Oracle® APEX

Developer's Companion



Release 26.1
G27196-01
June 2026



Oracle APEX Developer's Companion, Release 26.1

G27196-01

Copyright © 2026, Oracle and/or its affiliates.

Primary Author: Steve Muench

Contributing Authors: CM Dietrich

This software and related documentation are provided under a license agreement containing restrictions on use and disclosure and are protected by intellectual property laws. Except as expressly permitted in your license agreement or allowed by law, you may not use, copy, reproduce, translate, broadcast, modify, license, transmit, distribute, exhibit, perform, publish, or display any part, in any form, or by any means. Reverse engineering, disassembly, or decompilation of this software, unless required by law for interoperability, is prohibited.

The information contained herein is subject to change without notice and is not warranted to be error-free. If you find any errors, please report them to us in writing.

If this is software, software documentation, data (as defined in the Federal Acquisition Regulation), or related documentation that is delivered to the U.S. Government or anyone licensing it on behalf of the U.S. Government, then the following notice is applicable:

U.S. GOVERNMENT END USERS: Oracle programs (including any operating system, integrated software, any programs embedded, installed, or activated on delivered hardware, and modifications of such programs) and Oracle computer documentation or other Oracle data delivered to or accessed by U.S. Government end users are "commercial computer software," "commercial computer software documentation," or "limited rights data" pursuant to the applicable Federal Acquisition Regulation and agency-specific supplemental regulations. As such, the use, reproduction, duplication, release, display, disclosure, modification, preparation of derivative works, and/or adaptation of i) Oracle programs (including any operating system, integrated software, any programs embedded, installed, or activated on delivered hardware, and modifications of such programs), ii) Oracle computer documentation and/or iii) other Oracle data, is subject to the rights and limitations specified in the license contained in the applicable contract. The terms governing the U.S. Government's use of Oracle cloud services are defined by the applicable contract for such services. No other rights are granted to the U.S. Government.

This software or hardware is developed for general use in a variety of information management applications. It is not developed or intended for use in any inherently dangerous applications, including applications that may create a risk of personal injury. If you use this software or hardware in dangerous applications, then you shall be responsible to take all appropriate fail-safe, backup, redundancy, and other measures to ensure its safe use. Oracle Corporation and its affiliates disclaim any liability for any damages caused by use of this software or hardware in dangerous applications.

Oracle®, Java, MySQL, and NetSuite are registered trademarks of Oracle and/or its affiliates. Other names may be trademarks of their respective owners.

Intel and Intel Inside are trademarks or registered trademarks of Intel Corporation. All SPARC trademarks are used under license and are trademarks or registered trademarks of SPARC International, Inc. AMD, Epyc, and the AMD logo are trademarks or registered trademarks of Advanced Micro Devices. UNIX is a registered trademark of The Open Group.

This software or hardware and documentation may provide access to or information about content, products, and services from third parties. Oracle Corporation and its affiliates are not responsible for and expressly disclaim all warranties of any kind with respect to third-party content, products, and services unless otherwise set forth in an applicable agreement between you and Oracle. Oracle Corporation and its affiliates will not be responsible for any loss, costs, or damages incurred due to your access to or use of third-party content, products, or services, except as set forth in an applicable agreement between you and Oracle.

For information about Oracle's commitment to accessibility, visit the Oracle Accessibility Program website at http://www.oracle.com/pls/topic/lookup?ctx=acc&id=docacc.

Oracle customer access to and use of Oracle support services will be pursuant to the terms and conditions specified in their Oracle order for the applicable services.

Contents

Preface

Audience i
Dedication i
Related Documents i
Conventions i
About the Author ii

1 Introduction

1.1 Browser-Based Enterprise App Development 1
1.2 Model-Driven, Direct Data Access 2
1.3 Extensible, Powerful Components 3
1.4 Responsive, Polished User Interface 3
1.5 Complete Application Platform 4
1.6 Open, Agent-Ready APEXlang 4
1.7 Spec-Driven App Scaffolding 5
1.8 Pricing and Deployment Options 6
1.9 Active, Helpful Community 6
1.10 App Development as a Service 8
1.11 Learning Resources and Free Workspaces 9

2 Working with Local Data

2.1 Database Concepts Primer 1
2.1.1 Database Tables 2
2.1.2 Structured Query Language (SQL) 3
2.1.2.1 Basic SQL Statements 4
2.1.2.2 SQL Mentor at Your Side 5
2.1.3 Database Views, Triggers, and Packages 6
2.1.3.1 Database Views 7
2.1.3.2 Database Triggers 7
2.1.3.3 Database Packages 8
2.2 Working with Sample Data 10
2.3 Installing a Sample Dataset 12

Copyright © 2026, Oracle and/or its affiliates.

2.4 Importing Data from a File 13
2.5 Creating Data Models with Quick SQL 13
2.6 Creating a Data Model Using Natural Language 15

3 Simplifying User Experience Design

3.1 Starting a New Application 2
3.2 Creating Pages Quickly 6
3.3 Working in Page Designer 7
3.3.1 Key Page Designer Areas 7
3.3.1.1 Rendering, Layout, and Property Editor 8
3.3.1.2 Configuring Settings with Property Editor 9
3.3.1.3 Configuring Dynamic Behavior 11
3.3.1.4 Validation, Processing, and Branching 13
3.3.1.5 Searching Your Page 14
3.3.1.6 Help on Any Setting at Your Fingertips 14
3.3.2 Page Layout Based on Template with Slots 15
3.3.3 Adding Regions in Page Designer 16
3.3.4 Adding Items in Page Designer 18
3.4 Running and Iteratively Editing Pages 19
3.5 Adding Interactivity to Pages 20
3.6 Universal Theme Design System 22
3.6.1 Theme Styles 23
3.6.2 Universal Theme Reference App 23
3.6.3 The Role HTML and CSS Play to Format Data 26
3.7 Report Columns and Form Items 27
3.7.1 Configuring Report Region Columns 28
3.7.2 Synchronizing Column Changes 29
3.7.3 Deleting or Commenting Out Columns 30
3.7.4 Page Items in a Form Region 32
3.7.5 Flexible Form Page Item Layout 32
3.8 Cascading, List-Driven Items 34
3.8.1 List-Driven Items Available 35
3.8.2 Defining the List of Values 37
3.8.2.1 Defining a Local List of Values 38
3.8.2.2 Additional Shared LOV Features 40
3.8.2.3 Converting Local LOVs to Shared 40
3.8.3 Configuring Cascading Lists 40
3.9 Simple Layout for Any Screen Size 42
3.9.1 Flexible Grid of Columns and Rows 42
3.9.2 Templates and Options Set Look and Feel 52
3.9.2.1 Page Template Sets the Scene 53

Copyright © 2026, Oracle and/or its affiliates.

3.9.2.2 Region Template Styles a Section 54
3.9.2.3 Item Template Controls the Label 55
3.9.3 Nest Regions to Create Any Layout 56
3.10 Launching Like Native Apps 58

4 Referencing Data Values in Pages

4.1 Using Data Values in HTML Expressions 1
4.2 Using Data Values in Template Directives 2
4.3 Directives Reference in Your Dock 4
4.4 Using the Context-Sensitive Help Tab 4
4.5 Using Data Values in SQL and PL/SQL 5
4.5.1 Working with Page and App Data Values 6
4.5.2 Working with Column Data Values 6
4.5.3 Values Bind as Strings 7
4.5.3.1 Working with Data Values as Numbers 7
4.5.3.2 Working with Data Values as Dates 9
4.5.3.3 Working with Delimited Values 11
4.5.4 Using Data Values in Named Program Units 12
4.5.5 Configure Items to Submit and Return 13

5 Visualizing and Analyzing Data

5.1 Listing Data with Classic Reports 2
5.2 Personalizable Interactive Reports 3
5.3 Flexibly Styling and Acting on Data 4
5.4 Info Tiles Using Cards 6
5.5 Info at a Glance with Charts & Dashboards 8
5.5.1 Choosing a Chart Type 9
5.5.2 Defining Chart Series 9
5.5.3 Including Related Data on a Chart 11
5.6 Showing Data on Maps 12
5.6.1 Choosing a Map Layer Type 13
5.6.2 Adding a Layer of Points 14
5.6.3 Conveying Quantities with a Heat Map 15
5.6.4 Simplifying Points and Distance 16
5.6.5 Distance from a Point of Reference 17
5.7 Placing Events on a Calendar 19
5.7.1 Configuring a Basic Calendar 20
5.7.2 Adding Create and Edit Links 22
5.7.3 Changing Events with Drag and Drop 23
5.7.4 Joining In Additional Info Using a View 24

Copyright © 2026, Oracle and/or its affiliates.

5.7.5 Enhancing Event Display 25
5.8 Exploring Hierarchical Data in a Tree 27
5.8.1 Configuring a Basic Tree Display 27
5.8.2 Tracking and Remembering Selection 30
5.8.3 Adding Icons to the Tree 31
5.8.4 Parent/Child Data in a Tree 33
5.9 Dynamically Formatting Data 35
5.9.1 Generating HTML Tags to Present Data 36
5.9.2 Using CSS for Styling 37
5.9.3 Referencing Page Item Values 38

6 Enhancing Search for End-Users

6.1 Narrowing a Search Using Facet Filters 1
6.1.1 Defining Your Facets 3
6.1.2 Configuring Range Facets 3
6.1.3 Behavior of Multiple Selections 4
6.1.4 Counts, Dependency, and Charts 5
6.1.5 Working with Multi-Value Columns 6
6.1.6 Displaying Facets in Add Filter Dialog 6
6.1.7 Finding Text Across Columns 7
6.1.8 Showing Filter Chips and Total Row Count 8
6.2 More Space for Results with Smart Filters 9
6.2.1 Defining Your Filters 10
6.2.2 Smart Filters User Experience 11
6.2.3 Configuring Suggestions 13
6.2.4 Even More Space for Search Results 14
6.3 Unified Search Results Across Sources 15
6.3.1 Defining a Search Configuration 15
6.3.2 Using Search Configurations 17
6.3.3 Search Results Formatting Options 19
6.3.3.1 Custom Search Region Row Template 20
6.3.3.2 Custom Search Configuration Template 20
6.4 Showing Search Results in Other Regions 21
6.4.1 Custom Unified Search Experience 21
6.4.2 Showing Filtered Data in Another Region 22
6.4.2.1 Getting Filtered Data Primary Keys 23
6.4.2.2 Using Filtered Data Primary Keys 24
6.4.2.3 Refreshing the Target Region 24
6.5 Using Page Items for Query by Example 25
6.5.1 Using Null Display Value 25
6.5.2 Filtering on a Page Item's Value 26

Copyright © 2026, Oracle and/or its affiliates.

6.5.3 Filtering on a Multi-Selection Page Item 27
6.5.4 Optimal Approach to Query by Example 27
6.5.5 Disabling Warn on Unsaved Changes 30

7 Adding Functionality with Plug-ins

7.1 Reusable Components Using HTML Skills 1
7.1.1 Template Component Overview 2
7.1.2 Custom Attributes for Data and Settings 3
7.1.3 Slots Enable Component Composition 4
7.1.4 Applying Template Using Directives 6
7.1.5 Controlling Component Availability 7
7.1.6 Templates Comprising the Component 7
7.1.7 Presenting Grouped Data 8
7.1.8 Configuring Pagination Support 10
7.1.9 Enabling Row Selection 11
7.1.10 Attribute Scopes and Availability to Templates 13
7.1.11 Action Positions and Action Templates 14
7.1.12 Defining Files and File URLs to Load 15
7.2 Using a Plug-in from apex.world 16
7.2.1 Tracking Employee Training Status 16
7.2.2 Downloading and Importing a Plug-in 18
7.2.3 Configuring the Kanban Plug-in 19

8 Understanding Page Flow & Session State

8.1 Navigating to Business Functions 1
8.2 App Options in the Navigation Bar 2
8.3 Configuring a Wizard Page Flow 3
8.3.1 Creating a Multi-Step Wizard 4
8.3.2 Multi-Step Wizard in Page Designer 6
8.3.3 Shared List Defines Wizard Pages 7
8.3.4 Branches Link Pages to Define Flows 8
8.3.5 Render, Submit, Branch, Repeat 9
8.3.5.1 Wizard Example Page Lifecycle 10
8.3.5.2 Computing Values on Render or Submit 11
8.3.5.3 Validating Submitted Data 12
8.3.5.4 Custom Processing on Render or Submit 12
8.3.5.5 Best Practice Use of Invoke API 13
8.3.5.6 Making Almost Any Element Conditional 16
8.4 Additional Page Flow Features 16
8.4.1 Understanding AJAX and Redirect 17

Copyright © 2026, Oracle and/or its affiliates.

8.4.2 Breadcrumbs Visualize Drilldown Path 18
8.4.3 Modal Dialogs and Drawers 23
8.5 Understanding Session State Management 25
8.5.1 Use Cases for Temporary Storage 25
8.5.2 Session Context for Temporary Values 26
8.5.3 How APEX Manages User Session State 26
8.5.4 User-Specific Temporary Storage 28
8.5.5 Clearing Session State 28
8.5.5.1 Clearing State with Link Target Settings 28
8.5.5.2 Clearing State with a Page Process 30
8.5.6 Application Items 31
8.5.7 Using Temporary Collections 32
8.5.7.1 Key Collections Concepts 33
8.5.7.2 Tips for Maintainable Collection Use 34
8.5.8 Temporary Files 37
8.5.9 Inspecting Session State for Debugging 38
8.6 Studying a List and Edit Flow 39
8.6.1 Creating List and Edit Pages 40
8.6.2 Running the List and Edit Pages 42
8.6.3 Linking to the Edit Page 45
8.6.4 Retrieving Existing Row by Primary Key 46
8.6.5 Memory Only Storage for Form Page Items 49
8.6.6 Saving Changes with a Page Process 50
8.6.7 Conditional Database Action Buttons 51
8.6.8 Refreshing List On Dialog Close 53

9 Editing Data in a Grid

9.1 Getting to Know the Interactive Grid 1
9.2 Editing Collection Data with a Grid 11
9.3 Understanding and Customizing Grid Behavior 18

10 Working with Master/Detail Data

10.1 Reviewing Master/Detail Wizard Patterns 1
10.2 Configuring Master/Detail Grids 2
10.3 Showing Master/Detail Data Side by Side 6
10.4 Drilling Down to Edit Master and Details 14

Copyright © 2026, Oracle and/or its affiliates.

11 Controlling Access with Users and Roles

11.1 Requiring Authenticated Access to a Page 2
11.2 Authenticating Access with APEX Accounts 3
11.3 Assigning Users to Roles 5
11.4 Shaping Experience with Rules and Roles 6
11.4.1 Declaring Authorization Scheme Rules 7
11.4.2 Authorizing Menu Access 9
11.4.3 Authorizing Page Access 9
11.4.4 Authorizing Page Item or Column Access 11
11.4.5 Authorizing Button Access 12
11.4.6 Authorizing Edit Link Column Access 13
11.4.7 Experiencing Authorization in Action 14
11.5 Enabling Row-Level Data Security Policy 17
11.5.1 Using SQL and PL/SQL Authorization Rules 18
11.5.1.1 Defining Authorization Rule with SQL 18
11.5.1.2 Using PL/SQL for an Authorization Rule 19
11.5.2 Enabling Row-Level Data Security 20
11.5.2.1 Ensuring Execute Privilege on DBMS_RLS 21
11.5.2.2 Defining the Security Policy Function 21
11.5.2.3 Adding and Enabling the Security Policy 22
11.5.2.4 Referencing Other Context Information 24
11.6 Refining Salary Review with Data Security 25
11.6.1 Switching to Data Security-Enabled View 25
11.6.2 Driving Read-Only with Authorization Rule 26
11.6.3 Adjusting Salary Review Page Access 27
11.6.4 Enabling Salary Review Menu for Managers 28
11.6.5 Testing Enhanced Salary Review Page 28
11.7 Integrating an External Identity Provider 30
11.7.1 Defining Users in an IAM Domain 31
11.7.2 Creating Groups in an IAM Domain 32
11.7.3 Assigning IAM Domain Users to Groups 32
11.7.4 Configuring Confidential App for APEX 33
11.7.5 Defining Credential for Identity Provider 35
11.7.6 Authenticating with an External Provider 36
11.7.7 Enabling Groups in Post-Auth Procedure 37
11.7.8 Sourcing App Groups from Custom Code 40
11.7.9 Switching Authorization Rules to Custom 41
11.7.10 Making Social Sign-in the Current Scheme 42
11.7.11 Experiencing External Identity Provider 43
11.8 Exploring Context for Authorization Rules 44

Copyright © 2026, Oracle and/or its affiliates.

12 Applying Artificial Intelligence

12.1 Configuring a Generative AI Service 1
12.2 Exploring Data with Natural Language 2
12.3 Defining an Agent to Achieve a Goal 9
12.4 Creating an Agent-Driven Chatbot 18
12.5 Generating Text to Save Users Time 19
12.6 Finding by Meaning with Vector Search 21

13 Integrating Remote Data

13.1 Using Remote Oracle and MySQL Data 1
13.1.1 Referencing a REST-Enabled SQL Service 2
13.1.2 REST-Enabled SQL Service URL 4
13.1.3 Using a REST-Enabled SQL Service 5
13.2 Configuring Secure Web Credentials 5
13.3 Using REST APIs as Data Sources 6
13.3.1 Using a Simple HTTP REST Data Source 7
13.3.2 Service URL Uses Remote Server 9
13.3.3 Data Profile Helps Turn JSON Into Rows 10
13.3.4 Retrieving Remote Data Page by Page 14
13.3.5 REST Operations and Database Actions 16
13.3.6 Configuring a Request Body Template 19
13.3.7 Supply and Return Values with Parameters 21
13.4 Invoking a REST Operation Declaratively 21
13.5 Using REST APIs in Code 26
13.5.1 Processing REST Source Rows in Code 26
13.5.1.1 Querying REST Source Rows in Code 26
13.5.1.2 Modifying REST Source Rows in Code 28
13.5.2 Invoking a REST Operation Programmatically 34
13.5.2.1 Configuring REST Source for Invocation 35
13.5.2.2 Invoking REST Operation in Code 38
13.5.3 Calling REST APIs Without a Data Source 39
13.6 Extra Fusion Apps, ORDS, OData Features 40
13.7 Additional REST Data Source Features 41
13.7.1 Caching a GET Operation's Response 42
13.7.2 Synchronizing Data Locally 42
13.7.3 Adding Computed Data Profile Columns 42
13.7.4 Configuring Default Filtering and Sorting 43
13.7.5 Filter, Sort, Join, or Aggregate Remote Data 44
13.7.5.1 Understanding the JSON_TABLE Query 44
13.7.5.2 Locally Filtering and Sorting Remote Data 45

Copyright © 2026, Oracle and/or its affiliates.

13.7.5.3 Using Custom SQL Involving Remote Data 46
13.7.6 Creating a Custom REST Source Plug-in 47
13.8 Debugging REST API Errors 48

14 Exposing APIs for Integration

14.1 Reviewing Action Items API Requirements 1
14.1.1 Exploring Action Item Teams Example 2
14.1.2 Understanding the Integration Scenario 4
14.2 Reviewing REST API Basics 5
14.2.1 Reviewing HTTP Methods for REST APIs 5
14.2.2 Inspecting JSON Syntax 6
14.2.3 Constructing JSON in SQL 7
14.2.4 Working with JSON in PL/SQL 9
14.2.4.1 Parsing JSON in PL/SQL 9
14.2.4.2 Treating an Element More Specifically 10
14.2.4.3 Constructing JSON in PL/SQL 11
14.2.4.4 Serializing JSON in PL/SQL 12
14.3 Planning Your ORDS Service Module 12
14.3.1 Choosing a Schema Alias 13
14.3.2 Identifying URL Templates in a Module 13
14.3.3 Choosing GET Handlers to Read Data 13
14.3.4 Creating a View for GET Handlers 14
14.4 Using SQL Developer Web REST Designer 15
14.4.1 Accessing SQL Developer Web REST Designer 15
14.4.2 Navigating ORDS Service Definitions 16
14.5 Creating and Testing Read-Only APIs 20
14.5.1 Defining a Read-Only Collection Query 20
14.5.2 Testing Handler with curl or Postman 21
14.5.3 Filtering/Paging Collection Query Endpoints 24
14.5.4 Configuring a Collection Item Handler 26
14.6 Enforcing Business Logic in REST APIs 28
14.6.1 Reviewing Rules for Data Changes 29
14.6.2 Defining Text Messages for Errors 29
14.6.3 Using Text Messages in Pages and PL/SQL 30
14.6.4 Handling Errors in a Common Way 30
14.6.5 Coding Business Rules for Reuse 32
14.6.6 Experiencing Validation Failure in UI 36
14.6.7 Receiving Validation Error in REST API 37
14.7 Controlling REST API Behavior Completely 39
14.7.1 Providing Complete Set of ORDS Handlers 40
14.7.2 Using Handler Bind Variables 41

Copyright © 2026, Oracle and/or its affiliates.

14.7.3 Keeping PL/SQL Handlers Maintainable 41
14.7.4 Defining Explicit Handler Parameters 42
14.7.5 Reusing Common REST API Handler Code 43
14.7.6 Reading and Writing Binary Content 45
14.8 Layering Packages for Modularity 46
14.8.1 Studying API Handler Package Layers 46
14.8.2 Examining Core Action Item API Logic 49
14.8.2.1 Constructing or Querying an Action Item 49
14.8.2.2 Retrieving Action Items with Paging 51
14.8.2.3 Modifying Action Items with Pre/Post Logic 52
14.8.2.4 Inspecting the Core Action Item API Body 57
14.9 Simplifying REST APIs in Oracle 26ai 64
14.9.1 Using JSON Type and Duality Views 64
14.9.1.1 Storing and Accessing JSON Type Data 65
14.9.1.2 Defining JSON Relational Duality View 68
14.9.1.3 Querying JSON Relational Duality View 69
14.9.1.4 Modifying JSON Using a Duality View 70
14.9.1.5 Flexing the Duality of Duality Views 71
14.9.1.6 Preventing Lost Updates with ETAG 72
14.9.1.7 Comparing _id and RESID in Duality View 74
14.9.2 Exploring ORDS AutoREST for Duality View 74
14.9.2.1 Enabling AutoREST for a Duality View 75
14.9.2.2 Exercising Duality View AutoREST Endpoint 77
14.9.2.3 Enforcing More Complex Business Logic 78
14.9.3 Configuring ORDS Handlers for 26ai API 78
14.9.4 Seeing How Duality View Simplifies Logic 80
14.9.4.1 Saving JSON Using SQL Without Parsing 81
14.9.4.2 Transforming JSON Declaratively in SQL 81
14.9.4.3 Processing and Accessing JSON Easily 82
14.9.4.4 Querying Predefined JSON Structure 82
14.9.4.5 Inspecting the Core 26ai Action Item API 82
14.10 Securing APIs with Role-Based Access Control 85
14.10.1 Defining ORDS Role and Privileges 86
14.10.2 Issuing OAuth Client for Authentication 87
14.10.3 Using REST API with an OAuth Client 89

15 Geocoding Addresses

15.1 Geocoding with User Confirmation 1
15.1.1 Producing GeoJSON from Coordinates 2
15.1.2 Avoiding SQL Expression Update Error 3
15.1.3 Identifying Address Text Fields 4

Copyright © 2026, Oracle and/or its affiliates.

15.1.4 Confirming an Address Interactively 4
15.1.5 Extracting Coordinates for Storage 7
15.1.6 Unprotecting Client-Set Hidden Items 9
15.1.7 Exploring Additional Page Features 9
15.1.7.1 Clicking Map Point to Open Edit Dialog 10
15.1.7.2 Refreshing Region on Dialog Close 11
15.1.7.3 Stretching to Fill Vertical Space 13
15.1.7.4 Preventing Enter Key from Submitting 20
15.1.7.5 Triggering Geocoding with Enter Key 21
15.1.7.6 Including Context Info in Dialog Title 22
15.1.7.7 Providing App-Wide Helper Functions 24
15.1.7.8 Using Translatable Text in JavaScript 28
15.2 Geocoding in the Background 29
15.2.1 Selecting Items for Server-side Geocoding 30
15.2.2 Referencing Coordinates Item Values 30
15.2.3 Processing Collection of Matches 31
15.2.4 Configuring Match Mode 32

16 Uploading, Viewing, and Downloading Files

16.1 Reviewing Break Room and Referrals Pages 1
16.1.1 Sharing Photos in the Break Room 2
16.1.2 Submitting a Referral to Earn a Bonus 3
16.1.3 Studying Supporting Tables and Views 4
16.2 Storing Employee ID in an Application Item 7
16.2.1 Defining an Application Item 7
16.2.2 Computing an Application Item at Login 8
16.2.3 Referencing the App Item Value 9
16.3 Uploading an Image to a BLOB Column 10
16.3.1 Configuring Upload Image Item Source 10
16.3.2 Controlling Image Upload User Experience 12
16.3.3 Enlarging Image Upload Icon Preview 13
16.3.4 Configuring Application-Wide Style Rules 14
16.3.5 Posting a Break Room Photo 16
16.4 Displaying BLOB Column Images 17
16.4.1 Showing BLOB Column Images on Cards 18
16.4.2 Previewing Image in a Form Region 20
16.4.3 Displaying BLOB Images in Classic Reports 22
16.4.4 Using BLOB Images in Interactive Reports 24
16.4.5 Including BLOB Images in Content Row 26
16.4.6 Viewing BLOB Images in Search Results 28
16.4.7 Showing BLOB Images in Interactive Grid 30

Copyright © 2026, Oracle and/or its affiliates.

16.4.7.1 Using GET_BLOB_FILE_SRC for Image URL 31
16.4.7.2 Configuring Reference Image Page Item 33
16.4.7.3 Serving Inline BLOB Images with a Page 34
16.4.7.4 Using an Inline Image Serving Page 36
16.4.7.5 Weighing X01 Against a Secure Page Item 37
16.4.8 Using BLOB Column Image URLs Anywhere 41
16.5 Uploading a File to a BLOB Column 42
16.5.1 Selecting an Open Job to Add Referral 43
16.5.1.1 Using Trigger Actions to Set Selected Job Info 43
16.5.1.2 Adjusting UI When Selected Job Changes 45
16.5.2 Passing Client-Side Data to Dialogs 46
16.5.2.1 Opening Server-Generated Dialog URL 47
16.5.2.2 Passing Client Data Directly to Dialogs 51
16.5.3 Configuring File Upload to a BLOB Column 60
16.5.4 Ensuring the Automatic DML Process 62
16.5.5 Controlling File Upload Item Behavior 63
16.5.6 Submitting a Job Referral CV 64
16.6 Previewing BLOB Column PDF Files Inline 66
16.6.1 Creating PDF Viewer Template Component 66
16.6.2 Serving an Inline PDF from a BLOB Column 67
16.6.3 Using PDF Viewer in Job Referrals Page 69
16.6.4 Allowing Embedding from Same Domain 72
16.7 Downloading a File for Offline Use 73
16.7.1 Configuring Single-File Download Action 73
16.7.2 Downloading ZIP Archive of Multiple Files 74
16.8 Uploading Multiple Images or Files 75
16.8.1 Handling Temporary Uploaded Files 76
16.8.1.1 Controlling Retention of Temporary Files 76
16.8.1.2 Referencing Temp Files by Unique Name 76
16.8.1.3 Displaying Temp File Names and Types 77
16.8.1.4 Processing Temporary Uploaded Files 77
16.8.1.5 Deleting Temporary Uploaded Files 78
16.8.2 Posting Multiple Break Room Photos 78
16.8.2.1 Configuring Multiple File Image Upload 79
16.8.2.2 Experiencing Post Multiple Photos Flow 79
16.8.2.3 Using a Collection to Capture Image Titles 81
16.8.2.4 Joining Collection with Temp Files Table 81
16.8.2.5 Preparing and Saving Images with Titles 83
16.9 Using an OCI Object Bucket for Files 95
16.9.1 Understanding Object Bucket URL Format 96
16.9.2 Defining OCI Web Credential for REST APIs 97
16.9.3 Listing Buckets with REST Data Source 98

Copyright © 2026, Oracle and/or its affiliates.

16.9.4 Using REST Data Source for Bucket Files 101
16.9.5 Serving Inline File from Object Bucket 102
16.9.6 Browsing and Selecting Bucket Files 105
16.9.6.1 Displaying Bucket Files with Content Row 105
16.9.6.2 Configuring Content Row for Bucket Files 106
16.9.6.3 Enhancing REST Data with SQL Expressions 108
16.9.6.4 Defining and Using Global Substitutions 110
16.9.6.5 Configuring Selection, Paging, and Avatar 112
16.9.7 Storing an Uploaded File in Object Bucket 112
16.9.8 Exploring Pre-Authenticated Requests 115
16.9.8.1 Understanding Object Bucket Visibility 115
16.9.8.2 Creating Pre-Authenticated Requests 116
16.9.8.3 Using Pre-Authenticated Requests 119

17 Loading External Data

17.1 Exploring External Data Examples 1
17.2 Creating a Data Load Definition 3
17.3 Augmenting Data Load Profile Columns 6
17.4 Generating and Using a Data Loading Page 8
17.4.1 Creating a Data Loading Page 8
17.4.2 Using a Data Loading Page 9
17.4.3 Understanding Data Loading Pages 11
17.4.3.1 Specifying the File and Uploading It 12
17.4.3.2 Validating the Uploaded File Type 15
17.4.3.3 Letting User Select Worksheet 16
17.4.3.4 Previewing External Data Before Loading 18
17.4.3.5 Loading Data with Native Page Process 20
17.5 Parsing Data Without Data Load Definition 21
17.6 Loading Multiple Files Including ZIPs 21
17.6.1 Adding XML, CSV, JSON Data Load Definitions 22
17.6.2 Setting File Upload to Handle Multiple Files 23
17.6.3 Processing Multiple Uploaded Files 24
17.6.4 Uploading Multiple Data Load Files 29

18 Grouping Logic and Background Execution

18.1 Exploring Employee Excellence for HR Reps 2
18.2 Outlining Logic Using Execution Chains 3
18.2.1 Setting Up Context After Submit 4
18.2.2 Making Processing Conditional 4
18.2.3 Sequencing Child Processes 5

Copyright © 2026, Oracle and/or its affiliates.

18.2.4 Nesting Execution Chains 6
18.2.5 Choosing Descriptive Process Names 7
18.3 Offloading Work to the Background 8
18.3.1 Configuring Chain to Run in Background 9
18.3.2 Outlining the Award Review Process 10
18.3.3 Printing Personalized Certificate 12
18.3.4 Emailing User with Attachments 12
18.3.5 Push Notifying User on Opted-in Devices 14
18.4 Serializing Background Process Jobs 15
18.5 Understanding Cloned Session State 17
18.5.1 Isolating Background Session State 17
18.5.2 Configuring Temporary File Handling 18
18.6 Reporting Background Progress 18
18.6.1 Reporting Background Process to Users 18
18.6.2 Accessing Background Process Status 19
18.6.3 Querying Progress of Background Process 20
18.6.4 Showing Background Process Progress Bar 21
18.6.5 Stretching Classic Report's Last Column 22
18.6.6 Clamping Text to a Fixed Number of Lines 24
18.7 Loading Data in the Background 26
18.8 Limiting and Throttling Background Jobs 27
18.9 Adjusting Resources for Background Jobs 28
18.10 Debugging Background Processes 29
18.10.1 Reviewing App Background Processes 29
18.10.2 Investigating Background Process Errors 30
18.10.3 Viewing Background Process Session State 32

19 Executing Repetitive Processing

19.1 Understanding Key Automation Concepts 1
19.2 Exploring an Automation Example 2
19.2.1 Winning Salary Increase Every Saturday 3
19.2.2 Identifying Automation Rows to Process 4
19.2.3 Configuring Supporting Automation Logic 5
19.2.4 Defining Automation Actions in Sequence 7
19.2.5 Establishing an Automation's Schedule 10
19.2.5.1 Using Interval Builder to Schedule Job 11
19.2.5.2 Using More Complex Schedule Expressions 12
19.2.6 Running an Automation on Demand 14
19.2.7 Viewing the Automation Log 14

Copyright © 2026, Oracle and/or its affiliates.

20 Notifying End Users

20.1 Displaying Success and Error Messages 1
20.1.1 Configuring Success and Error Messages 1
20.1.2 Using Validations for Multiple Errors 3
20.1.2.1 Validating By Returning True/False Result 4
20.1.2.2 Validating By Returning Error Text 6
20.1.3 Auto-Dismissing Success Messages 7
20.1.4 Showing Messages with Dynamic Actions 8
20.1.5 Showing Messages Programmatically 11
20.1.5.1 Adding an Error Message in PL/SQL 12
20.1.5.2 Displaying Messages with JavaScript 14
20.1.6 Customizing Error Message Display 19
20.1.6.1 Studying Default Error Message Behavior 19
20.1.6.2 Specifying an Error Handling Function 21
20.1.6.3 Examining Error Handling Function Code 22
20.1.6.4 Logging Error Info for Troubleshooting 24
20.1.6.5 Experiencing Custom Error Handling 26
20.2 Sending Emails with Templates 30
20.2.1 Setting Up Default From Email Address 30
20.2.2 Sending Email Without a Template 31
20.2.3 Defining Email Template with Directives 32
20.2.4 Using Email Template and Placeholders 33
20.2.5 Sending Emails from Business Logic 34
20.3 Delivering Push Notifications 34
20.3.1 Enabling Push Notifications in an App 35
20.3.2 Generating Key Pairs for Push Notifications 36
20.3.3 Letting Users Manage Notification Settings 37
20.3.4 Opting-In to Receive Push Notifications 38
20.3.5 Pushing a Smile to a Colleague 39
20.3.6 Overriding Notification Target Link 41
20.3.7 Pushing Notification from Business Logic 41

21 Producing Pixel-Perfect Reports

21.1 Setting Your App's Remote Print Server 1
21.2 Printing Report with an Excel Template 3
21.2.1 Formatting Data in Excel with Tags 3
21.2.2 Creating Layout for Document Template 5
21.2.3 Defining Report Query Sources 5
21.2.4 Assigning Sources a Data Loop Name 6
21.2.5 Using SQL Features in Report Sources 7

Copyright © 2026, Oracle and/or its affiliates.

21.2.6 Inspecting the Report Query JSON 8
21.2.7 Printing Report from a Page 9
21.2.8 Experiencing Report Generation as a User 11
21.3 Customizing Reports with Word Template 13
21.3.1 Formatting Data in Word with Tags 13
21.3.2 Setting Layout Data Loop Name 14
21.3.3 Overriding PDF Layout of Report Regions 15
21.3.4 Downloading Customized Report PDF 16

22 Translating Your Application

22.1 Exploring Woods HR Pages to Translate 1
22.2 Enabling the First Translation Language 3
22.3 Understanding Initial Text Message Sync 4
22.4 Exporting and Importing Translations 6
22.5 Determining User's Preferred Language 8
22.6 Experiencing App in Another Language 9
22.7 Translating Text as Your App Evolves 11
22.7.1 Adding a New Text Message 11
22.7.2 Using the Text Messages Picker 12
22.7.3 Entering New Text Directly Inline 14
22.8 Synchronizing Text Messages Anytime 15
22.9 Editing Translations in a Filterable Grid 16
22.10 Centralizing Reusable UI Text 17

23 Understanding Application Lifecycle

23.1 Identifying Application Artifacts 1
23.2 Exporting and Importing Applications 2
23.3 Working with Multiple Environments 4
23.4 Isolating Changes with Working Copies 5
23.5 Tracking Changes with Source Control 6
23.6 Summarizing Lifecycle Recommendations 7

24 Working with APEXlang

24.1 Why APEXlang Matters 2
24.2 Using APEXlang with AI Coding Agents 2
24.3 Exporting APEXlang from App Builder 3
24.4 Importing APEXlang in App Builder 5
24.5 Reading APEXlang Syntax 7
24.6 Using App Builder and External Tools Together 14

Copyright © 2026, Oracle and/or its affiliates.

24.7 Editing APEXlang in Visual Studio Code 15
24.8 Using SQLcl with APEXlang 19
24.9 APEXlang Inside App Builder 22

25 Automating Business Processes

25.1 Automating a Process with Workflow 1
25.1.1 Using the Diagram and Activities Palette 2
25.1.2 Surveying the Native Activities 3
25.1.3 Connecting and Routing Activities 4
25.1.3.1 Adding an Activity from the Palette 4
25.1.3.2 Naming a New Activity 5
25.1.3.3 Changing a Connection's Target Activity 6
25.1.3.4 Fine-Tuning Line Slope with Anchor Dots 7
25.1.3.5 Moving Connections with Grab Handles 9
25.1.3.6 Adding a Connection Between Activities 10
25.1.3.7 Bending a Connection with Elbow Dots 11
25.1.3.8 Adding Next Activity from Existing One 14
25.1.3.9 Configuring a Switch Connection 16
25.1.3.10 Inserting Activity Between Existing Ones 17
25.1.3.11 Moving a Group of Activities 19
25.1.3.12 Bringing Connections in Front of Activities 21
25.1.4 Resolving Errors Before Saving a Workflow 23
25.1.5 Associating a Table with a Workflow 23
25.1.6 Retrieving Data for an Activity 24
25.1.7 Defining Parameters and Variables 24
25.1.7.1 Creating a Workflow Parameter 25
25.1.7.2 Creating a Workflow Variable 26
25.1.8 Understanding Background Execution 26
25.1.9 Getting Approval or User Input in a Workflow 27
25.1.9.1 Assigning Work to a User's Task Inbox 27
25.1.9.2 Waiting for a Signal to Continue 77
25.1.10 Identifying Workflow Participants 90
25.1.10.1 Configuring Static Workflow Participants 90
25.1.10.2 Determining Participants Dynamically 91
25.1.10.3 Passing in Workflow Participants 92
25.1.11 Listing Workflows and Viewing Details 92
25.2 Exploring Patient Onboarding Example 94
25.2.1 Understanding the Current Process 95
25.2.1.1 Inquiring About a Procedure 95
25.2.1.2 Onboarding the Patient Manually 96
25.2.2 Identifying Automation Opportunities 96

Copyright © 2026, Oracle and/or its affiliates.

25.2.3 Starting a Patient Onboarding Workflow 97
25.2.4 Associating the Patients Table 97
25.2.5 Configuring a Meaningful Title 98
25.2.6 Building Patient Onboarding Visually 99
25.2.6.1 Applying Activities to Patient Onboarding 100
25.2.6.2 Evaluating Spatial Distance with an API 100
25.2.6.3 Switching Path Using a Condition 104
25.2.6.4 Predicting Results with Machine Learning 105
25.2.6.5 Visualizing Workflow Progress 109
25.2.6.6 Awaiting Completion of a Human Task 111
25.2.6.7 Emailing Users a Rejection 114
25.2.6.8 Provisioning a New User Account 120
25.2.6.9 Waiting Until Patient Completes Profile 126
25.3 Automating Business Entity Lifecycle Logic 133
25.3.1 Learning the Medical Procedure Lifecycle 134
25.3.2 Seeing the Medical Procedure Lifecycle 135
25.3.3 Modeling the Medical Procedure Lifecycle 136
25.3.4 Starting the Lifecycle Workflow 137
25.3.5 Following a Medical Procedure Lifecycle 139
25.3.5.1 Launching Woods Clinic Application 139
25.3.5.2 Scheduling a Medical Procedure 142
25.3.5.3 Receiving Medical Procedure Schedule 146
25.3.5.4 Discharging Patient After Procedure 147
25.3.5.5 Preparing Pixel-Perfect PDF Invoice 150
25.3.5.6 Reviewing Invoice with Insurance Provider 157
25.3.5.7 Regenerating Invoice in Background 160
25.3.5.8 Invoicing Patient for the Balance 163
25.3.5.9 Acknowledging Receipt of Payment 167
25.3.5.10 Confirming the Lifecycle is Complete 170
25.3.6 Notifying Workflow of Data Changes 172
25.3.6.1 Understanding Lifecycle Event Strategy 172
25.3.6.2 Simplifying Event Management 173
25.3.6.3 Adding Events Using a Trigger 175
25.3.6.4 Processing Events with an Automation 176
25.3.6.5 Continuing a Waiting Lifecycle Workflow 178
25.4 Maintaining Workflow Processes 180
25.4.1 Importing vs. Deleting an Application 180
25.4.2 Recognizing Development Mode 181
25.4.3 Debugging Business Process Logic 182
25.4.3.1 Querying Recent Debug Messages 182
25.4.3.2 Enabling Debug Tracing Proactively 183
25.4.3.3 Viewing Debug Trace Information 184

Copyright © 2026, Oracle and/or its affiliates.

25.4.3.4 Adding Custom Debug Messages 186
25.4.4 Accessing Workflow and Task Data 187
25.4.5 Understanding Task and Workflow Purging 189
25.4.6 Activating an In Development Workflow 190
25.4.7 Deploying Workflows to Production 191
25.4.8 Deleting a Workflow with No Instances 192
25.4.9 Removing Development Instances 192

Index

Copyright © 2026, Oracle and/or its affiliates.

List of Figures

1-1	App Builder Landing Page for a DEMOAPEX Workspace on apex.oracle.com	2
1-2	Examples of APEX Apps on Mobile and Desktop	4
1-3	Using AI Coding Agent in VS Code to Fix Problem Highlighted in APEXlang File	5
1-4	Generating Scaffolded App with AI Produced Blueprints to Converge Requirements	6
1-5	APEX Community Discussion Forum, Built Using APEX	7
1-6	APEX World Community Site for News, Component Exchange, and More	8
1-7	Statistics for Large Internal APEX Instance	9
1-8	Browsing Other Learning Resources	9
2-1	Maintaining Database Objects in Object Browser	2
2-2	Two Tables with Primary Keys and Foreign Key	3
2-3	Experimenting with SQL in SQL Commands	5
2-4	APEX Assistant Helps Write SQL	6
2-5	Choosing Sample Data as a Data Source in the Create Page Wizard	11
2-6	Changing the Type of Sample Data in Page Designer	12
2-7	Loading an Excel Spreadsheet into a New Table in Data Workshop	13
2-8	Iteratively Designing a New Data Model with Quick SQL	15
2-9	Explaining the Data Model to APEX Assistant	16
2-10	Reviewing the Quick SQL Data Model Diagram	16
3-1	App Builder's Home Page Lists All the Applications in Your Workspace	2
3-2	Creating a New Application from the App Builder Home Page	3
3-3	Create an App by Assigning a Name, or Use a Wizard for More Options	4
3-4	Application Landing Page Shows a List of Pages the App Contains	5
3-5	Toggle Between a List and Icon View of Your Application's Pages	5
3-6	Create Page Wizard	6
3-7	Rendering, Dynamic Actions, Processing, and Page Shared Components Tabs	8
3-8	Using Property Editor with Rendering and Layout Tabs	9
3-9	Working with Properties of Multiple Selected Components	10
3-10	Searching the Property Editor to Find a Setting	10
3-11	Pinning a Property Editor Search to Work on Multiple Components	11
3-12	Define Triggered Actions on Buttons and Menus in the Rendering Tab	12
3-13	Dynamic Action Tabs Shows Event Handlers and Actions	13
3-14	Processing Tab Shows Page Submit Validation and Processing	14
3-15	Find and Navigate to Anything Anywhere on Your Page	14
3-16	Help Tab Shows Detailed Information About Selected Setting	15
3-17	Blank Page with Slots Based on Page Template	15
3-18	Region Palette in Page Designer	16

Copyright © 2026, Oracle and/or its affiliates.

3-19	Dragging and Dropping Regions from the Palette into Page Slots	17
3-20	Search Page for Patients in APEX Page Designer	18
3-21	Page Items Palette in Page Designer	18
3-22	Adding a Sort By Item and Search Facets in APEX Page Designer	19
3-23	Running Patients Search Page to See Map Not Refreshing Yet	20
3-24	Dynamic Actions Add Interactive Behavior	21
3-25	Map Reflects Filtered Result and Auto-Centers and Zooms	22
3-26	Customizing the Theme Style with Theme Roller	23
3-27	Universal Theme Components Reference	24
3-28	Calendar Reference	25
3-29	Map Region Reference in "Vita Dark" Theme Style	26
3-30	Browser Display of Simple HTML and CSS Example	27
3-31	Columns of an Interactive Report Region in Page Designer	29
3-32	Synchronizing Columns After Altering a Region's Data Source	30
3-33	Commenting Out a Report Region Column	31
3-34	Form Region Page Items	32
3-35	Form Region Source and Region Layout Properties	33
3-36	Flexible Layout of Form Region Page Items at Runtime	34
3-37	Single Selection List Items	35
3-38	Single- or Multi-Selection List Items	35
3-39	Multi-Selection List Items	36
3-40	Text with Autocomplete Page Item	36
3-41	Query by Example Select One Page Item with Null Display Value	37
3-42	Defining a List of Values with Static Values	39
3-43	Converting Locally-Defined List of Values to Shared	40
3-44	Department Page Item as Cascading Parent Item for Employees List	41
3-45	Dividing Horizontal Space Evenly Using a 12 Column Grid	43
3-46	Automatic Column Spanning Using Only "Start New Row"	44
3-47	Component Tree Dictates Layout Order	45
3-48	Automatic Column Spanning at Runtime	46
3-49	Manually Specifying Column Span	47
3-50	Effect of Manually Specifying Column Spanning at Runtime	47
3-51	Layout Responds Automatically to Narrower Screen	48
3-52	Registration Page on a Mobile Device with Even Narrower Screen	50
3-53	The Same Grid of Columns and Rows Operates at Page Level	51
3-54	Disabling "Start New Row" for Map Region	52
3-55	Disabling "Start New Row" for Map Region	52

Copyright © 2026, Oracle and/or its affiliates.

3-56	Universal Theme Page Templates	53
3-57	Standard Region Template Options in the Universal Theme Reference App	54
3-58	Three Query by Example Page Items Using Optional Item Template	55
3-59	Same Page Using Optional - Above Item Template	55
3-60	Items With Floating Item Template Stretch to Fill Column Span	56
3-61	Department's Required Item Template Displays Visual Marker and Inline Text	56
3-62	Regions Can Contain Other Regions and Items	57
3-63	Responsive APEX Web Page With Nested Regions	58
3-64	Tapping to Install an APEX as a Native App on iPhone	59
3-65	Full Screen iPhone App Experience with APEX Progressive Web App	61
3-66	APEX To Go Mobile Sample App on iPhone	63
4-1	Using {loop/} Directive to Format Tags as Chips	3
4-2	Template Directives in Universal Theme Reference App as PWA	4
4-3	Help Tab Provides Content-Sensitive Assistance	5
4-4	Numeric Range Using Two Number Field Page Items	8
4-5	Expected Validation Error Treating Numeric Items as Numbers	9
4-6	Date Range Using Two Date Picker Page Items	10
4-7	Expected Validation Error Treating Date Items as Dates	10
5-1	Classic Report Region Showing Employees Data with Pagination	2
5-2	Filtered, Highlighted, Grouped Employees with Computed and Aggregate Fields	3
5-3	Content Row Region with Pagination and Action Menu Entries	5
5-4	Content Row Configured for Multiple Row Selection	6
5-5	Employees Shown in a Cards Region	7
5-6	Cards with Media and an Action Button Configured	8
5-7	Native Chart Types to Choose From for a Chart Region	9
5-8	Sum of Employees Salaries and Commissions by Department	10
5-9	Stacked Bar Chart in Page Designer	11
5-10	Stacked Bar Chart Now Showing Joined Department Names	12
5-11	Five Different Types of Map Layer in the Create Page Wizard	13
5-12	Maps Showing Employees Addresses	15
5-13	Point Layer and Heat Map Layer on a Map	16
5-14	Showing the Office and Employees Distance Away	19
5-15	EMP_MEETINGS Table Records for EMP to EMP Scheduled Events	21
5-16	Four Meetings in Employee KING's Schedule with Dynamic Action on Select List	22
5-17	Configuring Create Link to Pass Start and End Date	23
5-18	Events Showing Purpose and Meeting With Employee Name	25
5-19	Employee KING's Week of Meetings	27

Copyright © 2026, Oracle and/or its affiliates.

5-20	Minimal Settings for a Working Tree Region	29
5-21	Basic Tree Region Showing EMP Management Hierarchy	30
5-22	Configuring Dynamic Action Step to Set Selected Node Id Into Hidden Page Item	31
5-23	Employee Hierarchy with Icons Based on Job	32
5-24	Exploring Parent/Child Data in a Tree	34
5-25	VT220-Style Chart of Employee Salaries Using Dynamic Content Region	36
5-26	Dynamic Content Region Using a Page Item Value	38
6-1	Faceted Search in the Left Column Filtering Classic Report in the Body	2
6-2	Search Facets in Page Designer	3
6-3	Specifying Range Facet Interval Low and High Values	4
6-4	Results Include Rows Matching Any Choice in Multi-Select Facets	5
6-5	Comparing Facet Value Distribution with Pie or Bar Chart	6
6-6	Applying Filters from the Add Filter Dialog	7
6-7	Finding Text Across Columns with the Search Facet	8
6-8	Row Count and Filter "Chips" Over Classic Report	9
6-9	Compact Smart Filters Region is a Single Search Field	9
6-10	Defining Multiple Filters in Page Designer	10
6-11	Filter Display Names and Icons Appear in Search Dropdown	11
6-12	Selecting One or More Options from a Filter	12
6-13	One Filter Used, Other Filters Remain Available	12
6-14	Applying Multiple Filters Using a Smart Filters Region	13
6-15	Typing Text to Search Across Columns or Find Filters	13
6-16	Suggestion Chips For Initial Filter to Apply	14
6-17	Smart Filters Region in the Navigation Bar	15
6-18	Mapping Search Configuration Columns to Standard Slots	16
6-19	Defining a Search Configuration's Link Target Page	17
6-20	Configuring a Search Region with Search Configurations	18
6-21	Finding and Drilling Down on Search Results	19
6-22	Consulting Help Tab for Search Region Result Row Template	20
6-23	Consulting Help Tab for Search Region Result Row Template	21
6-24	Show Smart Filtered Cards Region Data in Chart and Map	23
6-25	Custom Label Representing the Null Value in a Select List	26
6-26	Warn on Unsaved Changes for Query by Example Confuses Users	31
7-1	Reusing a Template Component Three Different Ways	3
7-2	Custom Attributes in the Settings Section	4
7-3	Configuring Slot Behavior in a Template Component	5
7-4	Using a Template Component's Slots in Page Designer	5

Copyright © 2026, Oracle and/or its affiliates.

7-5	Template Component Positions and Styles Composed Buttons and Regions	6
7-6	Different Usage of Content with Action Buttons Template Component	6
7-7	Templates Involved to Format a Multiple (Report) Component	8
7-8	Report Group Attributes in the Property Editor	9
7-9	Template Component Without and With Grouped Data	10
7-10	Normal and Grouped Data with Pagination	11
7-11	Row Selection Settings on Template Component Available as Multiple (Report)	11
7-12	Single & Multiple Selection Showing Entity Title	12
7-13	Processing Zero, One, and Multiple Selected Rows	13
7-14	Attributes of Different Scopes in Grouped Name/Value Pairs Component	14
7-15	Defining Menu and Submenu Options in an Action Position	14
7-16	End User Choosing a Template Component Row Action	15
7-17	Files List for a Template Component	15
7-18	Additional EMP Tables Tracking Training Course Progress	16
7-19	Employee Training Status Using Interactive Report	17
7-20	Kanban Board Mockup Showing Employee Training Status	17
7-21	Kanban Board Plug-in on apex.world	18
7-22	Importing a Plug-in Into Your Application	19
7-23	Defining Kanban Board Status Columns and Icons	20
7-24	Configuring the Kanban Region's SQL Query	20
7-25	Kanban Board Region Plug-in Shows Employee Training Status	21
8-1	Navigation Menu Lets Users Initiate Business Functionality	2
8-2	Navigation Bar List for Application Options and Logout	3
8-3	Three Step Wizard for Onboarding an Employee	4
8-4	Wizard Container on Each Page Shows Steps in a Flow	5
8-5	Defining the Wizard Steps in the Create Wizard Dialog	6
8-6	Wizard List Region Shows List of Pages as "Train" Stops	7
8-7	List Shared Component Defining Wizard Step Pages and Titles	8
8-8	Branches Connect Three Pages in Onboard New Employee Flow	9
8-9	Page Renders to Browser, then User Submits Page for Processing	11
8-10	Execute Code Page Process to Insert New Employee	13
8-11	Configuring Invoke API Process to Call Package Business Logic	15
8-12	Configuring Parameter Values for Invoke API	15
8-13	AJAX Requests and Redirects Complement Render, Submit, and Branch	18
8-14	Data Model for Employees Having One or More Dependents	19
8-15	Four-Level Drill Down Page Flow for Employees and Dependents	20
8-16	Editing Breadcrumbs Shared Component to Reflect Navigation Hierarchy	21

Copyright © 2026, Oracle and/or its affiliates.

8-17	Manage Employees Page Redirects to Employee Form Page for Edit	21
8-18	Drilling Down on Employee to See Their Dependents	22
8-19	Viewing Employee SMITH's Dependents and Drilling Down Again	22
8-20	Viewing SMITH's Dependent Janie: Breadcrumb Shows Path "Home"	23
8-21	Modal Dialog Form Page Called from a List Page	24
8-22	Modal Drawer Slides Out from the Start or End of the Window	24
8-23	Automatic Session State Management	27
8-24	Clearing Session State in Link Target Settings	29
8-25	Clear Cache Setting in Button Target Dialog	30
8-26	Using Clear Session State Page Process	31
8-27	App Computation Sets the Value of App Item After Authentication	32
8-28	Browsing Page Item Session State for Page 44	38
8-29	Browsing Application Items and Other Kinds of Session State	39
8-30	Create Page Wizard Choices for List and Edit Pages on the EMP Table	41
8-31	Configuring Breadcrumb, Navigation Menu Entry, and Icon	42
8-32	Navigation Menu Entry for Newly Created List Page	43
8-33	New Interactive Report Page Showing Employees	44
8-34	Interactive Report and Form Modal Drawer Pages in Action	45
8-35	Examining Wizard-Configured Link Column Settings	46
8-36	Form Initialization Page Process Retrieves Row by Primary Key	47
8-37	Form Region Using EMP Table as Data Source	47
8-38	Form Initialization Page Process Requires a Primary Key	48
8-39	All Form Region Page Items Use "Memory Only" Storage	49
8-40	List Page's (Create) Button Passes No Parameters to Edit Page	50
8-41	Declarative Form DML Handling with Native Page Process	51
8-42	Wizard-Created Form Page Delete Button	52
8-43	Wizard-Created Modal Form Page Cancel Button	53
8-44	Wizard-configured Close Dialog Page Process	54
8-45	Dynamic Action Event Handler for the Close Dialog Event	54
8-46	Dynamic Action Step to Refresh the Interactive Report	55
9-1	User Can Insert, Update, and Delete Multiple Rows Then Save All the Changes	2
9-2	Grid Offers Scrolling Pagination with Fetch on Demand	2
9-3	Date Picker is One of Many Item Types Available for Grid Column Editing	3
9-4	Grid Columns Can Use List-Driven Items for Data Entry	3
9-5	Actions Menu Presents the Grid Features You Enable for Users	4
9-6	Scrolling Through Large Dataset, Interactive Grid Fetches More Data On Demand	6
9-7	Updating a Grid Cell Using a Popup LOV	7

Copyright © 2026, Oracle and/or its affiliates.

9-8	SVG viewBox Area Used by Percent Dot on Bar Template Component Graphics	9
9-9	Visualizing Salaries as a Dot on a Bar Using a Template Component	11
9-10	Editing Shopping Cart Items in a Collection with Interactive Grid	12
9-11	Configuring Custom Grid PL/SQL to Save Cart Data to Collection	16
9-12	Custom When No Rows Found Message Improves Usability	18
9-13	Calling Helper Function to Refresh Grid Row on Dialog Close	21
9-14	Storing Primary Key Being Edited in a Hidden Item with a Computation	22
9-15	Configuring Item Values to Return to Calling Page on Dialog Close	22
9-16	Clearing the Edited Primary Key on Delete	23
10-1	Create Page Wizard Offers Three Master/Detail Patterns	2
10-2	Edit Multiple Master Rows and Respective Detail Rows, Then Save	3
10-3	Enter New Master Row with New Detail Rows as Part of Other Changes	4
10-4	Configuring the Master Region of a Detail Grid	5
10-5	Master Column	6
10-6	Side by Side Master/Detail Lets Users Search and See Details for Master Rows	7
10-7	Breakdown of Side by Side Page Template Slots and Regions in Them	8
10-8	Configuring an Alternative Classic Report Template	9
10-9	Find Media List Report Template	10
10-10	Study Row Template Substitutions	10
10-11	Configuring Value Attribute Pairs Classic Report Template	12
10-12	Editing the Selected Master Row in a Modal Drawer	13
10-13	Creating a New Detail Row for the Currently Selected Master	14
10-14	Master Detail in IR	15
10-15	Master/Detail Edit Page Lets User Maintain Master and Details Together	16
10-16	Editing Master and Detail Before Saving or Navigating Previous or Next	17
10-17	Defaulting Employee Department Foreign Key to Current Department Value	18
10-18	Configuring Form Navigation Items on Initialization Page Process	19
10-19	Execute Code Process with Editable Region Set Runs Once per Modified Grid Row	20
10-20	Creating a New Master Row and Multiple New Details at Once	21
11-1	Deliver User Experience Tailored to Each User's Functional Role	1
11-2	Deciding if a Page is Public or Requires Authentication	3
11-3	Oracle APEX Accounts Provide Default Authentication Scheme	3
11-4	Workspace Administrators Manage APEX User Accounts	4
11-5	Workspace Administrators Manage Users from Accounts List Page	4
11-6	Configuring Roles and Role Assignments	6
11-7	Configuring Authorization Scheme Rules Using Role Membership	7
11-8	Detail of Authorization Scheme Rule Based on Role Membership	8

Copyright © 2026, Oracle and/or its affiliates.

11-9	Applying Authorization Scheme to Navigation Menu Entries	9
11-10	Applying Authorization Scheme to a Page	10
11-11	Non HR Representatives See Error Navigating to Unauthorized Page	11
11-12	Applying Authorization Scheme to Columns	12
11-13	Applying Authorization Scheme to a Button	13
11-14	Applying Authorization Scheme to an Edit Link Column	14
11-15	End Users Experience and Features Adjust Automatically to their Role	15
11-16	Regular Employees Don't See Certain Columns, Create Button, or Edit Link	15
11-17	HR Representatives See Pay Columns, Create Button, and Edit Link	16
11-18	Only HR Representatives Can Use Salary Review Page	17
11-19	Two Additional Authorization Schemes Using PL/SQL and SQL	18
11-20	Using SQL to Define Authorization Scheme Rule for Managers	19
11-21	Using PL/SQL Expression to Combine Other Authorization Schemes	20
11-22	Change Salary Review to Use View with Data Security Policy	26
11-23	Make Sal and Comm Columns Read Only Unless User is HR Representative	27
11-24	Adjust Authorization Scheme to Let Managers and HR Reps Access	27
11-25	Adjust Menu Entry Authorization Scheme for Managers and HR Reps	28
11-26	Now Managers Like Jones See the Salary Review Page	29
11-27	Data Security Ensures Jones Sees Only Own Staff Members	29
11-28	Domain Users in Identity Cloud Service	31
11-29	Domain Groups in Identity Cloud Service	32
11-30	Domain Group Assignments in Identity Cloud Service	33
11-31	Domain Integrated App for Use by Woods HR APEX Application	34
11-32	Configuring Domain Integrated App OAuth Client	35
11-33	Web Credential for Use with IDCS Social Sign-in Authentication	36
11-34	Configuring Social Sign-in Authentication Scheme for IDCS	37
11-35	Configuring Post-Authentication Procedure to Enable Groups	38
11-36	Changing App to Source Groups from Custom Code	41
11-37	Changing Authorization Schemes to Use Custom Type for Dynamic Groups	42
11-38	Making Social Sign-in Authentication Scheme the Current One	43
11-39	Users Now Login to Woods HR Using OCI IAM Sign In	44
12-1	Typing a Request in Natural Language into the Interactive Report Search Field	2
12-2	Applied Report Features Appear as Chips, and Assistant Button Opens Chat Area	3
12-3	User Can Refine Results By Continuing the Conversation in the Chat Area	4
12-4	Interactive Report Features Like Highlighting Get Configured Using Natural Language	4
12-5	Charting and Pivoting Data Are Also Available Just by Asking	5
12-6	Hide or Show Columns and Add Break Groups with a Prompt	6

Copyright © 2026, Oracle and/or its affiliates.

12-7	Context You Add Helps Assistant Turn "the 80s" and "won an Oscar" into Filters	6
12-8	The Assistant Prompts the User for Clarification When Necessary	7
12-9	Report Context Helps Search with AI Assistant Better Understand User Requests	8
12-10	Column Context	9
12-11	Defining a Retrieve Data On-Demand Tool with Optional Parameters	11
12-12	Writing the SQL Query to Implement a Retrieve Data Tool	12
12-13	Defining an Execute Server-side Code On-Demand Tool with Required Parameters	12
12-14	Writing the PL/SQL Code to Implement an Execute Server-side Code Tool	14
12-15	Using the HR Assistant AI Agent in an AI Assistant Chatbot	15
12-16	Once get_employee_info Returns EMPNO, Assistant Completes Task Directly	16
12-17	Lifecycle of a Request Involving an AI Agent	17
12-18	Showing an AI Assistant Dialog Using a Trigger Action on a Button	19
12-19	Entering Patient Discharge Instructions	20
12-20	Preparing Discharge Letter Using AI Agent	21
12-21	Activities Standard Search Configuration on Activities Table Columns	22
12-22	Lucy's Choice of Words Produces No Matching Activities	23
12-23	Adding a VECTOR Column to an Existing Table in Object Browser	24
12-24	Creating a Vector Provider Based on a Database-Resident ONNX Model	25
12-25	Defining a Vector Search Type Search Configuration on the Activities Table	26
12-26	Configuring a Search Region to Use a Vector-Aware Search Configuration	27
12-27	Lucy Now Finds Meaningful Results, Even When Words Differ	27
12-28	Trying Another Search, Temporarily Studying Vector Distance of Results	28
12-29	Applying Best Score Proximity Cutoff to Search Configuration Results in a Content Row	31
12-30	Lucy Now Sees Only the Most Relevant Results	31
12-31	Searching by Meaning, Lucy Finds Exactly What She's Looking For	32
12-32	Multiple Activities Land Inside the Best Score Proximity Cutoff	33
13-1	Referencing a REST-Enabled SQL Service in Workspace Utilities	2
13-2	Providing Credentials for REST-Enabled SQL Service	3
13-3	Confirmation of Successfully Authenticated REST-Enabled SQL Connection	4
13-4	Using a Table in a Remote Database with REST-Enabled SQL	5
13-5	Defining a New REST Data Source Using Its Endpoint URL	9
13-6	Service Endpoint URL Base URL Comes from Remote Server Definition	10
13-7	Preview of Data Rows Returned from REST Data Source	11
13-8	Inspecting REST API Response: items Array and Nested links Array	12
13-9	Data Profile Columns Discovered Automatically by the Wizard	13
13-10	Adjusting the HIREDATE Data Profile Column's Data Type	14
13-11	Configuring a REST Data Source's Pagination Strategy	15

Copyright © 2026, Oracle and/or its affiliates.

13-12 Interactive Report Using a REST Data Source 15
13-13 Paging Through REST Data 5 Rows at a Time in an Interactive Report 16
13-14 Fetch Rows GET Operation for Discovered Simple HTTP REST Data Source 17
13-15 Complete Set of Operations to Support All Database Actions 17
13-16 Fetch Single Row Operation to Retrieve a Single Row by Primary Key 18
13-17 Ensuring the Primary Key Column is Enabled 19
13-18 Request Body Template in the PUT / Update Row Operation 20
13-19 Calling a REST Data Source Operation with Invoke API Page Process 22
13-20 Configuring REST Data Source Parameter Using a Page Item 22
13-21 Configuring Invoke API Page Process Parameters 23
13-22 REST Data Source with an Operation Used by Invoke API 24
13-23 REST Data Source Operation for Invoke API 25
13-24 Fetch Rows GET Operation to Retrieve One Employee by PK 32
13-25 Data Profile for Employee by PK (Simple HTTP) REST Data Source 33
13-26 Data Profile for Single Book Create REST API Response Payload 35
13-27 Data Profile Column Configuration for Estimated Delivery Date 36
13-28 POST Operation for Single Book Create REST API 37
13-29 Resulting REST Data Source for Simple Book Order 38
13-30 Interactive Report Based on Fusion Apps Sales Opportunities REST Data Source 48
13-31 Enable Debug Tracing to Diagnose HTTP 400 Error from REST Data Source 49
13-32 Identifying Most Recent Debug Trace and Drilling Down for Details 49
13-33 Quickly Navigating to an Error on the Debug Trace Timeline 50
13-34 Examining First Error and the Preceding REST API Call 51
13-35 Disabling Sorting on Assignment Mode Column 52
13-36 Problem Fixed: Users Can No Longer Choose the Non-Sortable Column 52
14-1 Application Page for Editing an Action Item and Assigned Team Members 3
14-2 Simple Schema for Action Items and Team Members Assigned to Them 4
14-3 Open Source Ticketing System Needs to Integrate with Your Action Item App 4
14-4 ORDS "v0" Module with Templates and Handlers for Action Items API 14
14-5 Opening SQL Developer Web from the App Builder 15
14-6 Navigating to View and Define REST Services in SQL Developer Web 16
14-7 Navigating to View and Define REST Modules in SQL Developer Web 17
14-8 Navigating to View and Configure a Module's Templates in SQL Developer Web 18
14-9 Navigating to View and Configure a Template's Handlers in SQL Developer Web 19
14-10 Listing Handlers for a Template in SQL Developer Web 20
14-11 Defining a Collection Query GET Handler for /v0/actionitems Template 21
14-12 Testing ORDS GET Handler on the Command Line with curl 22

Copyright © 2026, Oracle and/or its affiliates.

14-13	Formatting Output of curl Using jq Command Line Utility	23
14-14	Using Postman to Access Your New Action Items REST API	24
14-15	Defining an Collection Item GET Handler for /v0/actionitems/:id Template	27
14-16	Shared Component Text Messages Define Translatable Error Messages	29
14-17	Editing an Action Item and Assigned Team Members in a Page	37
14-18	Working with Action Items and Team Members Using Postman	39
14-19	ORDS "v1" Module with Templates and Handlers for Action Items API	40
14-20	An ORDS Template Handler Can Use PL/SQL for Complete Control	42
14-21	Defining a Custom Parameter for an ORDS Handler	43
14-22	Each ORDS Handler Call an ACTION_ITEMS_API_REST Procedure	47
14-23	Layered Package Design for Sharing Business Logic with a Page	48
14-24	Leo Updates the Title of the New Action Item	72
14-25	AutoREST-Enabling the ACTION_ITEMS_DV Duality View in SQL Developer Web	75
14-26	Configuring Object Alias for AutoREST-Enabled Duality View	76
14-27	Browsing AutoREST-Enabled Objects in SQL Developer Web	76
14-28	ORDS "v2" Module for Action Items API Using JSON Relational Duality View	79
14-29	Each ORDS Handler Call an ACTION_ITEMS_API_REST Procedure	79
14-30	Layered Package Design for Sharing Business Logic with a Page	80
14-31	Defining a Privilege in SQL Developer Web REST Designer	86
14-32	Granting the Action Items Privilege to External Application Integration Role	87
14-33	Store Client Secret Securely, then Share Securely with External App Team	88
14-34	ORDS OAuth Client Used to Authenticate External System Integration	88
15-1	Configuring Geocoded Address Page Item	4
15-2	Viewing KING's Existing Address in a Map Tooltip	5
15-3	Geocoded Address Displays Existing Address On Page Load	6
15-4	User Confirms Correct Address	6
15-5	Geocoded Address Map Updates to Show New Address	7
15-6	Extracting Longitude and Latitude from Geocoded Address	8
15-7	Disabling Value Protected on Hidden Item Set by Dynamic Actions	9
15-8	Clicking Refreshes on Return from Dialog to Show KING's Updated Address	11
15-9	Linking to Another Page from a Map Point	11
15-10	Map Refreshes on Return from Dialog to Show KING's Updated Address	12
15-11	Configuring a Dialog Close Dynamic Action Event Handler	13
15-12	Subtracting Height of Fixed Page Elements from Viewport Height	14
15-13	Determining the Element or Class to Target with a CSS Rule	15
15-14	Adding Inline CSS Rule to a Page	16
15-15	Vertical Stretch in Action on a Map Region in a Normal Page	17

Copyright © 2026, Oracle and/or its affiliates.

15-16	Relative CSS rem Unit Works Correctly at Any Zoom Percent	18
15-17	Vertical Stretching May Need to Account for Fixed Content Above and Below	19
15-18	Vertical Stretch in Action in a Model Drawer for a Geocoded Address Map	20
15-19	Adding CSS-Hidden Field to Stop Enter Key from Submitting Page	21
15-20	Triggering Address Geocoding Using the Enter Key	22
15-21	Showing ENAME in Modal Dialog Provides Useful Context	22
15-22	First Unsuccessful Attempt to Include Context Info in Dialog Title	23
15-23	The &P60_ENAME. in Dialog Title Comes Up Empty	23
15-24	List Page Shows Reference Path for a Static Application Files	27
15-25	Including Static App File Reference in Application JavaScript File URLs	28
15-26	Configuring a Translatable Text Message for Use in JavaScript	29
15-27	Configuring Match Mode for Server-side Geocoding	33
16-1	Break Room and Open Positions Referrals Pages in Woods HR App	2
16-2	Break Room Page Lets Employees Share Photos with Colleagues	3
16-3	Employees Submit Candidate CVs for Open Job Positions	4
16-4	Additional Tables to Support Woods HR Photos and CVs	5
16-5	Views Present Two "Interfaces" on Breakroom Photos Table	6
16-6	Shared Component Application Items List	8
16-7	Computing an Application Item's Value After Login	9
16-8	Using an Application Item to Default the Posting Employee ID	10
16-9	Configuring Source Properties for an Upload Image Item	11
16-10	Configuring Form DML Page Process	12
16-11	Configuring Properties That Control Image Upload Behavior	13
16-12	Defining a CSS Stylesheet in Static Application Files	15
16-13	Including a CSS Style Sheet in Every Page of Your App	15
16-14	Upload Image Page Item with Custom Preview Icon Size	16
16-15	User Sees Large Image Preview Before Clicking Upload	17
16-16	Cards Page Refreshes on Dialog Close to Show Latest Photo	17
16-17	Configuring Card Media Attributes to Display BLOB Column Image	19
16-18	Configuring Card Item Media from a CLOB Image Column	20
16-19	Configuring Display Image Form Page Item for Image BLOB	21
16-20	Previewing an Image from a BLOB Column in a Form Region	22
16-21	Configuring Display Image BLOB Column in Classic Report	23
16-22	Custom-Sized Images in a Classic Report	24
16-23	Configuring BLOB Image Column in an Interactive Report	25
16-24	BLOB Images in an Interactive Report	26
16-25	Configuring BLOB Image in a Content Row	27

Copyright © 2026, Oracle and/or its affiliates.

16-26 Content Row with BLOB Image in the Avatar Image Slot 28
16-27 Setting Up Search Configuration to Use a BLOB Image Column 29
16-28 Including BLOB Column Image in Search Page Results 30
16-29 Configuring HTML IMAGE tag to Display Only 32
16-30 Displaying BLOB Image Using Image URL via GET_BLOB_FILE_SRC 33
16-31 Form Image Item GET_BLOB_FILE_SRC References for BLOB Metadata 34
16-32 Pre-Rendering Invoke API Process to Download an Image 35
16-33 Configuring Break Room Image ID Parameter Value 36
16-34 Inline BLOB Images in an Interactive Grid Using Image-Serving Page 37
16-35 Passing Checksum-Protected Hidden Page Item for Image ID 38
16-36 Default Page Access Protection Setting Requires Checksum for URL Arguments 39
16-37 End Users Notice No Changes Using Checksum-Protected Image URLs 40
16-38 Clever Users See Error When Manually Modifying an Image URL 41
16-39 Using Computed Image URL in a Static Content Image Template 42
16-40 Selecting an Open Job Card with Existing Referrals 43
16-41 Selecting an Open Job with No Referrals Yet 44
16-42 Full Card Action Sets Select Job Referral Count and ID 45
16-43 Using True or False Dynamic Action Steps to Adjust the User Experience 46
16-44 Triggered Action Generates Dialog URL Before Opening It Using JavaScript 48
16-45 Calling JavaScript Helper Function to Open Dialog 49
16-46 "Always" Source Defaulting the Drawer Model URL Template 52
16-47 Computation Defaulting the Drawer Model URL Template 53
16-48 Opening Modal Drawer Passing Dynamic Values in URL Template 58
16-49 Adjusting Page Access Protection to Unrestricted to Accept Dynamic Parameters 59
16-50 Adjusting Session State Protection to Unrestricted to Accept Dynamic Parameters 60
16-51 Configuring File Upload Page Item 61
16-52 Ensure Automatic DML Page Process Targets the Form 62
16-53 Ensure Automatic DML Page Process Targets the Form 62
16-54 Database Action on Submitting Button Informs Automatic DML Process 63
16-55 Configuring File Upload Page Item Display Behavior 64
16-56 Submitting a New Job Referral Candidate's CV 64
16-57 Uploading a File to a BLOB Column 65
16-58 Previewing Newly Uploaded Job Referral Candidate's CV 65
16-59 Simple Template Component to More Easily View PDFs Inline 67
16-60 Pre-Rendering Invoke API Process to Download a PDF 68
16-61 Configuring Break Room Referral ID Parameter Value 69
16-62 Sourcing the PDF_URL in the Region's Query 70

Copyright © 2026, Oracle and/or its affiliates.

16-63 Configuring PDF URL 71
16-64 Candidate Picker Select List Refreshes PDF Viewer Template Component 71
16-65 Encountering an Error Due to "Embed in Frames" = Deny 72
16-66 Allowing Embedded Frame Only from the Same Domain 73
16-67 Downloading a Single File with the Native Dynamic Action 74
16-68 Downloading Multiple Files in a ZIP Archive with the Native Dynamic Action 75
16-69 Add Break Room Photos Page with Multiple Files Image Upload 79
16-70 Preview of Multiple Images Ready to Upload 80
16-71 Entering Titles for Multiple Posted Photos with Validation 80
16-72 Both New Photos with Titles Show Up Immediately in Break Room Photo Gallery 81
16-73 View Adds Titles to Temp Files by Joining with UPLOADED_PHOTOS Collection 82
16-74 Using an Execution Chain to Group Conditional Logic 84
16-75 Calling Page Button Target Link Sets P17_CLEAR = Y to Clear Collection 85
16-76 Invoke API Child Process Calls Procedure to Clear the Collection 86
16-77 Upload Button Triggers Invoke API to Add Image Names to Collection 87
16-78 Conditional Close Dialog Leaves User on Page After Image Upload to Set Titles 88
16-79 Collection Rows Determine if Image Upload or Titles Grid Displays 89
16-80 Image Preview Grid Allows Update and Delete, with No Toolbar 91
16-81 Saving Grid Edits Into Collection Using PL/SQL Code Target Type 93
16-82 Invoke API Calls Package Procedure to Save Temporary Photos 94
16-83 When Done Processing Temporary Images, Clear the Collection 95
16-84 Object Bucket Has a Namespace Used in REST APIs 97
16-85 Defining OCI Native Authentication Web Credential 98
16-86 Defining Object Buckets REST Data Source 99
16-87 Testing Object Buckets REST Data Source 100
16-88 Data Profile for Object Buckets REST Data Source 100
16-89 Defining Bucket Objects REST Data Source 102
16-90 Browsing and Selecting Bucket Files with a Content Row Region 106
16-91 Content Row Region Uses Bucket Objects REST Data Source 107
16-92 Configuring What Appears in Each Content Row Slot 108
16-93 Data Profile Columns for Bucket Objects REST Data Source 109
16-94 Use Application Definition Substitutions for Global Constants 111
16-95 Referencing Global Constants as Items 111
16-96 Configuring Avatar Image URL, Selection Behavior, and Pagination 112
16-97 Use a Private Object Bucket Unless It Stores Public Web Site Assets 116
16-98 Get File PAR Operation Has Base URL for Creating Pre-Authenticated Request 117
16-99 Strategy to Use PAR URLs to Serve Object Bucket Images Directly 119

Copyright © 2026, Oracle and/or its affiliates.

17-1	External Data in an Excel Spreadsheet	2
17-2	Creating a Data Load Definition for an Excel Spreadsheet	4
17-3	Mapping Source Columns from External Data to Target Columns	5
17-4	Reviewing Data Load Definition Results By Previewing Data	5
17-5	Excel Spreadsheet Has Employee Names in Mixed Case	6
17-6	Renaming ENAME to ORIG_ENAME Data Profile Column	7
17-7	Adding a SQL Expression Data Profile Column for ENAME	8
17-8	Creating a Data Load Page for an Existing Data Load Definition	9
17-9	Data Loading Page File Drop Zone	10
17-10	Preview of Excel Spreadsheet Data to Load	10
17-11	Success Message Confirms Number of Rows Loaded	11
17-12	Loaded Employees as HR Rep Susan in Employee Directory	11
17-13	File Upload Item in Data Loading Page	13
17-14	Dynamic Action Immediately Submits File Upload	14
17-15	After Submit Computation Populates Uploaded File Name	15
17-16	Validating Allowed File Types on Submit	16
17-17	Selecting Worksheet to Load When Multiple Ones Exist	17
17-18	Previewing the Data to Load with APEX_DATA_PARSER.PARSE	19
17-19	Ensuring Preview Sorts by Line Number	20
17-20	Configuring Native Data Load Page Process	21
17-21	Creating a Data Load Definition for an Example JSON File	23
17-22	Configuring File Upload Item to Allow Multiple Files	24
17-23	Calling Custom Data Load File Processing with Invoke API Page Process	25
17-24	Computing a Custom Success Message with Conditional Text	26
17-25	Selecting Four Different Kinds of Files at Once for Employee Data Loading	29
17-26	Success Message Confirms Eight Employees Loaded from Four Files	30
18-1	Woods HR App's Employee Excellent Page for HR Reps	2
18-2	Viewing Progress of Longer-Running Award Review	3
18-3	Execution Chains Let You Organize Page Processing Logic into an Outline	5
18-4	Child Processes Execute in Sequence Order in an Execution Chain	6
18-5	Nest Execution Chains to Form an Outline	7
18-6	Sending Email with a Native Page Process	8
18-7	Setting Execution Chain to Run in the Background	10
18-8	Returning Function Result into Hidden Page Item	11
18-9	Checking Whether Candidate Employee Qualified for Excellence Award	11
18-10	Emailing Employee Excellence Award Certificate to Recipient	13
18-11	JONES Receives Her Employee Excellence Award Certificate	14

Copyright © 2026, Oracle and/or its affiliates.

18-12	Sending Push Notification to Excellence Award Recipient	15
18-13	Serializing Background Process Job on Context Value	16
18-14	Background Process Error if Context Value Already Being Processed	17
18-15	Views Providing Progress of Background Processes	20
18-16	Displaying Progress Percentage as a Inline Bar Graph	22
18-17	Stretching Last Classic Report Column	22
18-18	Stretching Classic Report's Last Column Using CSS Class	23
18-19	Clamping Status Message Text to a Single Line	25
18-20	Ellipsis Displays when Text Length Exceeds Clamped Number of Lines	26
18-21	Enqueued Background Jobs Wait Their Turn to Execute	28
18-22	Inspecting Background Executions for the Current Session	30
18-23	Viewing Failed Background Execution Jobs	30
18-24	Identifying the Debug Trace Log for the Background Job	31
18-25	Jumping to First Error in the Debug Trace Timeline	31
18-26	Investigating Details of a Background Job in FAILED State	32
18-27	Drilling Down into an Active Session for a User	33
18-28	Reviewing Background Executions for Specific User Session	33
18-29	Viewing Session State for a Background Process Work Session	34
19-1	Martin Gets an Email That She's the Lucky Winner of a Raise	3
19-2	Saturday Employee Lottery Automation Source Query	5
19-3	Additional Code Execution Section Defines an Inline Initialization Procedure	7
19-4	Five Actions in the Saturday Employee Lottery Automation	8
19-5	Configuring Email Template and Placeholder Values for a Send E-Mail Action	10
19-6	Configuring the Automation Scheduler Expression with Interval Builder	12
19-7	Syntax for DBMS_SCHEDULER Calendar Expressions	13
19-8	Running an Automation from the List Page	14
19-9	Viewing the Automation Execution Log List	15
19-10	Inspecting Automation Execution Log Messages	15
20-1	Multiple Success Messages Get Concatenated	2
20-2	Messages Can Include HTML Markup	3
20-3	First Page Process Error Halts Processing and Displays Message	3
20-4	All Failed Validation Error Messages Display on Submit	4
20-5	Error Message Displays When Expression Evaluates to False	5
20-6	Validation Signals Failure by Returning the Error Text	7
20-7	Enabling Auto-Dismissal of Success Messages After Five Seconds	8
20-8	Dynamic Action Eagerly Calls CHECK_EMPNO Validation Function	9
20-9	Conditionally Displaying an Error Message if One Was Returned	10

Copyright © 2026, Oracle and/or its affiliates.

20-10	Validation Performs Same Check on Page Submit	11
20-11	Eagerly Displaying a Page-Level Error Message Before User Submits	11
20-12	Aggregate SQL Checks Can Veto the Transaction Using ADD_ERROR	14
20-13	Showing Eager Field-Level Error Messages Before Submitting the Page	15
20-14	Clearing Field-Level Errors Before Checking Changed Employee ID	16
20-15	Validating the Employee ID on the Server	17
20-16	Conditionally Showing Error Message Returned by Server-side Check	18
20-17	Reusing the Same Recipe to Eagerly Validate Department ID	18
20-18	By Default, Users See Constraint Violation Error Name	20
20-19	By Default, Users See Internal Errors	21
20-20	Configuring a Page's Error Handling Function	22
20-21	Error Handling Function Shows Custom Error for Constraint Violation	27
20-22	Error Handling Function Shows Simplified Database Exception Message	28
20-23	Error Handling Functions Show Custom Message for Uncommon Internal Error	29
20-24	Error Handling Function Logged Uncommon Internal Error for Review	29
20-25	Viewing Logged Error Details for Uncommon Internal Error	30
20-26	Sending Email with a Native Page Process Without a Template	31
20-27	Template Has Static ID for API Use and Can Use Template Directives	32
20-28	Defining an Email Template with Placeholders	33
20-29	Configuring Email Template and Placeholder Values for Send-Email Action	34
20-30	Enabling Push Notifications in Your Application	36
20-31	Generating Key Pair Credential for Signing Push Notifications	37
20-32	Adding Push Notification Settings Pages to Your App	38
20-33	Lucy Uses App's Settings to Opt-In to Notifications	39
20-34	Users Send Push Notification with Random Laughable Line	40
20-35	Delivering a Push Notification with Native Page Process	40
20-36	Lucy Receives the Notification and Laughs	41
21-1	Configuring App to Use Instance Settings for Remote Printing	2
21-2	Glimpse of the Oracle Document Generator Instance Settings	2
21-3	Using Microsoft Excel to Create Report Layout	5
21-4	Defining Report Layout for Excel Document Template	5
21-5	Configuring Report Query Data for Tenure Report	6
21-6	Configuring Report Source Data Loop to Match Layout Tags	7
21-7	Using CSS Class to Vertically Center Button with Page Items in the Same Row	10
21-8	Setting Application Item to Selected Department Number	10
21-9	Printing Tenure Report Using Report Query	11
21-10	HR Rep Susan Prints Tenure Report for Research Department	12

Copyright © 2026, Oracle and/or its affiliates.

21-11	Tenure Report PDF Result for Research Department	13
21-12	Microsoft Word Layout Document for Employee Directory	14
21-13	Specifying Layout Data Loop Name to Match Layout Tags	15
21-14	Configuring Interactive Report PDF Printing to Use Custom Layout	16
21-15	Employee Directory Interactive Report PDF Print Uses Custom Layout	17
21-16	PDF Download from Employee Directory Interactive Report	18
22-1	Employee Directory Page with an Interactive Report in Primary Language	2
22-2	Employee Details Form Page for HR Representatives Only in Primary Language	2
22-3	Enabling Application Translation in Globalization Settings	3
22-4	Adding the First Translation Language	4
22-5	Primary Language and First Translated Language Created and Synchronized	4
22-6	Page Title Changed to Reference Translatable Text Message by Name	5
22-7	All Translatable Text Undergoes Same Transformation	5
22-8	Identical Text References the Same Translatable Text Message	6
22-9	Exporting all Text Messages for a Language to an XLIFF File	6
22-10	Importing Translated Strings from XLIFF File	8
22-11	Configuring How Your App Determines the Current User's Language	9
22-12	User's Italian Browser Language Preference Drives App Language	10
22-13	Navigation Menu and Employee Directory Use Italian Translations	10
22-14	Employee Details Modal Drawer Uses Consistent Italian Translations	11
22-15	Creating New Text Message in Primary Language	12
22-16	Enabling Text Messages Picker	13
22-17	Choosing Message in Text Message Picker Opened with the Globe Button	13
22-18	Picked Text Message Included with Proper Substitution Syntax	14
22-19	Adding New Translatable Text Inline on a Page	15
22-20	Synchronizing Text Messages Creates All New Messages Required	16
22-21	Accessing the Text Messages Grid Edit for Bulk Changes	16
22-22	Filtered Grid Edit is Handy for Bulk Translation	17
22-23	Manually Entered and Synchronized Messages Show for Italian Users	17
22-24	Converting Application Text to Text Messages	18
23-1	Using Environment Banners for DEV, TEST, and PROD	5
24-1	Creating a Small Employees Application Using the Create App Wizard	3
24-2	Choosing APEXlang Format for Your Application Export in App Builder	4
24-3	Importing an Application in APEXlang Format in App Builder	6
24-4	Previewing the Application to Import and Choosing an Application ID	6
24-5	Enabling Workspace Parsing Schema for ORDS REST Services if Needed	7
24-6	Modified Employees Page in Page Designer	11

Copyright © 2026, Oracle and/or its affiliates.

24-7	Exporting Employees App (104) in APEXlang from SQL Developer	15
24-8	Previewing SQLcl Command to Export App 104 in APEXlang Format to a Parent Directory	16
24-9	Using APEXlang Code Completion While Editing an .apx File	16
24-10	VS Code Problems Panel Shows APEXlang Validation Errors Proactively	17
24-11	Clicking the “Play” Button to Import Your APEXlang Application to App Builder	18
24-12	Confirmation of a Successful APEXlang Application Import in VS Code	18
24-13	Verifying the Results of Importing the Latest Application Version from VS Code	19
24-14	Accessing an APEXlang View of the Current Page in Page Designer	23
24-15	Reviewing the APEXlang Representation of the Current Page in Page Designer	23
24-16	Reviewing APEXlang Diffs When Comparing Working Copy with Main App	24
25-1	Activities Palette Shows Available Types You Can Drag onto Diagram	3
25-2	Dragging an Activity from the Palette and Dropping onto the Canvas	5
25-3	Naming a New Activity in the Property Editor	6
25-4	Reattach a Connection by Dragging its Arrowhead or Starting Dot	7
25-5	Connection Anchor Dots Determine Line Slope	8
25-6	Reposition Anchor Dots in Source or Target Activities to Change Slope	9
25-7	Drag Handles on Horizontal and Vertical Segments of Selected Connection	10
25-8	Adding a New Connection Between Activities	11
25-9	Select a Line Then Single Click to Add an Elbow Dot	12
25-10	Repositioning an Elbow Dot with Drag and Drop	13
25-11	Add Additional Elbow Dots and Reposition Them to Create Any Routing	14
25-12	Adding a Next Activity From a Current One	15
25-13	Next Activity and Connection Appear on the Diagram	16
25-14	Configuring Properties on Connection Originating from a Switch	17
25-15	Using Content Menu to Add a New Activity Between Existing Ones	18
25-16	Dragging and Dropping a New Activity Between Existing Ones	19
25-17	Selecting and Moving a Group of Activities	20
25-18	Selecting a Set of Activities with a Shift+Drag Rectangle	21
25-19	Connection Arrow Tips Obscured by Activity Shapes	22
25-20	Use "Bring to Front" to Stack Connection Arrows On Top of Activity Shapes	22
25-21	Clicking Error Indicator to Review Pending Issues	23
25-22	Workflow Delivers Tasks to Potential Owners' Task List "Inbox"	28
25-23	Choosing an Action Task or Approval Task at Create Time	29
25-24	Approval Tasks Show Quick Action Buttons in the Task List	30
25-25	Action Tasks Have a (Complete) Button in Task Details Page	30
25-26	Configuring Task Participants	33
25-27	Task Instance Lifecycle: States and Actions that Change State	38

Copyright © 2026, Oracle and/or its affiliates.

25-28 Updatable Action Task Definition Parameters Model User-Supplied Data 42
25-29 Disable Visible Flag to Hide a Parameter from the Task Details Page 43
25-30 Configuring Task Potential Owners Using a Parameter Value 43
25-31 Configuring a Human Task - Create Activity to Use a Task Definition 44
25-32 Passing the Value of a Workflow Parameter to a Task Definition Parameter 45
25-33 Retrieving Task Parameter Values into Workflow Variables 46
25-34 Lucy Initiates a Workflow and Sees It In Her "Requested Tasks" Page 47
25-35 Creating a Task Details Page for a Task 48
25-36 Task Details Page URL Appears After You Create or Assign a Page 49
25-37 Leo Sees Task Assigned to Him in His "My Tasks" Page 50
25-38 Leo Asks for More Information to Clarify the Task 51
25-39 Leo Enters His More Information Question 52
25-40 Lucy Sees the Task Shows Up in Her "My Tasks" Page 52
25-41 Lucy Clarifies and Submits the Additional Information 53
25-42 Leo Enters Parameter Values 54
25-43 Leo Completes the Task 55
25-44 Lucy Reviews Completed Workflow in Her "Initiated Workflows" Page 56
25-45 Lucy Sees Request Completed in Her "Requested Tasks" Page 57
25-46 Lucy Reviews Task Details to See Numbers Leo Supplied 57
25-47 Commenting Out Regions and Adding Page Items to Enter Additional Data 59
25-48 Automatic Field Types and Required Fields Validation 60
25-49 Making Items Read Only Unless User is Assigned Owner 61
25-50 Loading Task Parameter Values Into Page Items Before Page Render 62
25-51 Configuring a Custom Validator for Page Items in Task Details Page 63
25-52 Validation Error if User Enters Same Number Twice 63
25-53 Saving Task Parameter Values 64
25-54 Participants for Purchase Office Supplies Task Definition 69
25-55 Parameters for Purchase Office Supplies Task Definition 70
25-56 Actions Source Query Retrieves Computed Title and Potential Owner Emails 71
25-57 Email Template Used to Send New Task Notification 72
25-58 Email Body HTML in the Template for New Task Notification 73
25-59 Create Action on Purchase Office Supplies Task Definition 74
25-60 Send E-Mail Action on Task Create Event 74
25-61 Enabling Deep Linking so User's Click Can Directly Link to Page 79
25-62 Send E-Mail Activity Includes Page Link from Additional Data Query 81
25-63 Configuring the Value of APP_EMAIL Substitution 82
25-64 Send Push Notification Uses Link from Additional Data Query 83

Copyright © 2026, Oracle and/or its affiliates.

25-65 Wait Activity user_provide_two_values Has No Timeout Type 83
25-66 Receiving the Notification Email and Clicking the Link 84
25-67 Page Collecting Two Values from the Notified User 85
25-68 Page Collecting Values from the User 86
25-69 Page Process Calls CONTINUE_ACTIVITY, Passing Data in Parameter Map 87
25-70 Inspecting the Details of the Completed Workflow 89
25-71 Configuring Workflow Participants 91
25-72 Title Appears in Console Page Listing Workflow Instances 93
25-73 Workflow Detail Page Displays Status of Selected Instance 94
25-74 Woods Clinic New Patient Procedure Inquiry Form 95
25-75 After Registering a Procedure Inquiry, a New Patient Awaits a Response 96
25-76 Workflow Version Starts in Development Mode Until You Activate It 97
25-77 Getting Started on a Patient Onboarding Workflow 98
25-78 Referencing Patient Columns by Name in the Workflow Title 99
25-79 Woods Clinic Patient Onboarding Workflow in App Builder 100
25-80 Geocoding New Patient Address in Registration Page 101
25-81 Invoking a PL/SQL Package API to Evaluate Service Range 102
25-82 Configuring Patient ID as Value of P_PATIENT_ID PL/SQL Parameter 103
25-83 Configuring a Switch Activity to Decide Based on a Workflow Variable 104
25-84 Assigning Outcome Condition to Connection Originating from a Switch 105
25-85 Configuring an Oracle Machine Learning AutoML Experiment 106
25-86 Inspecting the Algorithm "Leader Board" Results of the Experiment 107
25-87 Winning Neural Network Model for Woods Clinic Patient Onboarding 107
25-88 Two Combinations of Invoke API and Switch Using Machine Learning 108
25-89 Linking to a Workflow Diagram Page to Visualize Progress 110
25-90 Visualizing Onboarding Progress of New Patient Danny Deng 111
25-91 Task Definition Subject Uses Column Values from the Actions Source Table 112
25-92 Using Human Task Create Activity to Wait for Approval 113
25-93 Unified Task List Showing Patient Onboarding Approval Tasks for DAVID 114
25-94 Anya Ableton's Procedure Request Was Rejected for Being Too Risky 115
25-95 Configuring Send Email Activity and Email Template Placeholders 116
25-96 Native Activities for Alerting Users with Email or Push Notifications 117
25-97 Rejection Email Template with Two Placeholders 118
25-98 Referencing Additional Data Columns in an Activity 119
25-99 Anya Ableton Rejection Email from Woods Clinic Patient Onboarding Workflow 120
25-100 Cristina Cordero's Request Was Auto-Approved and Is Now Waiting 121
25-101 Provision Account Invoke API Activity Creates User Account 122

Copyright © 2026, Oracle and/or its affiliates.

25-102 Cristina Cordero Receives Her New Account Credentials 126
25-103 Cristina Cordero's Request Was Auto-Approved and Is Now Waiting 127
25-104 Configuring an Activity to Wait Indefinitely 128
25-105 Patient Portal for Cristina Cordero on First Login 129
25-106 Cristina Cordero Completes Registration - Date of Birth 129
25-107 Cristina Cordero Completes Registration - Proof of Insurance 130
25-108 After Completing Registration, New Patients Portal Dashboard Options Change 132
25-109 Verifying Cristina Cordero's Patient Onboarding is Completed 133
25-110 Patient Procedures Table in the Woods Clinic Data Model 134
25-111 Patient's Medical Procedure Statuses Define its Lifecycle 135
25-112 Kanban Board Shows Users How Procedures Progress Through Their Lifecycle 136
25-113 Procedure Lifecycle Workflow Coordinates Application Logic 137
25-114 On David's Last Login, He Ticked the Remember Me Checkbox 140
25-115 David Launches Woods Clinic APEX App from His Dock 140
25-116 Home Page for Woods Clinic App Launched as a Native App 141
25-117 Navigating to Scheduling Page from a Different Screen 142
25-118 David Schedules Danny Deng's Procedure 143
25-119 Scheduled Medical Procedures Appear on the Calendar 143
25-120 Office Staff See Danny Deng's Procedure Scheduled on Kanban Board 144
25-121 Checking a Workflow from Pending Procedures List 145
25-122 Confirming Danny Deng Was Notified of Appointment 146
25-123 Danny Deng Gets Email with His Appointment Information 147
25-124 Discharging the Patient and Completing the Procedure 148
25-125 Entering Patient Discharge Instructions 149
25-126 Preparing Discharge Letter Using Generative AI 150
25-127 Invoice Prepared After Procedure Completed 151
25-128 Report Query Provides JSON Data for Pixel-Perfect PDF Report Layout 152
25-129 Report Source Configures SQL Query and Data Loop Name 153
25-130 Studying Report Query JSON and Testing Query in VS Code 154
25-131 Report Layout Provides Word Document Template for Invoice 155
25-132 Microsoft Word Document Defines Pixel-Perfect PDF Layout 156
25-133 Opening the Prepared Invoice Details 158
25-134 Reviewing Pixel-Perfect PDF Invoice 159
25-135 Entering Insurance Reimbursement Amount 160
25-136 Regenerating Final Invoice After Entering Insurance Payment 161
25-137 Reviewing Dashboard While Invoice Regenerates in Background 162
25-138 Execution Chain Simplifies Regenerating Invoice in the Background 163

Copyright © 2026, Oracle and/or its affiliates.

25-139 Reviewing Regenerated Invoice Before Sending to Patient 164
25-140 David Marks Danny Deng's Invoice as Ready to Send to Patient 165
25-141 Danny Deng's Procedure Lifecycle Workflow Shows Invoice Emailed 166
25-142 Progress of Danny Deng's Procedure on Kanban Board 167
25-143 Danny Deng Receives Invoice for His Medical Procedure 168
25-144 David Marks Danny Deng's Medical Procedure Invoice as Paid 169
25-145 Kanban No Longer Shows Danny Deng's Procedure After Invoice is Paid 170
25-146 David Confirms Danny Deng's Medical Procedure is Paid 171
25-147 David Confirms Danny Deng's Procedure Lifecycle is Done 171
25-148 Strategy for Using Workflow to Orchestrate Patient Procedure Lifecycle 173
25-149 Lifecycle Events Handler Automation – Settings 176
25-150 Lifecycle Events Handler Automation – Source 177
25-151 Lifecycle Events Handler Automation – Actions 177
25-152 Lifecycle Events Handler Automation – Action Execution 178
25-153 Development Toolbar Appears When Running in Development Mode 181
25-154 Your Availability Setting Can Disable the Developer Toolbar 182
25-155 Enabling Info-Level Debug Tracing with the Developer Toolbar 184
25-156 Enabling Debug Tracing for Background Workflow Execution 184
25-157 Disabling Page Filter to View Recent Workflow Debug Requests 185
25-158 Examining Debug Exception Call Stack in Debug Viewer 186
25-159 APEX Dictionary Views Related to Workflow Instances 188
25-160 APEX Dictionary Views Related to Task Instances 189
25-161 Activating an In Development Workflow 190
25-162 Workflow Versions Accumulate Over Time 191

Copyright © 2026, Oracle and/or its affiliates.

List of Tables

4-1	Available Template Directives	3
7-1	Attribute Scopes and Availability in Templates	13
8-1	Collection Columns	33
8-2	Wizard-Created Form Page Button Settings	51
13-1	Supported Web Credential Types	6
13-2	Most Common Parameter Types	21
13-3	Data Profile Columns	44
13-4	REST Data Source Plug-in Contract	47
14-1	HTTP Methods and REST Operations	5
14-2	JSON Types and Their Representations	7
16-1	Three Approaches to Defaulting a Page Item Value	52
16-2	APEX_APPLICATION_TEMP_FILES Table Structure	76
16-3	EBA_DEMO_WOODSHR_FILE Program Units	83
25-1	Report Context Determines Which Task Instances Display	39
25-2	Important GET_TASKS Result Columns	40
25-3	Pre-Defined Task Events	67
25-4	Pre-Defined Task Substitutions	67
25-5	Report Context Determines Which Workflow Instances Display	92
25-6	Switch Decision Types	105
25-7	APEX Dictionary Views for Workflows	187
25-8	APEX Dictionary Views for Workflows	188

Copyright © 2026, Oracle and/or its affiliates.

Preface

Oracle APEX Developer's Companion explains when, why, and how to use Oracle APEX's broad range of features to deliver business value faster with less work. It covers all aspects of enterprise app building including data modeling, UX design, search, visualization, integration, automation, security, reporting, translation, and application lifecycle management.

- Audience
- Dedication
- Related Documents
- Conventions
- About the Author

Audience

Oracle APEX Developer's Companion is intended for all application developers of any skill level who are building database-centric web applications using Oracle APEX.

Dedication

This book is dedicated to the memory of Joel R Kallman, Allan Sitterson, and Robert Muench. All left us too soon and are greatly missed.

Related Documents

For more information, see these Oracle resources:

- Oracle APEX Documentation Library

Conventions

The following text conventions are used in this document:

Convention	Meaning
boldface	Boldface type indicates graphical user interface elements associated with an action, or terms defined in text or the glossary.
italic	Italic type indicates book titles, emphasis, or placeholder variables for which you supply particular values.
`monospace`	Monospace type indicates commands within a paragraph, URLs, code in examples, text that appears on the screen, or text that you enter.

Copyright © 2026, Oracle and/or its affiliates.

About the Author

Steve Muench joined Oracle in 1990, supporting SQL*Forms 2.3 and 3.0 in the days when Oracle application development meant green screens and 80-column terminals. In 1992, he joined the Oracle Forms development team as product manager, helping shape its first graphical UI versions. In 1993, he co-authored the *Oracle: Forms Developer's Companion* for Maverick Publishing.

From 1994 to 1997, Steve contributed to Project Sedona, a radically object-oriented research effort that evolved into Oracle JDeveloper and Application Development Framework. He initially worked as a product manager and wrote the manual *Oracle ADF Developer's Guide for Forms 4GL Developers*. He progressed to developer and architect roles on the ADF Business Components team, working often with internal Fusion Applications developers to understand new requirements and deliver declarative features addressing them.

Along the way, on nights and weekends, he wrote *Building Oracle XML Applications* for O'Reilly & Associates (2000) and evangelized the possibilities of combining SQL, XML, and XSLT before the database included those capabilities. Oracle's latest JSON support stands on the shoulders of the native XML features that followed.

Since 2021, he has been a senior architect on the Oracle APEX development team. Throughout his career, he has helped many internal teams building ADF and APEX apps make the smartest use of the platforms. The *Oracle APEX Developer's Companion* is the natural culmination of three and a half decades spent supporting, building, using, and writing about Oracle application development tools, interacting every day with the developers who work with them.

Copyright © 2026, Oracle and/or its affiliates.

1 Introduction

Oracle APEX is a complete low-code platform for rapidly building secure, scalable enterprise apps of any size and complexity.

Using APEX **App Builder** in a browser, you define pages and features by configuring powerful built-in components and data sources. You add code only for custom business logic that *only you* could write. With Oracle APEX *you* deliver immediate business value by leveraging a full range of critical capabilities in a single development and runtime environment. This guide helps you understand key areas of APEX functionality and how to use those features in your apps.

- Browser-Based Enterprise App Development
  Build, run, and manage enterprise applications you can trust entirely in a browser.
- Model-Driven, Direct Data Access
  Build apps from declarative metadata with direct access to Oracle Database.
- Extensible, Powerful Components
  Use high-function, data-savvy components to build rich web applications.
- Responsive, Polished User Interface
  Create responsive, polished user interfaces that adapt to desktop, mobile, and installed app use.
- Complete Application Platform
  Build complete apps with workflow, AI, reporting, files, REST APIs, and integration.
- Open, Agent-Ready APEXlang
  Use APEXlang to generate and edit APEX apps that tools and coding agents can validate.
- Spec-Driven App Scaffolding
  Blueprints turn a functional specification and data model description into a working, scaffolded app.
- Pricing and Deployment Options
  Oracle APEX is *cost-effective* and offers flexible deployment options.
- Active, Helpful Community
  Connect with a worldwide APEX community for support, learning, and reusable plug-ins.
- App Development as a Service
  Offer self-service app development with multi-tenant APEX workspaces and isolated data.
- Learning Resources and Free Workspaces
  Find APEX learning resources and request a free workspace for hands-on practice.

1.1 Browser-Based Enterprise App Development

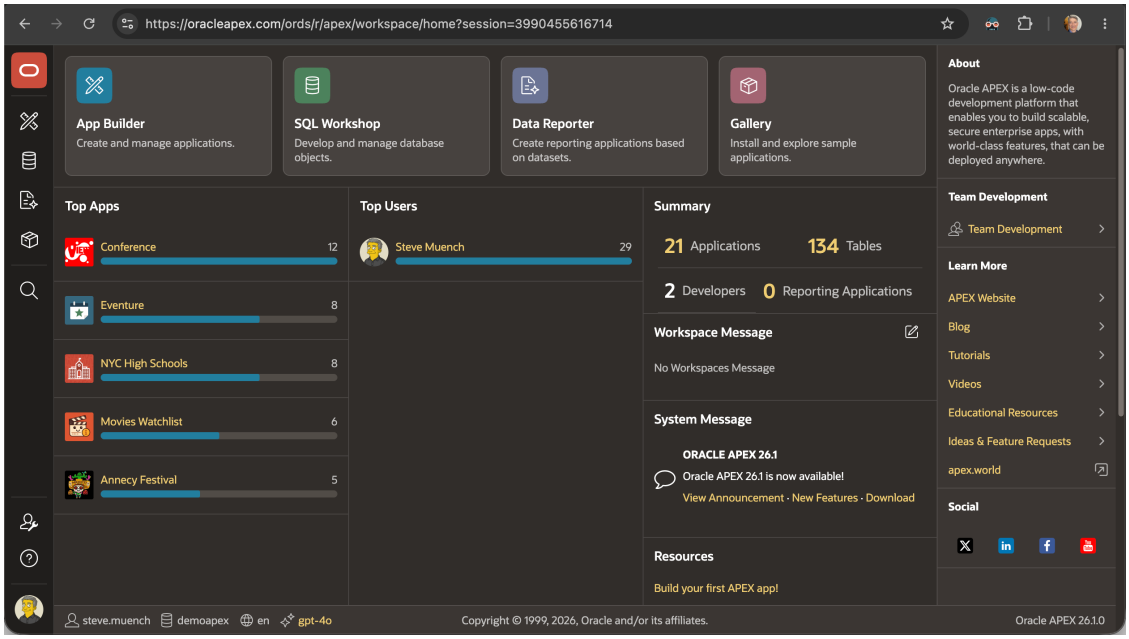
Build, run, and manage enterprise applications you can trust entirely in a browser.

A web browser is all you need to develop and run APEX apps. This includes all aspects of the enterprise application lifecycle from designing a data model and loading data, to implementing business logic and configuring user interfaces, as well as issue tracking, deployment, monitoring, and access control.

Copyright © 2026, Oracle and/or its affiliates.

Teammates of different skill levels collaborate in a shared workspace to create, deliver, maintain, and enhance applications. Each developer uses a **working copy** of the main app to isolate the individual feature or fix. When ready, in the browser they merge their changes back into the main app for testing and deployment. The figure below shows the landing page for App Builder for a `DEMOAPEX` workspace.

Figure 1-1 App Builder Landing Page for a DEMOAPEX Workspace on apex.oracle.com



1.2 Model-Driven, Direct Data Access

Build apps from declarative metadata with direct access to Oracle Database.

Oracle APEX is a model-driven execution engine, dynamically generating application pages at runtime based on metadata stored in the database. You configure your pages and features using declarative settings, adding custom business logic only where needed using SQL, PL/SQL, or JavaScript.

Development is rapid and iterative: you see changes immediately by simply refreshing the browser page. APEX renders pages and handles end-user interaction based on your app definition, so your solutions automatically benefit from platform enhancements and technology upgrades.

Co-locating the application engine with the data makes Oracle APEX fast, reliable, and secure. It leverages the proven and robust capabilities of Oracle's data platform including transactions, concurrency, session-state management, high availability, disaster recovery, analytics, SQL, PL/SQL, and more. By directly accessing data with no network overhead, your APEX apps get fast response times, reduced CPU load, and simplified management and debugging. The Oracle database securely manages any kind of data in a consistent way, so your APEX apps get maximum data-handling speed and flexibility using a single set of low-code skills. APEX's built-in authentication and authorization guarantee only intended users see the application pages and data you allow them to work with.

Copyright © 2026, Oracle and/or its affiliates.

1.3 Extensible, Powerful Components

Use high-function, data-savvy components to build rich web applications.

The Oracle APEX platform's many *native* components include regions that simplify searching for, creating, editing, visualizing, analyzing, mapping, and processing the information your app requires.

You add these components to your pages and configure what data they should work with. The components work the same way whether the data comes from the local database or from a remote data source when needed. The vibrant worldwide APEX developer community has built hundreds of additional components you can download from the `apex.world` website.

Developers who *may* be familiar with the basics of Hypertext Markup Language (HTML) and Cascading Style Sheets (CSS) can create reusable template components for the rest of the team to use just like built-in ones. Using template directives, your components produce conditional output based on any number of custom attributes that your teammates configure declaratively in Page Designer. This can be a **no code** experience, or more sophisticated template components can make use of JavaScript as needed.

1.4 Responsive, Polished User Interface

Create responsive, polished user interfaces that adapt to desktop, mobile, and installed app use.

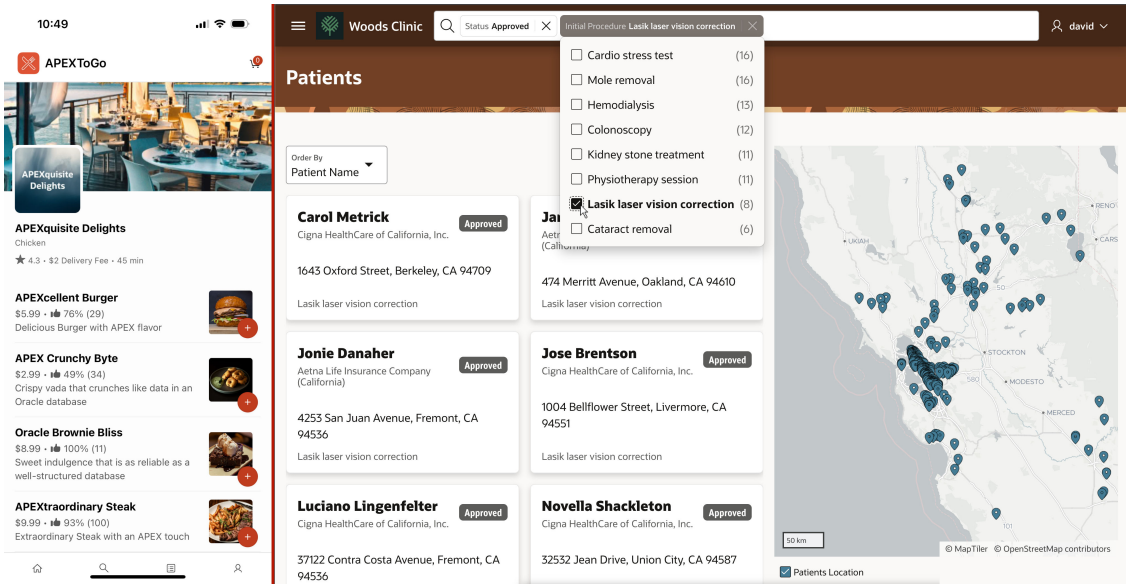
In APEX Page Designer, developers use an intuitive grid layout to position and size components, relative to others on the page. Their pages automatically adapt to any screen size on desktop and mobile devices, and the **Universal Theme** design system ensures pages are consistent and professional with no expertise required.

Choose a built-in theme style, and customize it interactively in the browser to override colors, fonts, icons, or other aspects to match a particular corporate look and feel.

With a single click, you can let users install your apps to launch them like native ones from their device's home screen, dock, or taskbar. Your apps can leverage native device facilities like location, tailored data entry keyboards, and multiple cameras. By following the best practices illustrated in the *APEX To Go* sample in the gallery, your apps look and work just like native mobile apps with the advantage that they are always up to date.

Copyright © 2026, Oracle and/or its affiliates.

Figure 1-2 Examples of APEX Apps on Mobile and Desktop



1.5 Complete Application Platform

Build complete apps with workflow, AI, reporting, files, REST APIs, and integration.

Your APEX apps can easily go well beyond data management. You can automate business processes and human approvals, apply artificial intelligence, and predict outcomes using machine learning. Producing pixel-perfect reports using Microsoft Word or Excel templates is easy.

Extending Oracle Fusion Applications, working with remote REST data, loading external data files, and uploading images and files are simple. The platform also makes it easy to automate repetitive tasks, expose REST APIs for integration, offload work to the background, and notify end-users of interesting events.

1.6 Open, Agent-Ready APEXlang

Use APEXlang to generate and edit APEX apps that tools and coding agents can validate.

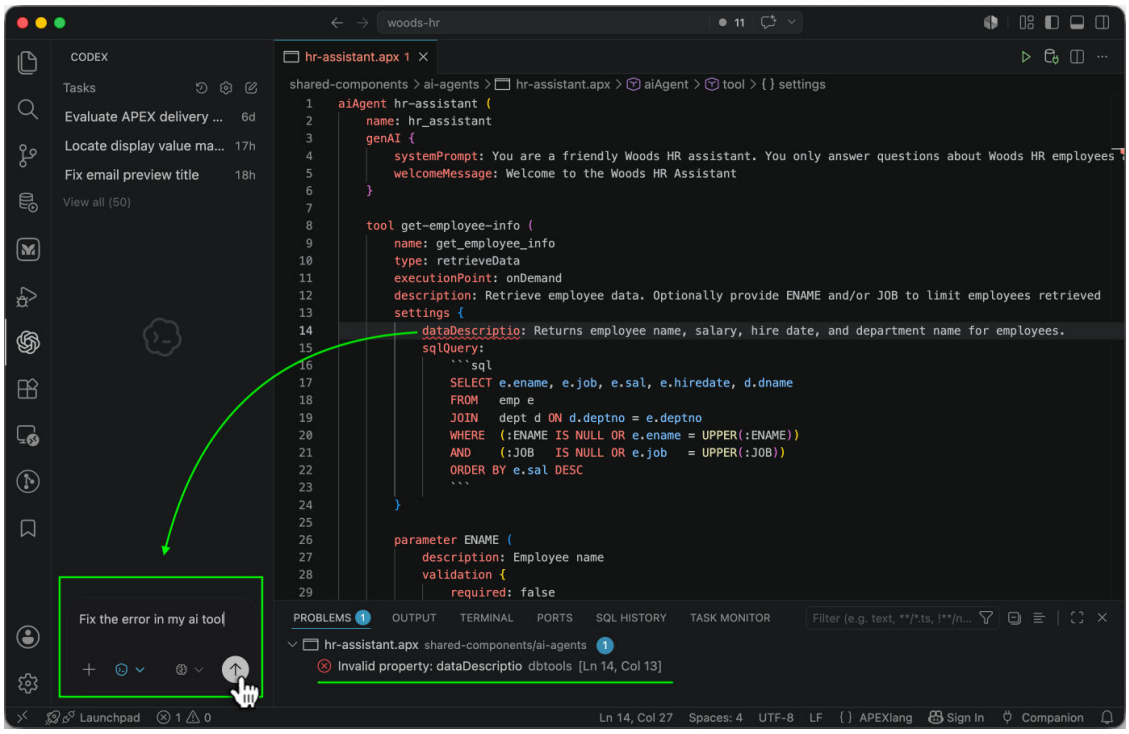
APEX applications are metadata-driven. APEXlang represents that metadata in an open, readable text format that developers, external tools, and coding agents can create and edit. SQLcl can validate APEXlang files without a database connection, giving tools and coding agents the feedback they need to correct errors before import. Open source APEXlang skills help agents create and evolve APEX apps, while the SQL Developer extension for Visual Studio code offers code completion, proactive error highlighting, and built-in import and export support. Developers can work in App Builder, in APEXlang with their favorite external tools, or both. All paths use the same open application metadata.

Because APEXlang targets the APEX application model, generated definitions run on the APEX platform rather than as standalone application code. The platform provides runtime services for sessions, authentication, authorization, transactions, data access, page rendering, approvals, and workflow. Your app can also define AI Agents with Tools that give an LLM a contract for looking up permitted data or performing allowed actions. The generated artifacts

Copyright © 2026, Oracle and/or its affiliates.

are human-readable metadata with a documented shape, so developers can review, maintain, and enhance them even when a coding agent helped author them.

Figure 1-3 Using AI Coding Agent in VS Code to Fix Problem Highlighted in APEXlang File



1.7 Spec-Driven App Scaffolding

Blueprints turn a functional specification and data model description into a working, scaffolded app.

APEX blueprints help teams refine specifications quickly and avoid costly surprises later. Start with a draft functional spec and a data model description, then use an AI coding agent with Oracle's detailed blueprint generation prompt to produce a readable plan for the app. This structured Markdown document uses a simple, predictable format to describe proposed pages, navigation, reports, forms, charts, filters, actions, and initial page behavior using curated patterns. Stakeholders can review it while requirements are still easy to change. More important, they can try a working, scaffolded app that implements the current specification.

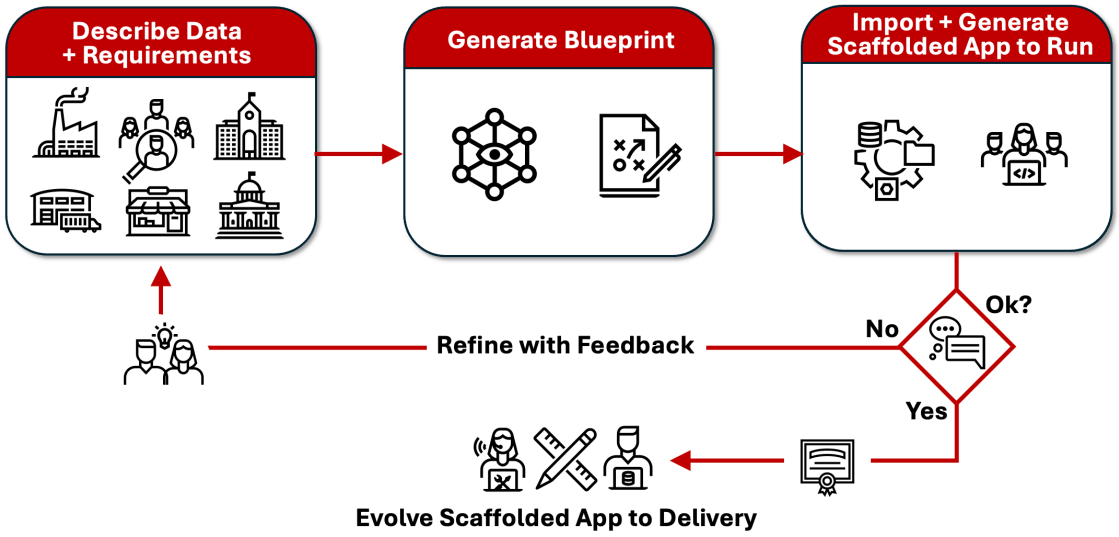
APEX imports the blueprint and deterministically generates a scaffolded app from APEXlang templates that represent tested building blocks. Key users can see real pages, follow real navigation, and react early. Gaps become visible. Vague requirements become specific. You can revise the specification, regenerate the blueprint, and produce another scaffolded app until stakeholders and builders agree on what the app needs to do.

This creates a fast prototyping loop. The scaffold is not the finished application. It helps the team refine the specification, expose missing requirements, and agree on the application's shape. After stakeholders sign off, the final scaffold becomes the foundation for the app you deliver. You and the development team then refine the user experience, add business logic, and implement integrations using any combination of App Builder and AI coding assistants with

Copyright © 2026, Oracle and/or its affiliates.

APEXlang skills. You finish the app with the detail and judgment real systems require, and stakeholders get the application they helped shape.

Figure 1-4 Generating Scaffolded App with AI Produced Blueprints to Converge Requirements



1.8 Pricing and Deployment Options

Oracle APEX is *cost-effective* and offers flexible deployment options.

Oracle APEX is free to use with any Oracle AI Database license, including the Oracle Database 26ai Free. It is also available as a fully-managed cloud service with usage-based pricing. Oracle APEX has no limits on applications, developers, or end-users, and the apps you build can run on-premises, in the Oracle Cloud, or in third-party clouds with no changes.

1.9 Active, Helpful Community

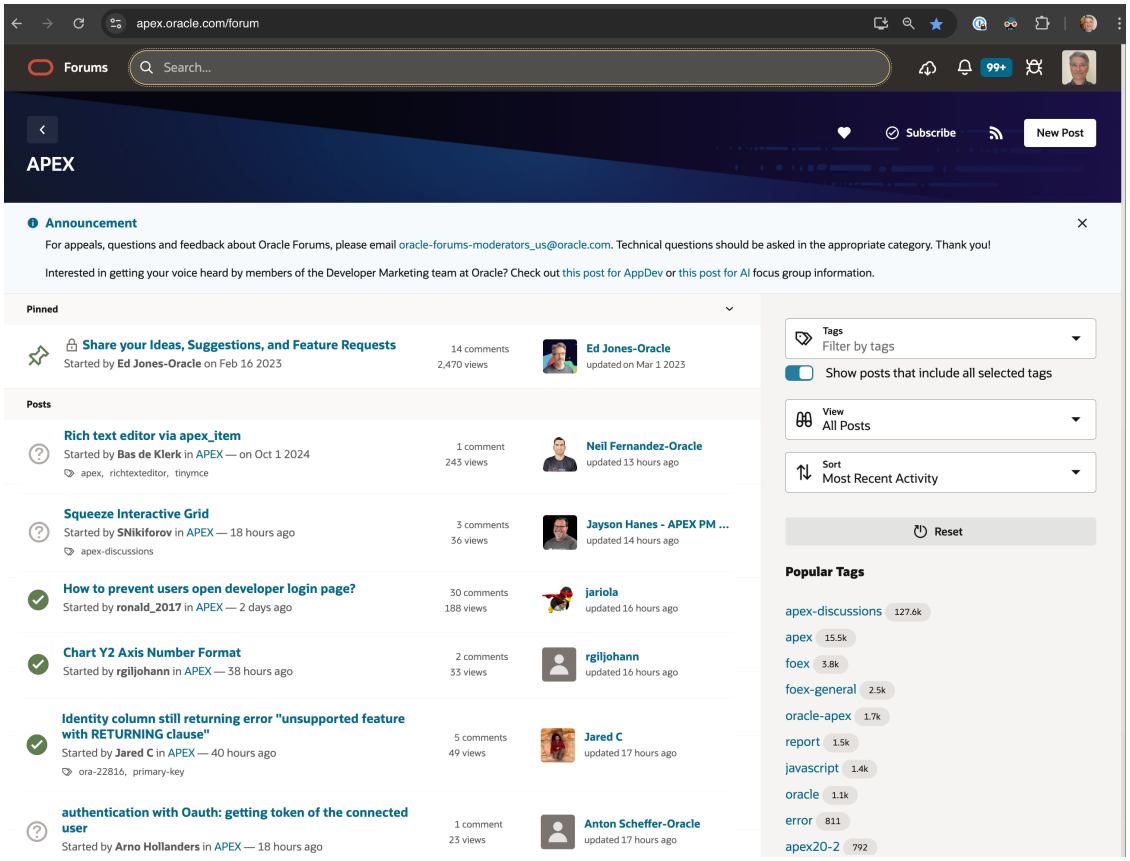
Connect with a worldwide APEX community for support, learning, and reusable plug-ins.

The worldwide Oracle APEX developer community is a vibrant, supportive group of hundreds of thousands of developers. They organize many annual user group conferences and developer meet-ups around the world to share APEX knowledge.

Community members help each other every day on the APEX Forum at oracleapex.com/forum. The site is an Internet-scale Oracle APEX app, that easily handles over a million requests a day for all developer questions about all Oracle products. The figure below shows a page of discussion threads where both Oracle and community APEX experts answer questions from the community.

Copyright © 2026, Oracle and/or its affiliates.

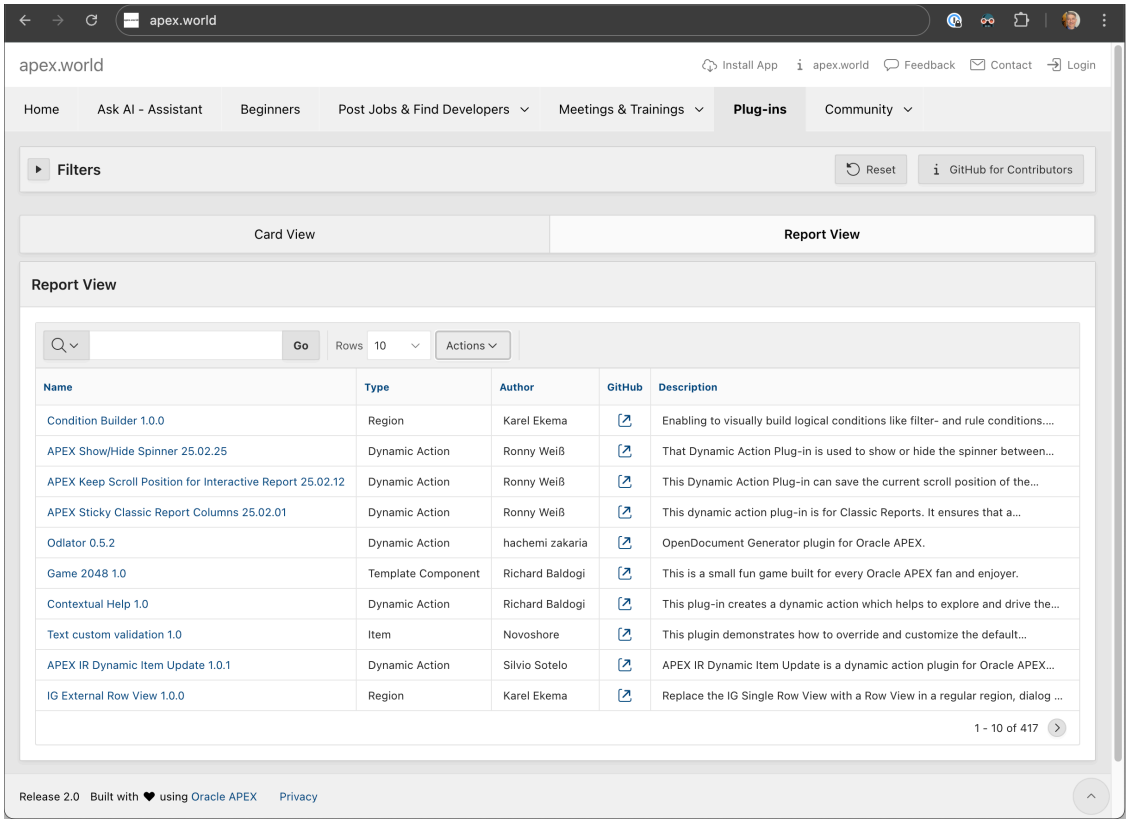
Figure 1-5 APEX Community Discussion Forum, Built Using APEX



The APEX community also maintains the apex.world website shown below, where you can find the latest APEX-related product news, blog articles, meetings, trainings, social media posts, and over 400 reusable components called **plug-ins**.

Copyright © 2026, Oracle and/or its affiliates.

Figure 1-6 APEX World Community Site for News, Component Exchange, and More



1.10 App Development as a Service

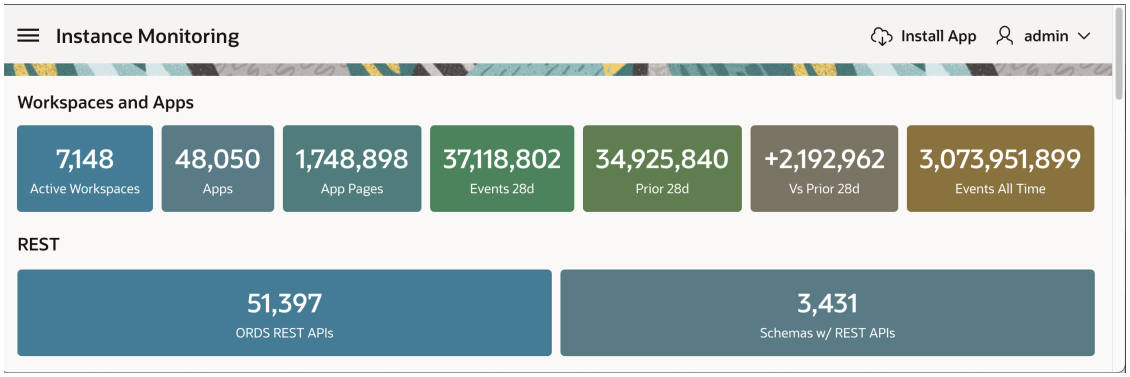
Offer self-service app development with multi-tenant APEX workspaces and isolated data.

Using Oracle APEX, any company can offer an **App Dev as a Service** option, empowering staff to rapidly turn ideas into apps. Oracle APEX is a multi-tenant low-code platform organized into workspaces. Each developer or team can use their own workspace on the same APEX instance and separate workspace schemas provide data isolation. APEX's self-service provisioning mode lets developers request a new workspace and get a link to log in to their workspace in a confirmation email. New requests can be approved manually or automatically.

Oracle extends this capability to all internal stakeholders. Tens of thousands of employees of every skill level, job description, line of business, and country use APEX *internally* to create apps that help run Oracle's day-to-day operations. The figure below shows statistics for one of Oracle's largest internal APEX instances supporting over 7000 active workspaces and thousands of production applications. Any Oracle employee can request a workspace and start building applications in minutes.

Copyright © 2026, Oracle and/or its affiliates.

Figure 1-7 Statistics for Large Internal APEX Instance



1.11 Learning Resources and Free Workspaces

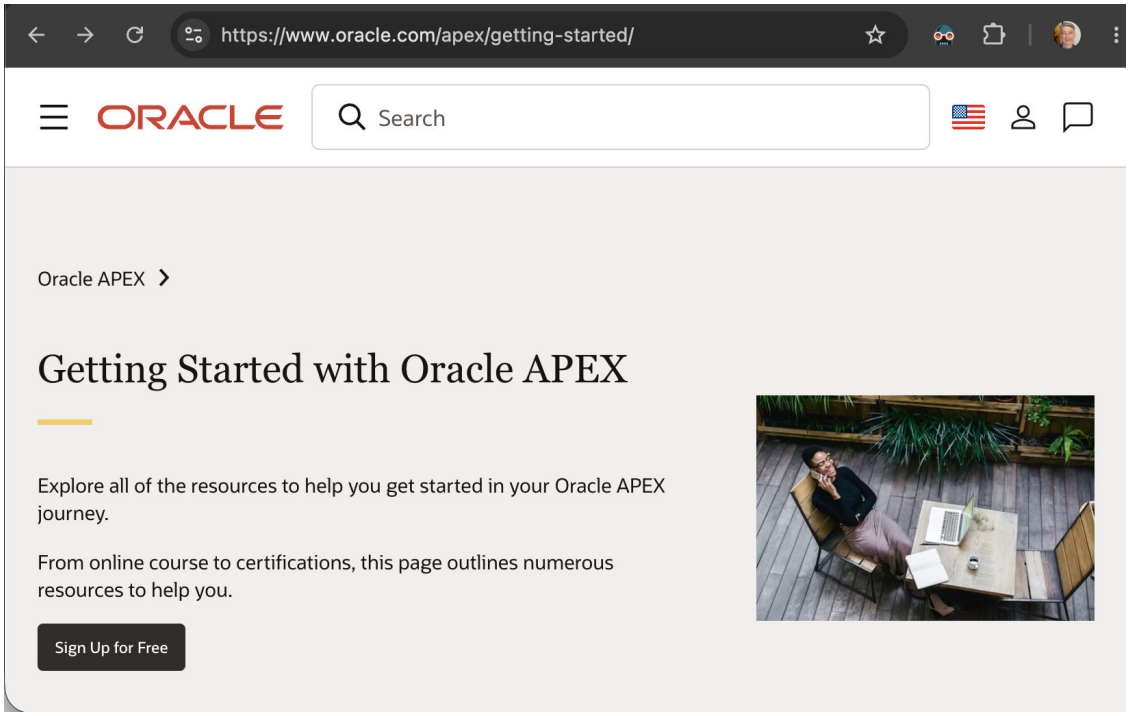
Find APEX learning resources and request a free workspace for hands-on practice.

The *Getting Started* page at oracle.com/apex/getting-started is your hub for quick access to documentation, training and certifications, labs and tutorials, videos, and additional resources about Oracle APEX.

As shown below, you can click **Sign Up for Free** to request a free workspace. It gives you access with no time limit to all features of the latest version of the APEX platform right from your browser for learning, exploration, or prototyping.

You can also provision an Always Free-Tier tenancy on Oracle Cloud Infrastructure. Oracle APEX is pre-installed in every OCI Autonomous AI Database.

Figure 1-8 Browsing Other Learning Resources



Copyright © 2026, Oracle and/or its affiliates.

Copyright © 2026, Oracle and/or its affiliates.

2

Working with Local Data

Enterprise applications thrive on data, and Oracle APEX makes working with any data simple and efficient. While it's also easy to integrate remote data, most APEX applications work primarily with data in local tables.

- Database Concepts Primer
  The Oracle APEX engine and App Builder run inside the Oracle database.
- Working with Sample Data
  While you are learning Oracle APEX, if you don't already have an existing set of database tables to experiment with, you can work initially with built-in sample data.
- Installing a Sample Dataset
  In addition to working with built-in sample data, you can also install additional sample datasets that include multiple tables related to a functional area.
- Importing Data from a File
  The APEX Builder's **Data Workshop** lets you easily import data from Excel spreadsheets or files in JSON, XML, or Comma-Separated Values (CSV) format.
- Creating Data Models with Quick SQL
  Create tables and relationships quickly by describing your data model in Quick SQL shorthand.
- Creating a Data Model Using Natural Language
  You can create a data model by using natural language to describe what data your application needs to manage.

2.1 Database Concepts Primer

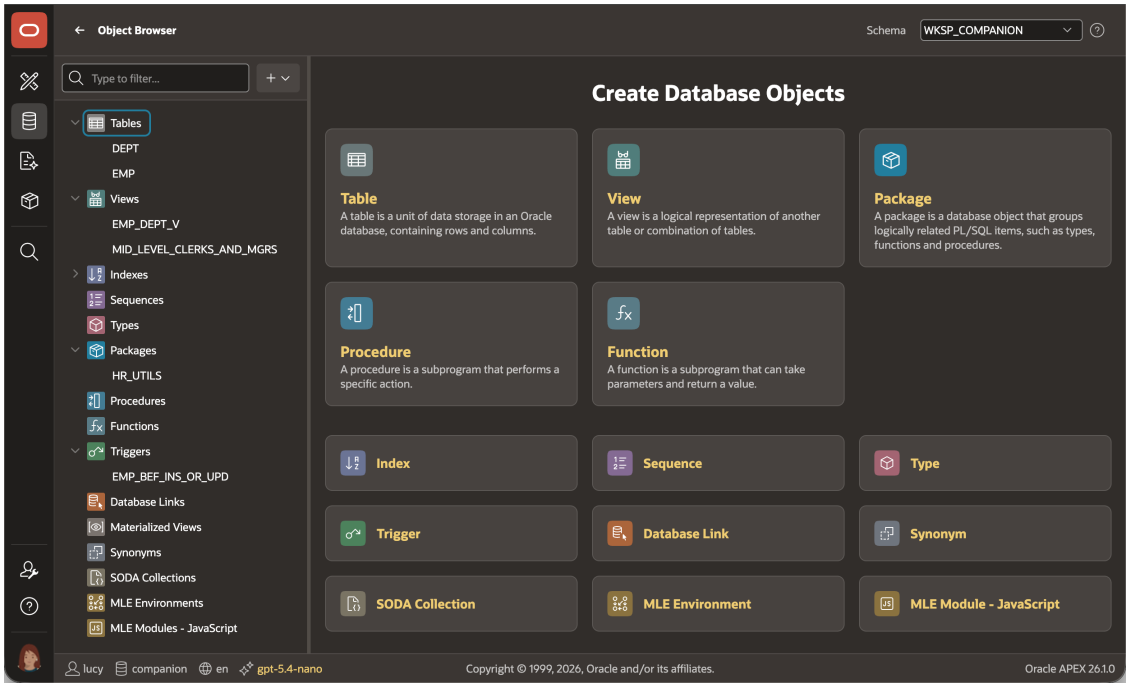
The Oracle APEX engine and App Builder run inside the Oracle database.

When your applications work with data in the *same* database, your development activities and applications benefit from zero-latency access. This performance advantage is unique to *local* database access, since using remote data from *another* Oracle database or from REST APIs requires network requests.

You create and maintain all the local database objects your application needs using the **Object Browser** in App Builder. In addition to creating tables for application data storage, you can also create views to easily reuse specific queries by name, define triggers to enforce business logic, organize custom code into packages, and more. The figure shows the **Object Browser** landing page, with the names of tables, views, a package, and a trigger visible in the navigator on the left.

Copyright © 2026, Oracle and/or its affiliates.

Figure 2-1 Maintaining Database Objects in Object Browser



- Database Tables
  You organize an application's data into tables, each representing a business entity like an employee or a department. Each column in a table corresponds to a property of that entity, like an employee's hire date or a department's location. Every column has a data type that defines the kind of information it holds, like text, numbers, or dates.
- Structured Query Language (SQL)
  While building your Oracle APEX application, you use the industry-standard Structured Query Language (SQL) to specify the data you want to work with in a simple, declarative syntax.
- Database Views, Triggers, and Packages
  While *tables* store your application data, it's important to understand how views, triggers, and packages complement them.

Related Topics

- Managing Database Objects with Object Browser in *Oracle APEX SQL Workshop Guide*

2.1.1 Database Tables

You organize an application's data into tables, each representing a business entity like an employee or a department. Each column in a table corresponds to a property of that entity, like an employee's hire date or a department's location. Every column has a data type that defines the kind of information it holds, like text, numbers, or dates.

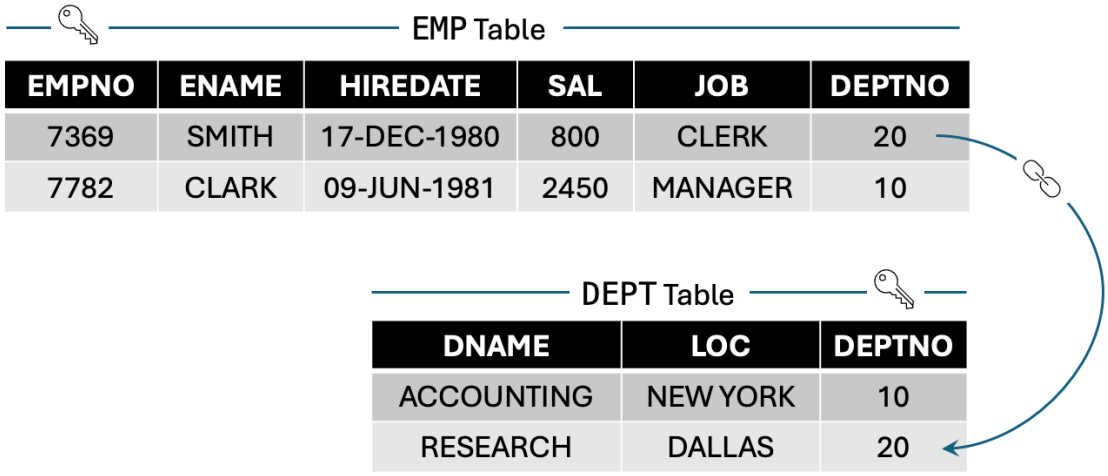
A table's **primary key** column uniquely identifies each row. This key lets you establish relationships between tables. By using it, you can link a row of data in one table to a related row in another. A **foreign key** is a column in one table that references the primary key value of another, ensuring data consistency and enforcing data integrity.

The figure below shows two tables representing the data for employees and the departments in which they work. The `EMPNO` column is the `EMP` table's primary key. The `DEPTNO` column plays

Copyright © 2026, Oracle and/or its affiliates.

that role in the DEPT table. Both display below with a key icon. The EMP table's DEPTNO column is a foreign key. Its value references the primary key of the DEPT table, indicating the department to which each employee belongs.

Figure 2-2 Two Tables with Primary Keys and Foreign Key



The APEX Builder's **Quick SQL** editor and **Create Data Model Using AI** wizard simplify creating an initial set of tables for your application, while the **Object Browser** makes it easy to evolve those tables over time.

Related Topics

- Managing Tables in *Oracle APEX SQL Workshop Guide*

2.1.2 Structured Query Language (SQL)

While building your Oracle APEX application, you use the industry-standard Structured Query Language (SQL) to specify the data you want to work with in a simple, declarative syntax.

You use SQL to indicate *what* data to retrieve or modify, without regard for *how* that will happen. While some spell out the acronym, most pronounce "SQL" like a movie "sequel".

Solid Foundation, Gold Standard

Using SQL, you stand on the shoulders of giants. Smart people at the world's most trusted IT companies have collaborated since the early 1970's to thoughtfully evolve the SQL language to earn its reputation as the gold standard of declarative data languages. It is comprehensive, but luckily you can learn as you go, starting with just the basics. As additional requirements arise, you can learn new SQL commands as needed.

- Basic SQL Statements
  You can use SQL productively for Oracle APEX app development by learning a small set of basic operations.
- SQL Mentor at Your Side
  APEX AI Assistant is your SQL mentor. After registering the generative AI service APEX should use, this AI-powered design-time companion is available in every code editor in App Builder. It can help you write, learn, and debug SQL on demand.

Copyright © 2026, Oracle and/or its affiliates.

2.1.2.1 Basic SQL Statements

You can use SQL productively for Oracle APEX app development by learning a small set of basic operations.

You use SQL's `SELECT` statement to retrieve data to display in a page or to reference in custom business logic. Its `FROM` clause names the desired table. When a table has a foreign key column, the `JOIN` clause lets you also retrieve data from related tables at the same time. Since a `SELECT` command is a request for information from your database, it is also known as a *query*. It produces results that answer some question about your application data like, "Who are all the employees that meet this criteria?"

Using other SQL statements, you can `INSERT` new rows of data into a table, `UPDATE` existing rows as needed, or `DELETE` rows no longer needed. When a logical business transaction requires multiple changes, you either save them all as a unit with `COMMIT` or abandon them all with `ROLLBACK`.

Your SQL commands can specify conditions on which rows in a table to select, update, or delete. Using a `WHERE` clause, you identify the participating rows using filter expressions. You sort the results with an `ORDER BY` clause. For example, the `SELECT` statement below retrieves the employee name, salary, and location of the department in which they work from related `EMP` and `DEPT` tables. It returns only employees whose salary is between 800 and 2450 and whose job is either "CLERK" or "MANAGER". The `JOIN … ON` clause asks to relate departments to employees using their respective `DEPTNO` value. The results are sorted by salary in descending order, that is, highest salary first.

```
SELECT emp.ename, emp.sal, dept.loc
  FROM emp
  JOIN dept ON emp.deptno = dept.deptno
 WHERE sal BETWEEN 800 AND 2450
   AND job IN ('CLERK','MANAGER')
 ORDER BY sal DESC
```

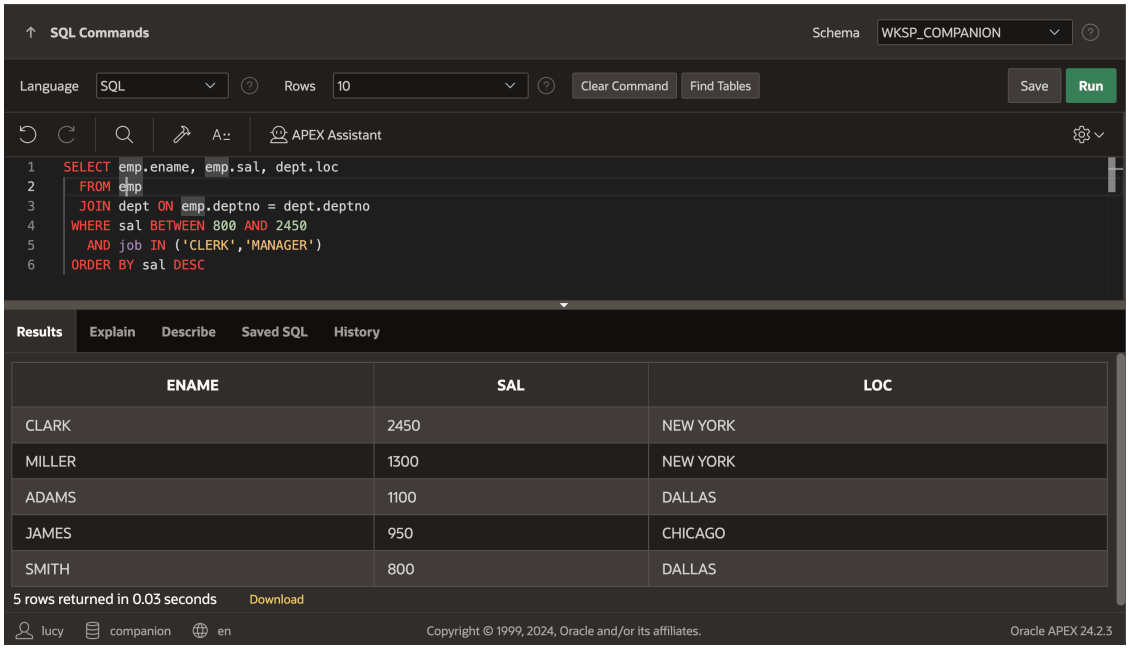
The query produces the following result rows:

```
ENAME          SAL LOC
--------- ------- -----------
CLARK         2450 NEW YORK
MILLER        1300 NEW YORK
ADAMS         1100 DALLAS
JAMES          950 CHICAGO
SMITH          800 DALLAS
```

APEX's native components can formulate SQL statements for you, given just a table name and the set of columns a page references. However, when writing custom business logic or fine-tuning the data for a particular part of a page, knowing SQL basics will be your superpower in daily development. In either case, APEX automatically augments SQL statements for you to add filtering, ordering, pagination, and aggregation to support end-user requirements. SQL Workshop's **SQL Commands** editor shown below is a simple sandbox for your SQL experiments.

Copyright © 2026, Oracle and/or its affiliates.

Figure 2-3 Experimenting with SQL in SQL Commands



Related Topics

- Using SQL Commands in *Oracle APEX SQL Workshop Guide*

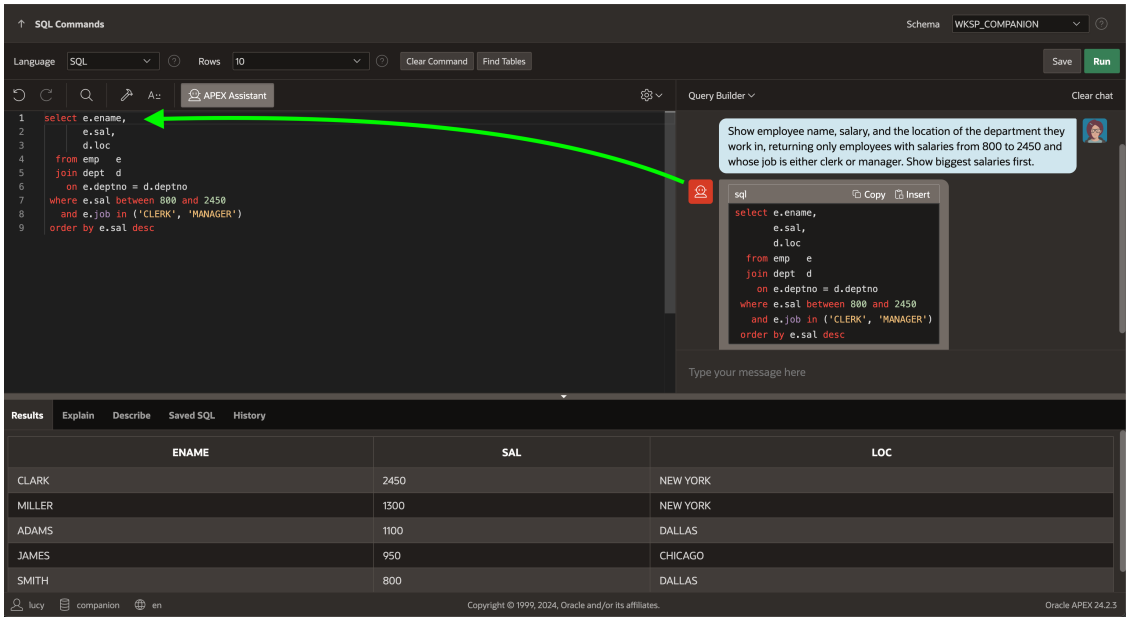
2.1.2.2 SQL Mentor at Your Side

APEX AI Assistant is your SQL mentor. After registering the generative AI service APEX should use, this AI-powered design-time companion is available in every code editor in App Builder. It can help you write, learn, and debug SQL on demand.

To write SQL, explain in English what data you need to work with, and APEX AI Assistant can help generate the statement to get the job done. To improve your understanding of SQL, APEX AI Assistant can help answer any questions or doubts you have. As you develop, it can also help suggest corrections to SQL when you encounter errors.

Copyright © 2026, Oracle and/or its affiliates.

Figure 2-4 APEX Assistant Helps Write SQL



Using Code Editor for SQL

In addition to accessing APEX AI Assistant in the **SQL Workshop**, **SQL Commands** page, you can also use it in the SQL Code Editor. This modal dialog is available throughout App Builder. Wherever you can specify SQL, APEX AI Assistant is one click away to help turbocharge your SQL authoring and education journey.

Related Topics

- Using APEX Assistant in *Oracle APEX App Builder User's Guide*

2.1.3 Database Views, Triggers, and Packages

While *tables* store your application data, it's important to understand how views, triggers, and packages complement them.

While you define a view using SQL, you write triggers, procedures, functions, and packages in PL/SQL, the procedural language extension to SQL. If your database is Oracle 26ai, you can also use JavaScript.

The **Object Browser** in App Builder lets you easily create and maintain any tables, views, triggers, and packages that your application requires.

- Database Views
  A database view is a named SQL query you can reuse to simplify data access across your APEX applications.
- Database Triggers
  Database triggers are a useful way to validate data and compute column values in one place, no matter which application access it.
- Database Packages
  Database packages contain a specification of available procedures and functions. Oracle recommends you use them to organize custom business logic.

Copyright © 2026, Oracle and/or its affiliates.

2.1.3.1 Database Views

A database view is a named SQL query you can reuse to simplify data access across your APEX applications.

You can create a database **view** to name a reusable `SELECT` statement. It identifies a curated subset of data appropriate to a specific business use case. The query can filter and join the data from related tables as needed. Then you can use the view by name in your application pages and business logic. The **Object Browser** in App Builder lets you create and maintain your views.

For example, the statement below creates a view named `mid_level_clerks_and_mgrs` using the `SELECT` statement from a previous section:

```
CREATE OR REPLACE VIEW mid_level_clerks_and_mgrs AS
SELECT emp.ename, emp.sal, dept.loc
  FROM emp
  JOIN dept ON emp.deptno = dept.deptno
 WHERE sal BETWEEN 800 AND 2450
   AND job IN ('CLERK','MANAGER')
```

Once you define the view, you can select data from it just like a table, including adding additional filtering and ordering:

```
SELECT ename, sal, loc
  FROM mid_level_clerks_and_mgrs
 WHERE loc LIKE '%A%'
ORDER BY sal
```

This produces a result of:

```
ENAME             SAL LOC
---------- ---------- ----------
SMITH              800 DALLAS
JAMES              950 CHICAGO
ADAMS             1100 DALLAS
```

Related Topics

- Managing Views in *Oracle APEX SQL Workshop Guide*

2.1.3.2 Database Triggers

Database triggers are a useful way to validate data and compute column values in one place, no matter which application access it.

You can define a trigger on a table to execute custom business logic before or after a row in that table is inserted, updated, or deleted. If the trigger fires before the operation, it can enforce validation rules and compute calculated column values if necessary before the row is saved. You can create and maintain your triggers in App Builder using **Object Browser**.

For example, the following trigger rounds an employee's salary to the closest multiple of 10 and raises an error if the salary value is greater than 9000. The syntax is easy to understand. It executes before insert or update on the `EMP` table for each row affected. Notice how it can

Copyright © 2026, Oracle and/or its affiliates.

reference the value of EMP table columns using the special bind variable syntax :NEW.*column_name*.

```
CREATE OR REPLACE TRIGGER emp_bef_ins_or_upd
BEFORE INSERT OR UPDATE ON emp FOR EACH ROW
BEGIN
    -- If incoming SAL value exceeds 9000, then raise error
    IF :NEW.sal > 9000 THEN
        raise_application_error(-20001,'Maximum salary is 9000');
    END IF;
    -- Round to nearest multiple of ten. The -1 means
    -- 1 digit to the *left* of the decimal point
    :NEW.sal := ROUND(:NEW.sal, -1);
END;
```

After creating this trigger on the EMP table, an attempt to set the employee JAMES' salary to *9954* using the following SQL statement results in an error:

```
UPDATE emp
   SET sal = 9954
 WHERE empno = 7900 /* JAMES */
```

As expected, this produces the following error and prevents the statement's intended change:

```
ORA-20001: Maximum salary is 9000
```

If instead you update JAMES' salary to *954* using a similar UPDATE statement, the update succeeds. However, if you SELECT the salary again, the trigger rounds the value 954 to 950.

Related Topics

- Managing Triggers in *Oracle APEX SQL Workshop Guide*

2.1.3.3 Database Packages

Database packages contain a specification of available procedures and functions. Oracle recommends you use them to organize custom business logic.

A package **specification** lists the procedures and functions available to call from application pages, triggers, or other packages. It also defines the names and data types of parameters these named program units accept. Parameters let a caller pass data into the function or procedure (IN), receive data back out of it (OUT), or pass data in both directions (IN OUT). Any IN parameter can be mandatory or optional. When not supplied explicitly, optional parameters take on the defined default value. The key distinction between functions and procedures is that a function returns a result value, while a procedure does not.

Public program units are a great way to provide application functionality for teammates to use in pages, workflows, or custom code. They just call a function or procedure, supplying any required parameter values. You can create and maintain your packages in App Builder using SQL Workshop's **Object Browser**.

For example, the hr_utils package specification below defines one procedure and one function. The p_additional_withholding parameter of the handle_employer_contribution

Copyright © 2026, Oracle and/or its affiliates.

procedure is optional, with a default value of `0` (zero). The `p_total_contribution` parameter returns information back `OUT` to the caller, while the other parameters pass `IN` values as input.

```
CREATE OR REPLACE PACKAGE hr_utils AS
    -------------------------------------
    PROCEDURE handle_employer_contribution(
        p_empno                  IN  NUMBER,
        p_additional_withholding IN  NUMBER DEFAULT 0,
        p_total_contribution     OUT NUMBER);
    -------------------------------------
    FUNCTION years_employed(
        p_hiredate IN DATE)
        RETURN        NUMBER;
END hr_utils;
```

You implement a package's publicly declared program units in a corresponding package body by writing custom business logic that remains private. The public/private distinction is important. It means you can change how a package's procedure or function is written without affecting the pages or other program units that use it. If you are distributing your application, you can optionally **wrap** a package body's code to prevent it from being readable.

The `hr_utils` package body below defines the behavior of the procedure and function in the package's public specification. Notice that the body can contain other program units, like the `years_ago` helper function, that are not accessible from outside the package. The `years_employed` function calls this package-private `years_ago` function.

```
CREATE OR REPLACE PACKAGE BODY hr_utils AS
    -------------------------------------
    FUNCTION years_ago(
        p_date IN DATE)
    RETURN NUMBER
    IS
    BEGIN
        RETURN FLOOR(MONTHS_BETWEEN(SYSDATE, p_date) / 12);
    END years_ago;
    -------------------------------------
    PROCEDURE handle_employer_contribution(
        p_empno                  IN NUMBER,
        p_additional_withholding IN NUMBER DEFAULT 0,
        p_total_contribution     OUT NUMBER)
    IS
    BEGIN
        apex_debug.enter('handle_employer_contribution');
        -- Interesting contribution code here, eventually
        -- assigning a value to p_total_contribution
    END handle_employer_contribution;
    -------------------------------------
    FUNCTION years_employed(
        p_hiredate IN DATE)
        RETURN        number
    IS
    BEGIN
        RETURN years_ago(p_hiredate);
    END years_employed;
END hr_utils;
```


Copyright © 2026, Oracle and/or its affiliates.

When a page, trigger, or another package calls a procedure like `handle_employer_contribution`, it provides values for the `IN` parameters and a location to store the `OUT` parameters. Using PL/SQL named parameter syntax, you can pass parameters in any order. Each value's purpose is clear.

```
-- Handle the employee's pension contribution
hr_utils.handle_employer_contribution(
    p_additional_withholding => 1500,
    p_empno                  => :P71_EMPLOYEE_ID,
    p_total_contribution     => :P71_TOTAL);
```

Using the APEX Builder's native "Invoke API" page process or workflow activity, it's also possible to **declaratively** call package procedures or functions. Simply configure the values for every parameter right in the Page Designer or Workflow Designer. This makes it even easier for less-technical teammates to take advantage of the functionality available to them in packages.

Related Topics

- Managing Packages in *Oracle APEX SQL Workshop Guide*

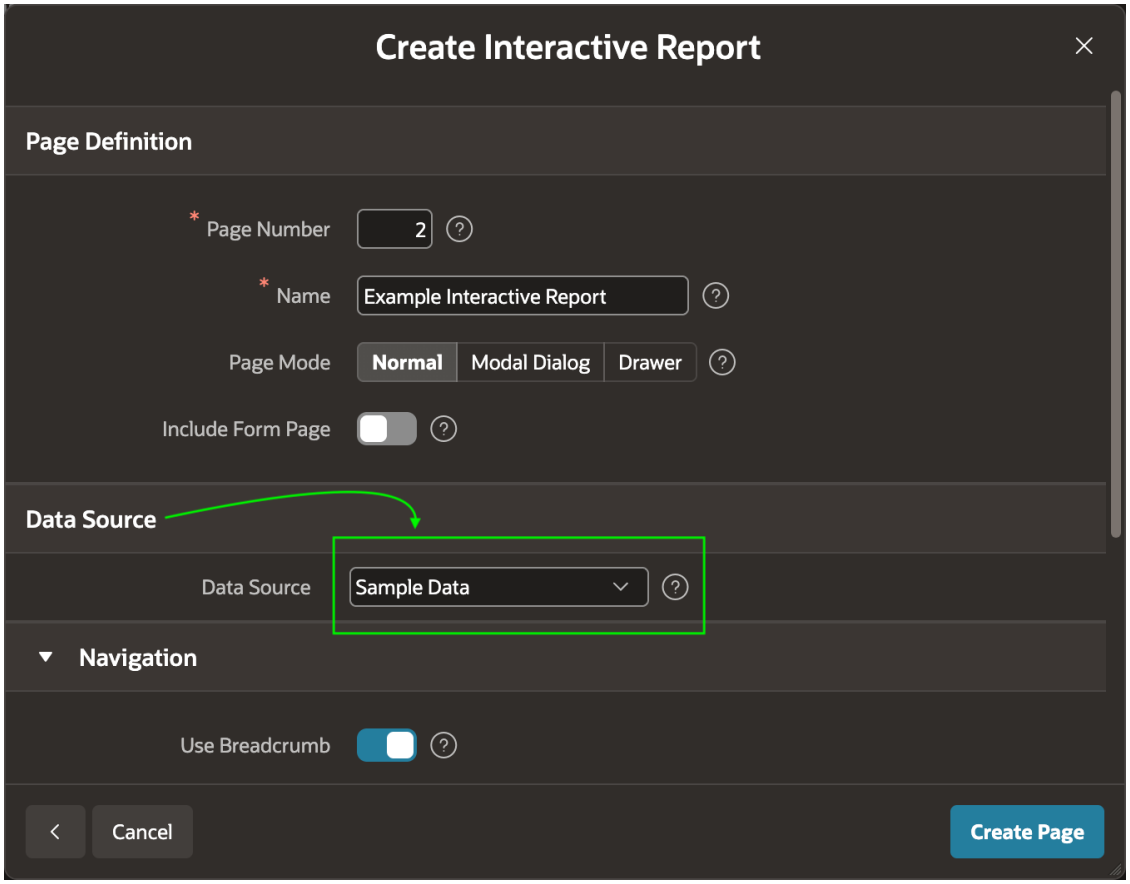
2.2 Working with Sample Data

While you are learning Oracle APEX, if you don't already have an existing set of database tables to experiment with, you can work initially with built-in sample data.

When creating new pages in App Builder, just select **Sample Data** for the Data Source as shown below.

Copyright © 2026, Oracle and/or its affiliates.

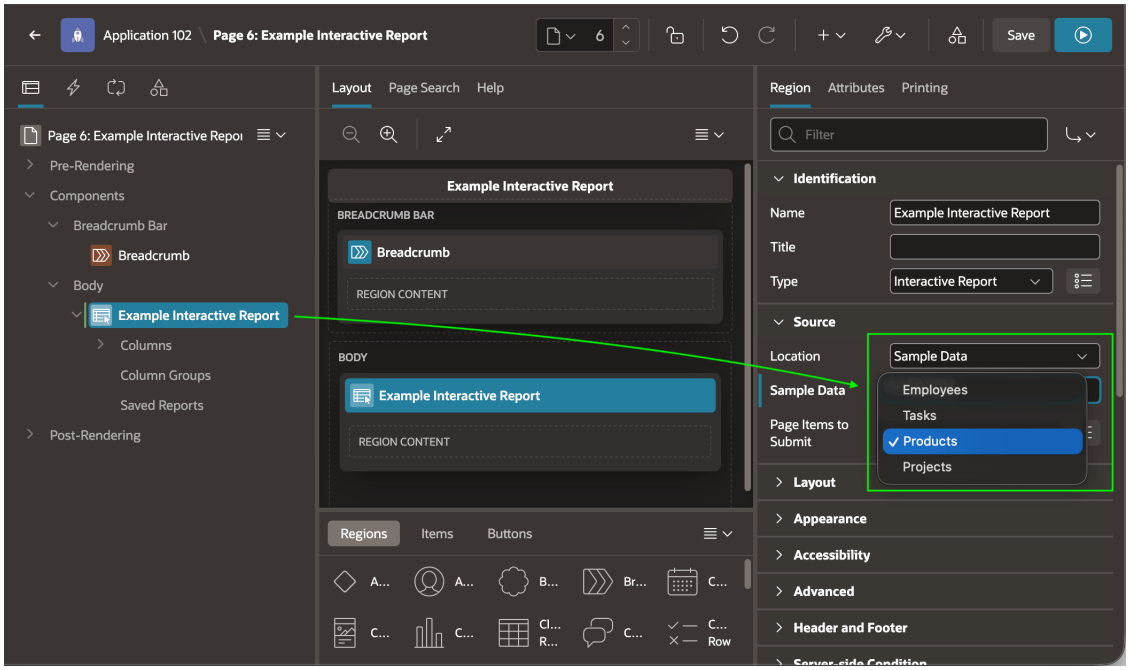
Figure 2-5 Choosing Sample Data as a Data Source in the Create Page Wizard



The built-in sample data defaults to using **Products**. As shown below, you can use the Property Editor in Page Designer to change to use **Employees**, **Tasks**, or **Projects** data instead.

Copyright © 2026, Oracle and/or its affiliates.

Figure 2-6 Changing the Type of Sample Data in Page Designer



2.3 Installing a Sample Dataset

In addition to working with built-in sample data, you can also install additional sample datasets that include multiple tables related to a functional area.

Each sample dataset contains one or more tables with sample data related to the following use cases:

- **Countries** – Countries, populations, and capitals
- **Customer Orders** – Customers, stores (with longitude/latitude), products, and orders
- **EMP / DEPT** – The venerable `EMP` and `DEPT` tables with employees and departments
- **HR Data** – Human Resources tables used by Oracle Education
- **Health Updates** – Patient health status updates
- **Project Data** – Projects, milestones, and tasks
- **Tasks Spreadsheet** – Spreadsheet-like table with tasks, dates, status, assignee, cost, and budget.

All sample data is in English. However, when installing the EMP / DEPT dataset, you can also choose Arabic, Chinese, Czech, French, German, Italian, Japanese, Korean, Polish, Russian, or Spanish. Wherever possible, the examples in this guide use the simple `EMP` and `DEPT` tables from the Employees and Departments sample dataset.

Related Topics

- Using Sample Datasets in *Oracle APEX SQL Workshop Guide*

Copyright © 2026, Oracle and/or its affiliates.

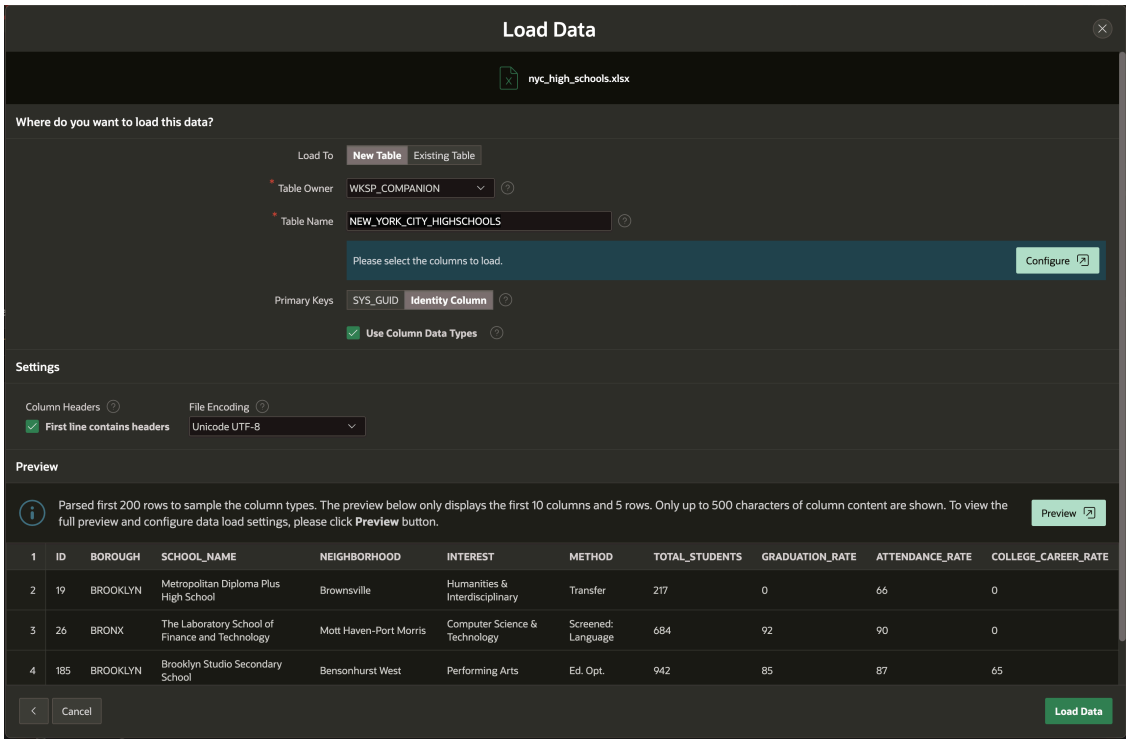
2.4 Importing Data from a File

The APEX Builder's **Data Workshop** lets you easily import data from Excel spreadsheets or files in JSON, XML, or Comma-Separated Values (CSV) format.

Many enterprise applications start small by importing existing data from an Excel spreadsheet. Migrating the data from a file to a central database ensures a single source of truth that all users can see simultaneously and modify where allowed. You can load data from a file on your computer, or one that resides in the cloud using an object bucket in Oracle Cloud Infrastructure (OCI).

Data Workshop parses the file to be loaded and automatically infers appropriate data types for the columns it identifies. You can load selected columns into an existing table, or into a new table **Data Workshop** creates. The figure shows loading a spreadsheet with data about New York City public high schools into a new table. The **Load Data** wizard previews the file's data and lets you configure which columns to load.

Figure 2-7 Loading an Excel Spreadsheet into a New Table in Data Workshop



Related Topics

- Using Data Workshop in *Oracle APEX SQL Workshop Guide*

2.5 Creating Data Models with Quick SQL

Create tables and relationships quickly by describing your data model in Quick SQL shorthand.

An application's **data model** is the set of tables and relationships that store the information it manages. When you are ready to create a new application, APEX Builder's **Quick SQL** editor is a productive way to iteratively design its tables. Using a shorthand syntax, you type in the

Copyright © 2026, Oracle and/or its affiliates.

business entities in your data model one per line, and indent the attributes and data types under each entity. You relate tables by identifying the foreign keys.

For example, imagine you need to build an application to manage a conference. The Quick SQL syntax below defines tables for `ROOMS`, `PRESENTERS`, `SESSIONS`, `ATTENDEES`, and `AGENDAS`. Quick SQL automatically creates a primary key named `ID` for each table, so you don't have to list it explicitly. The `NAME` and `TITLE` columns store variable-length character data with a maximum length of 100 characters (`vc100`), while `ROOMS` has two number columns to store its map coordinates. Each session has a `STARTS` date (with time) and a number `DURATION` in minutes. Notice how the `/fk` or `/references` annotation identifies a column as a **foreign key** along with the name of the table it references. In this example, each row in the `SESSIONS` table has a `ROOM_ID` for the room in which the session occurs, and a `PRESENTER_ID` for the person presenting the session. Similarly, each row in the `AGENDAS` table has a `SESSION_ID` for the session and a `ATTENDEE_ID` for the conference-goer who wants to attend it.

```
rooms
    name vc100
    latitude num
    longitude num

presenters
    name vc100

sessions
    title vc100
    starts date
    duration num
    room_id /fk rooms
    presenter_id /fk presenters

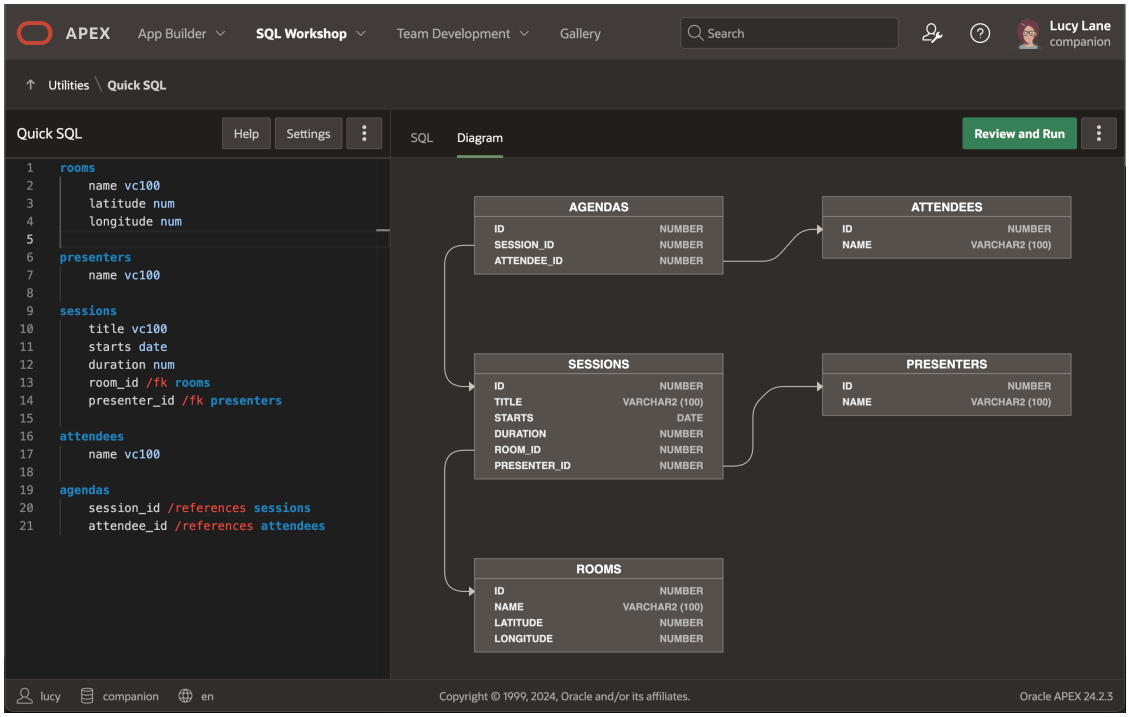
attendees
    name vc100

agendas
    session_id /references sessions
    attendee_id /references attendees
```

Whenever you pause your typing in the Quick SQL editor, as shown below, the **Diagram** tab immediately updates its visual representation of the tables, columns, and relationships. At any time you can peek at the **SQL** tab to see the statements needed to create the new tables. When you are happy with the data model, just click **Review and Run** to save the SQL script and create your tables.

Copyright © 2026, Oracle and/or its affiliates.

Figure 2-8 Iteratively Designing a New Data Model with Quick SQL



Related Topics

- Using Quick SQL in *Oracle APEX SQL Workshop Guide*

2.6 Creating a Data Model Using Natural Language

You can create a data model by using natural language to describe what data your application needs to manage.

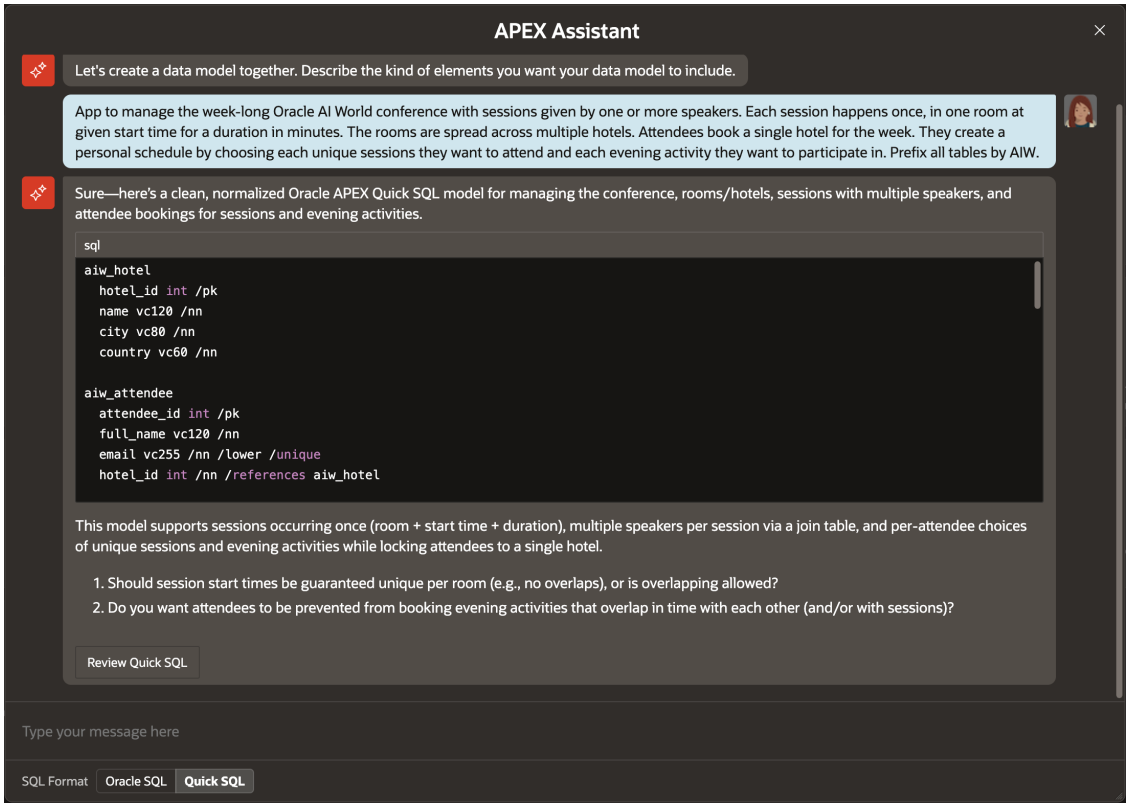
When you know the kind of app you need to build but need advice on the data model, APEX AI Assistant can help. APEX Builder's **Create Data Model Using AI** wizard opens a chat window where you explain the nature of your application. For example, if you need to build a conference management application, you could type in a prompt like:

> App to manage the week-long Oracle AI World conference with sessions given by one or more speakers. Each session happens once, in one room at given start time for a duration in minutes. The rooms are spread across multiple hotels. Attendees book a single hotel for the week. They create a personal schedule by choosing each unique sessions they want to attend and each evening activity they want to participate in. Prefix all tables by AIW.

As shown below, APEX AI Assistant responds with a proposed data model in Quick SQL notation with suggested tables, columns, data types, and relationships. You can carry on the conversation requesting adjustments if necessary, then click **Review Quick SQL** to inspect the data model in the Quick SQL diagram.

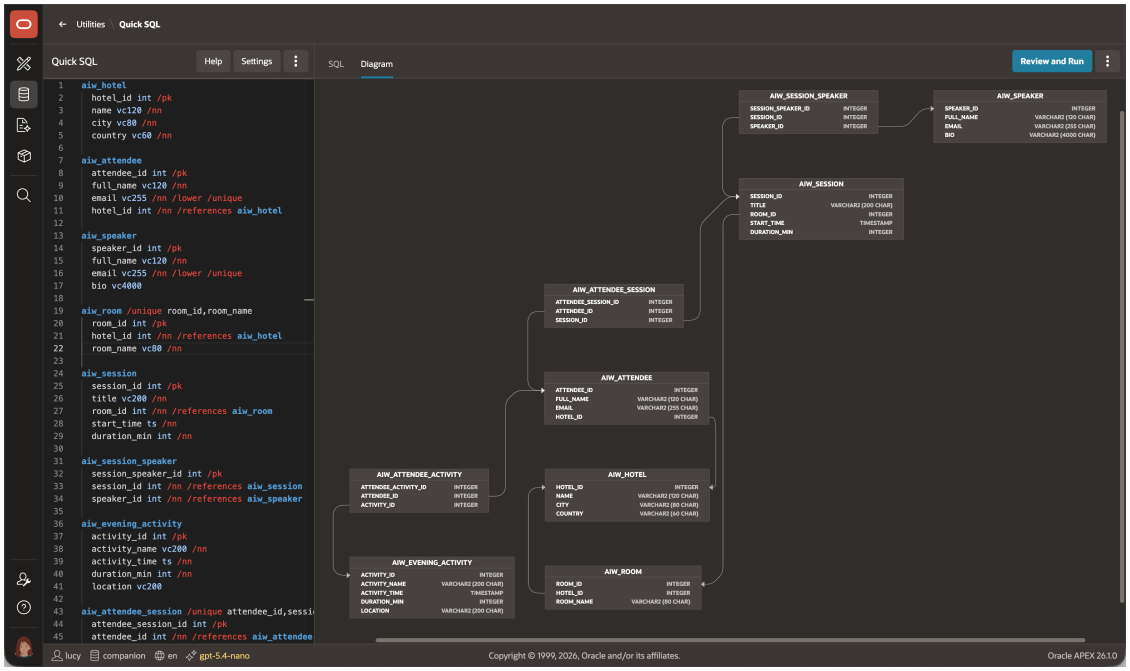
Copyright © 2026, Oracle and/or its affiliates.

Figure 2-9 Explaining the Data Model to APEX Assistant



Based on your prompt, APEX AI Assistant creates a data model like the one shown below. If you notice anything you forgot, you can edit the Quick SQL until you're ready to click **Review and Run** to save the script and create the tables.

Figure 2-10 Reviewing the Quick SQL Data Model Diagram



Copyright © 2026, Oracle and/or its affiliates.

Related Topics

- Creating a Data Model Using AI in *Oracle APEX SQL Workshop Guide*

Copyright © 2026, Oracle and/or its affiliates.

3

Simplifying User Experience Design

Oracle APEX simplifies designing experiences for users working with your application data.

In App Builder, creating and enhancing pages is productive, iterative, and immediate. You can create one using a wizard, or start with a blank page. Then, you use the APEX Page Designer to configure page elements and can run it at any time. Changes you save in Page Designer are instantly visible in the running page, so you can fluidly evolve its look and features. The APEX runtime executes your page definition, so there is no code generation or deployment step to wait for. You immediately experience the page *exactly* as users see it just by refreshing the browser.

With minimal effort and no low-level web developer skills, laying out your page is intuitive and adding interactivity is simple. Common tasks are straightforward, like defining list-driven items and items whose available choices depend on the value of others. Your pages look professional on desktop and mobile devices of any screen size and they can launch like native apps.

- Starting a New Application
  Create a new app from App Builder, then review its starter pages and page list.
- Creating Pages Quickly
  After creating a new application or editing an existing one, as shown below, the Create Page wizard shows a gallery of the most common page types.
- Working in Page Designer
  You create and enhance your application pages in Page Designer.
- Running and Iteratively Editing Pages
  At any time, you can run your page in the browser to use it like an end-user does.
- Adding Interactivity to Pages
  Add interactive behavior to your pages using Dynamic Actions.
- Universal Theme Design System
  Applications you build with Oracle APEX use a built-in design system called **Universal Theme**.
- Report Columns and Form Items
  Most region types have an associated data source with a set of columns APEX infers automatically.
- Cascading, List-Driven Items
  Apps often need end-users to pick one or more choices from a list of valid options, and sometimes the available items depend on a previous selection the user makes.
- Simple Layout for Any Screen Size
  Pages in your APEX application look polished and professional at any screen size with minimal effort.
- Launching Like Native Apps
  With a single click, you can let any user install and launch your APEX app like a native app.

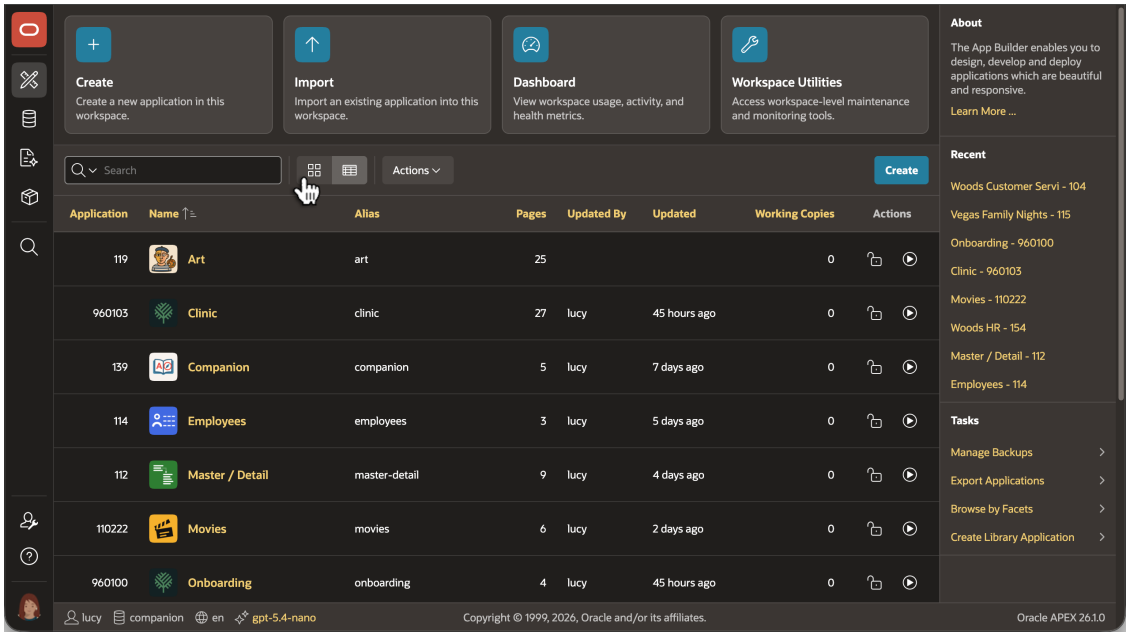
Copyright © 2026, Oracle and/or its affiliates.

3.1 Starting a New Application

Create a new app from App Builder, then review its starter pages and page list.

App Builder is your home base for working on applications in your workspace. As shown below, your existing apps display in a list with details, or using the **View Icons** toolbar button next to the **Search** field you can switch to an icon view.

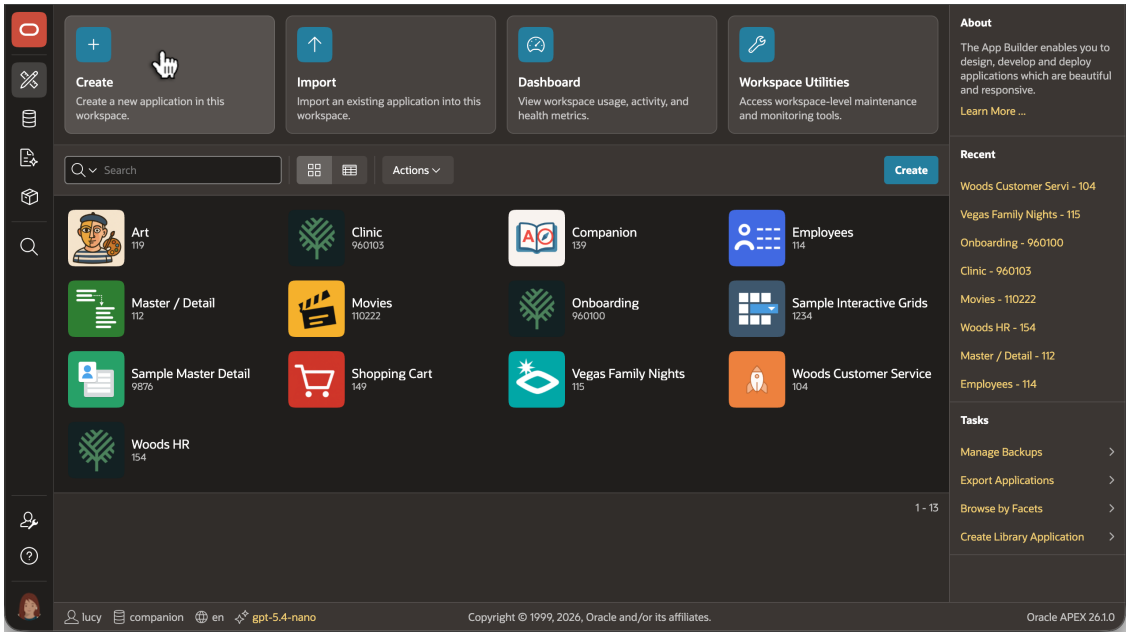
Figure 3-1 App Builder's Home Page Lists All the Applications in Your Workspace



To create a new application, click the **Create** button along the top.

Copyright © 2026, Oracle and/or its affiliates.

Figure 3-2 Creating a New Application from the App Builder Home Page



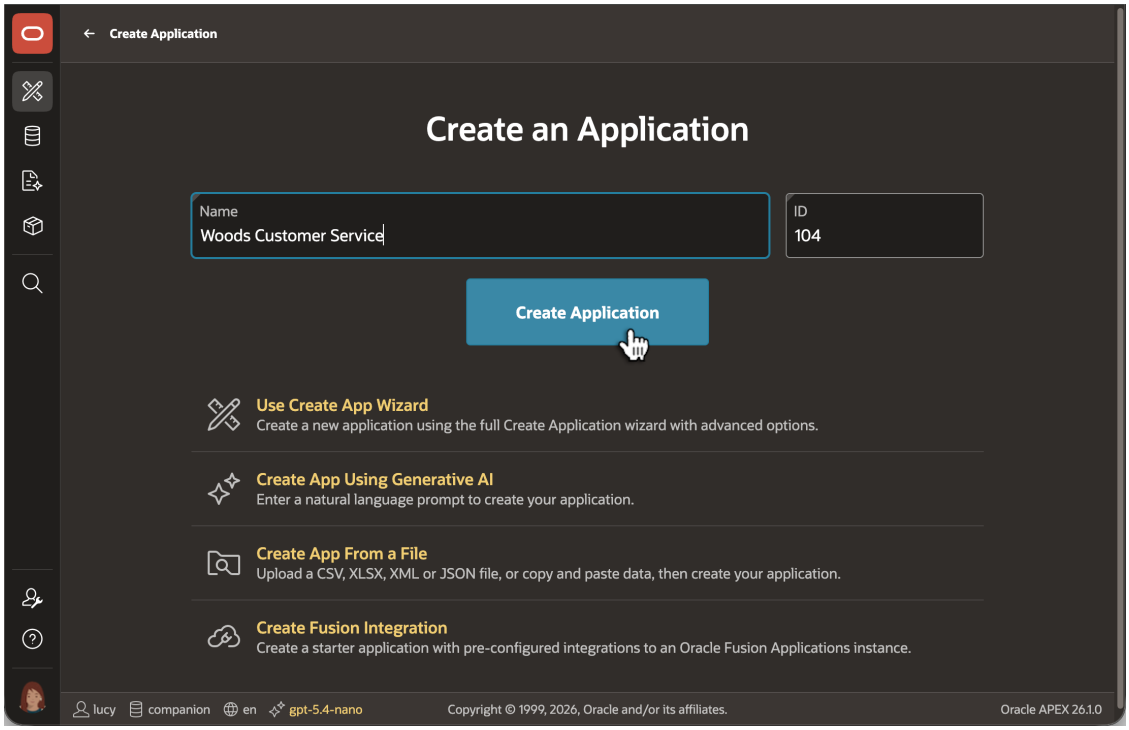
To start simple, just enter an application name and click (**Create Application**). Every APEX application in your workspace has a numeric ID as well. You can choose a number, or use the available ID App Builder suggests.

Notice below that there are additional options to:

- **Use Create App Wizard** - generate common page types and features for existing tables
- **Create App Using Generative AI** - explain the pages to generate in natural language
- **Create App from a File** - Upload data from a file then generate pages for it
- **Create Fusion Integration** - Set up an app to extend Fusion Applications.

Copyright © 2026, Oracle and/or its affiliates.

Figure 3-3 Create an App by Assigning a Name, or Use a Wizard for More Options



If you just enter an name and click (**Create Application**), your new app starts with three pages:

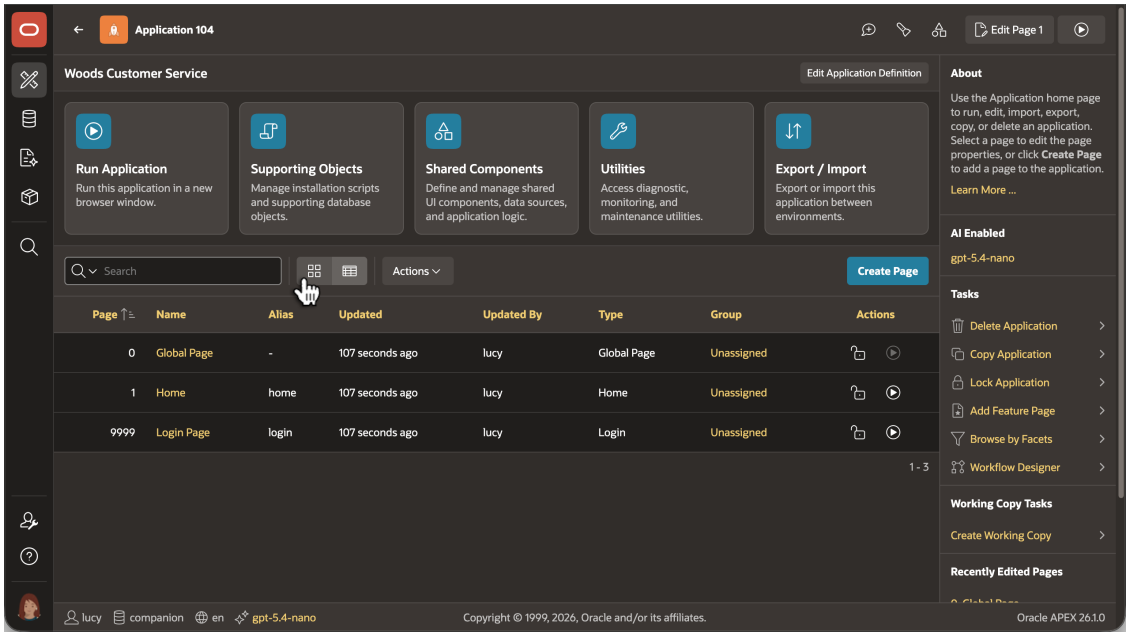
- **Home** - the initial page users see when you run the app
- **Global Page** - a special page whose components appear on every other page
- **Login Page** - the page user's see when they provide credentials to access the app.

> **Tip**
>
> By default, every page includes each component you add to the Global Page. To control when a global page component appears, add a condition to it. See Making Almost Any Element Conditional.

Each page in an application has both a number and user-friendly "alias" to uniquely identify it. The page alias appears in the browser address bar when navigating to a page. Notice both show in the page list below. If you prefer seeing less detail about your pages, use the toolbar button next to the **Search** field to toggle to icon view.

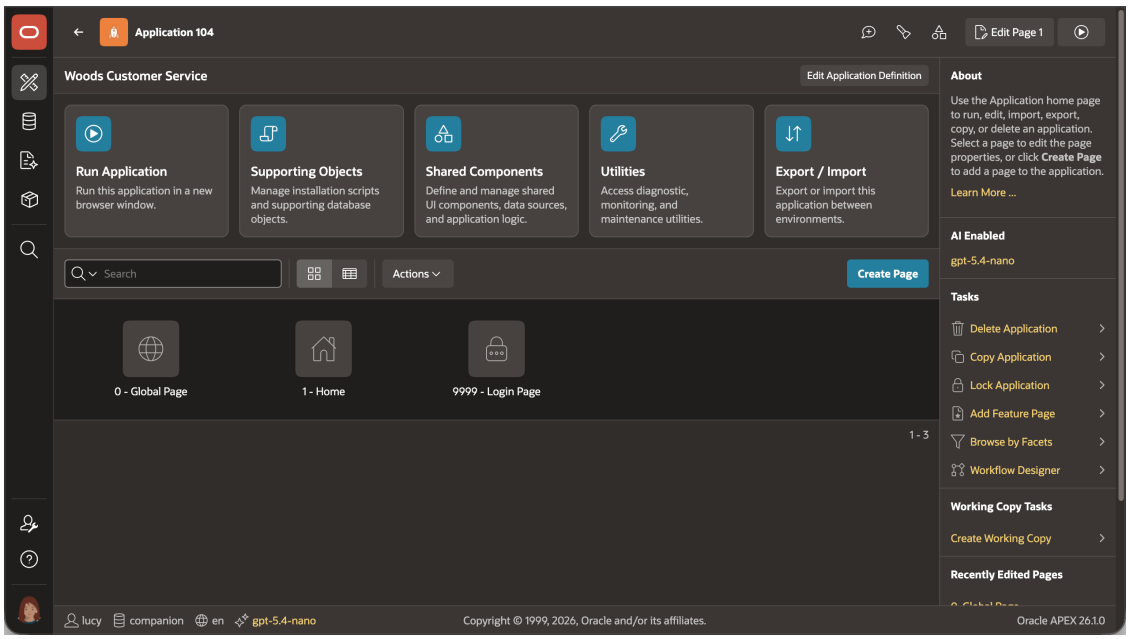
Copyright © 2026, Oracle and/or its affiliates.

Figure 3-4 Application Landing Page Shows a List of Pages the App Contains



Each page's icon in this view reflects its page type. In both icon and list views, when your application has more pages than will fit in a screenful, pagination links appear.

Figure 3-5 Toggle Between a List and Icon View of Your Application's Pages



Related Topics

- Using the Full Create Application Wizard in *Oracle APEX App Builder User's Guide*
- Creating an App Using Generative AI in *Oracle APEX App Builder User's Guide*

Copyright © 2026, Oracle and/or its affiliates.

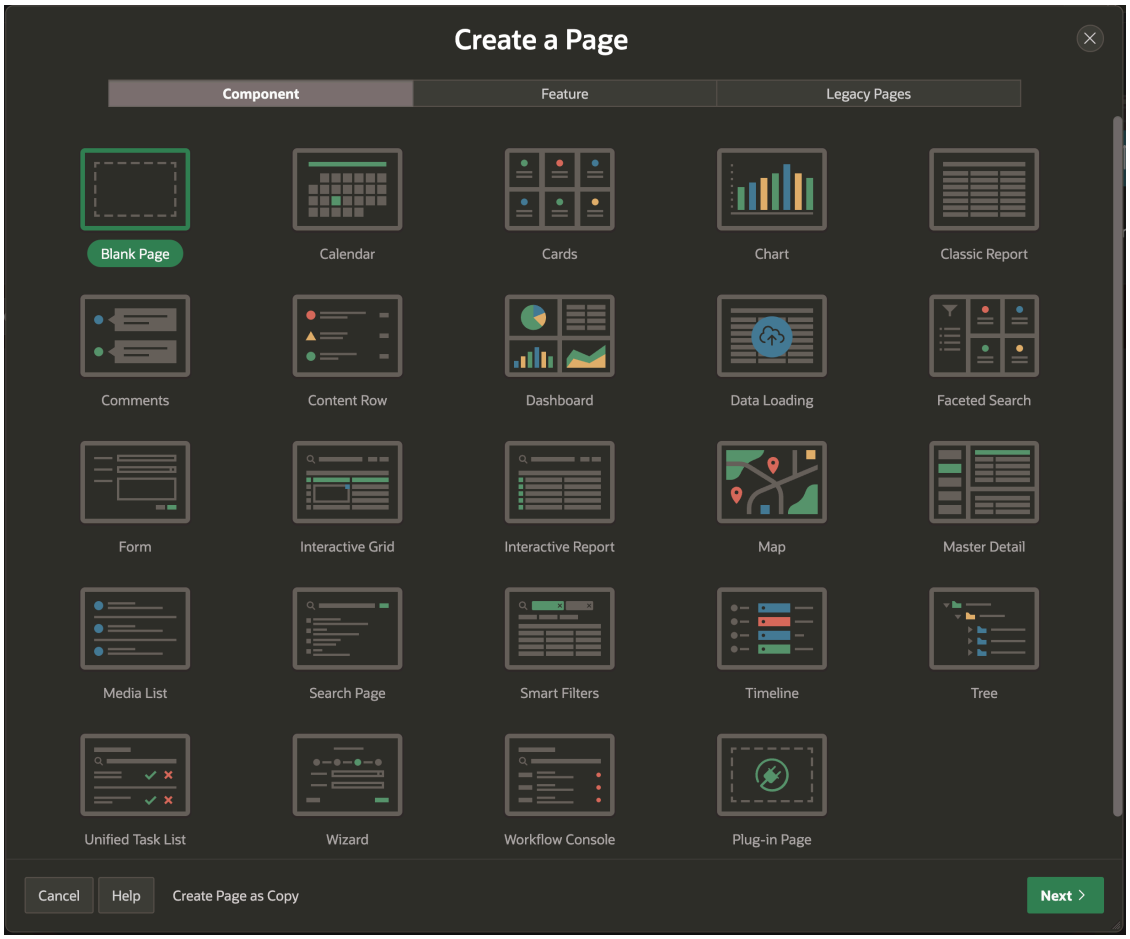
- Creating an Application from a File in *Oracle APEX App Builder User's Guide*
- Integrating APEX with Fusion Applications in *Oracle APEX App Builder User's Guide*

3.2 Creating Pages Quickly

After creating a new application or editing an existing one, as shown below, the Create Page wizard shows a gallery of the most common page types.

After picking a page type, the wizard lets you start with sample data or select a data source. You can opt to include additional context with a breadcrumb as well as add an application menu entry. In a few clicks, you get a page you can iteratively enhance in the Page Designer and run anytime. For several of the report region types, the wizard also lets you create a related form page to create new rows, edit the data of a selected row, or delete rows no longer needed.

Figure 3-6 Create Page Wizard



Related Topics

- Creating a Report Using the Create Page Wizard in *Oracle APEX App Builder User's Guide*
- Creating a Form Using the Create Page Wizard in *Oracle APEX App Builder User's Guide*

Copyright © 2026, Oracle and/or its affiliates.

- Creating Master Detail Forms in *Oracle APEX App Builder User's Guide*
- Creating Unified Task Lists in *Oracle APEX App Builder User's Guide*
- Creating the Workflow Console and Details Page in *Oracle APEX App Builder User's Guide*

3.3 Working in Page Designer

You create and enhance your application pages in Page Designer.

Page Designer includes tabs showing information about rendering, layout, dynamic actions, and processing. You use the Property Editor to configure settings for any component in these tabs. Additional areas like the component palette, search tab, and help tab add additional just-in-time assistance to let you quickly build the pages your apps require.

- Key Page Designer Areas
  The main areas of Page Designer you use to configure your page are the Property Editor and the **Rendering**, **Layout**, **Dynamic Actions**, **Processing**, and **Page Search** tabs. As you select one or more components in any of these tabs, the Property Editor updates to reflect your selection.
- Page Layout Based on Template with Slots
  Your chosen template defines the layout of each page. The template's named *slots* arrange the areas of the page that can contain visual components like regions, page items, or buttons.
- Adding Regions in Page Designer
  The APEX Page Designer's region palette includes all the region types you can drag and drop onto a slot in your page.
- Adding Items in Page Designer
  You can use the **Page Items** palette in Page Designer to add UI controls to the page.

3.3.1 Key Page Designer Areas

The main areas of Page Designer you use to configure your page are the Property Editor and the **Rendering**, **Layout**, **Dynamic Actions**, **Processing**, and **Page Search** tabs. As you select one or more components in any of these tabs, the Property Editor updates to reflect your selection.

The **Help** tab works in reverse. By clicking on the name of any property in the Property Editor, the **Help** tab updates to display context-sensitive help about that particular setting.

- Rendering, Layout, and Property Editor
  In the default arrangement of Page Designer tabs, the **Rendering** tab is grouped together with the **Dynamic Actions**, **Processing**, and **Page Shared Components** tabs, and all present a tree of their respective contents.
- Configuring Settings with Property Editor
  You use the Property Editor to review and change the settings of components on the page.
- Configuring Dynamic Behavior
  Use trigger actions to configure dynamic behavior for buttons and menus, and event handlers in the **Dynamic Actions** tab to react to value changes, dialog closure, and other user interface events.
- Validation, Processing, and Branching
  Use the **Processing** tab shown below to review and declaratively configure what happens when the end-user submits the page.

Copyright © 2026, Oracle and/or its affiliates.

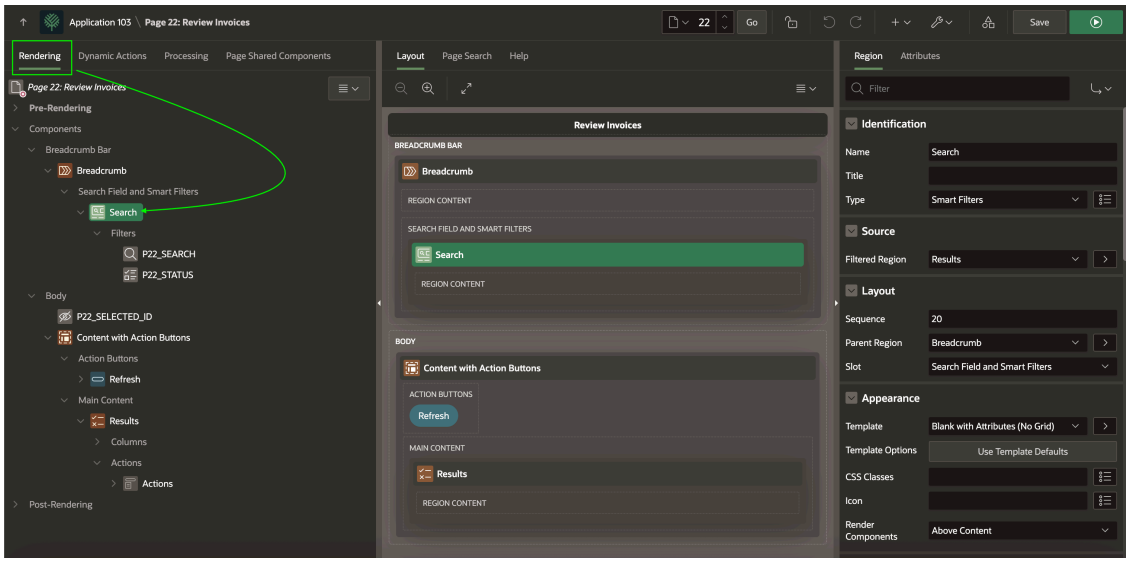
- Searching Your Page
  The **Page Search** tab lets you search your page to find anything you're looking for.
- Help on Any Setting at Your Fingertips
  The **Help** tab shows detailed information about any setting you select in the Property Editor.

3.3.1.1 Rendering, Layout, and Property Editor

In the default arrangement of Page Designer tabs, the **Rendering** tab is grouped together with the **Dynamic Actions**, **Processing**, and **Page Shared Components** tabs, and all present a tree of their respective contents.

As shown below, their tab names show when there is enough horizontal space. The **Rendering** tab displays the hierarchy of regions, page items, and buttons on the page. Tree headings reflect the different page template and region slots where components can appear.

Figure 3-7 Rendering, Dynamic Actions, Processing, and Page Shared Components Tabs

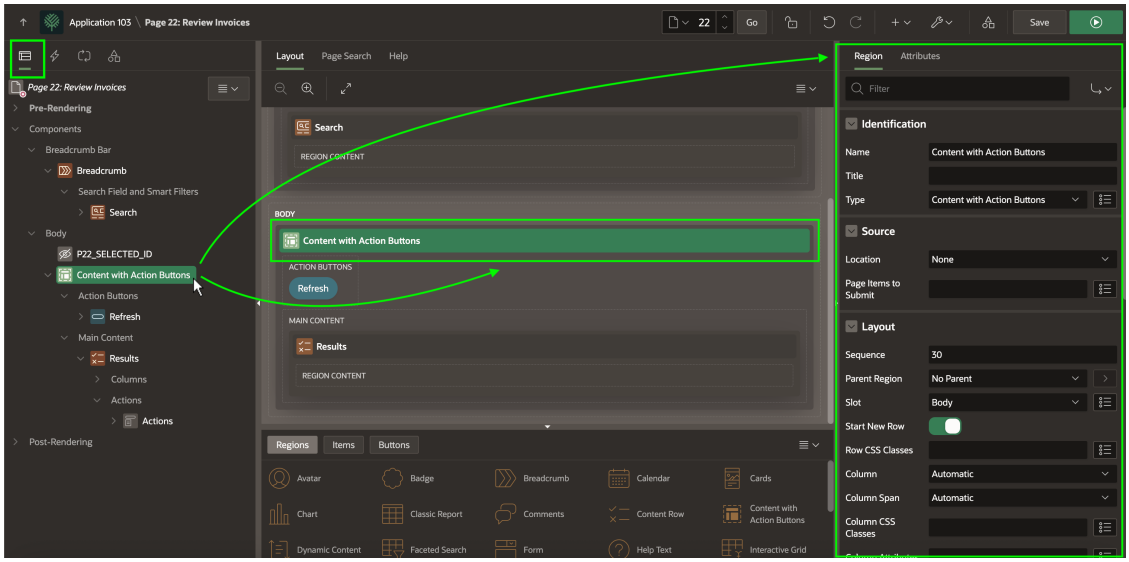


Selecting a component in the **Rendering** tab highlights it in the **Layout** tab, and updates the Property Editor to reflect the component's settings. The figure below shows how these same four tabs display only an appropriate icon when the panel width is narrower. You can use the vertical splitters between panels to adjust their widths. By clicking the small arrow in the splitter, you can collapse or expand the side columns as needed.

If useful, you can drag any tab to another position in its current panel, or into any position in a different panel. To reset the tab arrangement, choose **Reset Layout** from the **Utilities** dropdown menu. Its icon is a wrench with a down-pointing chevron.

Copyright © 2026, Oracle and/or its affiliates.

Figure 3-8 Using Property Editor with Rendering and Layout Tabs



3.3.1.2 Configuring Settings with Property Editor

You use the Property Editor to review and change the settings of components on the page.

The Property Editor displays all settings for the selected component. Some components have settings that appear across multiple tabs. The property editor supports multiple selection, searching for a selected property, and pinning a search to examine the same setting across multiple components.

Configuring Settings for Multiple Components

When you select multiple components in the **Rendering**, **Dynamic Actions**, or **Processing** tabs, the Property Editor shows the settings in common to all selected components. To perform multiple selection, you can:

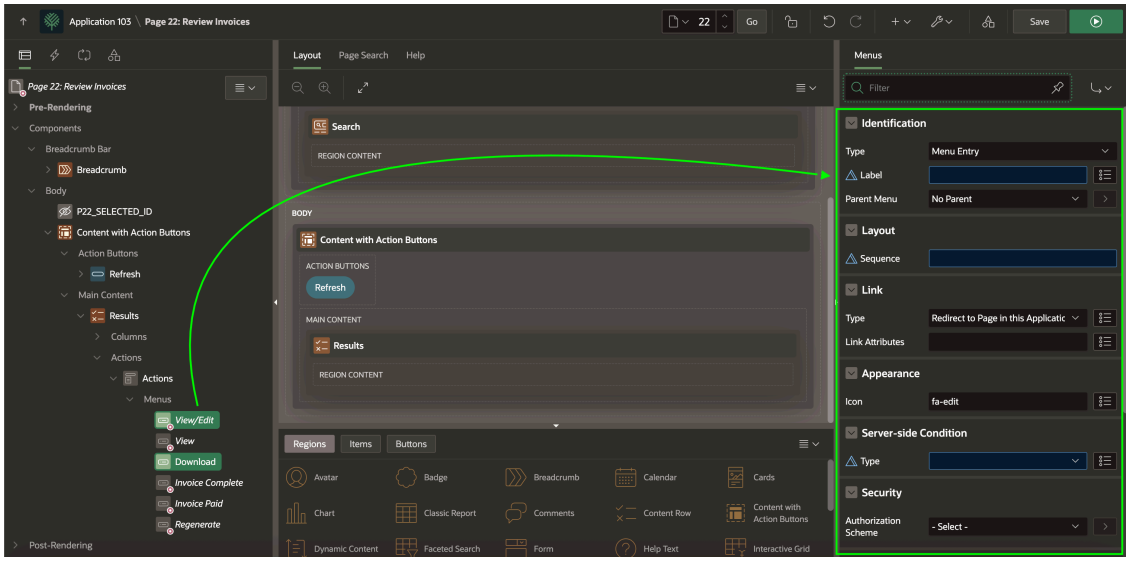
- select one component then Shift-select another to select a range, or
- Control- or Command-select any set of components.

As shown below, when the selected components have the same value for a common setting, like **Icon**, the property value displays. When their values for a property differ, a delta symbol appears and the property shows empty.

Setting any property value while multiple components are selected sets that same value on all selected components.

Copyright © 2026, Oracle and/or its affiliates.

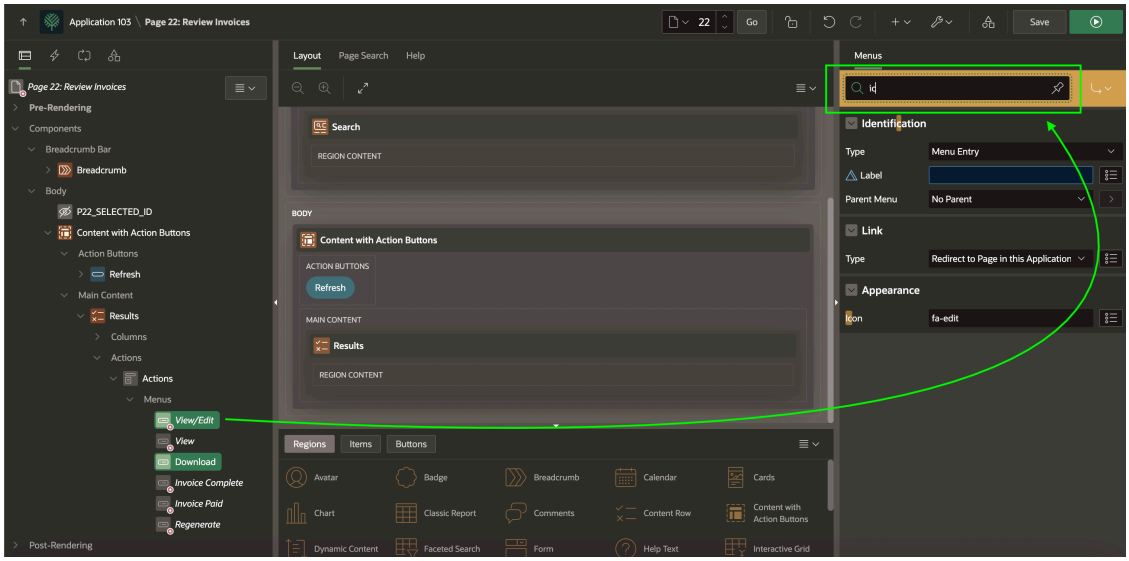
Figure 3-9 Working with Properties of Multiple Selected Components



Searching for a Particular Setting

Use the **Filter** search field at the top of the Property Editor to find a particular setting by name. As shown below, after typing `ic` into the field, the editor displays a filtered view of properties whose section title or property name contains the search term. Search results include the *Identification* group and the *Icon* setting, both of which contain `ic` in their name. This property search helps you quickly narrow the focus to just the property you need to change. Press the Escape key while the search field has focus to clear the search and see all settings again.

Figure 3-10 Searching the Property Editor to Find a Setting



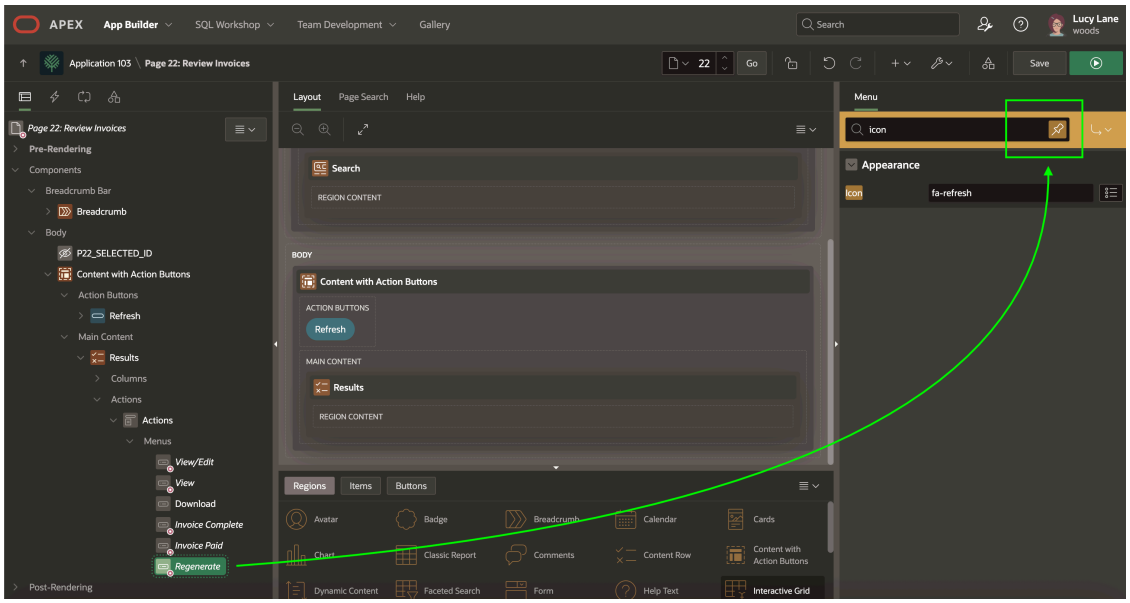
Pinning a Property Editor Search

If you plan to inspect or change a particular property across multiple components, select them all, search in the Property Editor to find the setting, and then set their common property in one

Copyright © 2026, Oracle and/or its affiliates.

go. As shown below, you can also search the Property Editor *first* to find the setting and then click **Pin** in the search field. This "freezes" the search until you click **Pin** again to unpin the search. With the search pinned, you can visit any number of components in the page and see or change only the properties in the filtered Property Editor.

Figure 3-11 Pinning a Property Editor Search to Work on Multiple Components



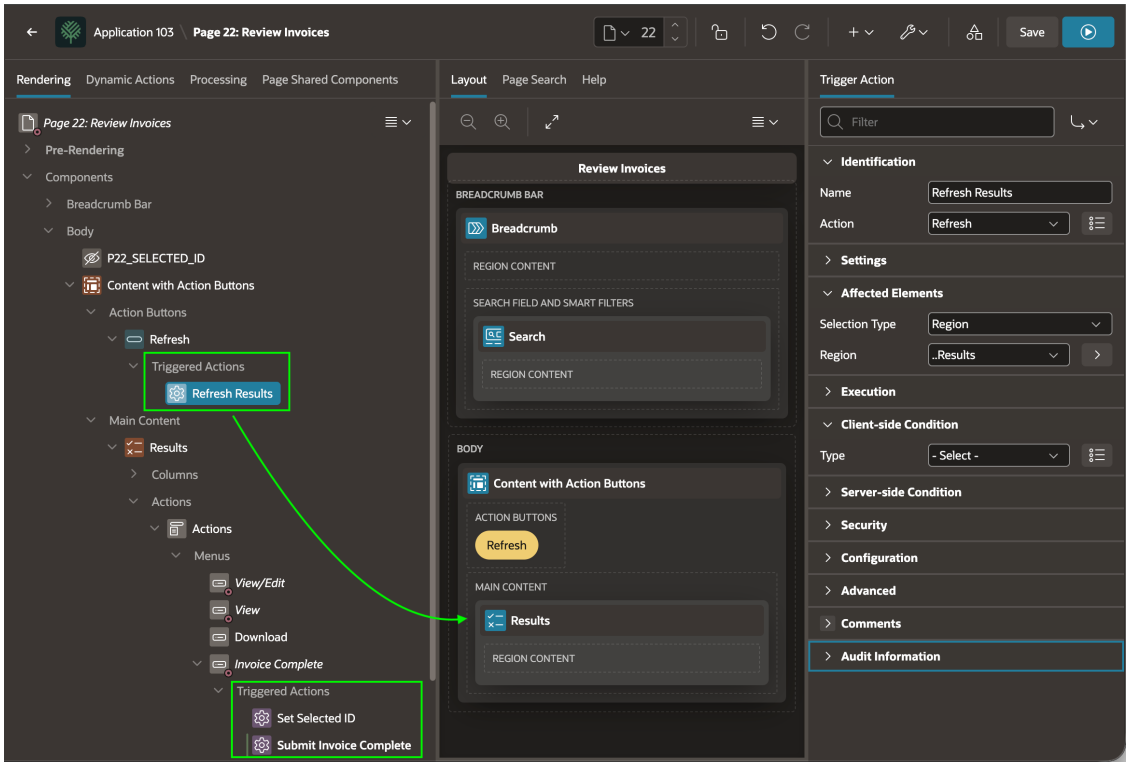
3.3.1.3 Configuring Dynamic Behavior

Use trigger actions to configure dynamic behavior for buttons and menus, and event handlers in the **Dynamic Actions** tab to react to value changes, dialog closure, and other user interface events.

When your user needs to perform an action by clicking a button or menu, you can add Triggered Actions directly to the element in the **Rendering** Tab. As shown below, the **Refresh** button triggers an action to refresh the **Results** region. The **Invoice Complete** menu item sets the ID of the selected invoice and submits the page to perform business logic related to completing an invoice.

Copyright © 2026, Oracle and/or its affiliates.

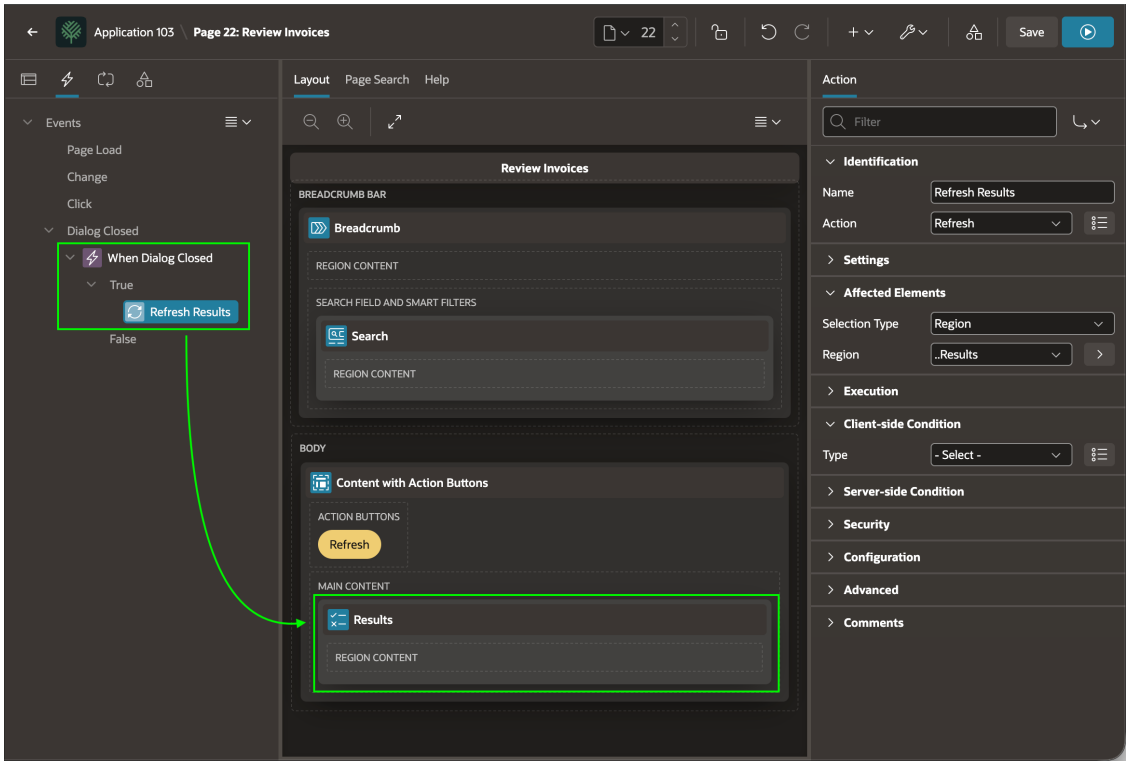
Figure 3-12 Define Triggered Actions on Buttons and Menus in the Rendering Tab



To handle any other user interface events, you can use dynamic action event handlers. The **Dynamic Actions** tab shows all event handlers defined on the page, and the sequence of Dynamic Action steps that you've configured to execute when each event occurs. As shown below, when the user closes a modal dialog this page invokes, this page handles that event by refreshing the **Results** region.

Copyright © 2026, Oracle and/or its affiliates.

Figure 3-13 Dynamic Action Tabs Shows Event Handlers and Actions



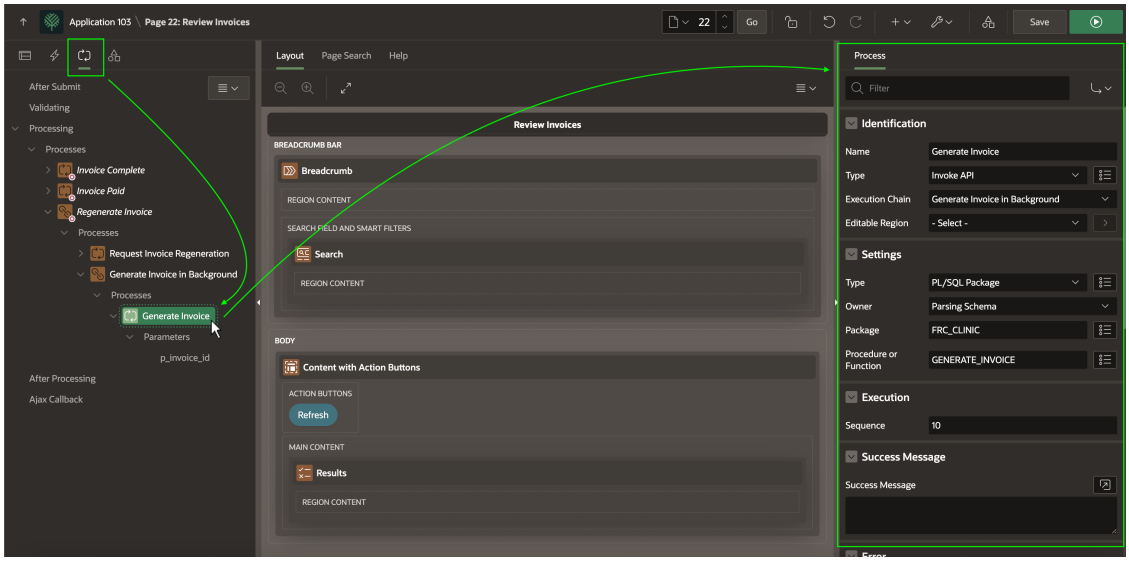
3.3.1.4 Validation, Processing, and Branching

Use the **Processing** tab shown below to review and declaratively configure what happens when the end-user submits the page.

You can define validations to ensure data correctness, page processes to perform work of various kinds or to do more complex validation checks, and branches that determine the next page the user sees if no validation errors occur. After selecting any validation, page process, or branch in the tree, the Property Editor updates to shows its settings.

Copyright © 2026, Oracle and/or its affiliates.

Figure 3-14 Processing Tab Shows Page Submit Validation and Processing

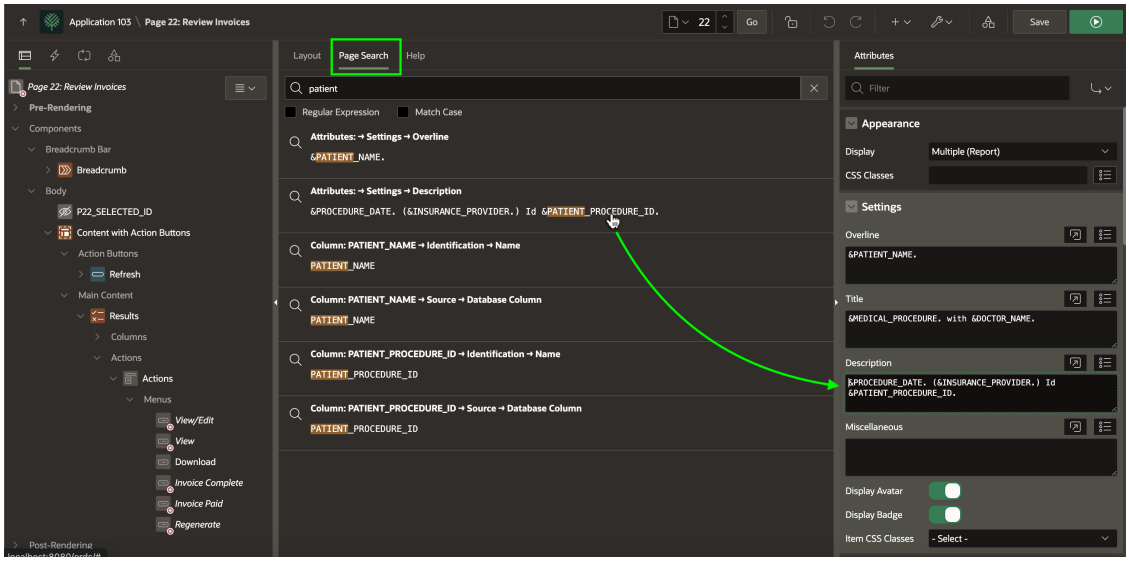


3.3.1.5 Searching Your Page

The **Page Search** tab lets you search your page to find anything you're looking for.

Clicking on any search result in the **Page Search** tab shows its settings in the Property Editor. The figure shows the results of searching the Review Invoices page (22) for the text `patient`, followed by clicking on the second result. Its properties display immediately in the Property Editor.

Figure 3-15 Find and Navigate to Anything Anywhere on Your Page



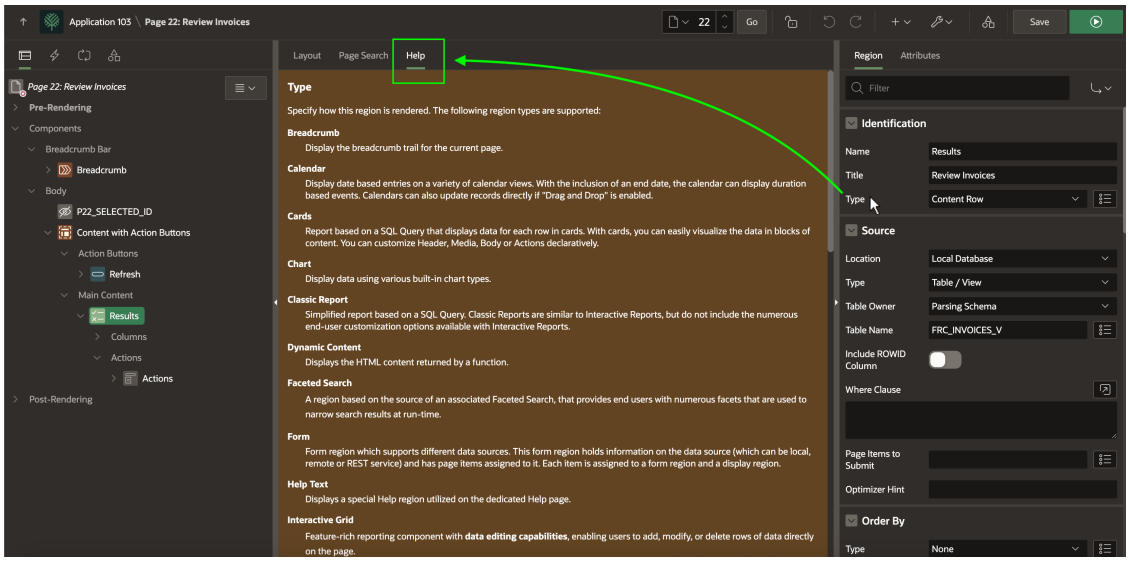
3.3.1.6 Help on Any Setting at Your Fingertips

The **Help** tab shows detailed information about any setting you select in the Property Editor.

Copyright © 2026, Oracle and/or its affiliates.

As shown below, after clicking on the **Type** setting label, the **Help** tab updates to explain the setting, available options, and usage examples. For settings supporting substitution strings, the **Help** tab information confirms the kinds of substitutions allowed, the syntax to use, whether you can use template directives, and often includes examples of usage.

Figure 3-16 Help Tab Shows Detailed Information About Selected Setting

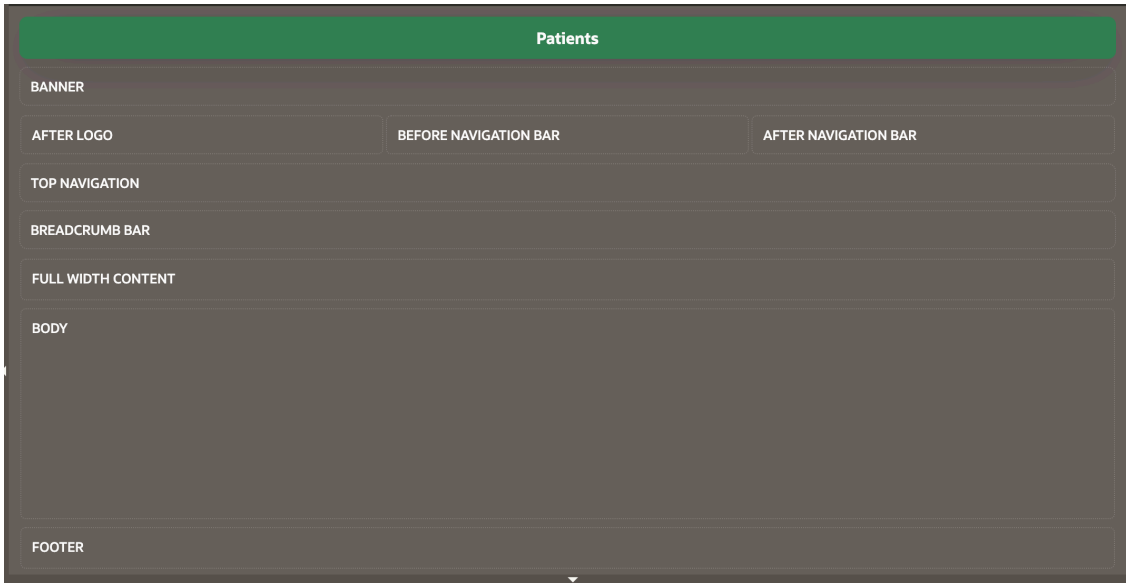


3.3.2 Page Layout Based on Template with Slots

Your chosen template defines the layout of each page. The template's named *slots* arrange the areas of the page that can contain visual components like regions, page items, or buttons.

Page Designer shows the named slots where you can add user interface elements, and helps you understand their relative positioning. Use the **Hide Empty Slots** context menu option in the **Layout** panel to toggle whether slots display even when they contain no components.

Figure 3-17 Blank Page with Slots Based on Page Template



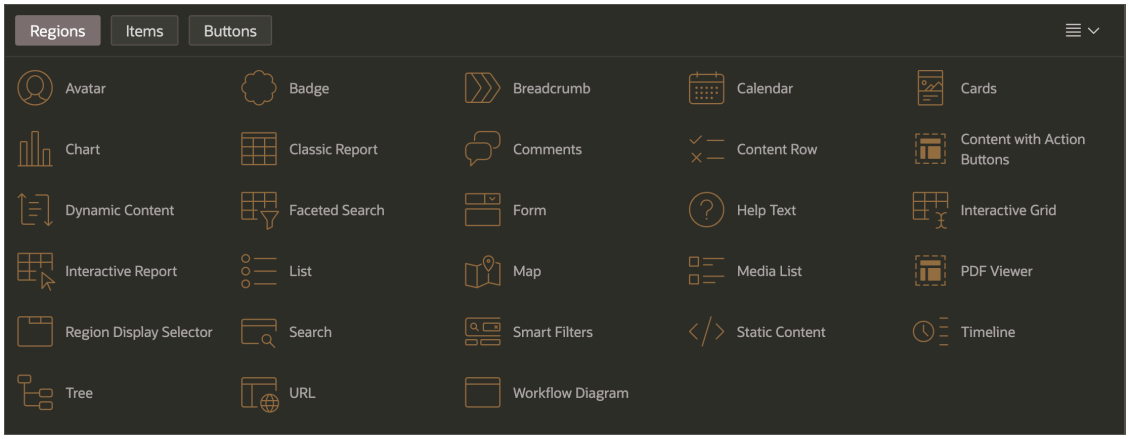
Copyright © 2026, Oracle and/or its affiliates.

3.3.3 Adding Regions in Page Designer

The APEX Page Designer's region palette includes all the region types you can drag and drop onto a slot in your page.

After dropping a region type into a slot, use the Property Editor to assign it a meaningful name and to set other properties that influence its look and behavior. If the region defaults to showing built-in sample data, change its **Location** property to another option like **Local Database** to use your own application data. For more information on each type of region, see Supported Region Types in *Oracle APEX App Builder User's Guide*.

Figure 3-18 Region Palette in Page Designer



Whether you jump start the process with the Create Page wizard or begin with a blank page, you add new regions to your page in the same way. Just drag one from the region palette and drop into a slot on the page.

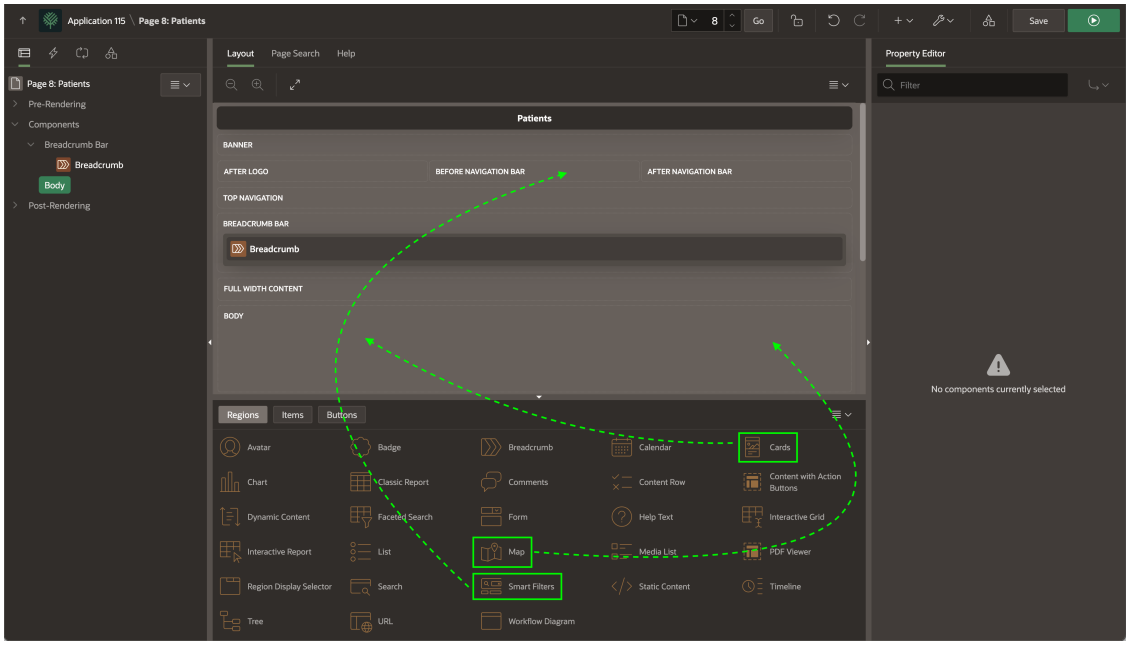
Alternatively, you can create a region using the **Create Region** option on the context menu in the Rendering tree, then set the region type and the desired slot name in the Property Editor.

For example, suppose you need to create a page to let staff of a medical clinic search for patients. If you start with a blank page, you could begin by dragging a **Cards** region to the **BODY** slot of the page, naming it *Search Results*, configuring its data source to be the patients table, and using the Property Editor to select the patient table's columns to show on each card.

Next, you might drag a **Smart Filters** search region, drop it into the **BEFORE NAVIGATION BAR** slot at the top of the page, name it *Search*, and set it to filter the cards region. Finally, to also show the filtered patient results on a map, you could drop a **Map** region into the **BODY** slot to the right of the cards region, name it *Map*, and configure its data source to use the filtered results of the cards region. For more details on this technique, see Showing Search Results in Other Regions.

Copyright © 2026, Oracle and/or its affiliates.

Figure 3-19 Dragging and Dropping Regions from the Palette into Page Slots

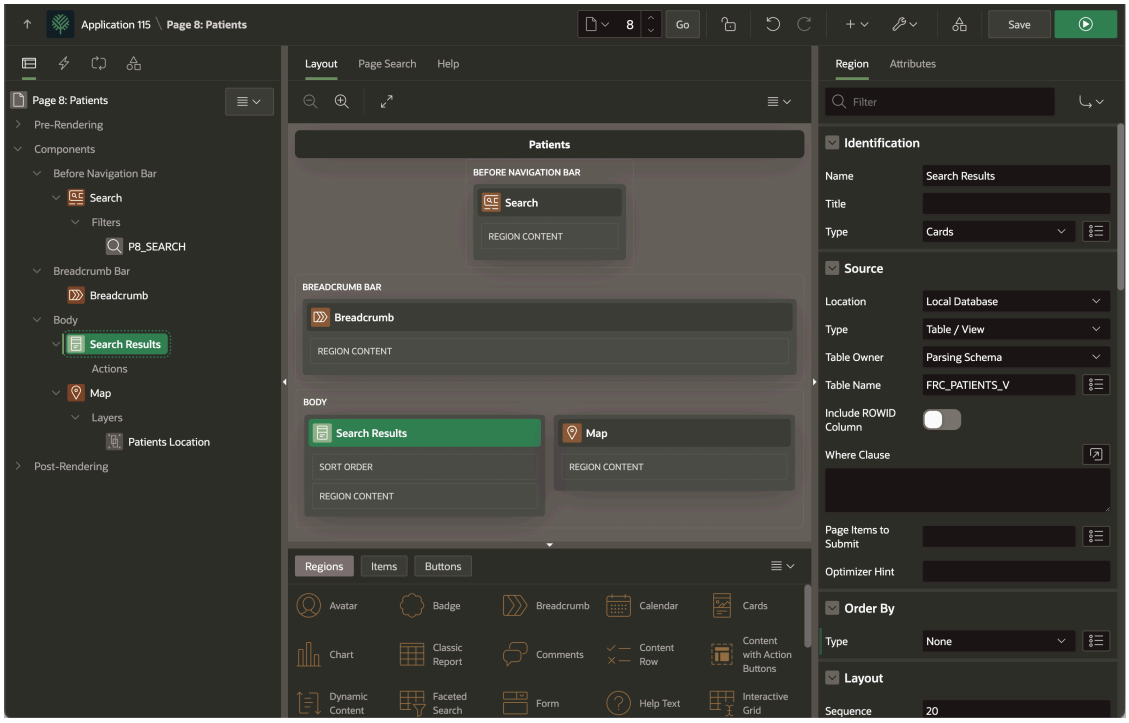


After performing these steps and choosing **Hide Empty Slots** from the layout context menu, your patients search page looks like the figure below.

The Page Designer panel on the left shows your page's components in a tree. The layout panel in the center is a visual representation of how regions and items are positioned relative to each other on the page. The Search region is at the top in the navigation bar, the breadcrumb in its appropriate bar, and the Search Results Cards and Map regions are side by side below that in the body of the page.

Copyright © 2026, Oracle and/or its affiliates.

Figure 3-20 Search Page for Patients in APEX Page Designer

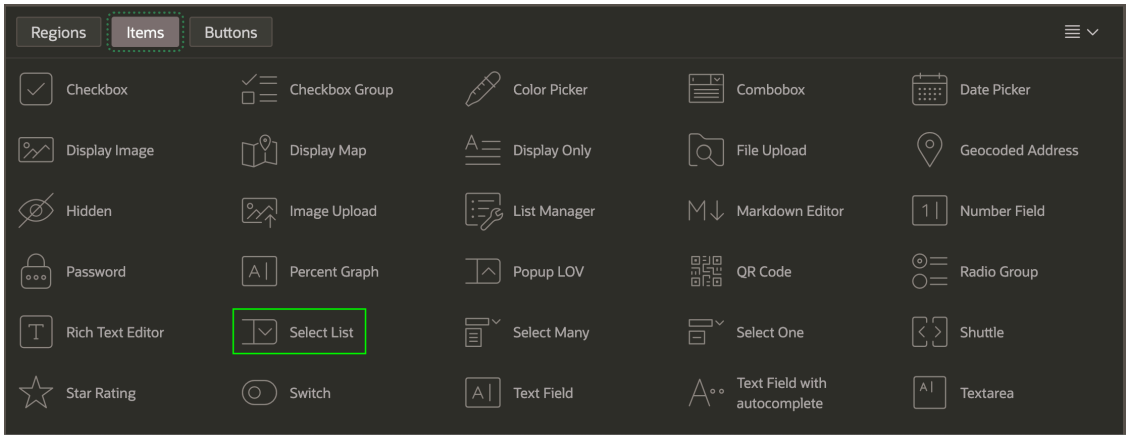


3.3.4 Adding Items in Page Designer

You can use the **Page Items** palette in Page Designer to add UI controls to the page.

Suppose we continue with the patients search page example from the previous section. After adding regions to the page, you might fine-tune the *Search Results* cards region to let the user choose a sorting order. A **Select List** item is a popular control to use for that purpose. As shown below, the page items palette in Page Designer contains all available UI controls, including a Select List. For more details on each page item type, see About Item Types in *Oracle APEX App Builder User's Guide*.

Figure 3-21 Page Items Palette in Page Designer

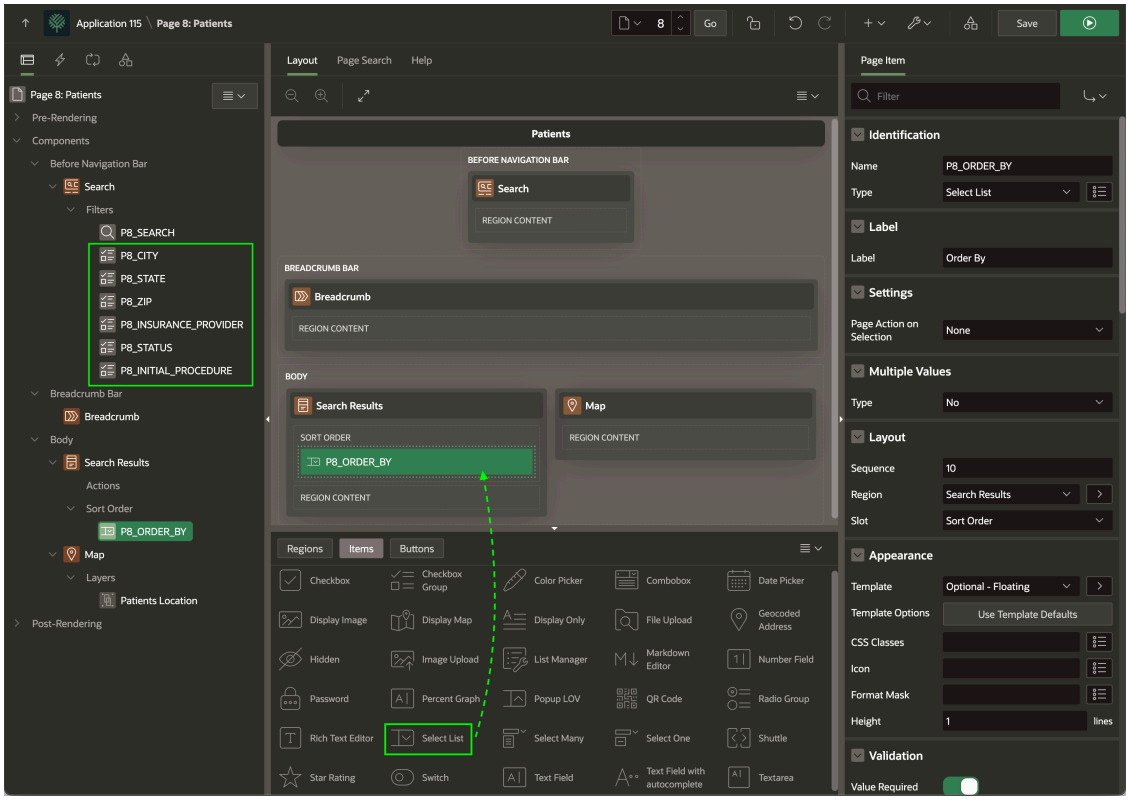


Copyright © 2026, Oracle and/or its affiliates.

To add a Select List to the Search Results region's **SORT ORDER** slot, you would drag and drop it as shown below, then use the Property Editor to name it `P8_ORDER_BY`, and configure its list of choices as static values. Then, you would select the Search Results region and configure the new Select List as its Order By item to specify the corresponding `ORDER BY` clauses. To learn more about user-driven sorting, see Managing Sorting with Order By Item in *Oracle APEX App Builder User's Guide*.

When added in Page Designer, the Smart Filters region starts with a single text search filter, but you can add other filters using the components tree. You do this by selecting the **Filters** node and choosing **Create Filter** from the context menu. Each additional filter defines a category of distinct values that users can select to interactively narrow results and find the patients they need. Notice below that filters have been added for City, State, Postal (Zip) Code, Insurance Provider, Status, and Initial Procedure. To learn more about search filters, see Search Filter Types in *Oracle APEX App Builder User's Guide*.

Figure 3-22 Adding a Sort By Item and Search Facets in APEX Page Designer



3.4 Running and Iteratively Editing Pages

At any time, you can run your page in the browser to use it like an end-user does.

When you run the page, it opens in a second browser tab, just next to the App Builder tab. This lets you see exactly how the page looks on any device or screen size. As shown below, the Smart Filters region in the example *Patients* page from the previous sections displays as a search field in the navigation bar at the top of the page.

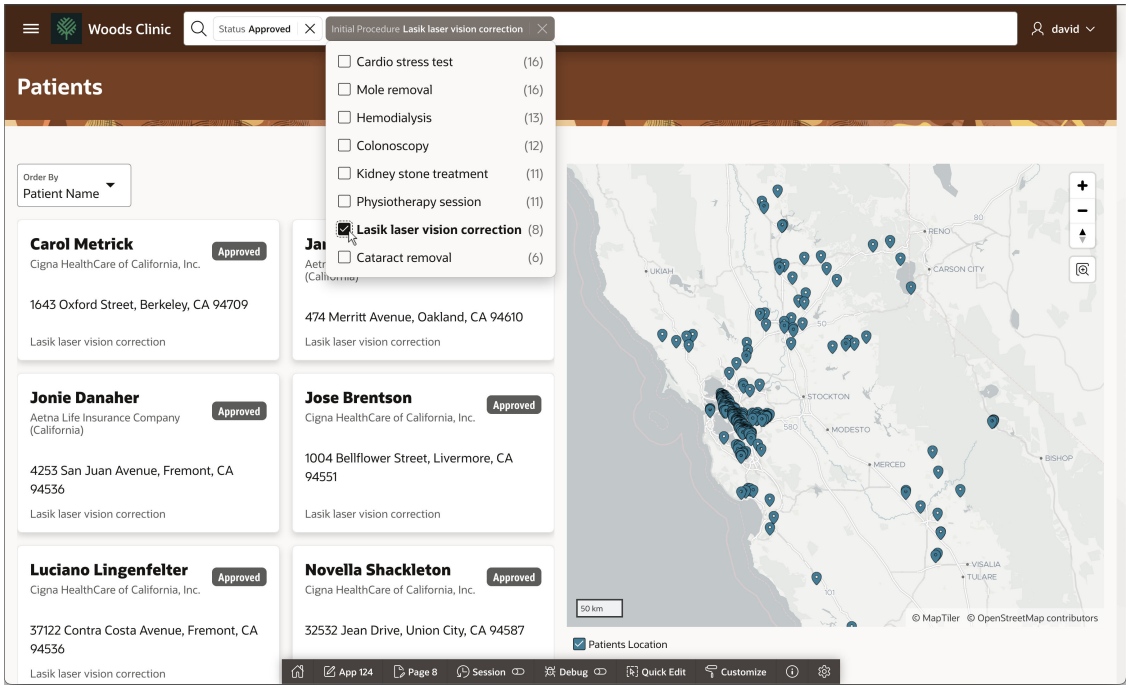
You could try selecting a value of *Approved* from the **Status** filter and *Lasik laser vision correction* from the **Initial Procedure** filter. You'd immediately notice the search criteria appear as "chips" in the search field, and the Search Results cards region now shows only the patients

Copyright © 2026, Oracle and/or its affiliates.

whose Lasik Laser Vision Correction medical procedure request has been approved. However, you notice the map region continues to show pins for *all* patients, instead of only the eight (8) patients that match the search filter criteria.

You can quickly get the map's filtered display working by configuring interactivity in the page: when the cards region is refreshed, the map region should also get refreshed. You see how that's done in the next section.

Figure 3-23 Running Patients Search Page to See Map Not Refreshing Yet



Notice the developer toolbar APEX adds along the bottom of the page when you run pages from App Builder. To iterate quickly, simply click the **(Page 8)** button in that toolbar to open it in Page Designer. To edit a particular *element* of the page, click the **(Quick Edit)** button there, then click on the region or item in the page you want to edit. In either case, you immediately jump to the Page Designer in the right context to carry out your next page configuration change.

3.5 Adding Interactivity to Pages

Add interactive behavior to your pages using Dynamic Actions.

Define one or more dynamic actions to run in response to an event that happens while a user interacts with the page. Common events include the change in value of a page item or selection of a list, the closing of a modal dialog, refreshing of a region, and many others. Button and menu clicks can use Triggered Actions. Handle all other events by defining a Dynamic Action event handler. APEX offers a large set of native dynamic actions that you can extend by using dynamic action plug-ins.

Some dynamic actions modify aspects of the page in the browser like showing or hiding items and enabling or disabling them. Other ones engage the APEX server to refresh a region's data or execute server-side code. Any actions involving a server request support declaratively sending the *latest* values of page items or report columns to the server. Just set the **Items to Submit** property to a comma-separated list of item or column names. While a report region's

Copyright © 2026, Oracle and/or its affiliates.

primary key column is always available to send, enable other columns' **Available on Client** setting as needed. Since the *Execute Server-side Code* dynamic action can also *modify* page item values, list any you want APEX to update in the page in the companion **Items to Return** property.

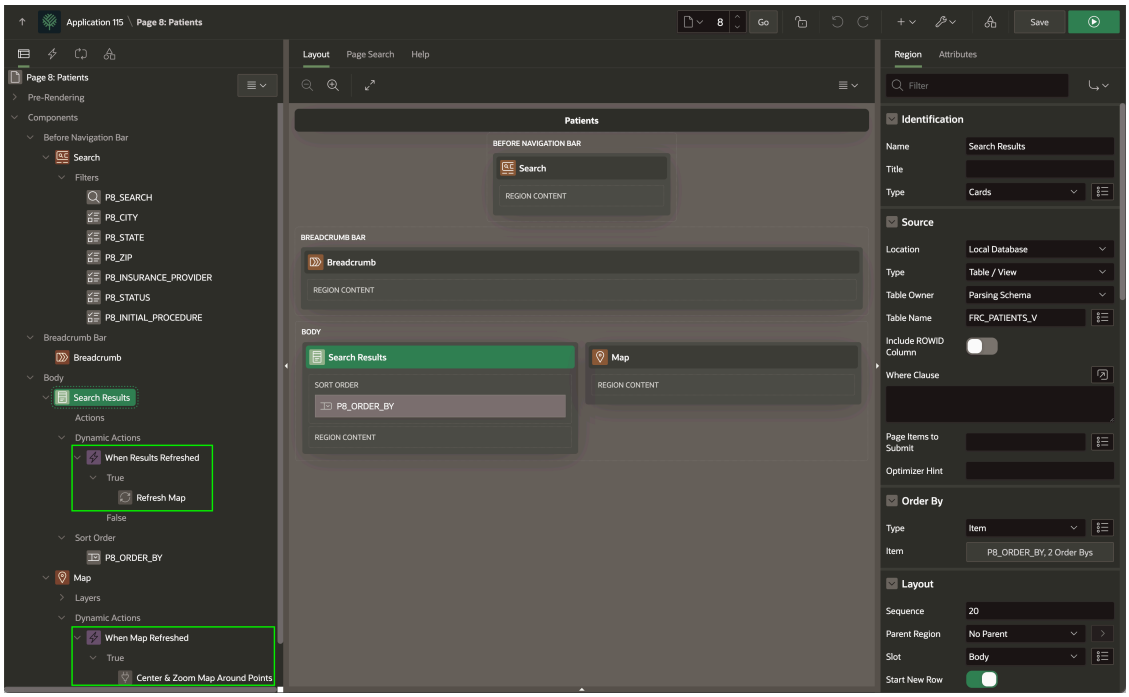
In the patients search page we're studying, recall that when the cards region is refreshed, the map region should also get refreshed. The event named **Page Change [Cards]** fires when a cards region is refreshed, so you can define a dynamic action related to this event to get the job done.

You'd start by selecting the **Search Results** region, and choosing **Create Dynamic Action** from the context menu. Then in the Property Editor you select the **Page Change [Cards]** event. You can give the dynamic action event handler a name like *When Results Refreshed* to make the intent more clear to colleagues who might have to enhance the page in the future. The set of built-in action types includes ones to hide or show elements, refresh elements, enable or disable elements, set or clear the value of items, execute custom code in the client or server, and many others. In this case, you need the one to refresh the Map region. So, you configure the type of the related action to be **Refresh** and set **Map** as the region to refresh. Those few clicks make the map automatically refresh whenever the cards region results change.

End users would definitely appreciate if the map would zoom and center itself around the filtered list of patients to display to save them having to do manual adjustments. Since a dynamic action is one of the kinds of **plug-in** components members of the APEX community can create and distribute, imagine you find a **Center & Zoom Map Around Points** dynamic action plug-in on `apex.world` or as part of a blog article or tutorial.

You can download and import it into your APEX app to put it to use. You select the **Map** region in the component tree and again choose **Create Dynamic Action**. You name this second event handler *When Map Refreshed* and set it to react to the **After Refresh** event on the Map region. Finally, you define the dynamic action to be of type **Center & Zoom Map Around Points [Plug-in]**. The figure below shows the result in Page Designer.

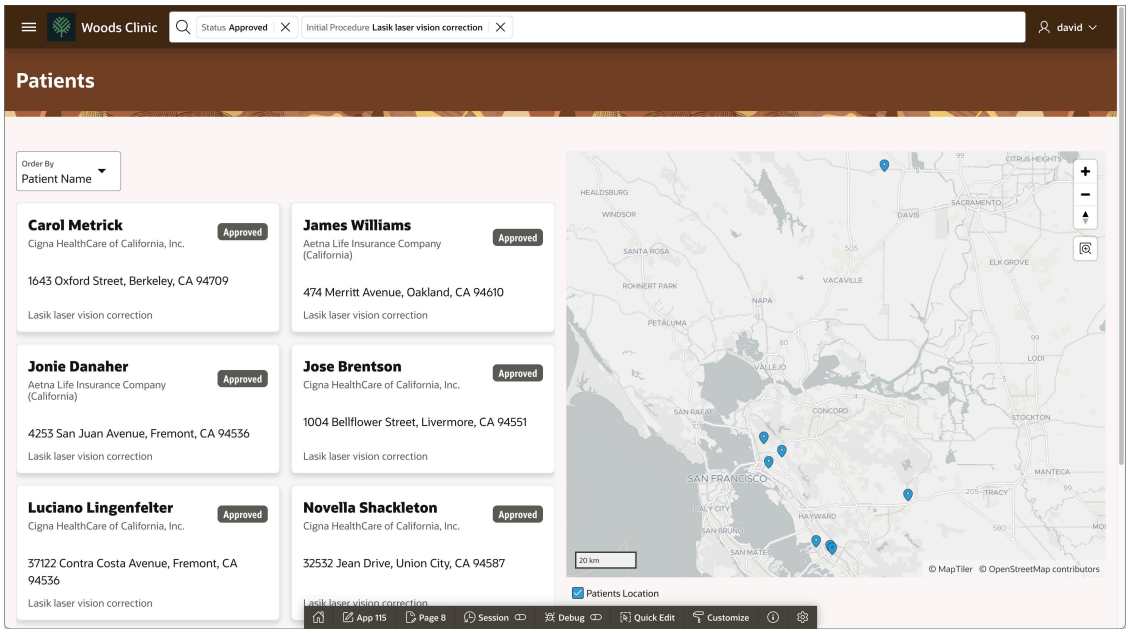
Figure 3-24 Dynamic Actions Add Interactive Behavior



Copyright © 2026, Oracle and/or its affiliates.

Now at runtime, as shown below, as the end-user fine-tunes their patient search both the cards and map regions reflect the narrowed results, and the map adjusts to fit the remaining patients' locations into the view.

Figure 3-25 Map Reflects Filtered Result and Auto-Centers and Zooms



Related Topics

- Managing Dynamic Actions in *Oracle APEX App Builder User's Guide*

3.6 Universal Theme Design System

Applications you build with Oracle APEX use a built-in design system called **Universal Theme**.

Technically speaking, Universal Theme consists of HTML templates, Cascading Style Sheets rules, icons, images, and JavaScript libraries. However, APEX shields you from having to understand these technologies. It's only important to remember that these resources cooperate to automatically ensure a consistent, professional look to all the pages you create on any desktop or mobile screen size.

- Theme Styles
  Universal Theme comes with several pre-defined styles.
- Universal Theme Reference App
  Bookmark the *Universal Theme* reference app at oracleapex.com/ut to quickly consult a live reference for many aspects of APEX user interface development.
- The Role HTML and CSS Play to Format Data
  While *Universal Theme* lets you deliver great results without being an expert in web technologies, understanding the role HTML and CSS play is useful when customizing data formatting.

Related Topics

- Understanding the Universal Theme

Copyright © 2026, Oracle and/or its affiliates.

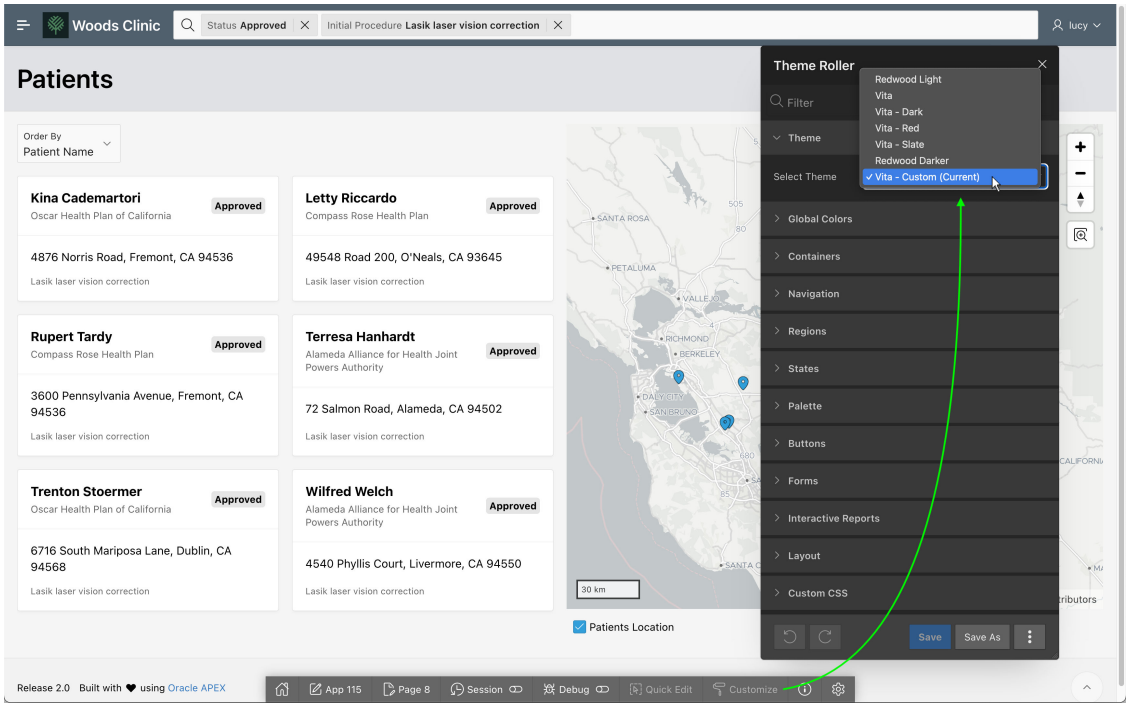
3.6.1 Theme Styles

Universal Theme comes with several pre-defined styles.

The **Redwood Light** style matches Fusion Applications using colors inspired by nature and is more visually spacious. The **Vita** style family variants use a more compact layout and other solid colors. The *Iris* style blends popular aspects of both. All are designed with accessibility in mind.

As shown below, the **Theme Roller** editor in the developer toolbar lets you switch theme styles to instantly see how your page looks in any style. It also lets you easily customize the colors of many theme elements and save the changes with a custom theme style name.

Figure 3-26 Customizing the Theme Style with Theme Roller



Related Topics

- Using Theme Roller in *Oracle APEX App Builder User's Guide*
- Developing Accessible Apps in *Oracle APEX Accessibility Guide*

3.6.2 Universal Theme Reference App

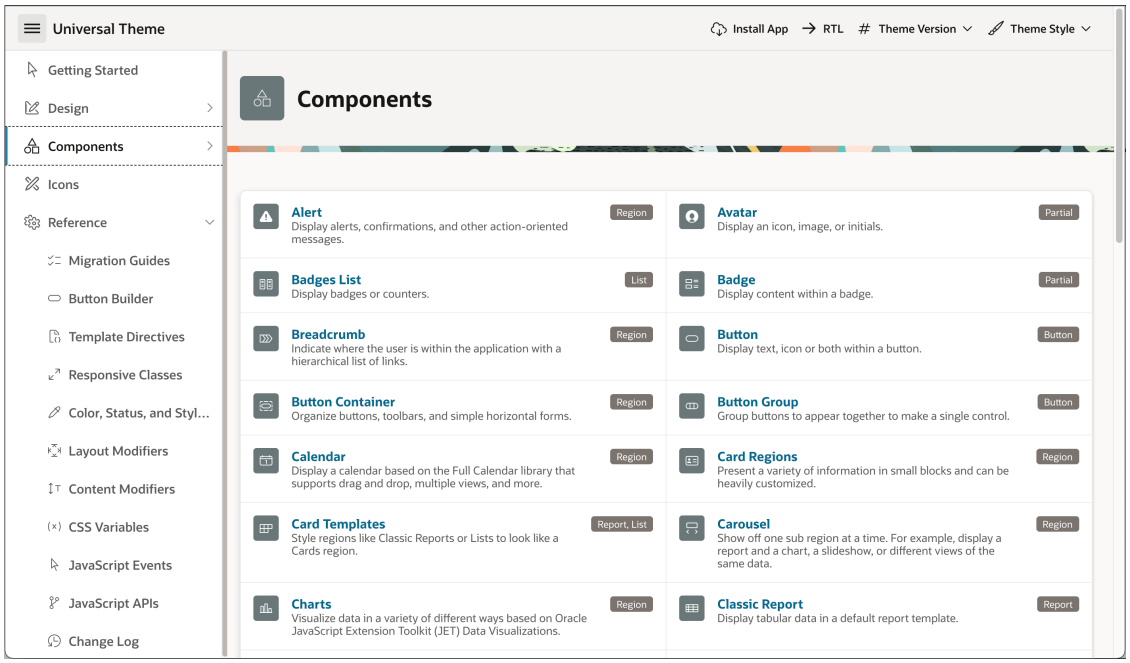
Bookmark the *Universal Theme* reference app at oracleapex.com/ut to quickly consult a live reference for many aspects of APEX user interface development.

The **Components** section shown below lists all the native APEX visual components you can use in your apps. Click on any component to see a live example along with steps to add the component to your page. In the **Icons** section, you can browse or search the library of predefined icons you can use on various UI elements to clarify their functionality for end-users.

Copyright © 2026, Oracle and/or its affiliates.

Each subsection under the **Reference** heading documents an important facet of APEX UI development you will find useful. For example, there you can learn about pre-defined CSS style **variables** and **classes** useful for adjusting your page elements' look and feel in a declarative way, **template directives** syntax for easy conditional formatting, a handy button builder, and more.

Figure 3-27 Universal Theme Components Reference

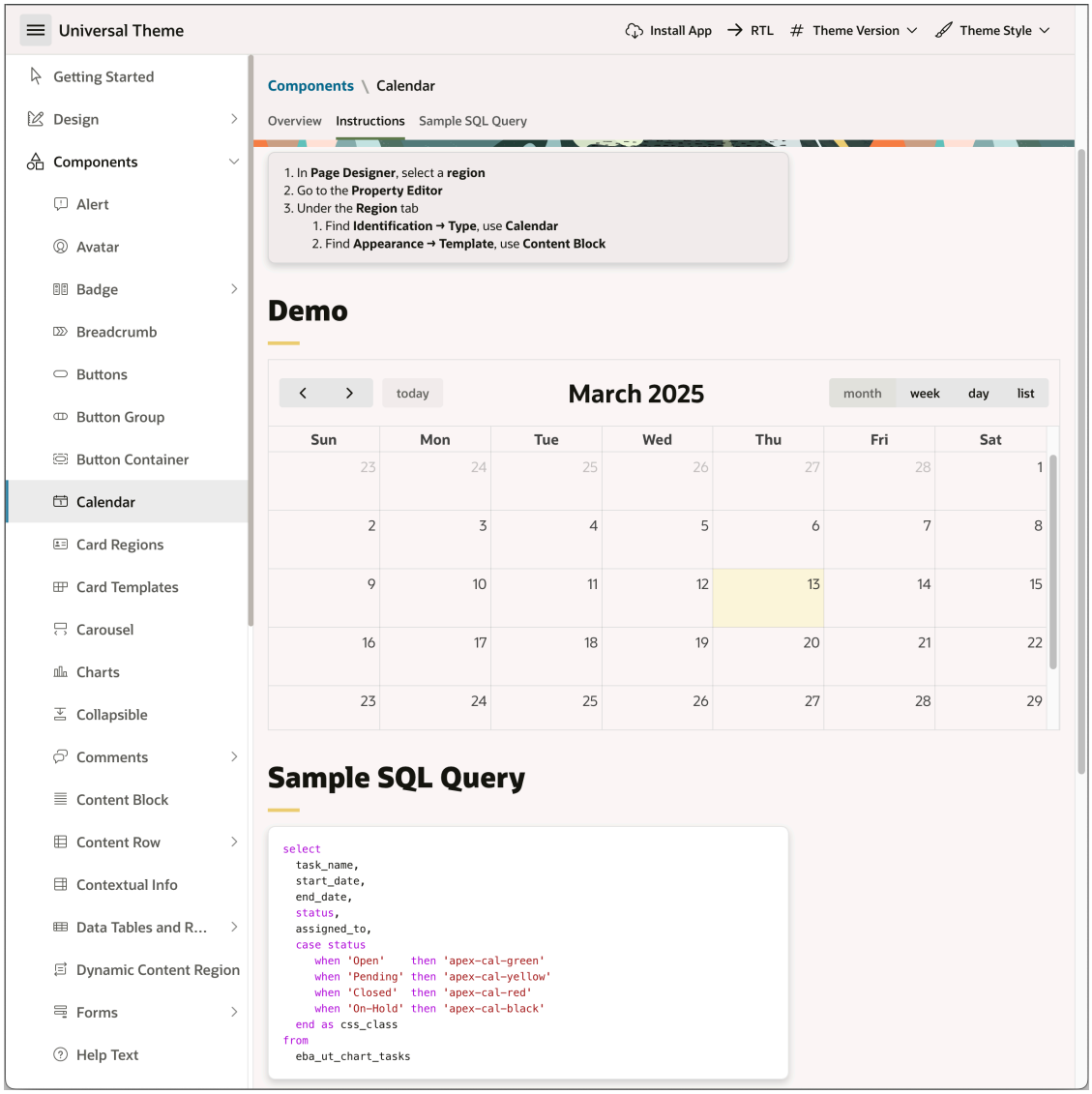


The figure below shows the reference for the **Calendar** region. Where relevant, you can also experiment interactively with different **template options** that control how the component looks and behaves. You can then set the same options in Page Designer after finding the combination that's best for your use case.

Notice it also suggests an example SQL `SELECT` statement to show the basic data the region requires. In the case of a calendar, you can see the data source needs a text value like `task_name` to give each calendar event a user-friendly label along with date (with time) values like `start_date` and `end_date` to define the event's start and duration. The query can also contain other columns whose data can appear in the event tooltip or in its customized event display text.

Copyright © 2026, Oracle and/or its affiliates.

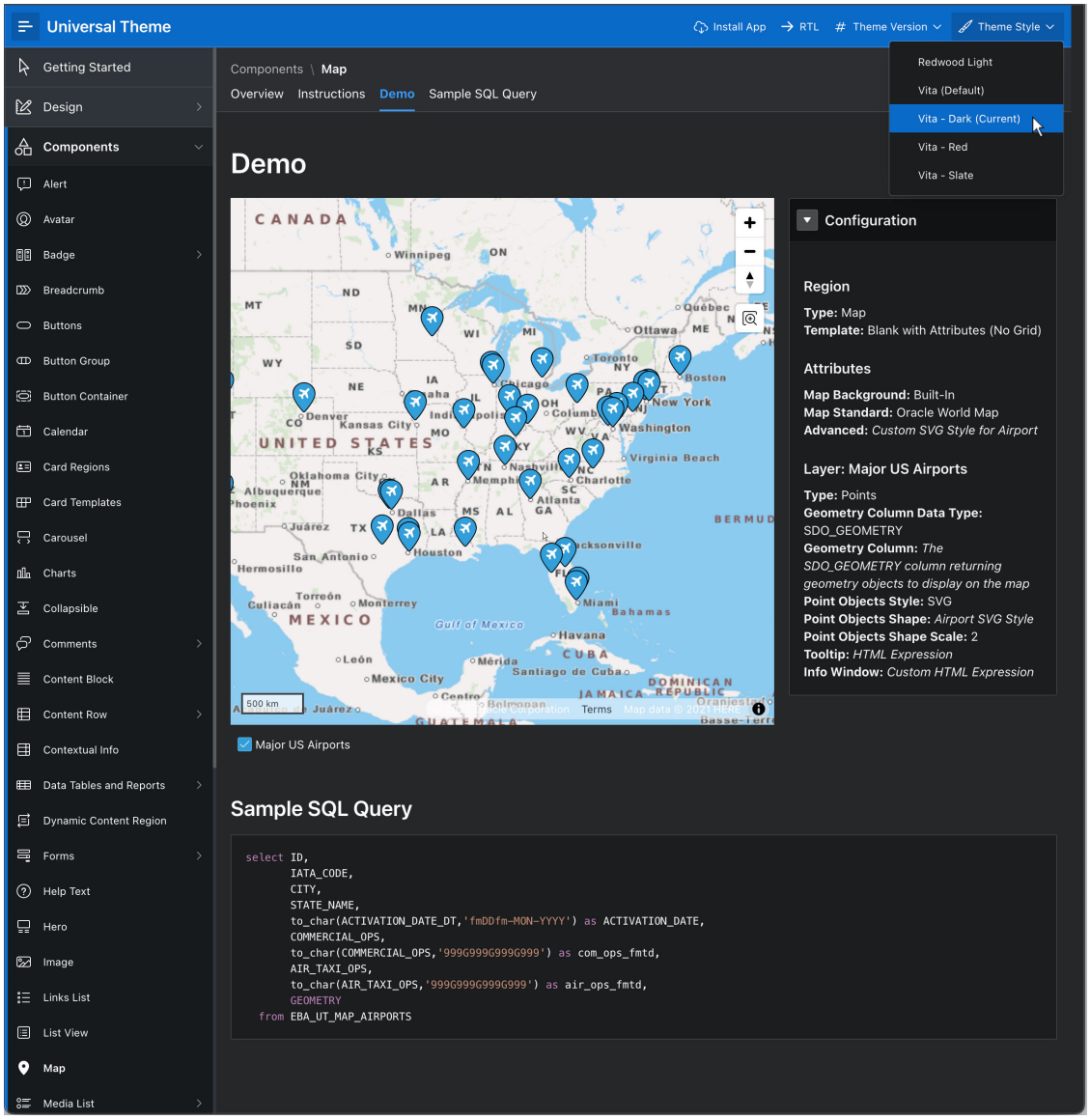
Figure 3-28 Calendar Reference



Using the reference app's navigation bar controls, you can toggle the writing direction and theme style. For example, the figure below shows the **Map** region reference page in the "Vita - Dark" theme style.

Copyright © 2026, Oracle and/or its affiliates.

Figure 3-29 Map Region Reference in "Vita Dark" Theme Style



3.6.3 The Role HTML and CSS Play to Format Data

While *Universal Theme* lets you deliver great results without being an expert in web technologies, understanding the role HTML and CSS play is useful when customizing data formatting.

Since APEX apps are web-based, they use standard tags to annotate or **mark up** data with details to help clearly present the information to users. The Hypertext Markup Language (HTML) uses **tags** like `<ul>` to indicate an unordered list of elements, `<li>` for each element in a list, `<strong>` to strongly emphasize some content, and many others. These tags introduce a logical presentation structure around the data to format. These tags can contain additional attributes like `class` whose value contains additional logical information about an element. For example, a tag like `<li class="is-selected">` might distinguish a list element the user has selected among other elements in the list. Tags and attributes work in concert with Cascading

Copyright © 2026, Oracle and/or its affiliates.

Style Sheets (CSS) rules that define how different structural elements on the page should appear.

Consider the simple example below showing the HTML markup for a list of four departments, with the *RESEARCH* department annotated as being selected.

```
<ul>
  <li><strong>ACCOUNTING</strong> → NEW YORK</li>
  <li><strong>OPERATIONS</strong> → BOSTON</li>
  <li class="is-selected"><strong>RESEARCH</strong> → DALLAS</li>
  <li><strong>SALES</strong> → CHICAGO</li>
</ul>
```

Your browser receives this markup to display along with CSS styling rules like the following. These two simple rules declaratively specify the background and foreground color of a selected list item and the relative font size of list item content with a strong emphasis.

```
/* Color selected list items differently */
li.is-selected {
    background-color: lightblue;
    color: midnightblue;
}
/* Increase font size of strong elements in list items by 20% */
li strong {
    font-size: 120%;
}
```

The combination of this HTML markup and CSS style rule produces a result like you see below. It shows the four department names in a bold, larger font in a bulleted list. The selected RESEARCH department appears with a light blue background.

Figure 3-30 Browser Display of Simple HTML and CSS Example



3.7 Report Columns and Form Items

Most region types have an associated data source with a set of columns APEX infers automatically.

Page Designer makes working with the columns of report regions and grids easy, as well as with page items in a form region. This is true even when new use cases require updating your application data model.

Copyright © 2026, Oracle and/or its affiliates.

- Configuring Report Region Columns
  If a region lets you configure settings on its data source columns, then they appear in Page Designer under a heading in the component tree.
- Synchronizing Column Changes
  If your region's data source needs to work with additional columns, or different set of them, your page can automatically adapt.
- Deleting or Commenting Out Columns
  Often, your table includes more columns than you need to show in a particular region. In that situation, you have two options: you can delete the column from the region's column list, or you can comment it out.
- Page Items in a Form Region
  When your page includes a **Form** region, the Page Designer detects and displays a page item for each column in the data source in the **Region Body** slot.
- Flexible Form Page Item Layout
  While report regions control the layout of their columns, form regions have a flexible layout.

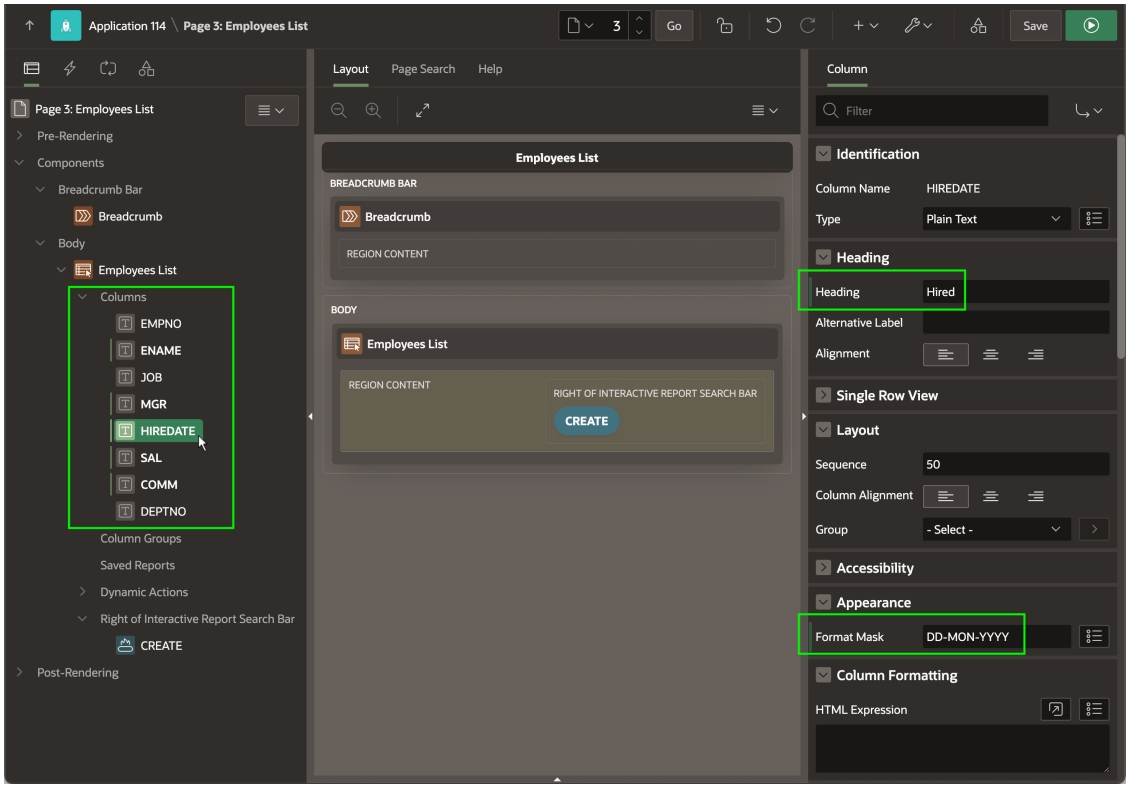
3.7.1 Configuring Report Region Columns

If a region lets you configure settings on its data source columns, then they appear in Page Designer under a heading in the component tree.

The figure below shows an **Employees List** region in Page Designer with its columns highlighted. Selecting a column like `HIREDATE` in the tree, the Property Editor lets you see and adjust its appearance and behavior. If you change properties like **Heading** and **Format Mask**, Page Designer indicates these unsaved changes with a vertical bar to the left of the property name as called out below. The next time you save the page, these change indicators disappear.

Copyright © 2026, Oracle and/or its affiliates.

Figure 3-31 Columns of an Interactive Report Region in Page Designer



3.7.2 Synchronizing Column Changes

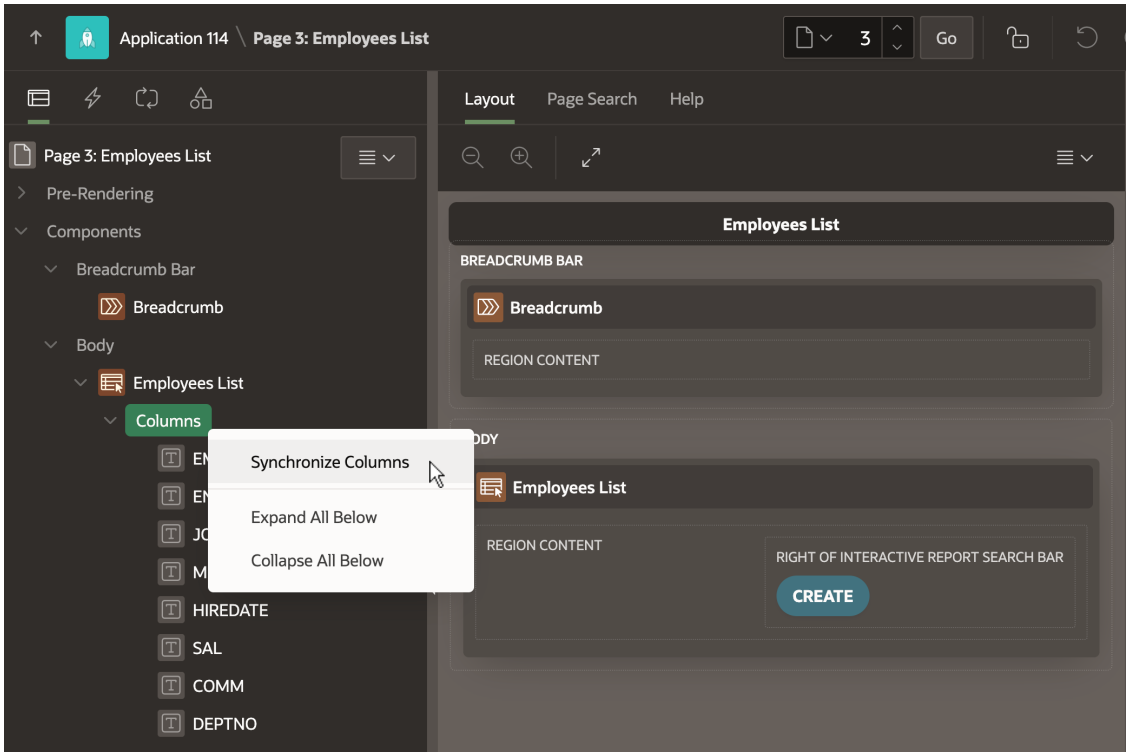
If your region's data source needs to work with additional columns, or different set of them, your page can automatically adapt.

If a region is based on a SQL query, then whenever you edit the `SELECT` statement Page Designer adds any new columns it detects to the list. However, if the region is based on a table or other data source, *you* decide when this synchronization occurs.

While developing, imagine you use **Object Browser** to add a new column to the `EMP` table that the **Employees List** region uses as its data source. To include this new column in your page, just select **Synchronize Columns** on the context menu as shown below. Page Designer adds any new columns to the end of the list. You can use drag and drop to rearrange the column list order as needed.

Copyright © 2026, Oracle and/or its affiliates.

Figure 3-32 Synchronizing Columns After Altering a Region's Data Source



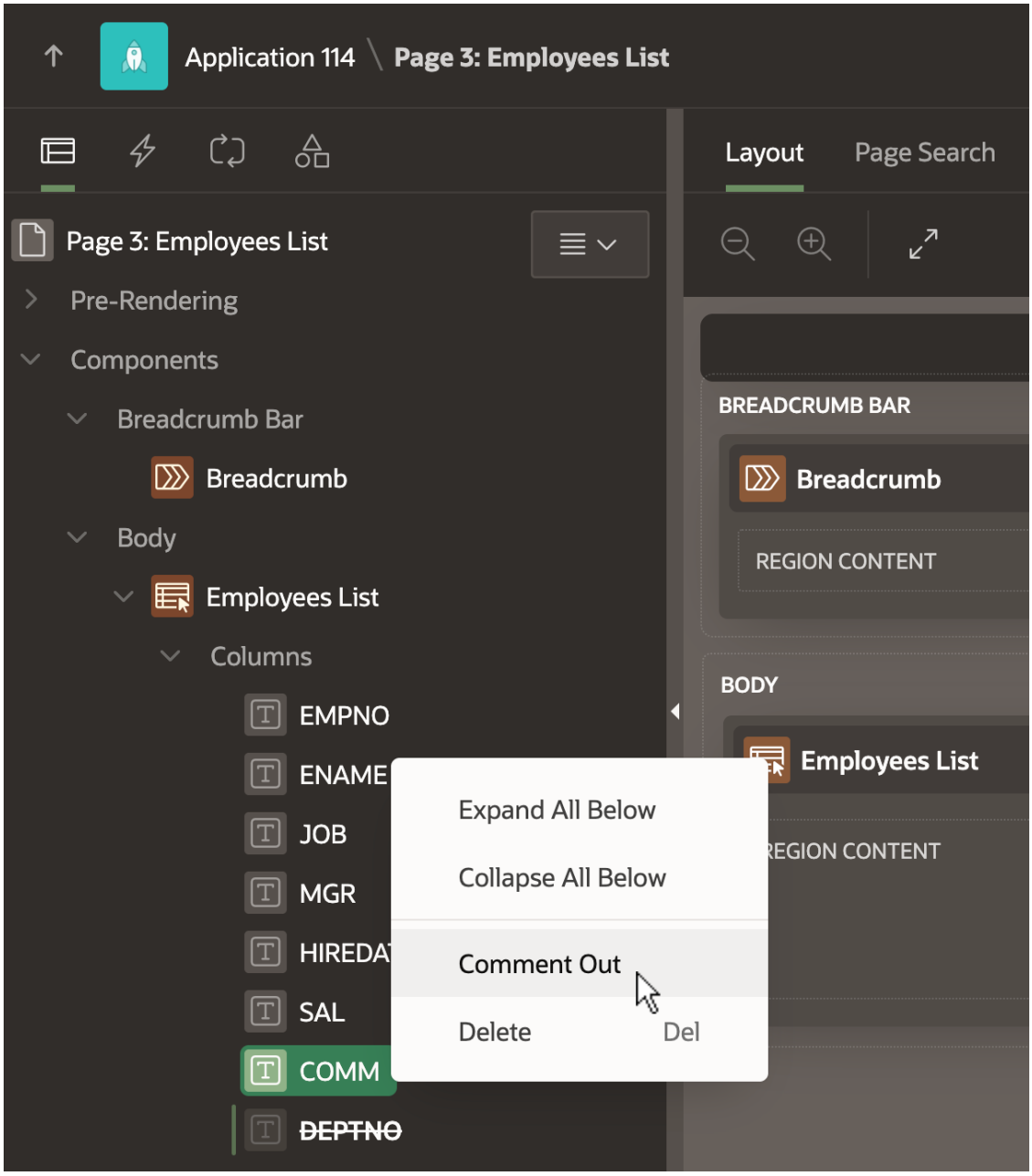
3.7.3 Deleting or Commenting Out Columns

Often, your table includes more columns than you need to show in a particular region. In that situation, you have two options: you can delete the column from the region's column list, or you can comment it out.

The figure shows the columns of an Employees List region in Page Designer. Commented out columns display with a line through their name like the `DEPTNO` column below. To comment out a column, select **Comment Out** from its context menu.

Copyright © 2026, Oracle and/or its affiliates.

Figure 3-33 Commenting Out a Report Region Column



Commenting out a column is useful while a page and its data source are still in active development, where you anticipate evolving the table further and repeating **Synchronize Columns** multiple times. If a column is commented out, it stays that way during synchronization. In contrast, if you had deleted the column from the region, then a subsequent **Synchronize Columns** action brings it back along with any newly added columns. Both techniques can be useful in the right situation. An **Uncomment** context menu option appears on any commented-out column that you can select to easily include the column in the region again.

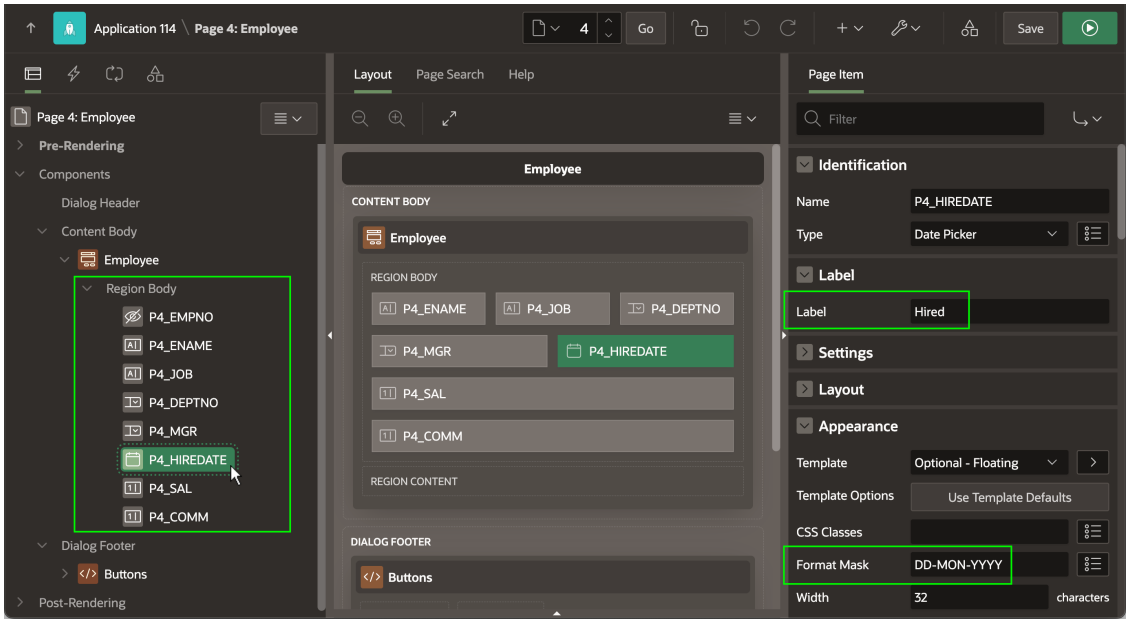
Copyright © 2026, Oracle and/or its affiliates.

3.7.4 Page Items in a Form Region

When your page includes a **Form** region, the Page Designer detects and displays a page item for each column in the data source in the **Region Body** slot.

By convention, page item names use the prefix of `P` followed by the page number. As shown below, after selecting an item like `P4_HIREDATE` in the tree the Property Editor shows its settings like **Label** and **Format Mask**. When you adjust your application data model, the form region works similarly to report regions. If the data source is a SQL query then Page Designer automatically adds page items for any new `SELECT` list columns in the query. If instead the form region is based on a table or other data source, then you select **Synchronize Page Items** on the context menu to include new columns when needed. Form region page items support the same **Comment Out**, **Uncomment**, and **Delete** options as report regions do.

Figure 3-34 Form Region Page Items



3.7.5 Flexible Form Page Item Layout

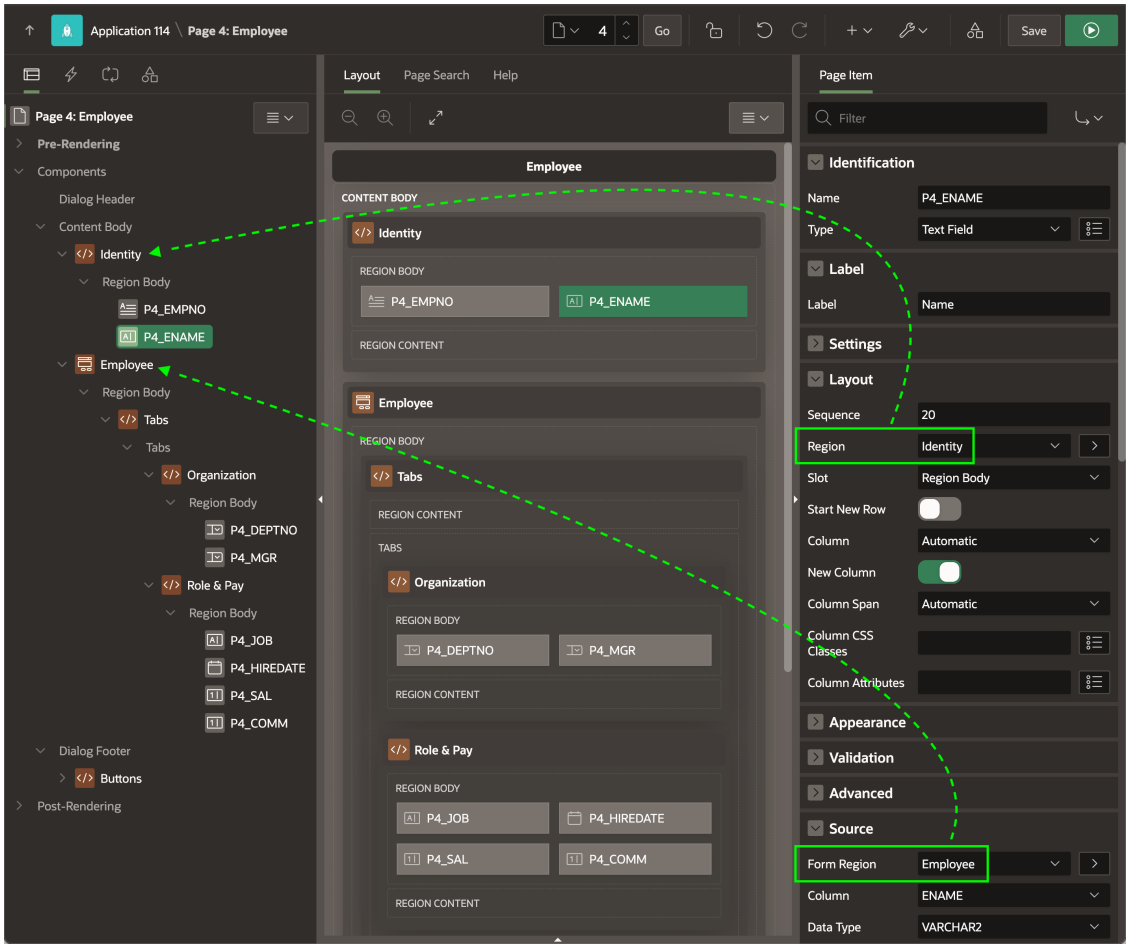
While report regions control the layout of their columns, form regions have a flexible layout.

Page Designer defaults form page items to reside in the form region's **Region Body** slot, but you can move them anywhere on the page. As shown below, a page item's **Form Region** property in the **Source** section of the Property Editor indicates what source region it belongs to. When a form region queries a row by primary key or saves modified data, it involves the page items that name it as a source. This lets you set the **Region** property in the **Layout** section to another region on the page as needed.

In this example, two of the items with the Employee form region as their source are in a Static Content region named **Identity** at the top of the page. The other items sourced from this form region are in an **Organization** tab or a **Role & Pay** tab using a **Tabs** Static Content region set to use the **Tabs Container** template.

Copyright © 2026, Oracle and/or its affiliates.

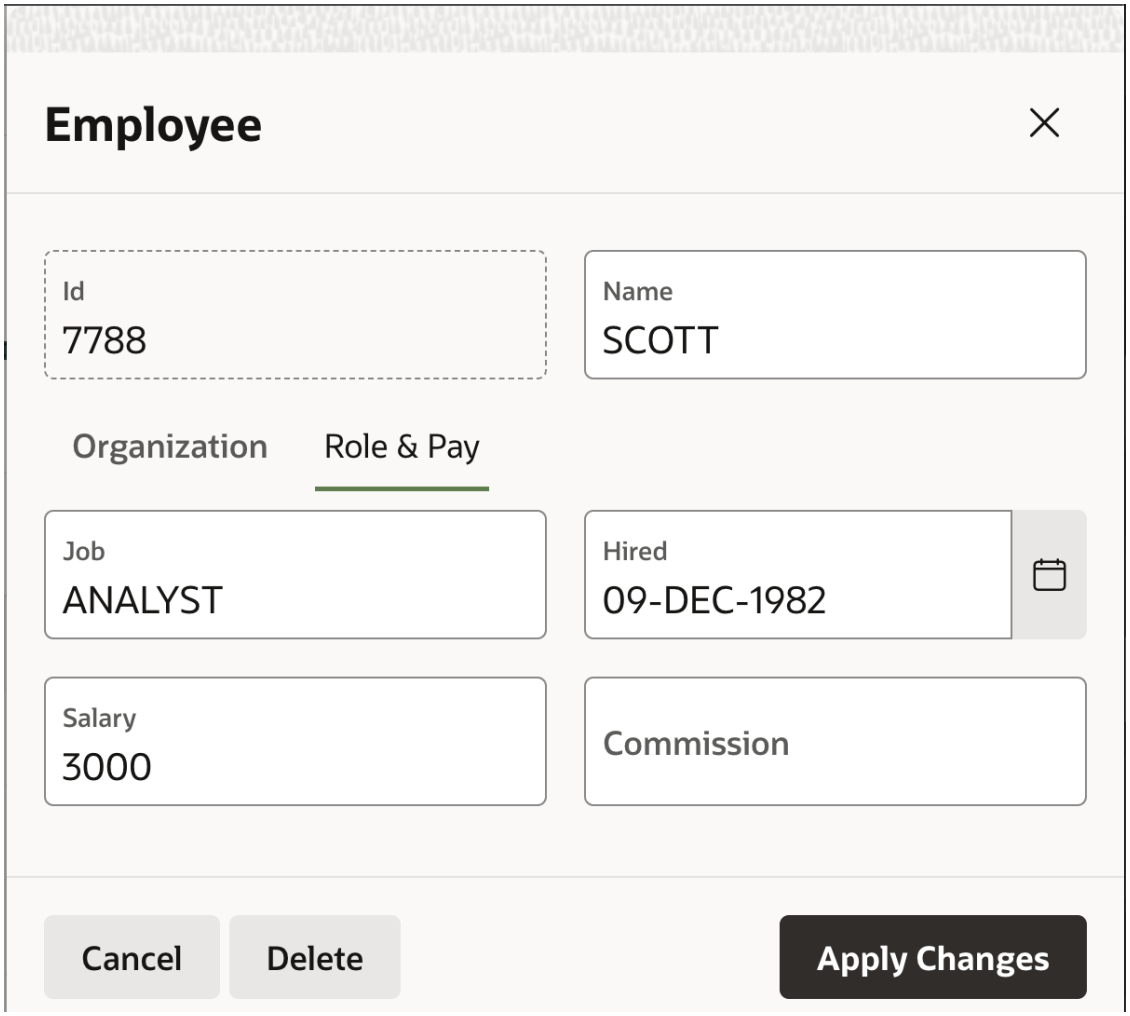
Figure 3-35 Form Region Source and Region Layout Properties



At runtime, the page above looks like the figure below. It shows the employee **Id** and **Name** fields side by side, above two tabs labeled **Organization** and **Role & Pay**. The latter tab is selected and contains **Job**, **Hired**, **Salary**, and **Commission** fields.

Copyright © 2026, Oracle and/or its affiliates.

Figure 3-36 Flexible Layout of Form Region Page Items at Runtime



3.8 Cascading, List-Driven Items

Apps often need end-users to pick one or more choices from a list of valid options, and sometimes the available items depend on a previous selection the user makes.

For example, after picking a particular department the user might need to select an employee in that department. Oracle APEX makes list-driven page items easy to configure. It offers many different ways to visualize and select options, and simplifies defining cascading lists.

- List-Driven Items Available
  Your pages can use ten different kinds of list-driven items.
- Defining the List of Values
  When you create a list-driven page item, you define its list of valid choices either locally on the page item itself or by referencing a shared component List of Values by name.
- Configuring Cascading Lists
  When you need one page item's value to influence the list of choices in another list-driven item, you can easily configure this **cascading** effect.

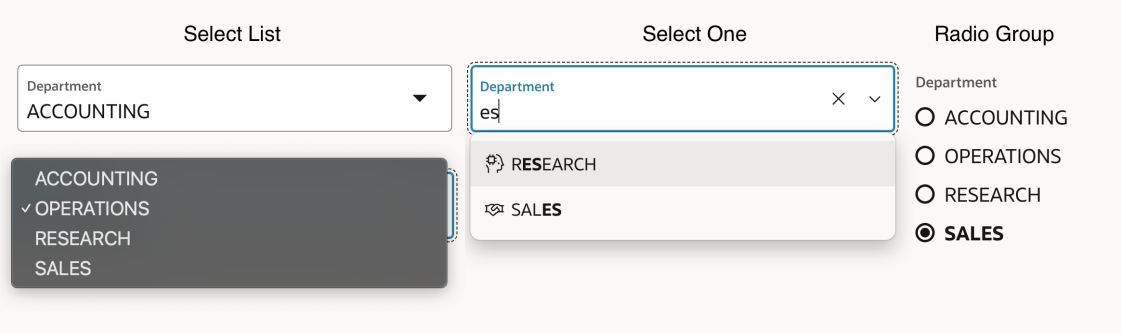
Copyright © 2026, Oracle and/or its affiliates.

3.8.1 List-Driven Items Available

Your pages can use ten different kinds of list-driven items.

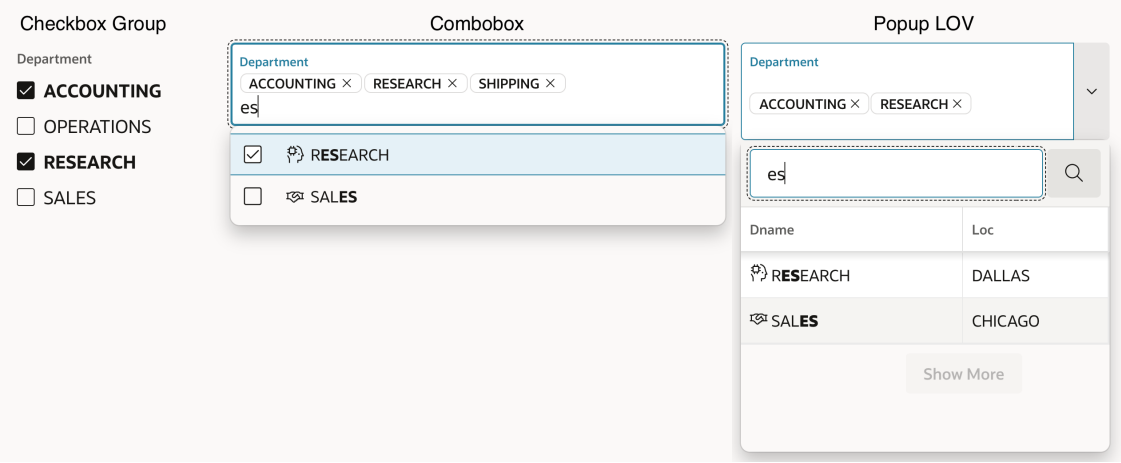
The **Select List**, **Select One**, and **Radio Group** items all support a single selection. The **Select One** item has a type-to-search capability, and lets you use icons and have full control over how list items appears in the dropdown. The figure below shows examples of Select List, Select One, and Radio Group page items displaying choices from the `DEPT` table. Notice how the user has typed `es` into the Select One and its list has narrowed to only show choices containing those letters. The Select One item also features icons in the list.

Figure 3-37 Single Selection List Items



The **Checkbox Group, Combobox**, and **Popup LOV** items all support either single or multiple selections. The Combobox item **also** lets a user type in a manual entry like "*SHIPPING*" below that's not in the list and supports icons and styling control on how list items appear. The Popup LOV can display additional columns and search across them. When used in single-selection mode, it can optionally return values of the additional list columns into other page items or grid columns in the page. Examples of the Checkbox Group, Combobox, and Popup LOV items appear in the figure below. Notice the Combobox and Popup LOV display selections as "chips", the user has typed `es` into to filter both's list options, and the icons that they support.

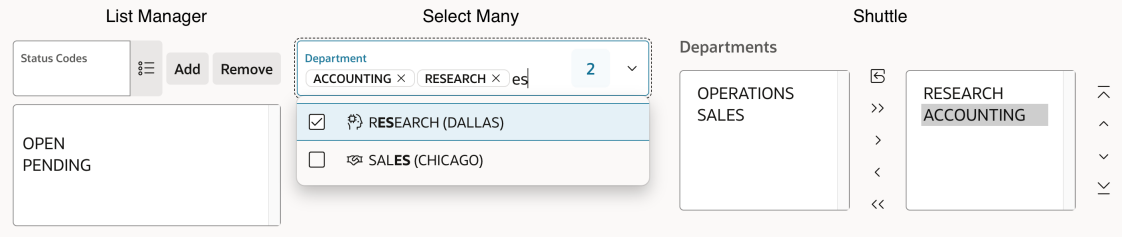
Figure 3-38 Single- or Multi-Selection List Items



Copyright © 2026, Oracle and/or its affiliates.

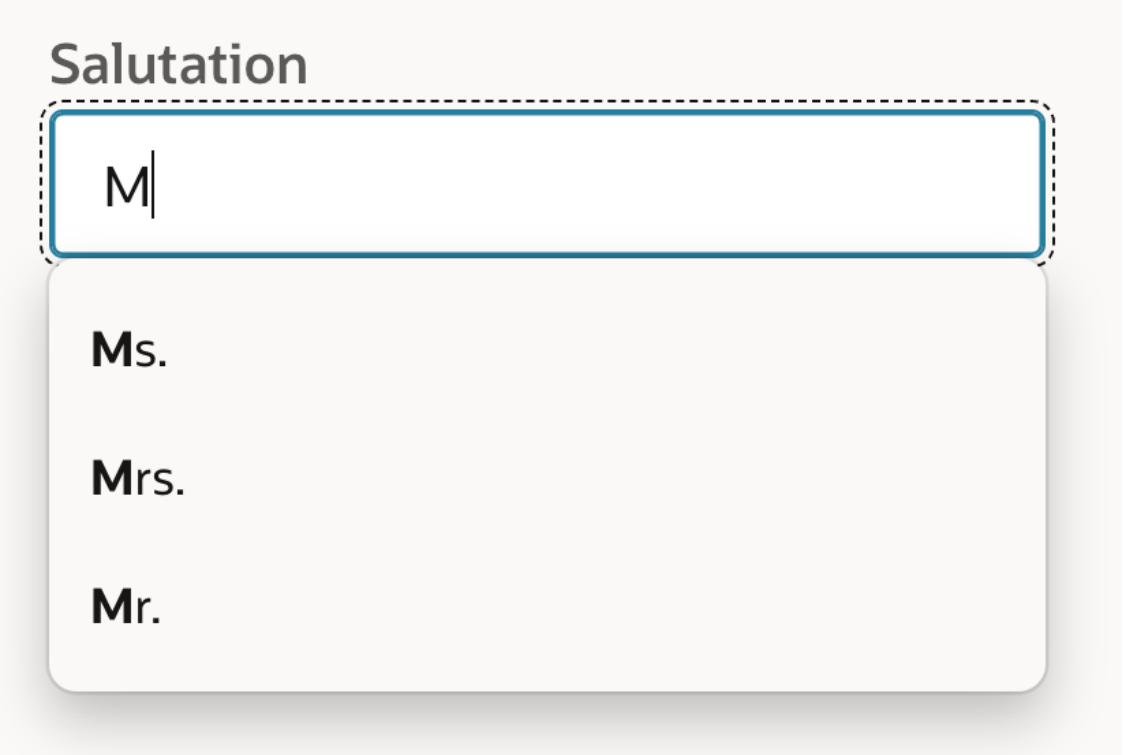
The **Select Many**, **List Manager**, and **Shuttle** items offer multiple selection only. The **List Manager** provides a search popup that combined with the **Add** button is used to add selected options to a list. The **Select Many** supports type-to-filter and gives styling control on how list items appear. The **Shuttle** has configurable controls for moving and reordering the selections. This figure shows examples of List Manager, Select Manager, and Shuttle in action.

Figure 3-39 Multi-Selection List Items



The **Text Field with Autocomplete** supports free form text entry, with autocomplete from a list. The user can type to search and pick from the autocomplete list of pre-defined entries, or type in any other value that's appropriate. Similar to the **List Manager**, its related list of options only uses the **Return Item**. As shown below, the user has typed the letter `M` in a **Salutation** field, and the choices **Ms.**, **Mrs.**, and **Mr.** appear.

Figure 3-40 Text with Autocomplete Page Item

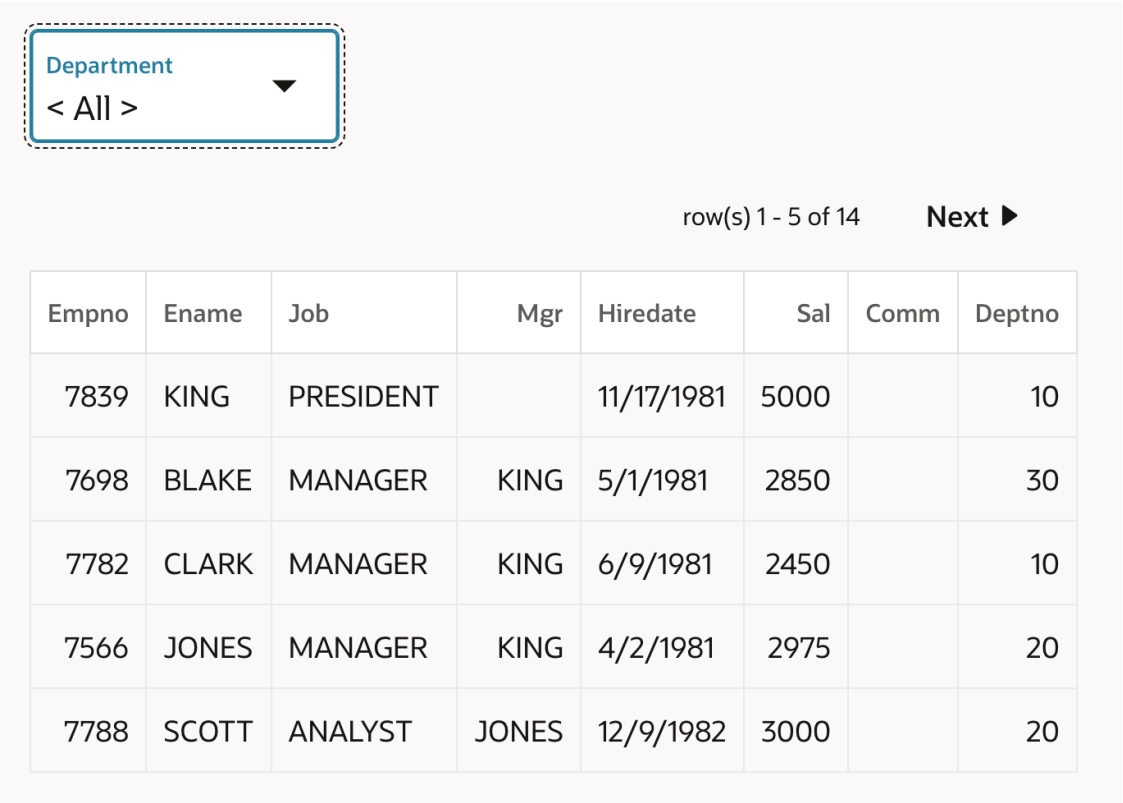


The **Select List**, **Radio Group**, and **Popup LOV** items support optionally including an item in the list that represents the `null` value, and if included it can have a custom label. This can be useful to display text that helps a user understand the meaning of **not** selecting a value. For

Copyright © 2026, Oracle and/or its affiliates.

example, the Department **Select List** page item shown below is used as an optional query by example criteria field. A **Null Display Value** of `All` helps the user understand that employees from all departments are retrieved if they don't select a specific department.

Figure 3-41 Query by Example Select One Page Item with Null Display Value



3.8.2 Defining the List of Values

When you create a list-driven page item, you define its list of valid choices either locally on the page item itself or by referencing a shared component List of Values by name.

If it's the first time you need a particular list of choices, then defining it locally is most immediate. Later, if you need to reuse a list you defined somewhere else, you can easily convert the locally-defined list to a shared **List of Values** component. Then you can reference it in multiple page items going forward.

In both cases, for each valid choice you provide a **Display Value** and a **Return Value**. The display value is what the user sees in the user interface. The return value gets assigned to the page item when the user makes a selection. For a list-driven page item configured for single selection, its value is the return value of the user's choice. For a multi-selection page item, the value is a delimiter-separated string containing the return values corresponding to the user's choices. The default delimiter is the colon (`:`), but you can configure it to be a different character when needed.

For example, for a page item showing a list of departments, the list of choices might be defined as follows:

Copyright © 2026, Oracle and/or its affiliates.

Display Value	Return Value
ACCOUNTING	`10`
RESEARCH	`20`
SALES	`30`
OPERATIONS	`40`

After a user chooses *SALES* from a single-selection page item `P5_SELECTED_DEPARTMENT`, its value would be *`30`*. For a multi-selection page item `P5_SELECTED_DEPARTMENTS`, after the user chooses *ACCOUNTING* and *RESEARCH* its value would be *`10:20`*.

- Defining a Local List of Values
  To define the available choices for a list-driven page item, you can specify a set of static values or provide a SQL query.
- Additional Shared LOV Features
  If you anticipate reusing a list of values multiple times, define a shared List of Values (LOV). This named component supports additional features and data sources when compared with a locally-defined one.
- Converting Local LOVs to Shared
  You can use the *Locally Defined* tab of the shared List of Values component page to quickly convert any locally-defined LOV to a named, shared one.

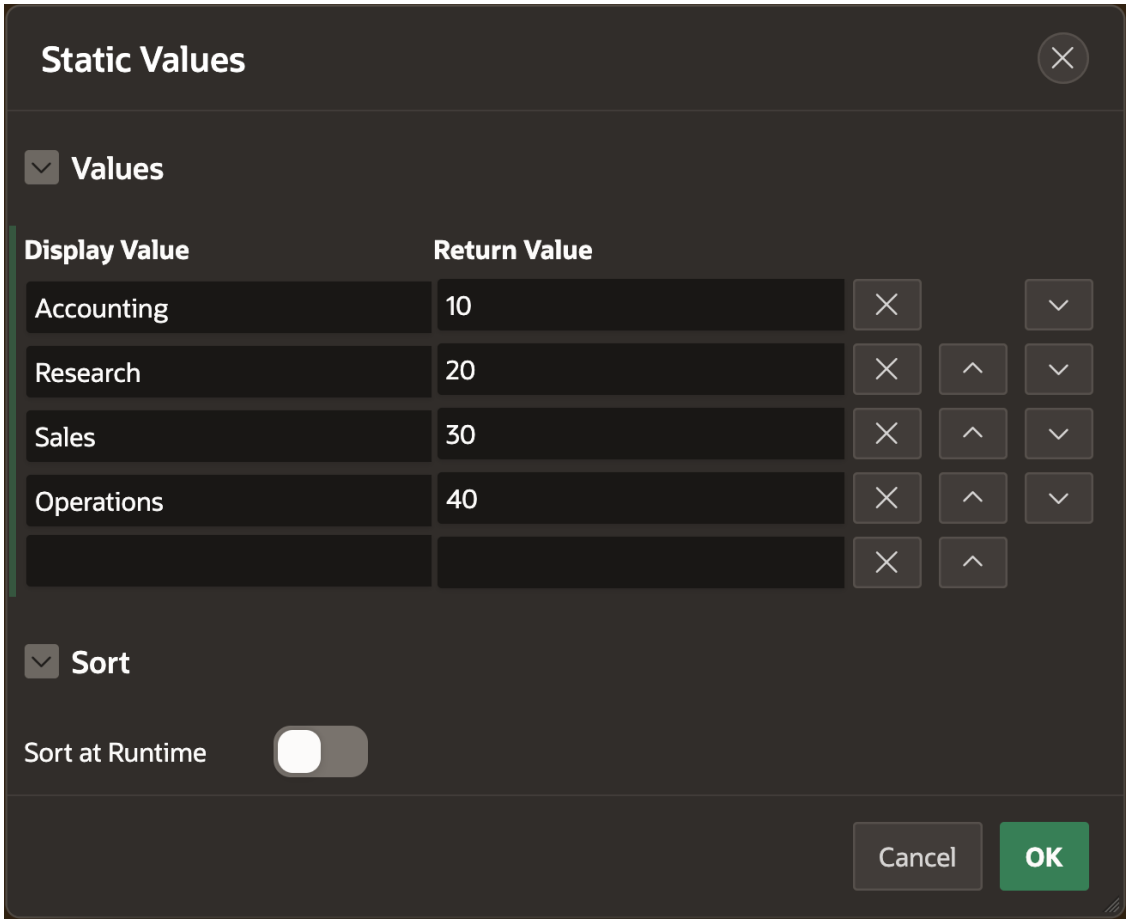
3.8.2.1 Defining a Local List of Values

To define the available choices for a list-driven page item, you can specify a set of static values or provide a SQL query.

When defining the static list of values, as shown below, you type in the **Display Value** and **Return Value** for each entry. Use the buttons next to the entries to delete and reorder them, and switch the **Sort at Runtime** toggle off if the items should appear exactly as entered. The default is to sort the items alphabetically on **Display Name**.

Copyright © 2026, Oracle and/or its affiliates.

Figure 3-42 Defining a List of Values with Static Values



When the choices for a list-driven item are not static, then use a SQL query to define the list. The query should have two columns in the `SELECT` list. The first provides the display value and the second is the return value. It's common, although not required, to alias the columns as `d` and `r`. For example, you can define an alphabetical list of departments using the `DNAME` column as the display value and the `DEPTNO` column as the return value using the query:

```
select dname as d, deptno as r
  from dept
 order by dname
```

The list of values query can optionally refer to values of one or more page items as bind variables. For example, if a page has an item named `P5_DEPARTMENT` containing a department number, then a list of values query for employees in that department can be:

```
select ename as d, empno as r
  from emp
 where deptno = :P5_DEPARTMENT
 order by ename
```


Copyright © 2026, Oracle and/or its affiliates.

3.8.2.2 Additional Shared LOV Features

If you anticipate reusing a list of values multiple times, define a shared List of Values (LOV). This named component supports additional features and data sources when compared with a locally-defined one.

For example, when you create a shared LOV with static entries, you can optionally configure a condition with each entry to indicate when it should appear in the list.

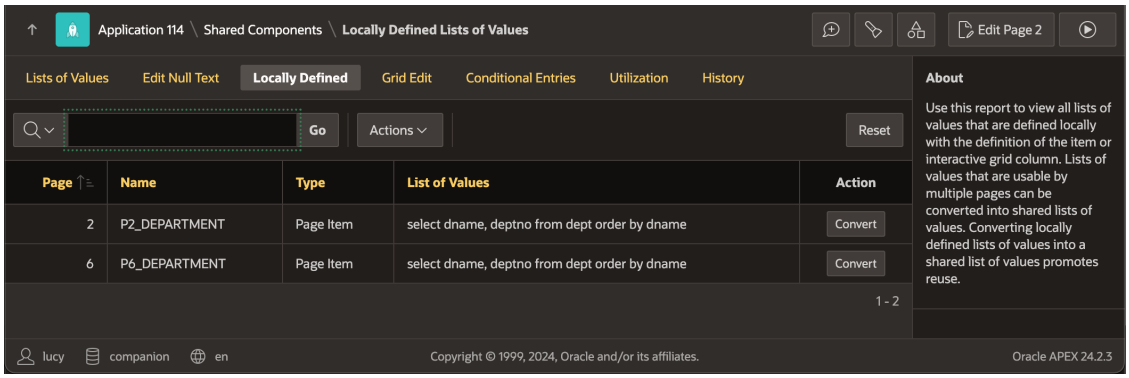
A shared LOV whose choices are *not* static is known as a **Dynamic List**. Its entries come from a SELECT statement or a remote data source. The query or data source can provide additional columns beyond the usual display value and return value. One of these additional columns can be the name of icon to display in each row, and you can use these extra columns in the custom list entry display formatting of **Select One**, **Select Many** or **Combobox** page items. With a Popup LOV, you can also display the additional columns, let users search on them, and return their values into other page items.

3.8.2.3 Converting Local LOVs to Shared

You can use the *Locally Defined* tab of the shared List of Values component page to quickly convert any locally-defined LOV to a named, shared one.

The figure shows the *Shared Components* List of Values list page. On its **Locally Defined** tab, two List of Values appear, each with a **Convert** button in the **Action** column of the table. Just click this button to convert the locally-defined LOV into a shared component. App Builder adjusts the original page to use the new shared LOV.

Figure 3-43 Converting Locally-Defined List of Values to Shared



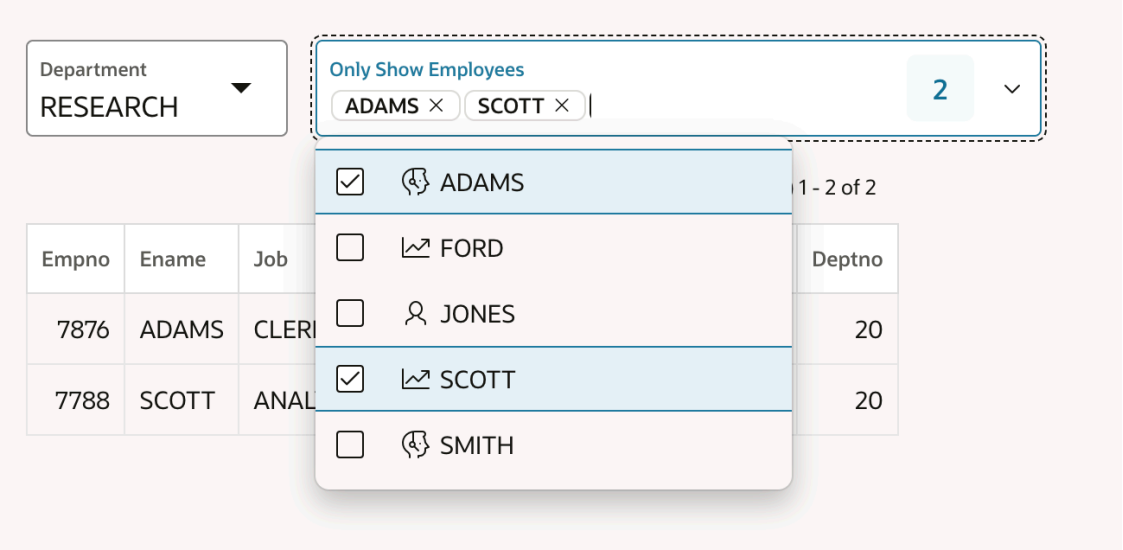
3.8.3 Configuring Cascading Lists

When you need one page item's value to influence the list of choices in another list-driven item, you can easily configure this **cascading** effect.

For example, in the **Only Show Employees** Select Many list below, the list of employees shown depends on the department selected in the Select List item to its left.

Copyright © 2026, Oracle and/or its affiliates.

Figure 3-44 Department Page Item as Cascading Parent Item for Employees List



This requires only two steps. First, the list of values query for the **Only Show Employees** page item references the value of the page item it depends on using a bind variable:

```
select ename as d,
       empno as r
from emp
where deptno = :P2_DEPARTMENT
order by ename
```

Then, you configure the **Parent Items(s)** property on the `P2_ONLY_EMPLOYEES` list item to name the page item it depends on: `P2_DEPARTMENT`. This property is in the **Cascading List of Values** section of the Property Editor.

Setting the **Parent Items(s)** property enables two automatic runtime behaviors. When a user changes `P2_DEPARTMENT`, then the value of `P2_ONLY_EMPLOYEES` gets cleared and its list of available choices is refreshed. If your list of values query depends on any other page items as bind variables, list their names in the **Items to Submit** property in the same section. The values of the page items mentioned in **Items to Submit** are sent along when the list gets refreshed, but there is no other automatic behavior related to the change in their values. It's only a parent item's value change that clears and refreshes the cascading list.

Notice in the figure above that the list displays an icon for each list entry. To use an icon or additional columns beyond the display value and return value, you need to define a shared List of Values component then reference it from the cascading list item. So, in practice the **Only Show Employees** Select Many list shown above references a shared List of Values component with the following query that defines the icon based on each employee's job:

```
select ename as d,
       empno as r,
       case job
            when 'PRESIDENT' then 'fa-badgerine'
            when 'ANALYST'   then 'fa-line-chart'
            when 'CLERK'     then 'fa-user-headset'
            when 'SALESMAN'  then 'fa-badge-dollar'
```


Copyright © 2026, Oracle and/or its affiliates.

```
            when 'MANAGER'   then 'fa-user'
       end as icon
from emp
where deptno = :P2_DEPARTMENT
order by ename
```

The example above shows a pair of cascading page items, but using the same technique, you can configure multiple levels of cascading. For example, a user could pick the brand of an automobile like *Volkswagen* from a `P2_BRAND` list. Then, they might select a particular model of Volkswagen like *Tiguan* from a `P2_MODEL` list that uses `P2_BRAND` as its parent item. Finally, the user could choose an available Tiguan engine configuration like *1.5 TSI 130 PS (96kW)* from a `P2_ENGINE` list that mentions `P2_MODEL` as its parent item. At each level, when the parent item is cleared, the cascading list item gets cleared and refreshed.

The same cascading list behavior is available among editable Interactive Grid region cells, with the same simple configuration steps. Your grid column's list of values query references the parent column's value as a bind variable using its column name, and you configure the **Parent Column(s)** property using one or more column names.

3.9 Simple Layout for Any Screen Size

Pages in your APEX application look polished and professional at any screen size with minimal effort.

Instead of focusing on exact positioning and sizing, you arrange regions and page items into a flexible grid of columns and rows. At runtime, your page elements respond automatically to the device's screen size by adjusting position and size to fit their relative layout to the space available. You test your page by running it. You see what users experience, with one-click access to Page Designer to make iterative refinements. Your live changes to browser width, or testing the page on different sized devices, give the immediate visual feedback you need to perfect your page.

- Flexible Grid of Columns and Rows
  Design a page's layout by arranging regions and page items into a flexible grid.
- Templates and Options Set Look and Feel
  Choosing from Universal Theme's available page templates, region templates, and item templates, you get professional results using a declarative approach. Each template offers options you can configure in Page Designer to fine-tune your page's look and feel.
- Nest Regions to Create Any Layout
  A region can contain other regions in addition to its primary content.

3.9.1 Flexible Grid of Columns and Rows

Design a page's layout by arranging regions and page items into a flexible grid.

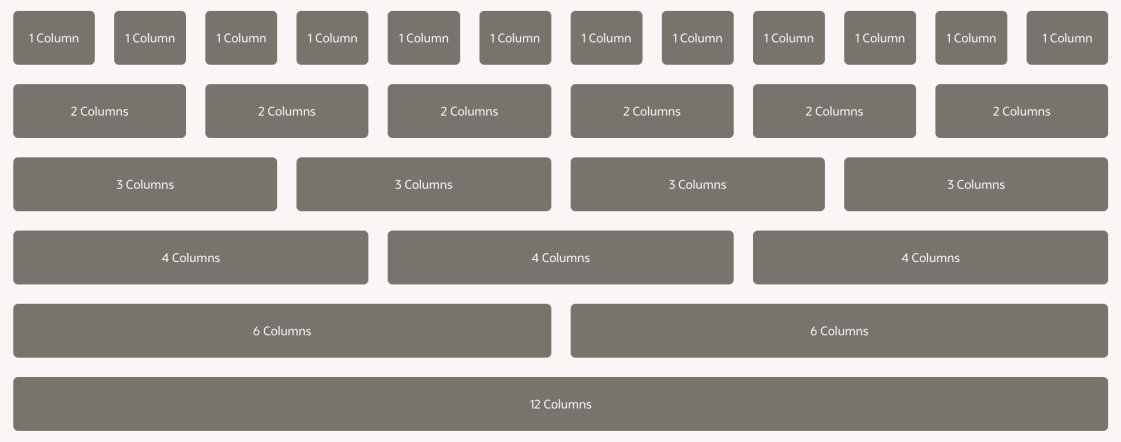
The grid of columns and rows uses three simple rules:

1. Each region's width is divided into 12 logical columns of equal size.
2. An element's start column and column span can be automatic or you can set them manually.
3. An element stays in the same row as the one before it, unless you configure it to start a new row.

Copyright © 2026, Oracle and/or its affiliates.

As shown below, APEX uses a 12 column grid since it gives six ways to divide a row evenly. This simplifies laying out elements in a visually appealing way by default.

Figure 3-45 Dividing Horizontal Space Evenly Using a 12 Column Grid



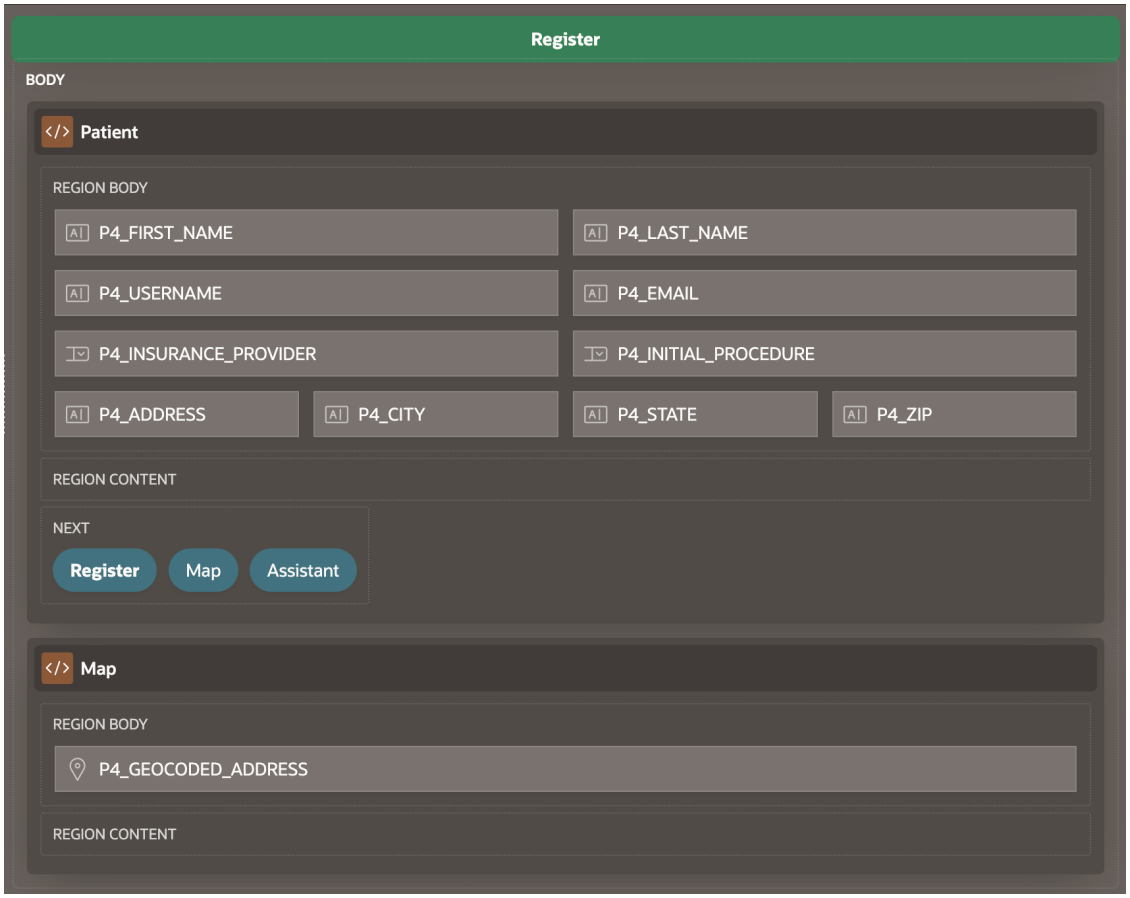
The figure below shows the Page Designer **Layout** pane for a patient registration page with a **Patient** region and a **Map** region. The **Patient REGION BODY** slot stretches its contents to fill the entire 12-column width. `P4_FIRST_NAME` starts in the first column, and takes up half the width: 6 columns. `P4_LAST_NAME` starts in column 7 and also spans 6 columns. In the fourth row starting with `P4_ADDRESS`, each of the four page items spans 3 columns. The **Map REGION BODY** slot similarly stretches its contents. Its single **Geocoded Address** page item starts in column one and spans all 12 columns.

The `P4_USERNAME`, `P4_INSURANCE_PROVIDER`, and `P4_ADDRESS` page items each has its **Start New Row** property set to true. Other page items have this property set to false, so each stays in the same row as the page item that comes before it in layout order.

Notice the **Patient** region's **NEXT** slot containing three buttons. It does *not* stretch its content to fill the region's available width. Instead, some slots meant for buttons do not use the grid layout. The buttons use only the width they need, with uniform space between them.

Copyright © 2026, Oracle and/or its affiliates.

Figure 3-46 Automatic Column Spanning Using Only "Start New Row"



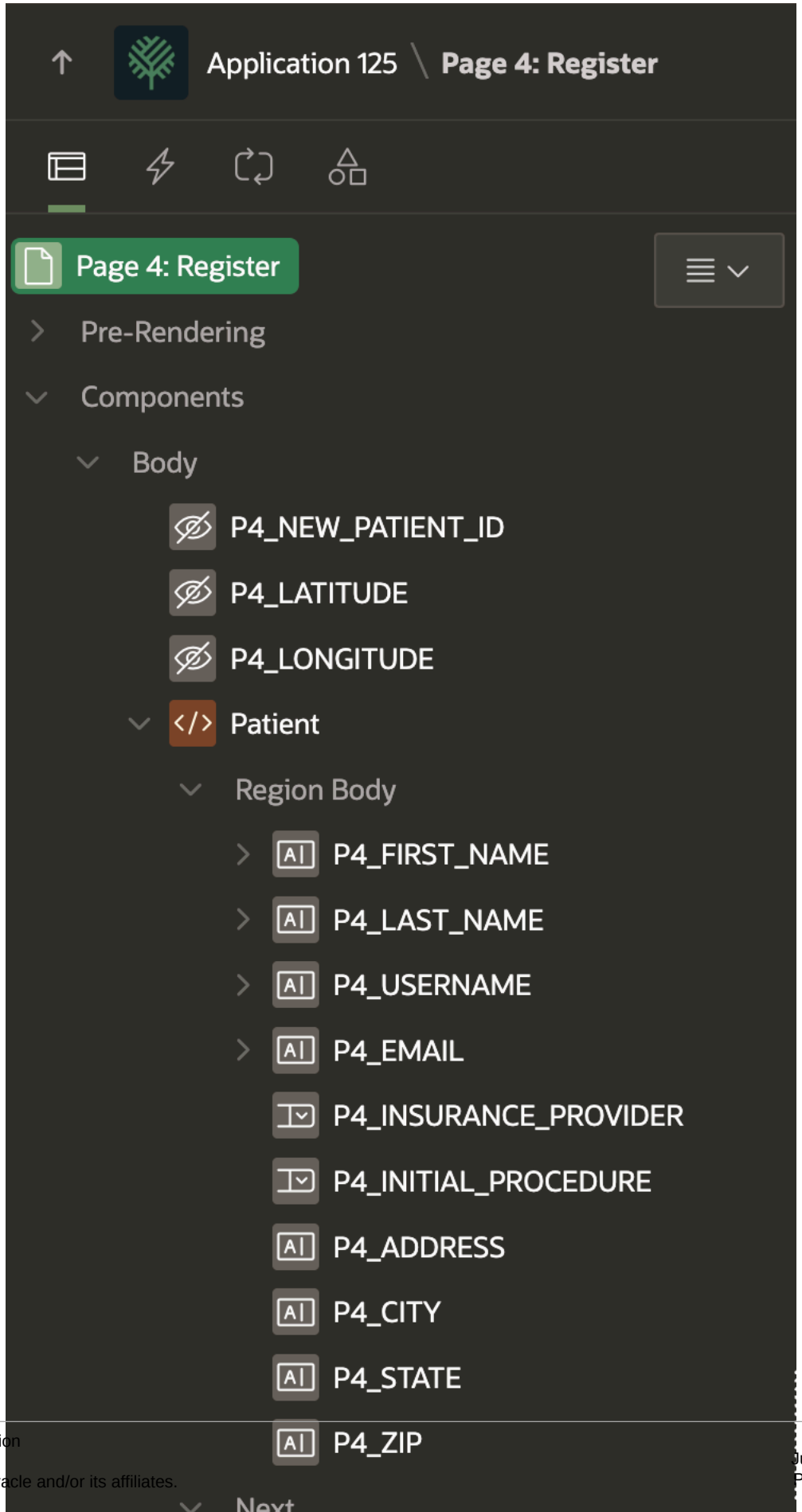
As shown below, the sequence of elements in the component tree in Page Designer sets the layout order. You can drag an element to reorder it in the tree, and the **Layout** pane immediately reflects the change. Alternatively, you can resequence items by changing the value of their **Sequence** property.

At the top of the tree, the elements whose icons are an eyeball with a diagonal line through it are hidden page items. These act like invisible page variables that users never see, so they do not appear in the **Layout** pane.

Besides the visible difference in the buttons' layout, another clue that the **NEXT** slot does *not* use the columns and rows grid is that the Property Editor has no **Start New Row** property for a button like **Assistant** in this slot. If that same button were instead in the **BODY** slot that *does* uses the grid, its **Start New Row** property would appear.

Copyright © 2026, Oracle and/or its affiliates.

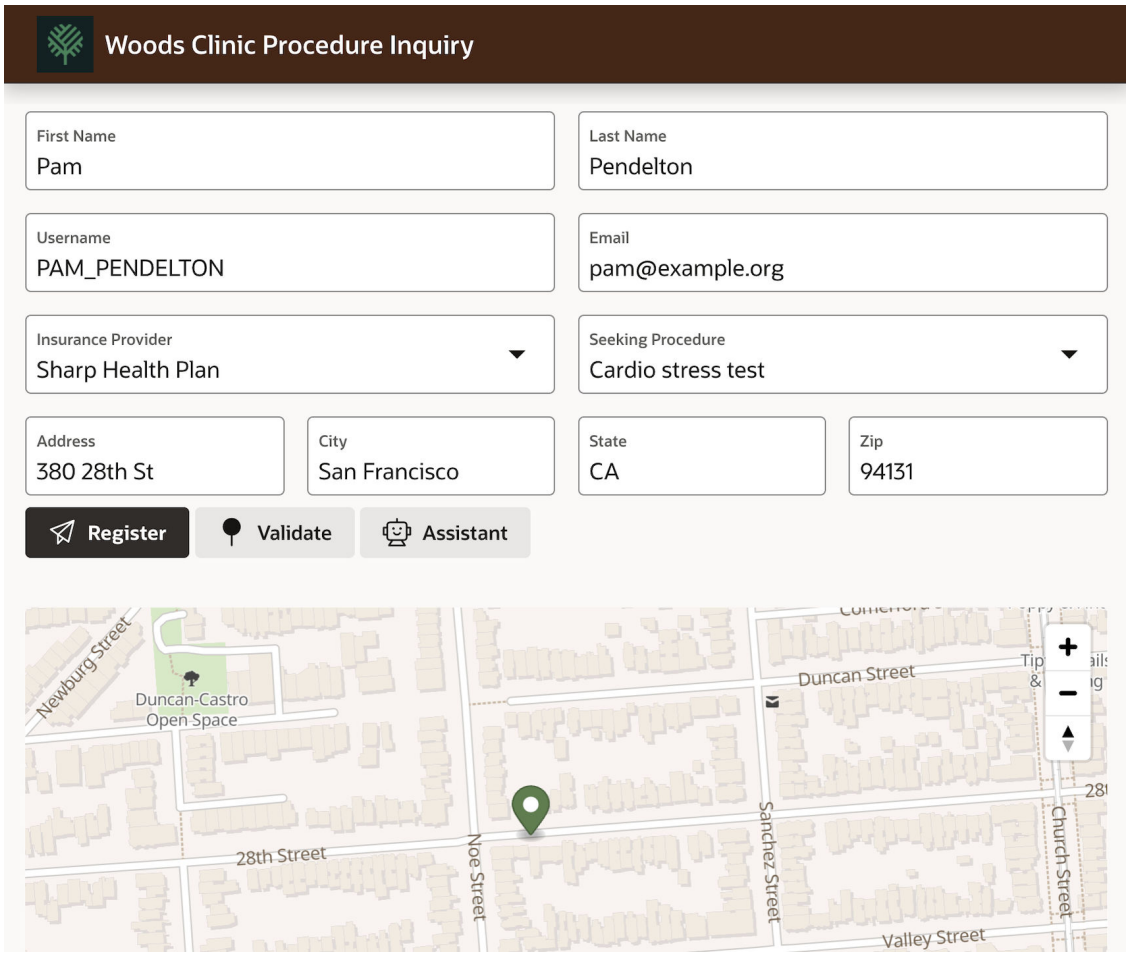
Figure 3-47 Component Tree Dictates Layout Order



Copyright © 2026, Oracle and/or its affiliates.

If you run this page, the figure below shows what you see in the browser. The page items in the **Patient** region's **Body** slot fill the available width, maintaining the relative layout from Page Designer. The **Map** region appears below the **Patient** region and also occupies all 12 logical columns. The buttons in the **NEXT** slot just use the space required by their label and icon.

Figure 3-48 Automatic Column Spanning at Runtime



By default, the **Column Start** and **Column Span** properties are set to **Automatic**. This means each element in a row tries to occupy the same amount of space if possible, and each starts in the next available column after the one that comes before it in layout order.

Sometimes you want to dedicate more width to a particular item or have it start in a specific column. You can manually set an item's **Column** property to explicitly start in a column from 1 to 12. Independently, you can set the **Column Span** property to the number of columns the item should occupy. Only legal combinations are allowed. So, for example, if you set an item to start in column 10 and also manually set a **Column Span**, then the only allowed values are 1, 2, or 3. Any more would cause that item to go beyond the 12th grid column.

In the figure below, both `P4_ADDRESS` and `P4_CITY` have their **Column Span** set to 4. This leaves 4 of the 12 columns to `P4_STATE` and `P4_ZIP` in that same row. Since they are set to **Automatic**, they evenly divide the remaining space with 2 columns each.

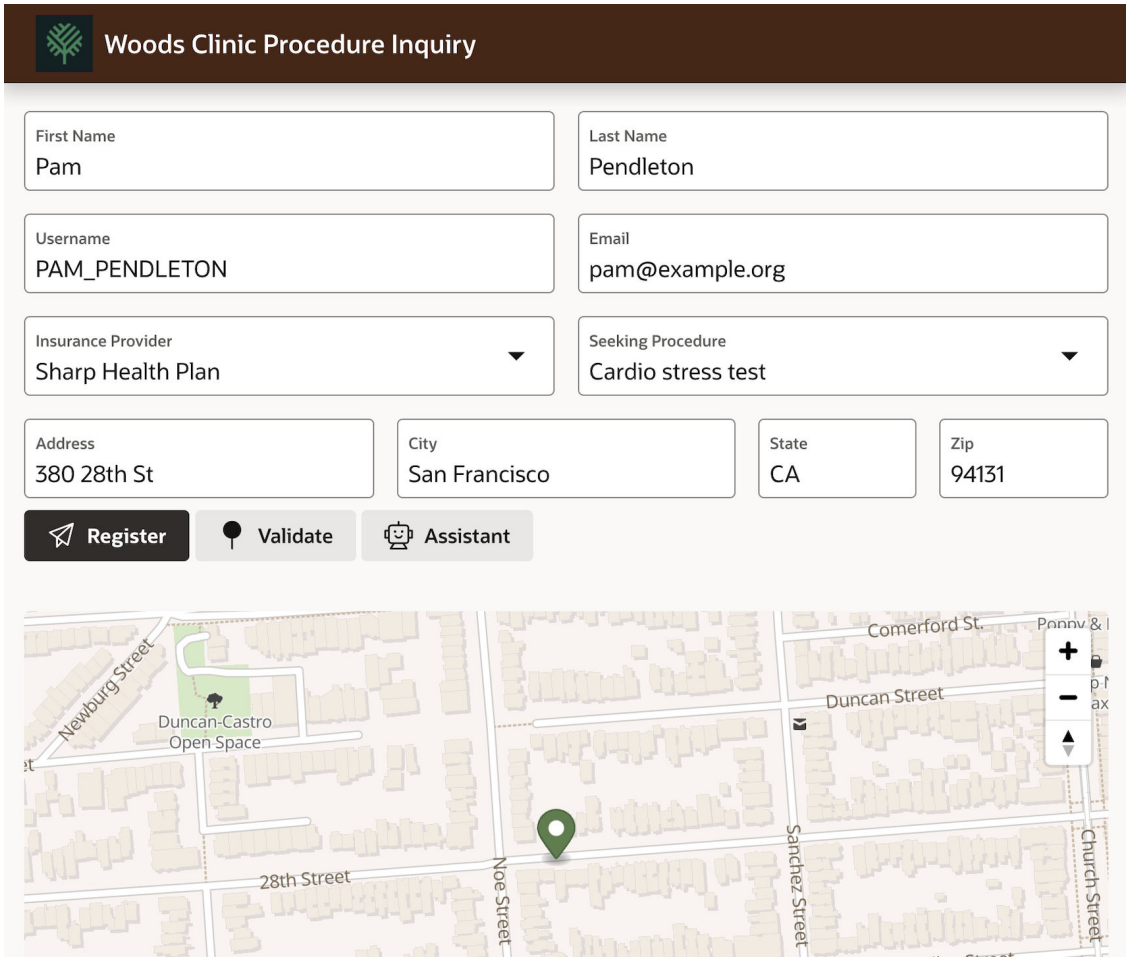
Copyright © 2026, Oracle and/or its affiliates.

Figure 3-49 Manually Specifying Column Span



Running the page, the figure below now reflects the custom column spans.

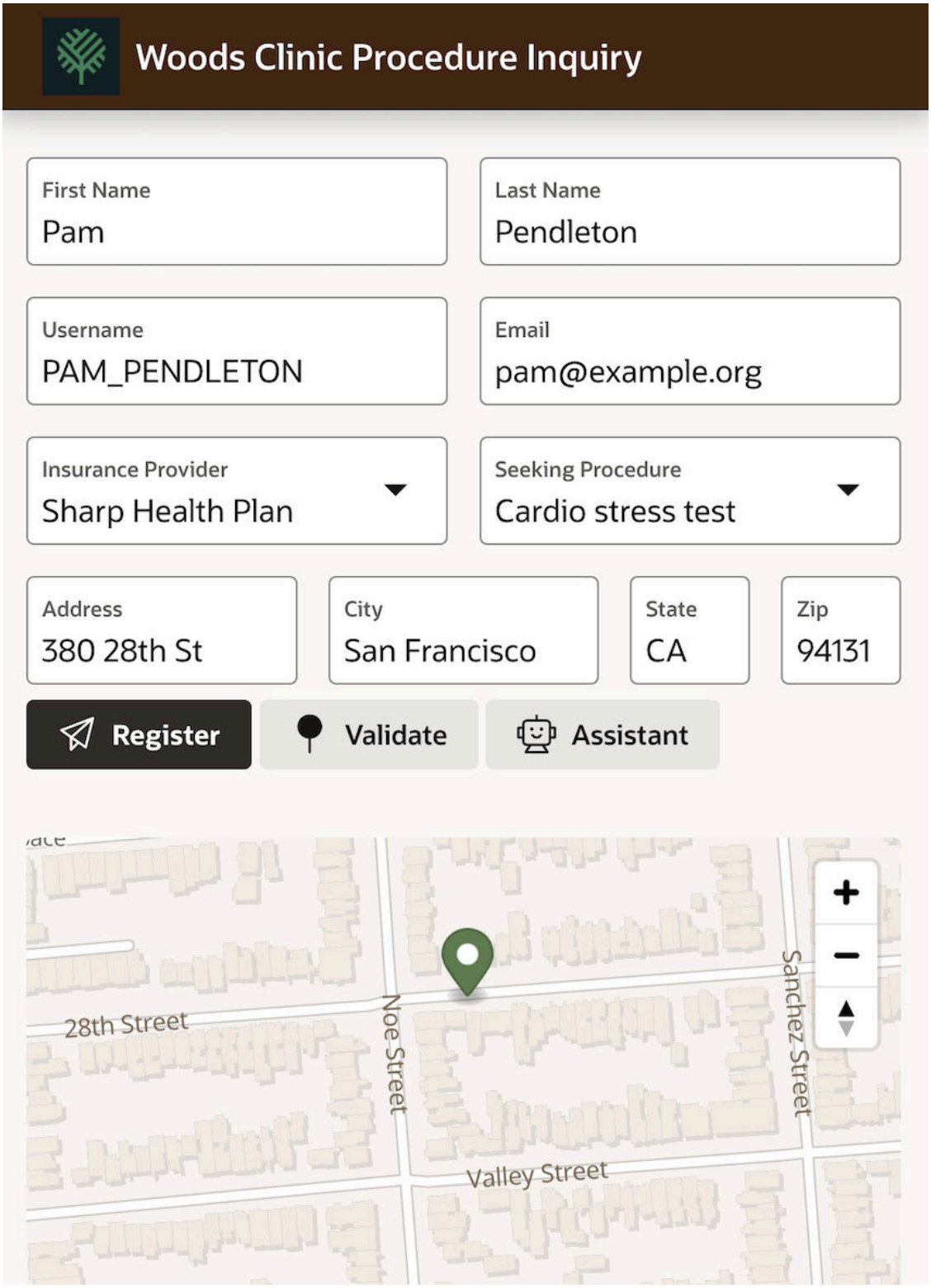
Figure 3-50 Effect of Manually Specifying Column Spanning at Runtime



Copyright © 2026, Oracle and/or its affiliates.

While you are testing the page, you can reduce the width of the browser window to see how the page responds to a new screen size. APEX preserves your logical layout as long as it can as the screen gets narrower. The figure below shows how the same page looks on a narrower mobile device.

Figure 3-51 Layout Responds Automatically to Narrower Screen

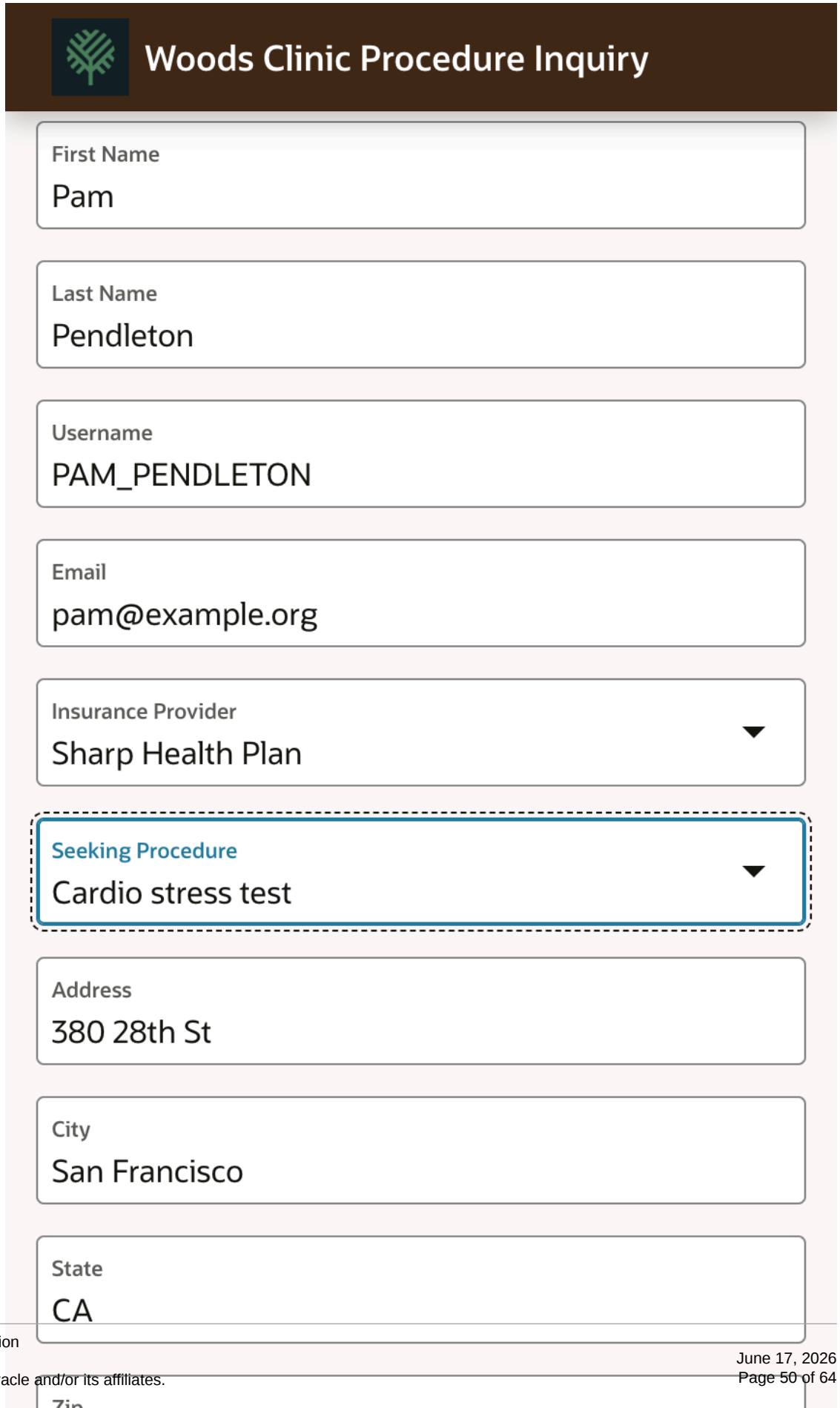


Copyright © 2026, Oracle and/or its affiliates.

However if you continue to make the window narrow enough, at some point it will switch to showing each item in its own row to preserve the usability of the page. For example, the figure below shows the same page on the width of an iPhone 14 screen. This is responsive design in action. Every APEX page you create behaves this way. There's no need to design different versions of the same page for different screen sizes. Your single logical layout adapts to any device you use it on automatically.

Copyright © 2026, Oracle and/or its affiliates.

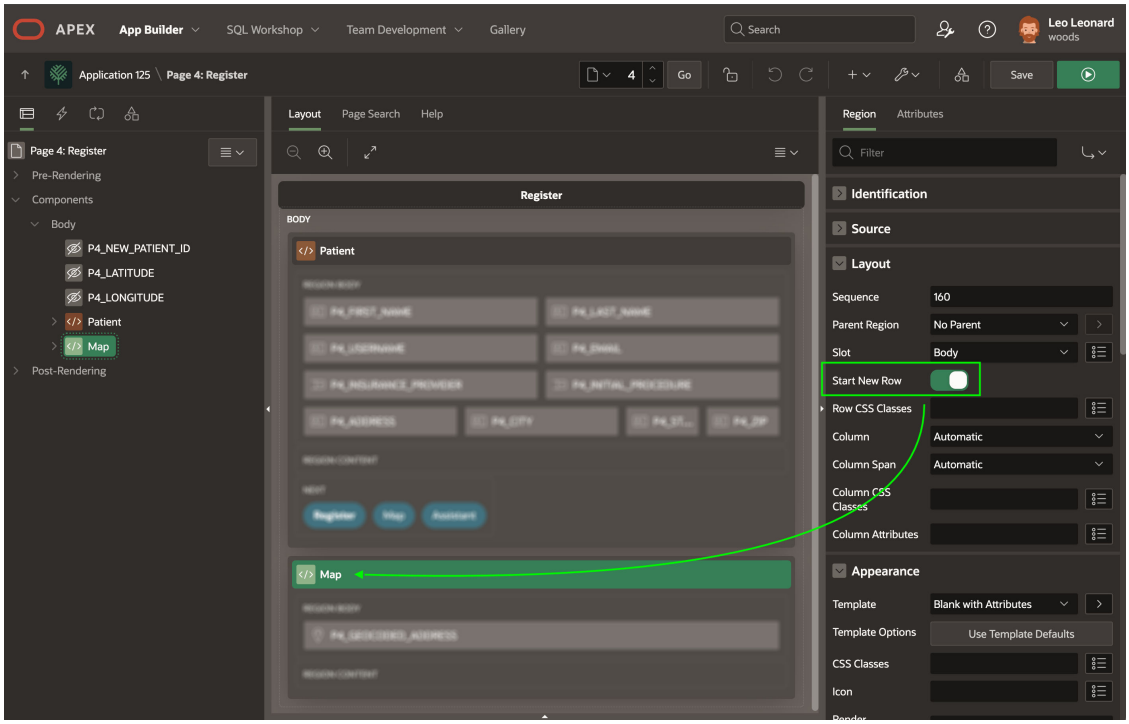
Figure 3-52 Registration Page on a Mobile Device with Even Narrower Screen



Copyright © 2026, Oracle and/or its affiliates.

So far, the **Map** region has appeared below the **Patient** region. The figure below explains why. The same simple to understand grid of columns and rows also operates at the page level. The component tree nodes representing the **Patient** and **Map** regions are collapsed to emphasize how they are elements at the same level of the tree. Their **Layout** pane contents are blurred to focus attention on how the regions themselves are two consecutive elements contained in the **BODY** of the page. Each is using the **Automatic** setting for **Column Span**, so each occupies the entire width available: all 12 logical columns in the page **BODY**. Notice the **Map** region has its **Start New Row** property set to true, so it starts in a new row.

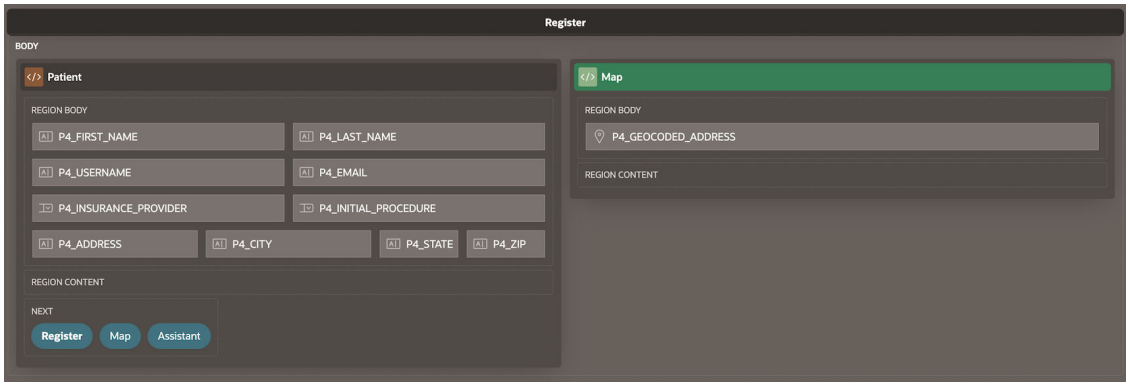
Figure 3-53 The Same Grid of Columns and Rows Operates at Page Level



By setting the **Map** region's **Start New Row** property to false, it stays on the same row as the element before it at the same level of the component tree. As shown below, this puts the **Map** region to the right of the **Patient** region in the same row of the page **BODY**. Since both regions use the **Automatic** setting for **Column Span**, they evenly split the space in their row. The same intuition for configuring **Column** and **Column Span** in a region's contents, also applies at page level. You could force the map to only occupy one **third** of the width instead of one half, simply by setting the region's **Column Span** to 4. This is because 4 is one third of the 12 available columns.

Copyright © 2026, Oracle and/or its affiliates.

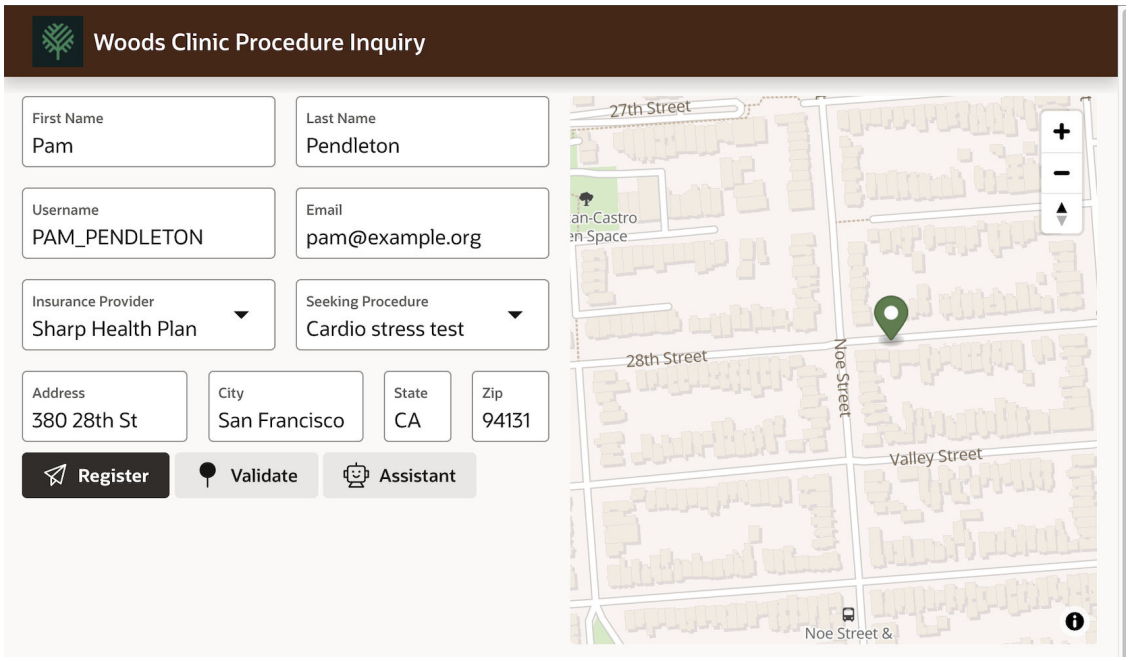
Figure 3-54 Disabling "Start New Row" for Map Region



Leaving the **Automatic** setting and running the page produces the result in the figure below. The **Patient** region occupies half the width of the page's body. Within that half of the page, the relative 12-column layout of the region's **BODY** contents is preserved. Since the **Map** region's only Geocoded Address item spans all 12 column in that region's **BODY** contents, the map fills its entire half of the page body.

Following the three simple rules illustrated here, laying out any page that works at any screen size becomes a breeze with a bit of practice.

Figure 3-55 Disabling "Start New Row" for Map Region



3.9.2 Templates and Options Set Look and Feel

Choosing from Universal Theme's available page templates, region templates, and item templates, you get professional results using a declarative approach. Each template offers options you can configure in Page Designer to fine-tune your page's look and feel.

Copyright © 2026, Oracle and/or its affiliates.

The *Universal Theme Reference App* at https://oracleapex.com/ut is your playground for live experiments.

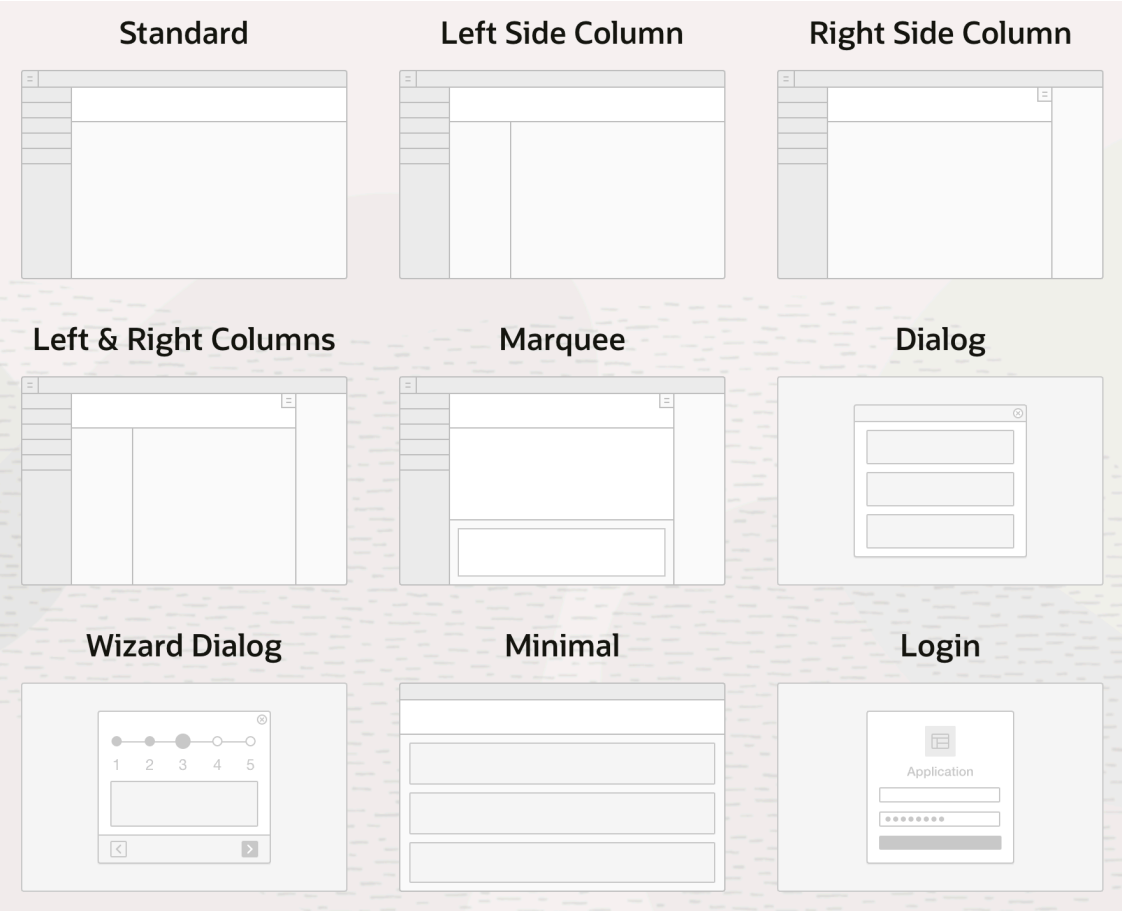
- Page Template Sets the Scene
  The page template you choose organizes the space on the page into slots that contain your page contents.
- Region Template Styles a Section
  A region's template organizes the layout of its section of the page.
- Item Template Controls the Label
  The item template controls how a page item renders its label.

3.9.2.1 Page Template Sets the Scene

The page template you choose organizes the space on the page into slots that contain your page contents.

The figure below shows the slots each page template provides. **Standard** is the default, and is one of the five that also display the application navigation menu. See the **Design**, **Page Templates** section of the Universal Theme reference app for additional details.

Figure 3-56 Universal Theme Page Templates



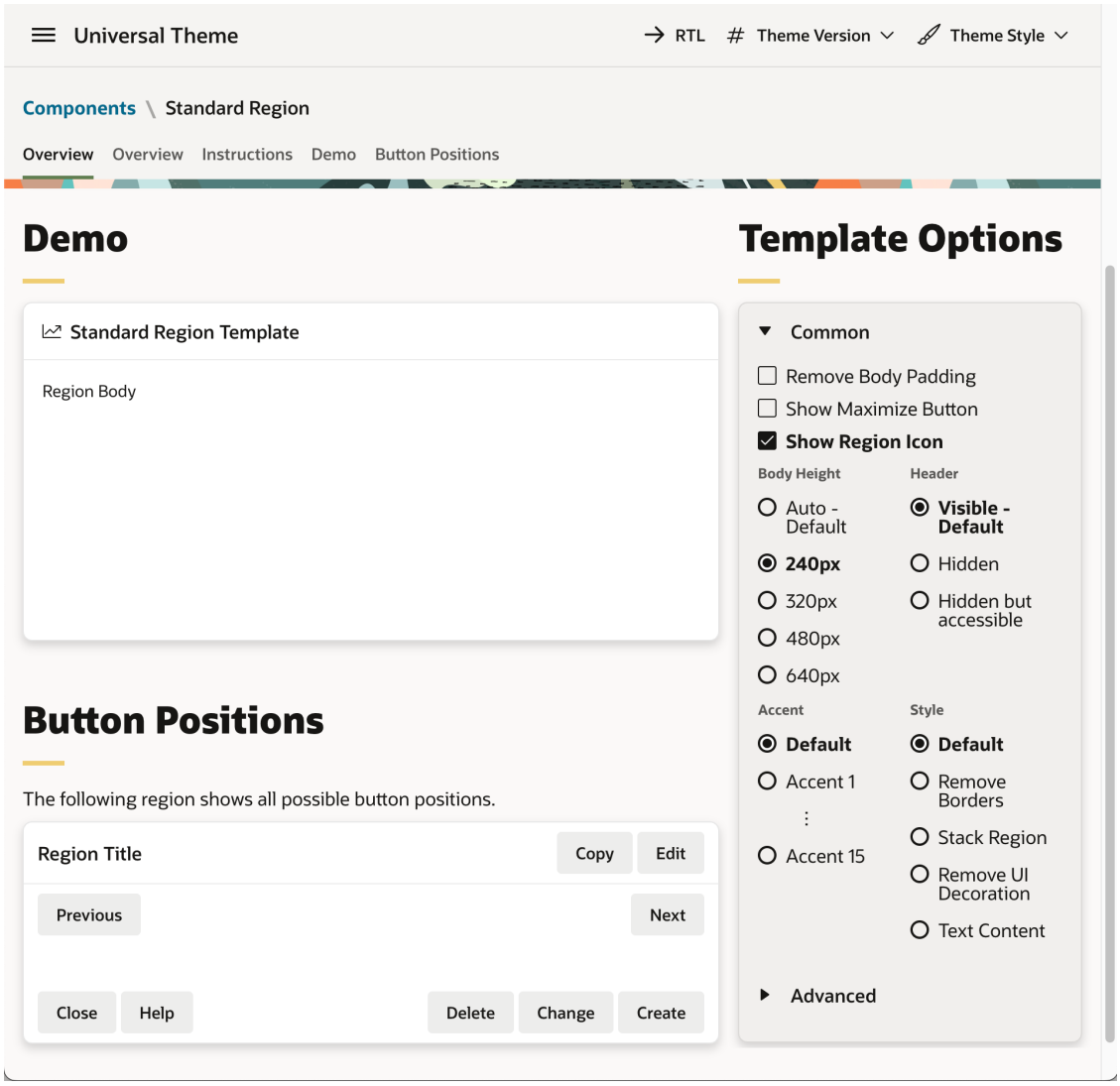
Copyright © 2026, Oracle and/or its affiliates.

3.9.2.2 Region Template Styles a Section

A region's template organizes the layout of its section of the page.

There are many different region templates to explore in the Universal Theme Reference App, but the default **Standard** template serves most common use cases. As shown below, its **Region Body** slot shows your primary contents using the 12-column grid layout. It has nine slots for positioning buttons, and simple options control its heading, border, and spacing. When appropriate, you can decide to show *only* the region's contents by using these options to hide its header and remove UI decoration.

Figure 3-57 Standard Region Template Options in the Universal Theme Reference App



To show region contents with no buttons or borders, the **Blank with Attributes** template is a one-click alternative to hiding the *Standard* template's header and using no UI decoration.

When you need to create a toolbar, but none of the page template's or **Standard** region's button slots are correct, then the **Buttons Container** template lets you create a toolbar somewhere else on the page.

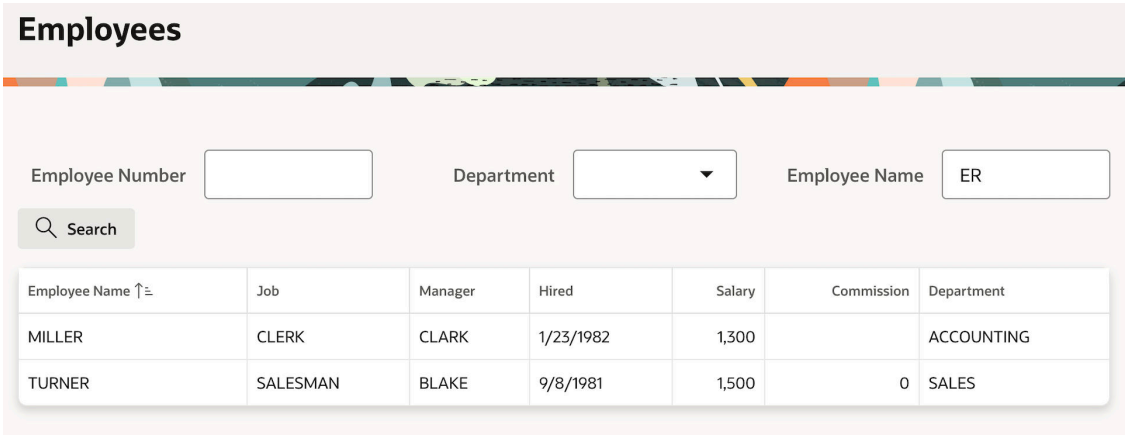
Copyright © 2026, Oracle and/or its affiliates.

3.9.2.3 Item Template Controls the Label

The item template controls how a page item renders its label.

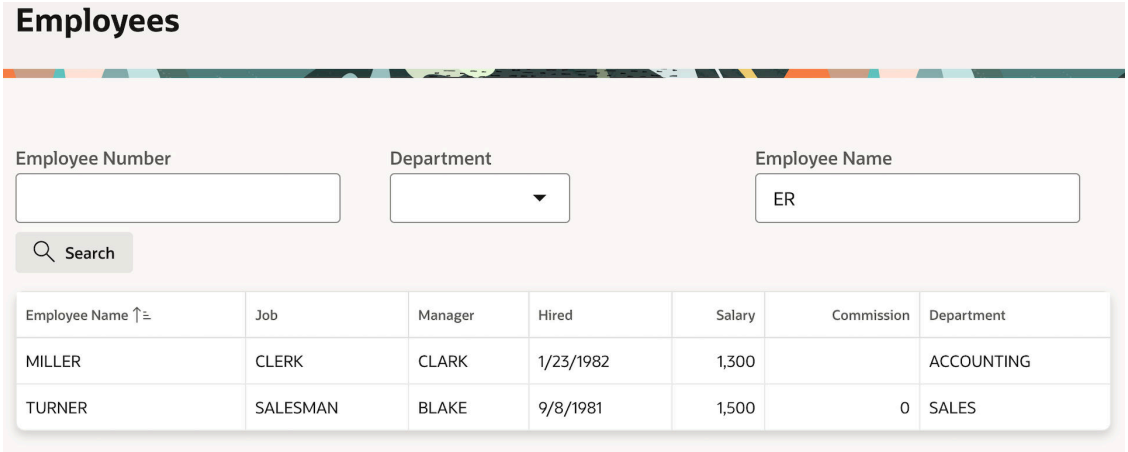
You use an item template to configure how the user sees a page item's label and required indicator. Set the item template to **Hidden** to have no label. To have the label appear before the item, use the **Optional** or **Required** template. In these three cases, the **Label Column Span** indicates the number of grid columns the label occupies. Within their respective column spans, the label is aligned to the end and the page item to the start. Notice that by default, the field is *not* stretched to fill its column span, although there is an option to enable that. In the figure below, the three query by example fields above the employees search results use the **Optional** item template.

Figure 3-58 Three Query by Example Page Items Using Optional Item Template



Using the **Optional - Above** or **Required - Above** item template, the labels show above the page item and there is no **Label Column Span** to set. As above, the fields do not stretch. The figure below shows the same page with its query by example fields using this template. All the page items use the **Automatic** column span setting, so the three items in the same row occupy four columns each.

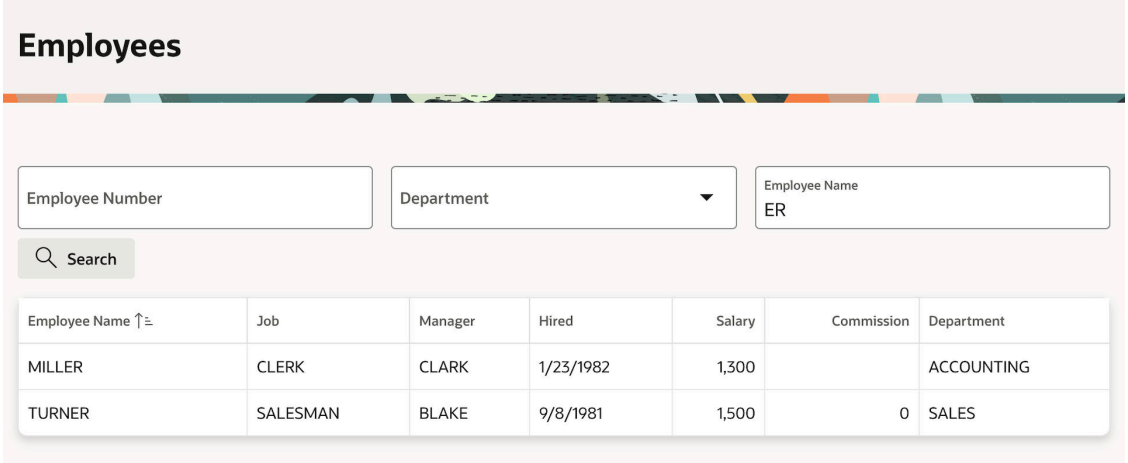
Figure 3-59 Same Page Using Optional - Above Item Template



Copyright © 2026, Oracle and/or its affiliates.

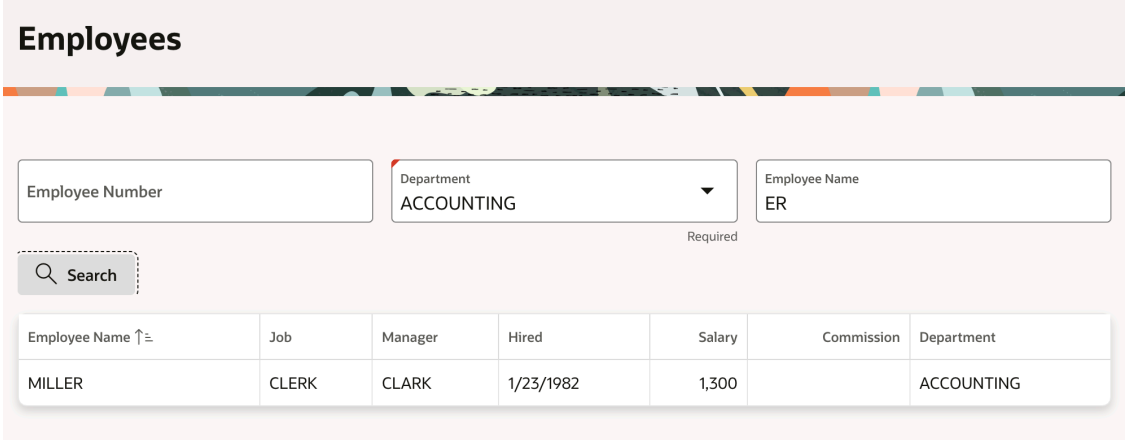
The label on items using the **Optional - Floating** or **Required - Floating** template appears inside the item. The label shrinks slightly and moves up when the field has a value. This item template stretches the item to fill its column span as shown below.

Figure 3-60 Items With Floating Item Template Stretch to Fill Column Span



The **Department** query by example field in the figure below uses a **Required - Floating** item template. Items with a required template display a visual marker, and if desired a template option lets you show an inline **Required** label as shown below.

Figure 3-61 Department's Required Item Template Displays Visual Marker and Inline Text



3.9.3 Nest Regions to Create Any Layout

A region can contain other regions in addition to its primary content.

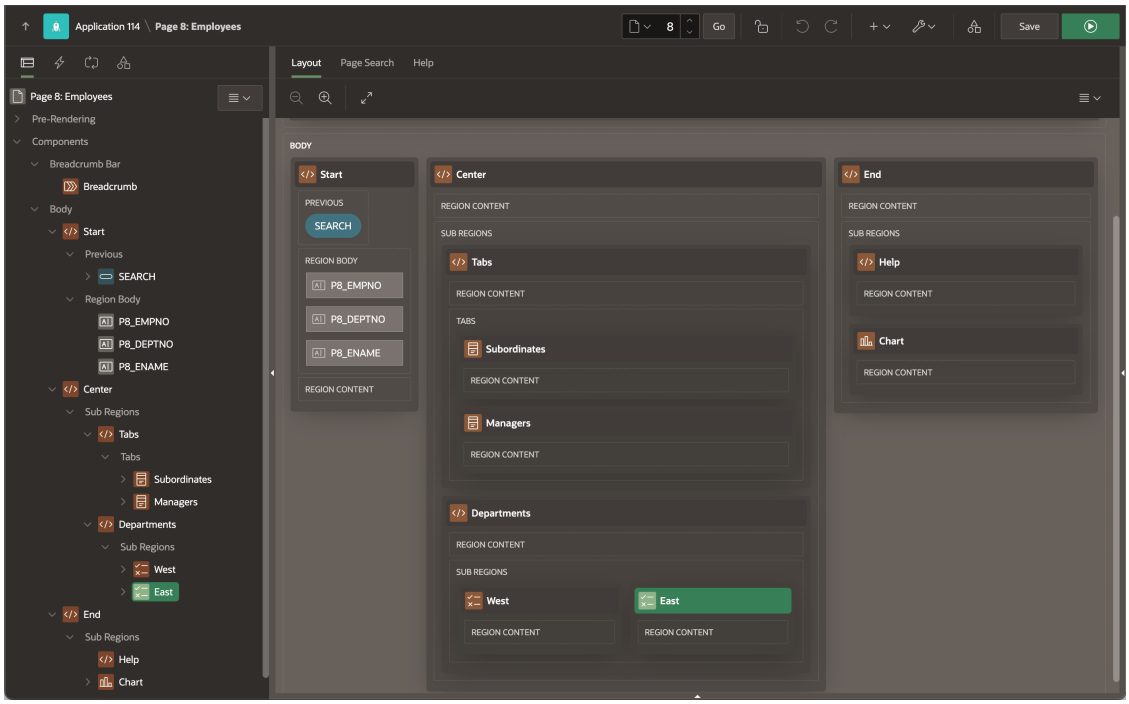
Both the **Layout** pane and the component tree display this relationship. Sub-regions have a **Parent Region** property you can see and change in the Property Editor. By nesting regions inside other regions and applying the grid layout rules, you can quickly produce any kind of layout.

Copyright © 2026, Oracle and/or its affiliates.

While any region can contain other regions, empty **Static Content** regions are most common for layout needs. For example, in the figure below, the page's **Body** slot has three static content regions named **Start**, **Center**, and **End**. Each uses the **Blank with Attributes** template, so none shows a header or borders. The **Start** region contains page items in its **Body** slot, while **Center** and **End** act as invisible containers for the regions inside.

These three top-level regions are in the same grid row of the page **Body**, so you can divide up the space in the 12-column grid by setting each region's **Column Span**. The **Start** region has **Column Span** of two (2) and also contains a **(Search)** button in its **Previous** slot. The **Center** region spans six columns and nests a **Tabs** region and a **Departments** region. The **End** region contains a **Help** region showing an informational message and a **Chart** region in two grid rows one above the other. The **Tabs** region contains two Cards regions: **Subordinates** and **Managers**, while **Departments** nests two Content Row regions in the same row side by side.

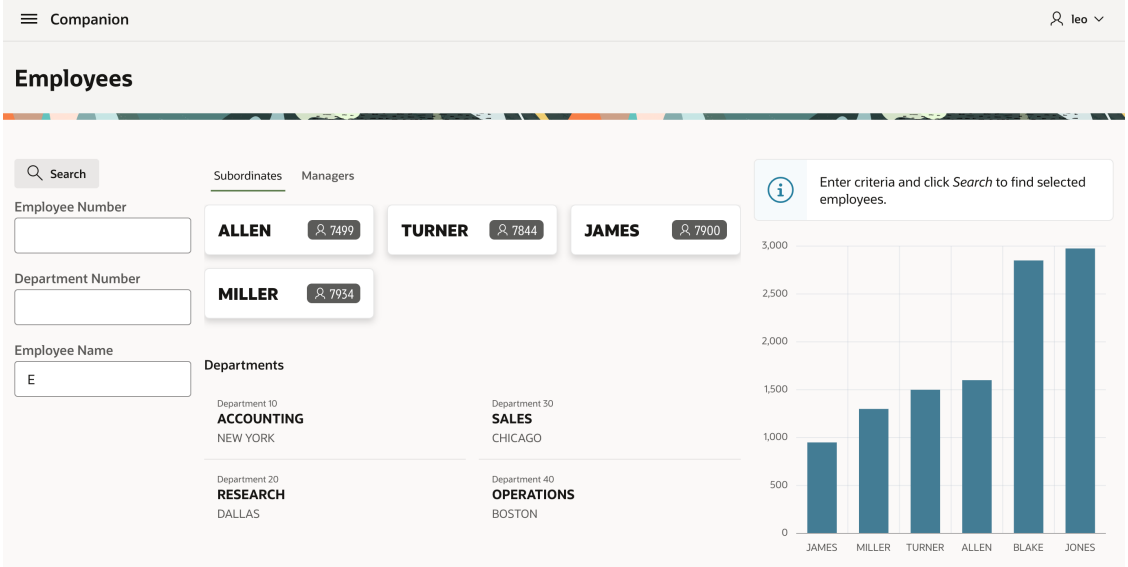
Figure 3-62 Regions Can Contain Other Regions and Items



The figure below shows how the page looks at runtime. It shows how the three top-level regions divide up the page's horizontal space according to their column span. Within each one, the nested region content are stacked vertically and horizontally as indicated in Page Designer. You'll quickly develop an intuition for using the grid layout rules with region nesting to create any layout you need to.

Copyright © 2026, Oracle and/or its affiliates.

Figure 3-63 Responsive APEX Web Page With Nested Regions



3.10 Launching Like Native Apps

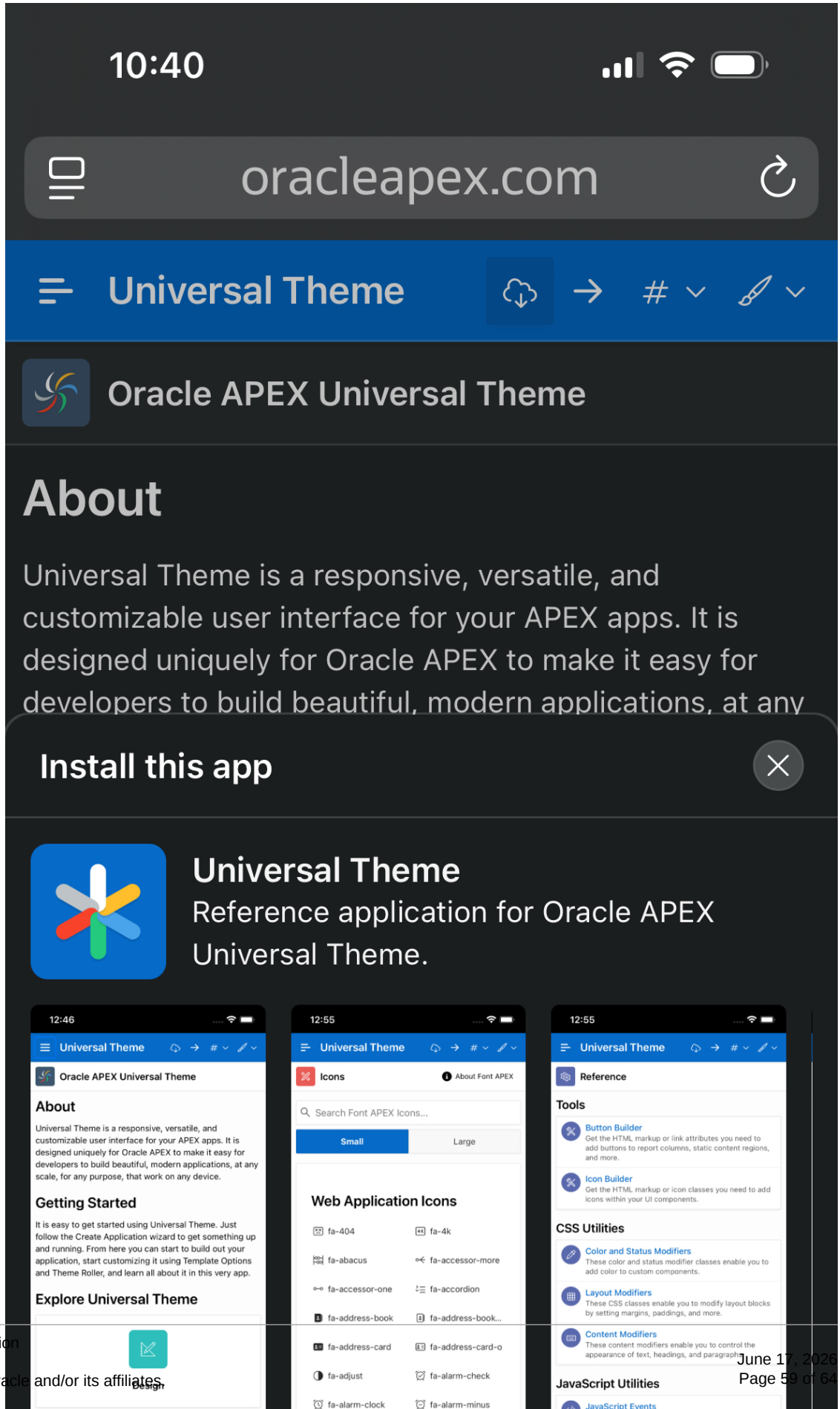
With a single click, you can let any user install and launch your APEX app like a native app.

By enabling the **Installable** switch in your application's **Progressive Web App** settings, you give users a simple installation experience on their smartphone or desktop. Your app icon appears in the device's Home Screen, in the Dock of their Mac, or in their Windows Taskbar. Tapping or clicking the icon launches a full-screen experience that looks and feels like other native apps on the platform. A key difference with progressive web apps is that your APEX app is always up to date, and you are always certain all users have your latest fix or feature.

You can try this experience yourself on your smartphone or desktop by visiting the Universal Theme Reference App at https://oracleapex.com/ut. As shown below, clicking the "cloud with download arrow" icon in the navigation bar opens the simple installation experience to guide you.

Copyright © 2026, Oracle and/or its affiliates.

Figure 3-64 Tapping to Install an APEX as a Native App on iPhone

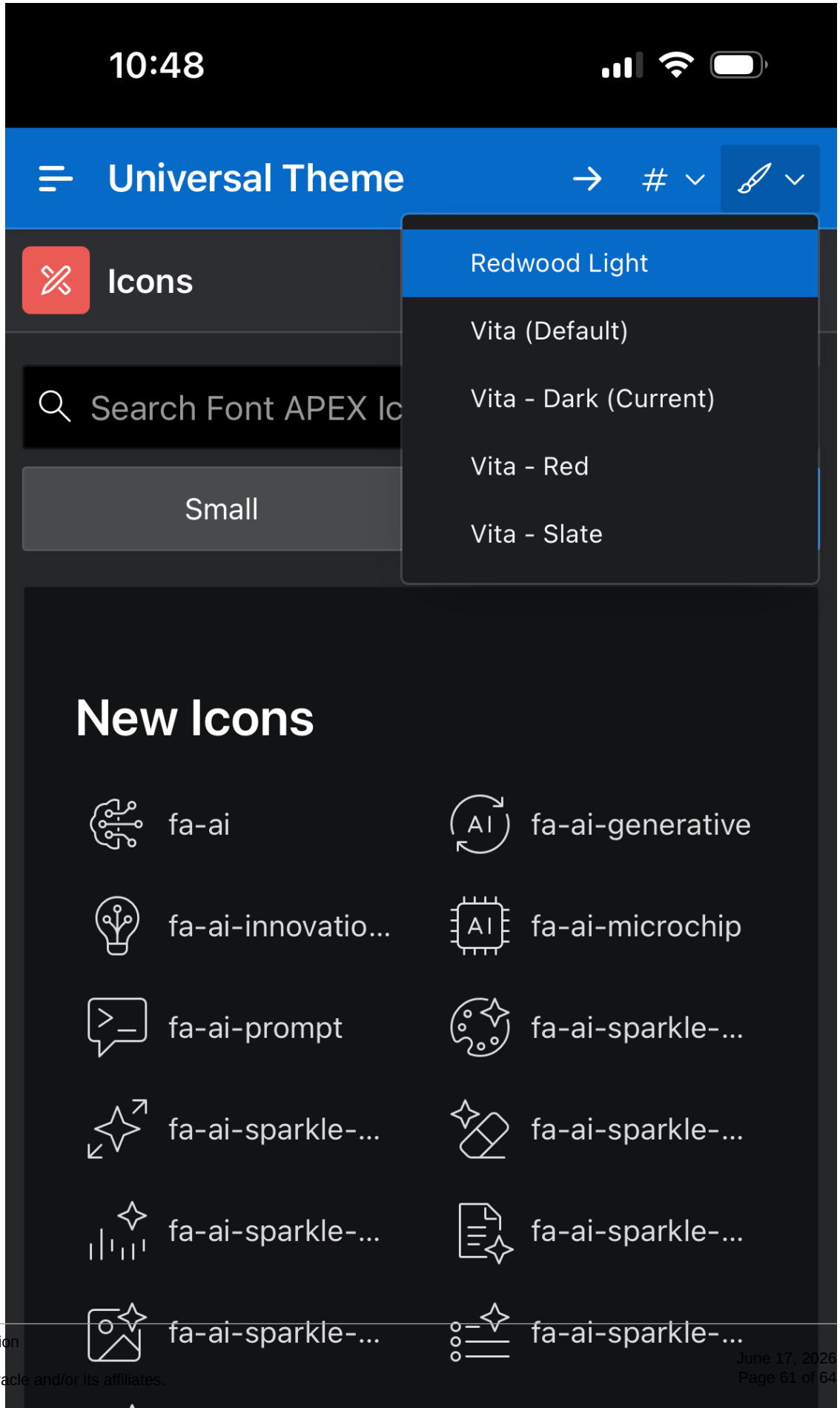


Copyright © 2026, Oracle and/or its affiliates.

After the quick installation process, tapping the **Universal Theme** app icon on your Home Screen opens the app in full screen where it works and behaves like any other native iPhone app. The figure below shows one of the pages in the reference app used for finding icons. You can install the **APEX Progressive Web App** reference app at https://oracleapex.com/pwa in an identical way.

Copyright © 2026, Oracle and/or its affiliates.

Figure 3-65 Full Screen iPhone App Experience with APEX Progressive Web App

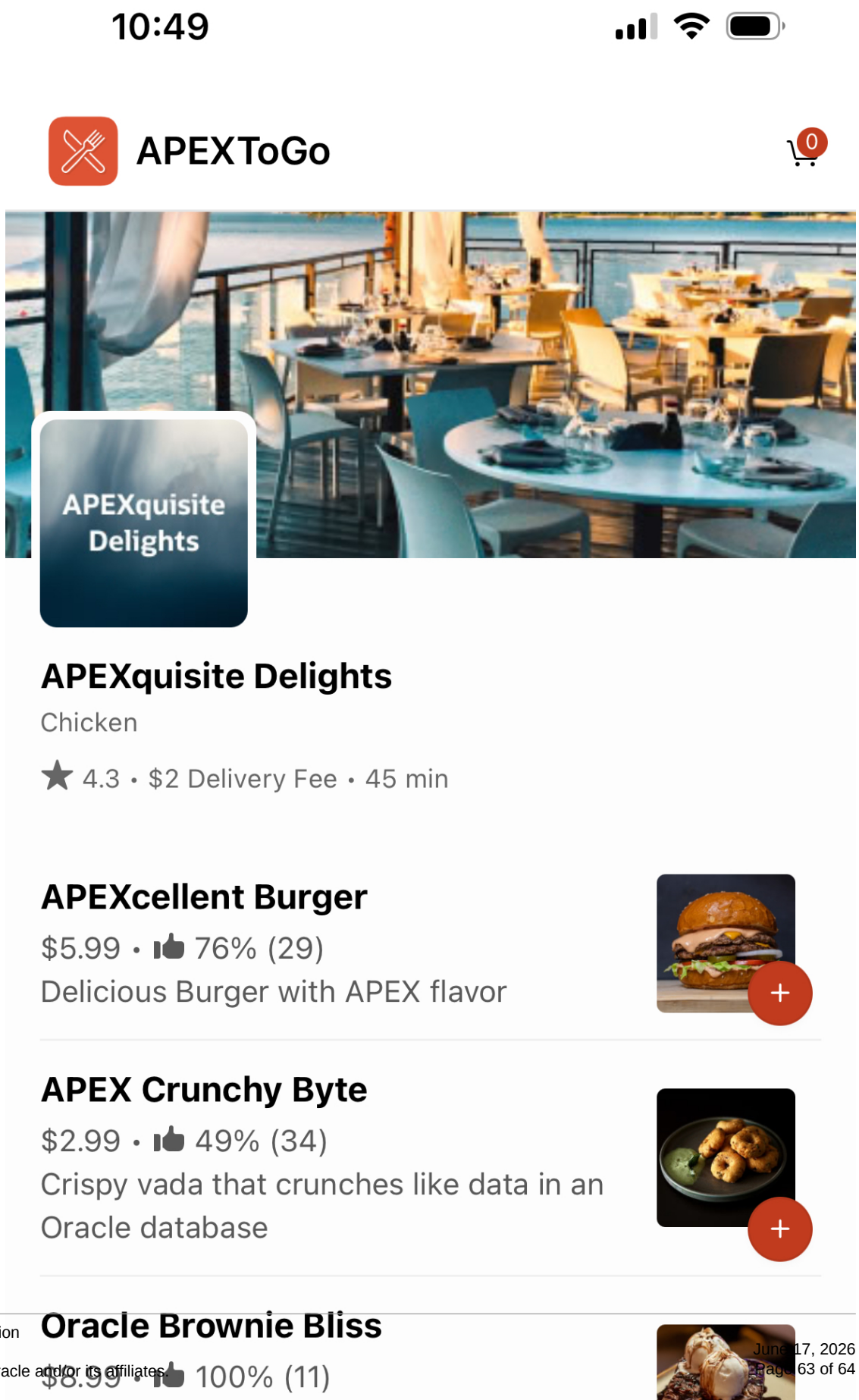


Copyright © 2026, Oracle and/or its affiliates.

Your APEX app can use mobile device features like front-facing (selfie) cameras and mobile location, and take advantage of optimized mobile keyboards for searching and entering numbers, email addresses, phone numbers, and URLs. Check out the **APEX To Go** mobile sample app shown below at https://apex.oracle.com/go/apextogo to learn more about creating APEX apps that are optimized for the mobile experience.

Copyright © 2026, Oracle and/or its affiliates.

Figure 3-66 APEX To Go Mobile Sample App on iPhone



Copyright © 2026, Oracle and/or its affiliates.

Related Topics

- Creating a Progressive Web App in *Oracle APEX App Builder User's Guide*

Copyright © 2026, Oracle and/or its affiliates.

4

Referencing Data Values in Pages

While working on your pages, you'll often need to reference the values of page items or report columns for formatting, in queries, and in business logic.

- Using Data Values in HTML Expressions
  When fine-tuning data formatting, to reference the value of page item `P1_NAME` use `&P1_NAME.` starting with an ampersand and ending with a dot. This also works to reference application items, built-in substitution strings, and column values in HTML expressions for regions like Cards, Content Row, Template Components, and Interactive Grid.
- Using Data Values in Template Directives
  Use *Template Directives* in your HTML expressions to conditionally format data or iterate over a list of delimited values.
- Directives Reference in Your Dock
  The Universal Theme reference app contains examples of each template directive.
- Using the Context-Sensitive Help Tab
  The context-sensitive **Help** tab in Page Designer is a handy resource. It's in the same tab group as the **Layout** tab by default.
- Using Data Values in SQL and PL/SQL
  Depending on the context, in your SQL and PL/SQL you use either `:NAME` bind variable syntax or `GET_TYPE()` functions in the `APEX_SESSION_STATE` package to use data values by name.

4.1 Using Data Values in HTML Expressions

When fine-tuning data formatting, to reference the value of page item `P1_NAME` use `&P1_NAME.` starting with an ampersand and ending with a dot. This also works to reference application items, built-in substitution strings, and column values in HTML expressions for regions like Cards, Content Row, Template Components, and Interactive Grid.

For example, consider a content row region based on the `DEPT` table where you need to reference the `DNAME` and `LOC` column values in the HTML expression for the **Overline** slot. You can use:

```
<strong>&DNAME.</strong> → &LOC.
```

In contrast, when working with columns in a Classic Report or Interactive Report, or when defining your own templates, use `#NAME#` instead. For example, imagine a classic report region based on the `DEPT` table where you want to format the department name column in a custom way. You can use the value of both `DNAME` and `LOC` columns in the `DNAME` column's **HTML Expression** as follows. If you also set the `LOC` column type to be *Hidden* then it won't occupy an additional column.

```
<strong>#DNAME#</strong> → #LOC#
```


Copyright © 2026, Oracle and/or its affiliates.

When using a page item or region source's column value in an HTML **attribute**, use the additional `!ATTR` qualifier as shown below to ensure the page formats the value correctly to be used for that purpose:

```
<a href="#TARGET_LINK!ATTR#"><strong>#DNAME#</strong></a>
```

If the column value *itself* includes HTML markup, use the `!RAW` qualifier to avoid the default escaping of characters like <, >, and & that have special meaning in HTML. For example, if a `DESCRIPTION_IN_HTML` column contains HTML markup, then reference it like this to ensure the browser sees the HTML markup as intended:

```
<i>#DESCRIPTION_IN_HTML!RAW#</i>
```

These qualifiers also work for `&NAME!RAW.` and `&NAME!ATTR.` use cases.

4.2 Using Data Values in Template Directives

Use *Template Directives* in your HTML expressions to conditionally format data or iterate over a list of delimited values.

When referencing data values inside the curly braces of a directive, use its name with no prefix or suffix.

Depending on the context, what you put *inside* the directive uses `#NAME#` or `&NAME.` notation. For example, in a Cards, Content Row, Template Components, and Interactive Grid region, use the `&NAME.` substitution syntax as follows to include the value of `TITLE` if it is not empty.

```
{if TITLE/}&TITLE.{endif/}
```

In a Classic or Interactive Report, or when defining your own templates, use `&NAME.` for page items, application items, or built-in substitution strings, but `#COLUMNNAME#` notation for column names. Also note that space before the closing `/}` is allowed.

```
{if TITLE /}#TITLE#{endif/}
```

Template directives make it easy to combine columns into a formatted result with appropriate conditional formatting. Imagine a `PERSONS` table with columns for `HONORIFIC`, `FIRST_NAME`, and `LAST_NAME`. In an email template, to format a greeting as "Dear Professor Smith," if an honorific is present and "Dear Jennifer," otherwise, you can use template directives like this:

```
Dear {if HONORIFIC/}#HONORIFIC# #LAST_NAME#{else/}#FIRST_NAME#{endif/},
```

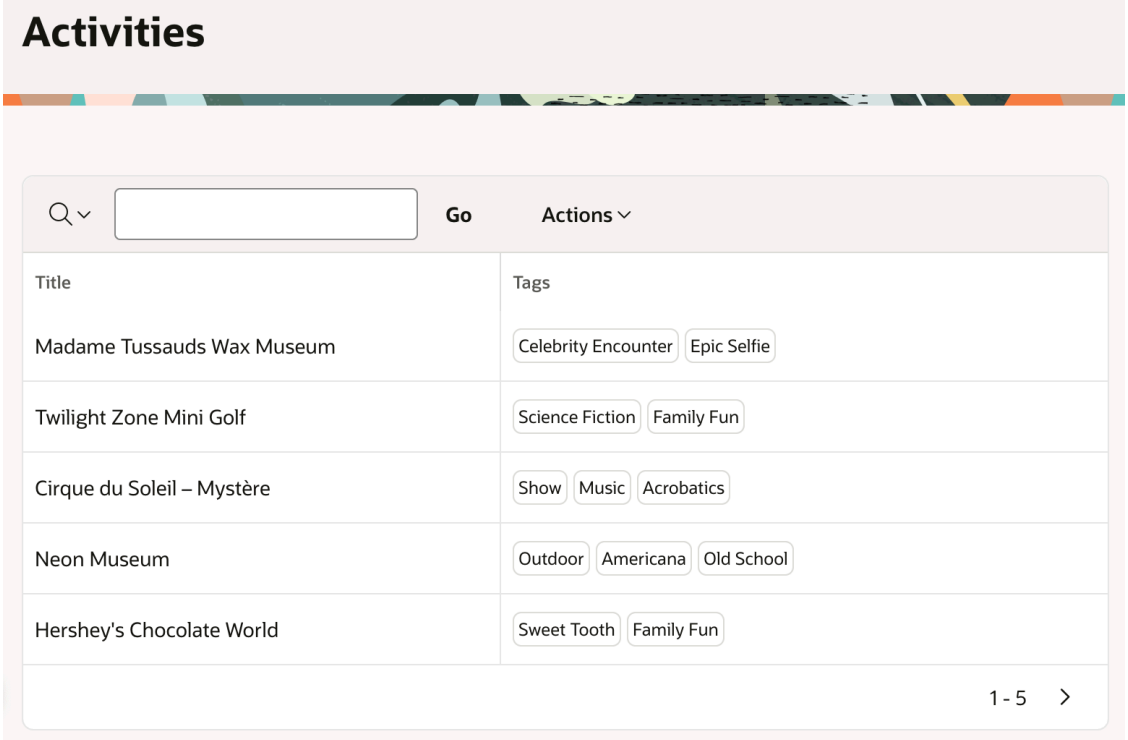
In an Interactive Report based on an `ACTIVITIES` tables, suppose a `TAGS` column contains a *colon-delimited* list of tag names. To format each tag as a **chip**, you can use the `{loop/}` directive in the column's HTML expression like this:

```
{loop ":" TAGS/}<span class="a-Chip">&APEX$ITEM.</span>{endloop/}
```

This produces the output you see below in the Tags column. The text of each tag appears inside a separate rounded rectangle in each row of the report.

Copyright © 2026, Oracle and/or its affiliates.

Figure 4-1 Using {loop/} Directive to Format Tags as Chips



APEX offers the template directives listed below. For more information, see Using Template Directives in *Oracle APEX App Builder User's Guide*.

Table 4-1 Available Template Directives

Purpose	Template Directive
Include contents when *NAME* has a value, or does *not* using optional {else/}.	{if *NAME*/} … {else/} … {endif/}

Include contents when value of *NAME* matches one of the {when/} directives, or when it *doesn't* match any of them using optional {otherwise/}.	{case *NAME*/}
	  {when X/} …
	  {when Y/} …
	  {otherwise/} …
	{endcase/}

Split *NAME* on delimiter (e.g. ",") and include contents for each value in the list using &APEX$ITEM. (or #APEX$ITEM#) and include index position in the loop using APEX$I (or #APEX$I#).	{loop "," *NAME*/} … &APEX$ITEM. … &APEX$I. … {endloop/}

Copyright © 2026, Oracle and/or its affiliates.

Table 4-1 (Cont.) Available Template Directives

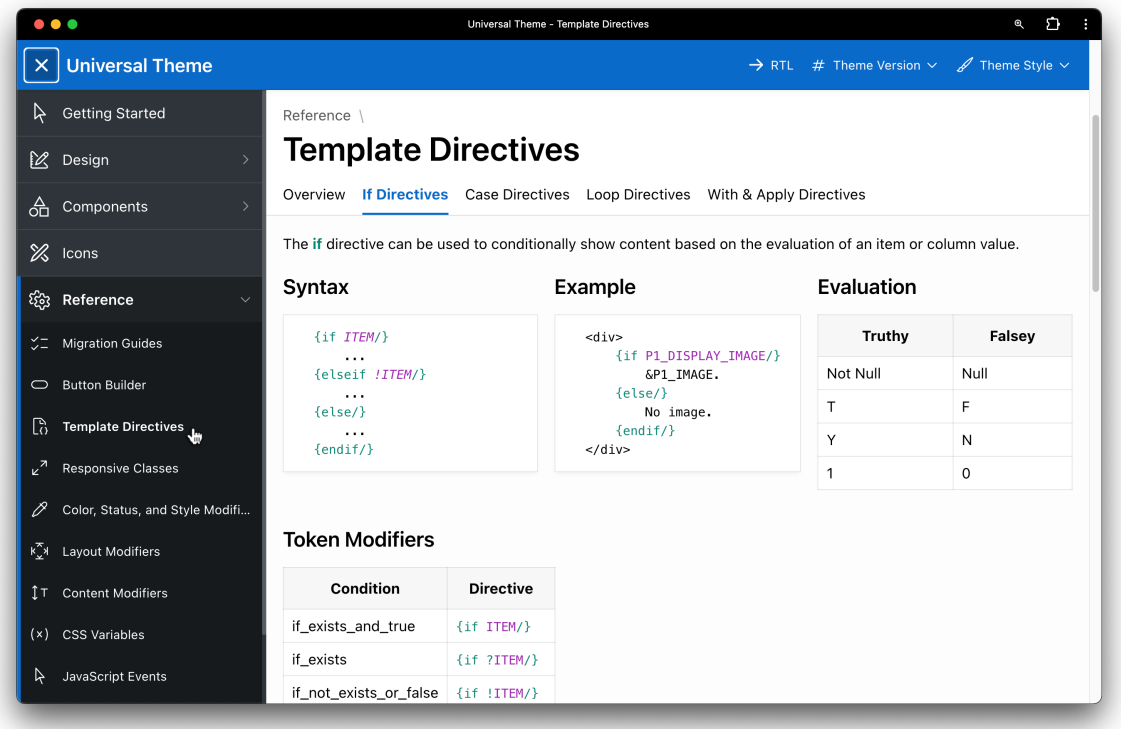
Purpose	Template Directive
Include the output of applying template *TEMPLATENAME* with *value1* and value of *NAME* data value for custom attributes *ATTR1* and *ATTR2*, respectively. Depending on context, use either #NAME# or &NAME. to reference data values as attribute values.	{with/} *ATTR1*:=*value1* *ATTR2*:=#*NAME*# {apply *TEMPLATENAME*/}

4.3 Directives Reference in Your Dock

The Universal Theme reference app contains examples of each template directive.

Visit and bookmark the *Universal Theme* reference app at oracleapex.com/ut. It supports *Progressive Web App* installation, so you can launch it anytime like a native app from your Dock or TaskBar. As shown below, the *Template Directives* page in its *Reference* section has examples of each template directive along with additional information about supported options.

Figure 4-2 Template Directives in Universal Theme Reference App as PWA



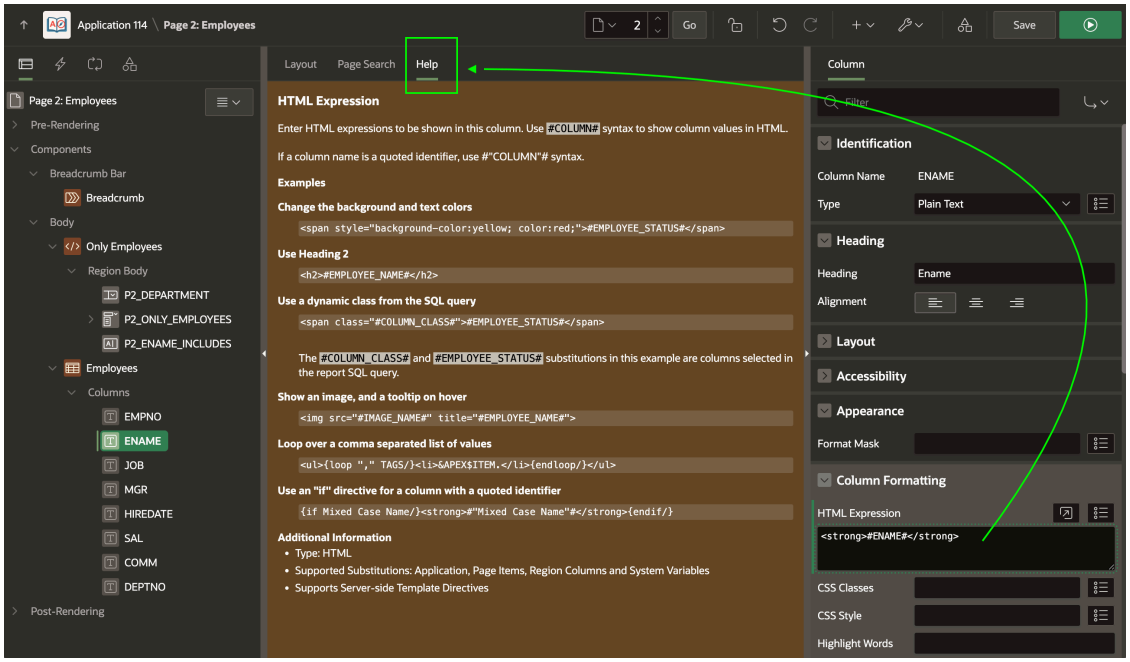
4.4 Using the Context-Sensitive Help Tab

The context-sensitive **Help** tab in Page Designer is a handy resource. It's in the same tab group as the **Layout** tab by default.

Copyright © 2026, Oracle and/or its affiliates.

For any property you select in the Property Editor, the **Help** tab shows you the substitutions the property supports and whether it allows Template Directives. Some entries also have helpful examples as shown below. In other editing pages in App Builder, click the help icon next to any field for similar information.

Figure 4-3 Help Tab Provides Content-Sensitive Assistance



4.5 Using Data Values in SQL and PL/SQL

Depending on the context, in your SQL and PL/SQL you use either `:NAME` bind variable syntax or `GET_TYPE()` functions in the `APEX_SESSION_STATE` package to use data values by name.

These include the values of page items, application items, and the names of built-in substitution strings like `APP_USER`, `APP_ID`, `APP_PAGE_ID`, and many others. For more information on the predefined names you can reference, see Using Available Built-in Substitution Strings in *Oracle APEX App Builder User's Guide*

- Working with Page and App Data Values
  In the SQL and the PL/SQL you write directly in APEX pages and component definitions, use `:NAME` bind variable syntax to reference the data value of a page item or application item by name.
- Working with Column Data Values
  When your SQL or PL/SQL processes rows for an Interactive Grid, in addition to page and application items you can also reference a *column* value in the current row using a `:COLUMNAME` bind variable.
- Values Bind as Strings
  Bind variables in your SQL and PL/SQL are always treated as strings, except when using database 26ai and explicitly setting a page item's data type to `BOOLEAN`.
- Using Data Values in Named Program Units
  In named program units like functions and procedures, to make code more reusable, you can accept values the business logic depends on as **input parameters**. Then pages

Copyright © 2026, Oracle and/or its affiliates.

invoking the function or procedure can pass in appropriate values by referencing page items.

- Configure Items to Submit and Return
  Whenever you reference a page item value as a bind variable in a region data source query or `WHERE` clause, make sure to list its name in the region's **Page Items to Submit** property.

4.5.1 Working with Page and App Data Values

In the SQL and the PL/SQL you write directly in APEX pages and component definitions, use `:NAME` bind variable syntax to reference the data value of a page item or application item by name.

Some common examples include using bind variables in:

- region SQL queries
- `WHERE` clauses
- local and shared component list of values
- validations
- page processes and workflow activities
- server-side condition expressions, and
- dynamic actions of type **Execute Server-side Code**.

Outside of data definition language (DDL) statements, anywhere SQL or PL/SQL allows an expression, you can use a bind variable instead. In PL/SQL, to change the value of a page item or application item use `:=` to assign a value to a `:NAME` bind variable. For example, if business logic in page 5 needs to change the value of the `P5_JOB` page item to `ANALYST` if it currently has the value `CLERK`, then the PL/SQL code looks like this:

```
-- Change clerk to analyst
IF :P5_JOB = 'CLERK' THEN
   :P5_JOB := 'ANALYST';
END IF;
```

4.5.2 Working with Column Data Values

When your SQL or PL/SQL processes rows for an Interactive Grid, in addition to page and application items you can also reference a *column* value in the current row using a `:COLUMNAME` bind variable.

This includes grid-related usages like:

- validations for a grid column
- page processes with their **Editable Region** property set to a grid, and
- dynamic actions of type **Execute Server-side Code** reacting to grid column value change.

In the dynamic action case, remember to mention the column names you're referencing in the **Items to Submit** and **Items to Return** properties. Those values are a comma-separated list of one or more column names *without* the preceding colon in the name.

In the page processing use case for new or modified grid rows, in addition to being able to change the value of a page item or application item, you can use a bind variable to set the

Copyright © 2026, Oracle and/or its affiliates.

value of a column by assigning an expression to a `:COLUMNNAME` bind variable using `:=` as shown below:

```
-- Change clerk to analyst in current grid row being processed
IF :JOB = 'CLERK' THEN
   :JOB := 'ANALYST';
END IF;
```

4.5.3 Values Bind as Strings

Bind variables in your SQL and PL/SQL are always treated as strings, except when using database 26ai and explicitly setting a page item's data type to `BOOLEAN`.

Even when you configure a page item like `P5_START_DATE` as a **Date Picker** or define a `P5_MAX_QUANTITY` item as a **Number Field**, APEX binds your references to `:P5_START_DATE` and `:P5_MAX_QUANTITY` as `VARCHAR2` string values. There are two exceptions to this rule. You can optionally configure the data type to be:

- `CLOB` for **Hidden**, **Text Area**, and **Rich Text Editor** page items
- `BOOLEAN` for *Switch* and *Checkbox* page items.
- Working with Data Values as Numbers
  If a page item contains a numeric value, using its value as a `VARCHAR2`-typed bind variable is fine when the database performs automatic type conversion.
- Working with Data Values as Dates
  If a page item contains a date value, then using its value as a `VARCHAR2`-typed bind variable can raise an error if the page item uses a different format mask than the application default.
- Working with Delimited Values
  When a page item contains multiple value separated by a delimiter like colon (`:`) or comma (`,`) you can use its multiple values in SQL or PL/SQL.

4.5.3.1 Working with Data Values as Numbers

If a page item contains a numeric value, using its value as a `VARCHAR2`-typed bind variable is fine when the database performs automatic type conversion.

For example, in a `WHERE` clause like the following, the database implicitly converts the string value of `:P5_MAX_QUANTITY` to a number to compare it with a number column like `QUANTITY`:

```
QUANTITY BETWEEN 1 AND :P5_MAX_QUANTITY
```

However, in other situations like a `CASE` statement, referencing `:P5_MAX_QUANTITY` in a position where the database expects a number produces an error. For example, consider the following usage:

```
QUANTITY BETWEEN 1 AND CASE CATEGORY
                          WHEN 'SMALL' THEN 10
                          WHEN 'LARGE' THEN :P5_MAX_QUANTITY
                          ELSE              15
                       END
```


Copyright © 2026, Oracle and/or its affiliates.

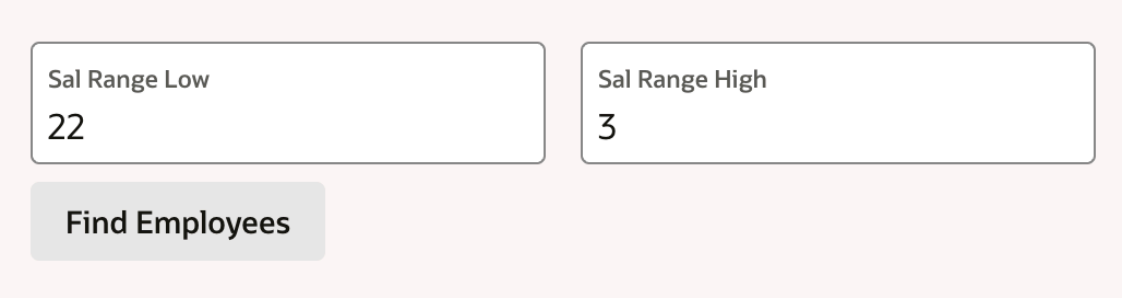
It produces the error below at runtime:

```
ORA-00932: expression (:1) is of data type CHAR,
           which is incompatible with expected data type NUMBER
```

The solution is to reference the value as a number using the `GET_NUMBER()` function in the `APEX_SESSION_STATE` package. This automatically accounts for any format mask you might have configured on the page item to correctly convert the value to a number result.

Another common situation is comparing the value of two page items containing numbers. Consider the following query by example page featuring a `P1_SAL_RANGE_LOW` and `P1_SAL_RANGE_HIGH` page items of type **Number Field**.

Figure 4-4 Numeric Range Using Two Number Field Page Items



If you define a validator **Check Salary Range** with the PL/SQL expression below, the user submitting a low value of 22 and a high value of 3 does *not* receive a validation error as expected. This happens because the string values containing digits 22 and 3 get compared in text order rather than numeric order.

```
:P1_SAL_RANGE_LOW <= :P1_SAL_RANGE_HIGH
```

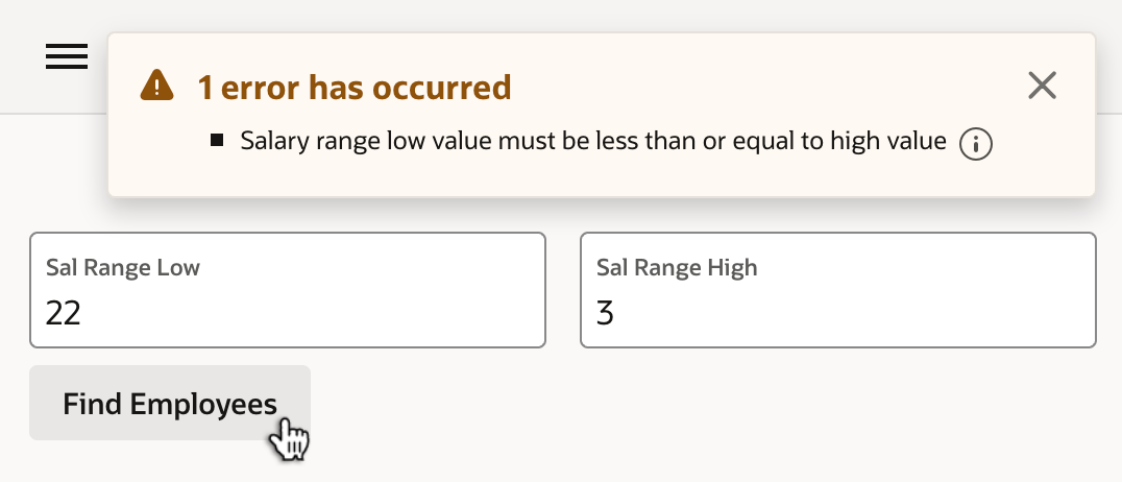
To have the values compared as numbers, rewrite the validation rule as follows. Notice the bind variable notation is replaced by passing the name of the page item to the `GET_NUMBER()` function.

```
apex_session_state.get_number('P1_SAL_RANGE_LOW')
 <= apex_session_state.get_number('P1_SAL_RANGE_HIGH')
```

With this solution in place, clicking **Find Employees** now produces the expected validation error as shown below, "Salary range low value must be less than or equal to high value."

Copyright © 2026, Oracle and/or its affiliates.

Figure 4-5 Expected Validation Error Treating Numeric Items as Numbers



Tip

When working with number values of *Interactive Grid columns*, convert the column's `VARCHAR2` value to a number using the `to_number` function with an appropriate format mask.

4.5.3.2 Working with Data Values as Dates

If a page item contains a date value, then using its value as a `VARCHAR2`-typed bind variable can raise an error if the page item uses a different format mask than the application default.

For example, consider the following where clause using the value of a **Date Picker** page item as a bind variable:

```
HIREDATE < :P34_HIRED_BEFORE
```

If the page item uses format mask `DD-MON-YYYY` and the application default is `DS` for the short date format of the current locale, then the page encounters the error below at runtime:

```
ORA-01858: A non-numeric character was found instead of a numeric character.
```

The solution is to reference the value as a date using the `GET_TIMESTAMP()` function in the `APEX_SESSION_STATE` package like this. Notice the bind variable notation is replaced by passing the *name* of the page item to the `GET_TIMESTAMP()` function.

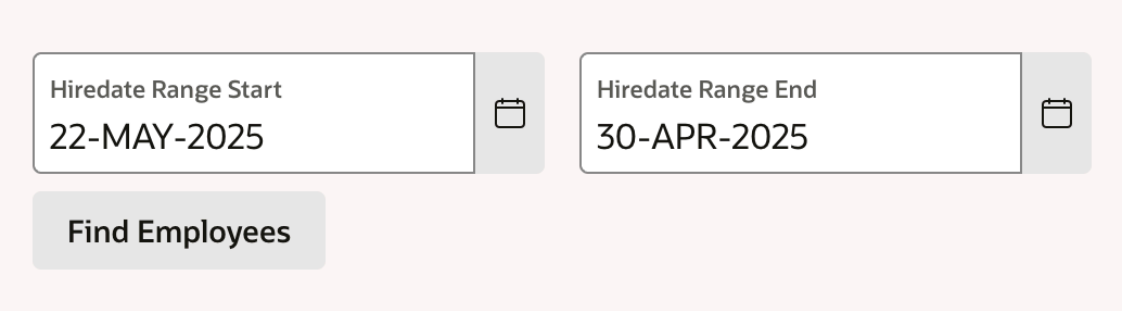
```
HIREDATE < apex_session_state.get_timestamp('P34_HIRED_BEFORE')
```

This automatically accounts for any format mask you might have configured on the page item to correctly convert the value to a date result. Forgetting this can also lead to unexpected results, especially when comparing date values.

Consider the following query by example page featuring a `P1_HIREDATE_RANGE_START` and `P1_HIREDATE_RANGE_END` page items of type **Date Picker**.

Copyright © 2026, Oracle and/or its affiliates.

Figure 4-6 Date Range Using Two Date Picker Page Items



If you define a validator **Check Hiredate Range** with the PL/SQL expression below, the user submitting a low value of `22-MAY-2025` and a high value of `30-APR-2025` does *not* receive a validation error as expected. This happens because the two string values get compared in text order rather than date order.

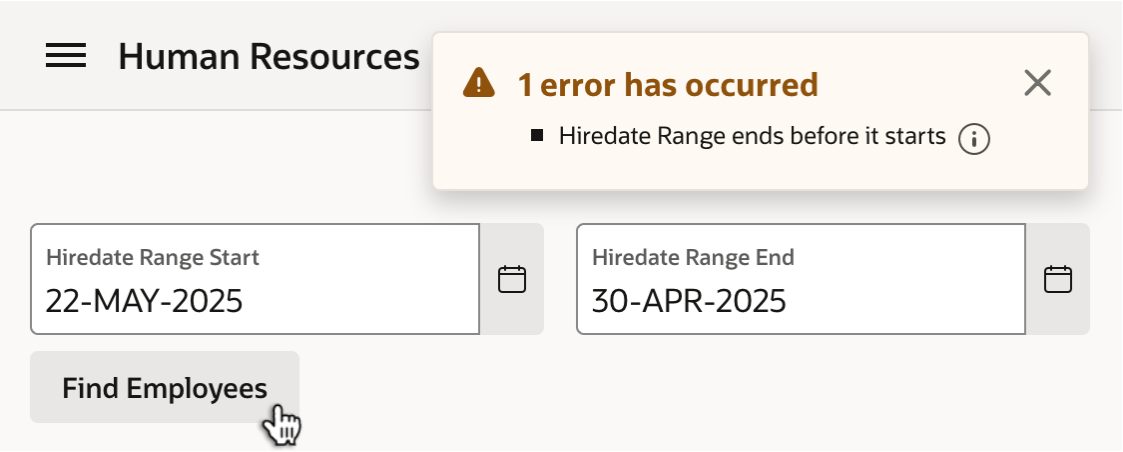
```
:P1_HIREDATE_RANGE_START <= :P1_HIREDATE_RANGE_END
```

To have the values compared as dates, rewrite the validation rule as follows:

```
apex_session_state.get_timestamp('P1_HIREDATE_RANGE_START')
 <= apex_session_state.get_timestamp('P1_HIREDATE_RANGE_END')
```

With this solution in place, clicking **Find Employees** now produces the expected validation error as shown below, "Hiredate Range ends before it starts."

Figure 4-7 Expected Validation Error Treating Date Items as Dates



Tip

When working with date or timestamp values of *Interactive Grid columns*, convert the column's `VARCHAR2` value to a date using the `to_date` or `to_timestamp` function with an appropriate format mask.

Copyright © 2026, Oracle and/or its affiliates.

4.5.3.3 Working with Delimited Values

When a page item contains multiple value separated by a delimiter like colon (`:`) or comma (`,`) you can use its multiple values in SQL or PL/SQL.

- Working with Delimited Number Values
  To work with delimited number values in SQL or PL/SQL, use the `apex_string.split_numbers()` function.
- Working with Delimited String Values
  To work delimited values in SQL or PL/SQL, use the `apex_string.split()` function.

4.5.3.3.1 Working with Delimited Number Values

To work with delimited number values in SQL or PL/SQL, use the `apex_string.split_numbers()` function.

Suppose `P5_SELECTED_EMPNOS` is a **Select Many** page item with colon (`:`) as delimiter. If you configure its list of values using a query with `ENAME` as the display value and `EMPNO` as the return value, then after the user selects multiple employees the value of `P5_SELECTED_EMPNOS` might look like the string "`7839:7369:7654`".

An initial thought might be to use the bind variable as follows:

```
select empno, ename, sal
  from emp
 where empno in (:P5_SELECTED_EMPNOS) /* Tempting, but problematic */
```

However, this technique fails when there are multiple delimited values in the page item. It would produce a runtime error like this:

```
ORA-01722: unable to convert string value containing ':' to a number:
ORA-03302: (ORA-01722 details) invalid string value: 7839:7369:7654
```

The solution involves *splitting* the string into its separate values using the `SPLIT_NUMBERS()` function in the `APEX_STRING` package. Then you can select the `COLUMN_VALUE` from its single-column table result. Notice the example below passes the colon (`':'`) as the second argument to indicate the value delimiter in use:

```
select empno, ename, sal
  from emp
 where empno in (select column_value
                   from apex_string.split_numbers(:P5_SELECTED_EMPNOS,':'))
```

To work with the multiple values in PL/SQL you can define a variable of type `apex_t_number`. This is APEX's built-in list of numbers type. Then you can call the same `split_numbers()` function:

```
declare
   l_selected_empnos apex_t_number;
begin
   l_selected_empnos := apex_string.split_numbers(:P5_SELECTED_EMPNOS,':');
   for j in 1..l_selected_empnos.count loop
```


Copyright © 2026, Oracle and/or its affiliates.

```
      -- Use the j-th employee number here
      -- l_selected_empnos(j)
   end loop;
end;
```

4.5.3.3.2 Working with Delimited String Values

To work delimited values in SQL or PL/SQL, use the `apex_string.split()` function.

Consider a **Select Many** page item named `P5_SELECTED_SIZES` that uses the vertical bar (`|`) as delimiter. If you configured its list of values using a query with `SIZE_DESCRIPTION` as the display value and `SIZE_CODE` as the return value, then after the user selects multiple sizes, the value of `P5_SELECTED_SIZES` might look like the string "`SM|MD|XXL`".

An initial thought might use the bind variable as follows:

```
select id, sku, price
  from items
 where item_size in (:P5_SELECTED_SIZES) /* Tempting, but problematic */
```

However, attempting this when there are multiple delimited values in the page item returns no rows because this syntax only matches an item row with an `ITEM_SIZE` value of the literal value "`SM|MD|XXL`".

The solution involves *splitting* the string into its separate values using the `SPLIT()` function in the `APEX_STRING` package. Then you can select the `COLUMN_VALUE` from its single-column table result. Notice the example below passes the vertical bar (`'|'`) as the second argument to indicate the value delimiter in use:

```
select id, sku, price
  from items
 where item_size in (select column_value
                       from apex_string.split(:P5_SELECTED_SIZES,'|'))
```

To work with the multiple values in PL/SQL, you can define a variable of type `apex_t_varchar2`. This is APEX's built-in list of strings type. Then you can call the same `split()` function:

```
declare
   l_selected_sizes apex_t_varchar2;
begin
   l_selected_sizes := apex_string.split(:P5_SELECTED_SIZES,'|');
   for j in 1..l_selected_sizes.count loop
      -- Use the j-th size code here
      -- l_selected_sizes(j)
   end loop;
end;
```

4.5.4 Using Data Values in Named Program Units

In named program units like functions and procedures, to make code more reusable, you can accept values the business logic depends on as **input parameters**. Then pages invoking the function or procedure can pass in appropriate values by referencing page items.

Copyright © 2026, Oracle and/or its affiliates.

In cases where that's not a priority, use the `GET_TYPE()` routines in the `APEX_SESSION_STATE` package to access the value of page items, application items, or built-in substitution string names. To work with data values as strings in your function and procedure code, the `V()` function is identical to `apex_session_state.get_varchar2()` and fewer characters to type.

4.5.5 Configure Items to Submit and Return

Whenever you reference a page item value as a bind variable in a region data source query or `WHERE` clause, make sure to list its name in the region's **Page Items to Submit** property.

If you reference multiple page items as bind variables, list each name separated by commas in the value of this property. While you precede the page item name with a colon when using it as a bind variable, for this region property, page item names without the colon. Failure to remember this step can result in an application malfunction. When you forget to include a page item's name, its bind variable inadvertently evaluates to `null`. This can cause a query to return no rows, or business logic to take the wrong execution path.

Similarly, when referencing a page item or grid column value as a bind variable in a Dynamic Action of type **Execute Server-side Code**, ensure you set the **Items to Submit** and **Items to Return** properties appropriately. Forgetting to do this can produce **No Data Found** errors or make your page appear to function incorrectly when server-side computed values are not returned to the page as expected.

Copyright © 2026, Oracle and/or its affiliates.

5

Visualizing and Analyzing Data

Oracle APEX's native region components simplify visualizing and analyzing data.

After choosing the region type that fits your requirements, configure the data it should use and adjust its settings in the Property Editor to fine-tune look and behavior. Consider installing the **Sample Reports** app from the Gallery to study additional useful examples that employ the different kinds of regions described in this section.

Using the **Refresh** dynamic action, you can refresh any of the regions in response to a user action to show updated data.

> **Tip**
>
> Sometimes your region's `WHERE` clause or SQL query references page item values as bind variables, for example `:P1_SAL < :P1_COMM`. Regardless of the region type, always make sure to list the names of these page items like `P1_SAL,P1_COMM` in the region's **Page Items to Submit** property. This ensures APEX refreshes the data using the latest values the end user set in the page. Failure to do so leaves their values `null`. This can inadvertently cause a query to return no data.

- Listing Data with Classic Reports
  The **Classic Report** region displays data in a table, with optional pagination.
- Personalizable Interactive Reports
  The **Interactive Report** region is a tabular display that lets users filter, sort, highlight, format, compute, aggregate, group, pivot, chart, download, and email the data you configure it to work with.
- Flexibly Styling and Acting on Data
  The *Content Row* region lets you flexibly style each row of data.
- Info Tiles Using Cards
  To show results as information tiles, use a *Cards* region.
- Info at a Glance with Charts & Dashboards
  Use a **Chart** region to present data visually using one of 18 different styles. You can add multiple charts on a page to create a dashboard. Most chart types can show multiple series of data.
- Showing Data on Maps
  Use a **Map** region to display data related to specific locations or areas on the earth. One map can show multiple layers when needed. For each layer you configure a table, SQL query, or other data source, and choose a layer type.
- Placing Events on a Calendar
  Use a *Calendar* region to display scheduled events with a month, week, day, or list view.
- Exploring Hierarchical Data in a Tree
  Use the *Tree* region to show hierarchical data like the management chain of employees in the `EMP` table. APEX can manage the hierarchy so you only need to configure the data to display in each tree node.

Copyright © 2026, Oracle and/or its affiliates.

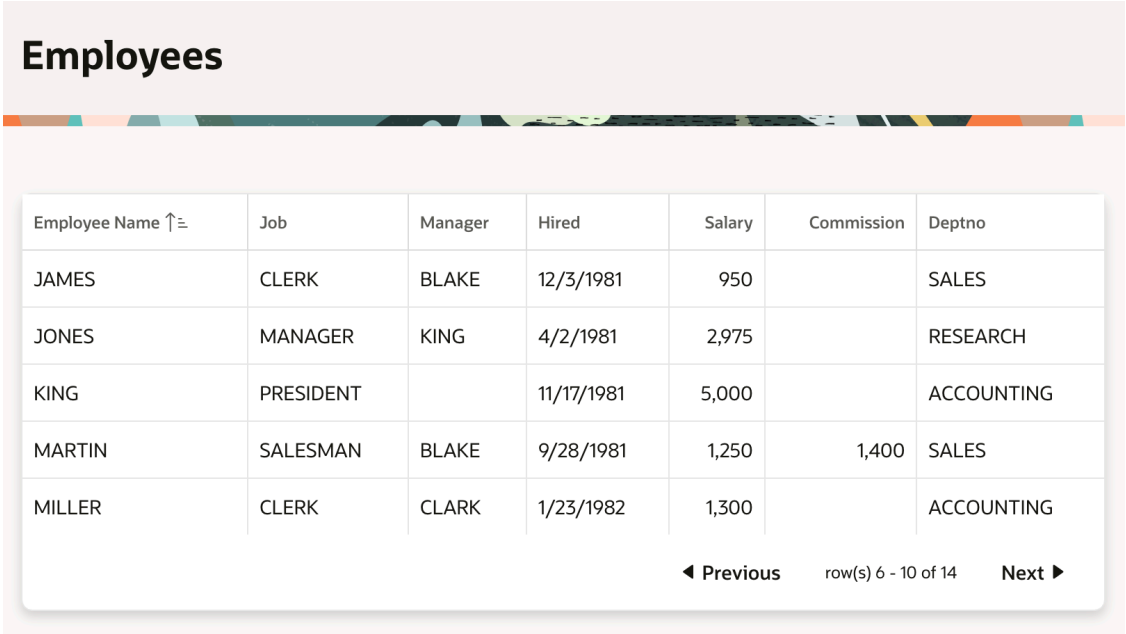
- Dynamically Formatting Data
  When none of APEX's other region types meet your requirements, the *Dynamic Content* region lets you programmatically generate all its HTML markup. Your code constructs a `CLOB` containing any mix of tags and data and returns it for the APEX engine to include in the appropriate place in the page.

5.1 Listing Data with Classic Reports

The **Classic Report** region displays data in a table, with optional pagination.

If the number of rows is reliably small, you can show them all at once. When the result set may be larger, configure pagination to efficiently retrieve rows page by page. By adjusting a column's settings in Page Designer, you control its display type, heading, format mask, sort behavior, alignment, and more. For a column of type **Plain Text**, you can use an **HTML Expression** to customize how it renders. You can mix HTML tags, conditional formatting directives, and column values, referencing even hidden ones using `#SOME_COLUMN_NAME#`. As shown below, when the value of a column like *Deptno* is a foreign key, set its type to **Plain Text (based on List of Values)** to have APEX automatically render its user-friendly display value instead.

Figure 5-1 Classic Report Region Showing Employees Data with Pagination



The **Standard** report template is the default, but the Universal Theme Reference App at oracleapex.com/ut shows other available templates like **Value Attribute Pairs**, **Media List**, **Search Results**, **Timeline**, and more. When none of the built-in templates meet your needs, you can create custom templates to format the rows and columns of data in a classic report in any way necessary.

For more information, see Managing Classic Reports in *Oracle APEX App Builder User's Guide*.

Copyright © 2026, Oracle and/or its affiliates.

5.2 Personalizable Interactive Reports

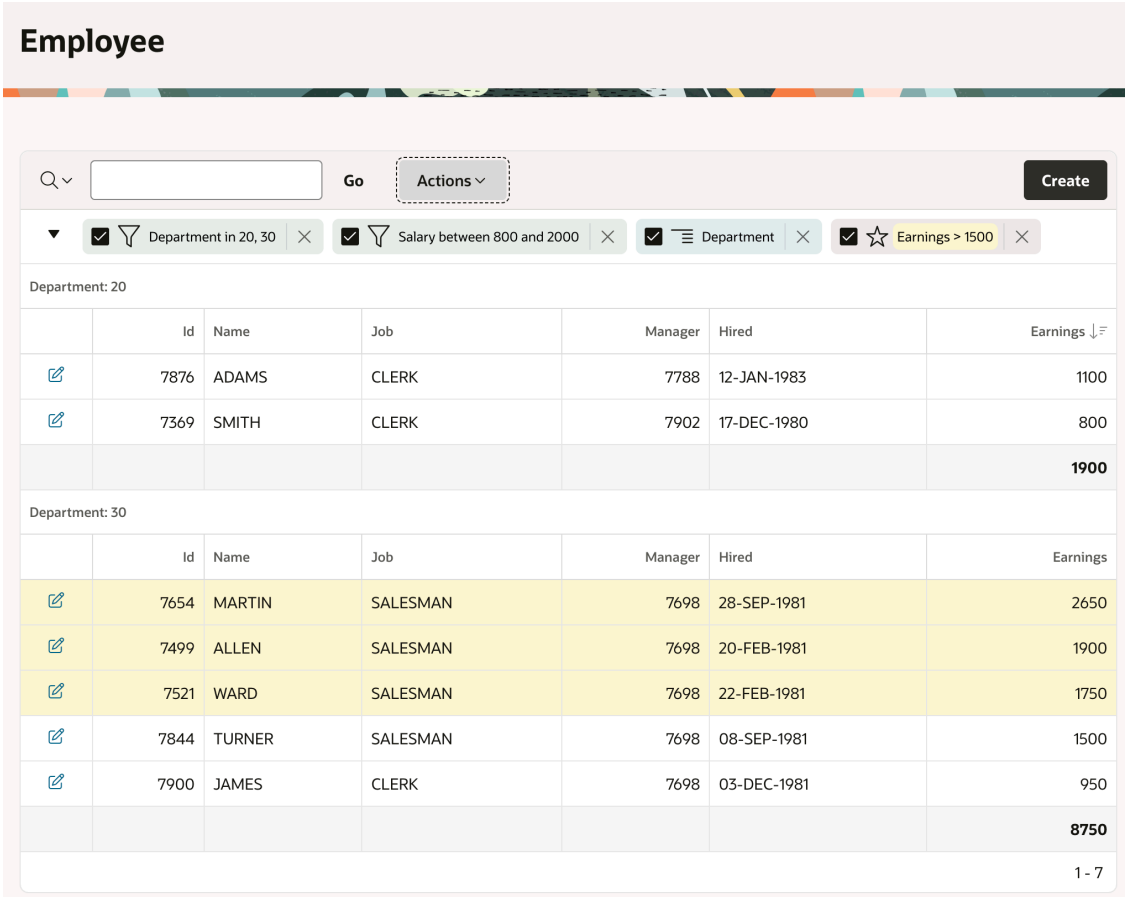
The **Interactive Report** region is a tabular display that lets users filter, sort, highlight, format, compute, aggregate, group, pivot, chart, download, and email the data you configure it to work with.

For each interactive report, you control the features end users can access. You can also pass **dynamic** filters when linking to a target page to proactively show the user only the rows they need to see.

The figure below shows an *Employees* interactive report region where the user has:

- added a computed *Earnings* column whose formula is *Salary* plus *Commission*
- hidden the *Salary* and *Department* columns
- defined an aggregate to show the sum of *Earnings*
- sorted the data by *Earnings* from highest to lowest
- filtered the results to show only departments 20 and 30 with salaries between 800 and 2000
- highlighted rows with *Earnings* greater than 1500, and
- formatted the results to list employees in each department in a separate section.

Figure 5-2 Filtered, Highlighted, Grouped Employees with Computed and Aggregate Fields



Copyright © 2026, Oracle and/or its affiliates.

When the type of an interactive report column is **Plain Text**, you can use an **HTML Expression** to customize how it renders. You can mix HTML tags, conditional formatting directives, and column values, referencing even *hidden* ones using `#SOME_COLUMN_NAME#`.

By default, your interactive report region has a *primary* default report configuration. Running your page from App Builder, you can set up all aspects of the primary report and save it as the **primary** default report for all users. This includes which columns should display by default, the order of the columns, the default sort order, and all other features available to configure.

You can also define and save one or more **alternative** default reports with a name that users can select from a list in the toolbar when multiple options exist. Any alternative default reports show in Page Designer under the **Saved Reports** heading in the component tree.

End users can also configure and save **private** reports for any number of reports they run frequently. The private reports show up in the same list as the public default reports, but each user only sees the ones they have created. They can also email any report on demand or subscribe to periodically receive an email containing a selected report.

For more information, see Managing Interactive Reports and Linking to Interactive Reports in *Oracle APEX App Builder User's Guide*.

5.3 Flexibly Styling and Acting on Data

The *Content Row* region lets you flexibly style each row of data.

For each of its different display slots, you can mix HTML tags, conditional formatting directives, and column values, referencing even hidden ones using `&SOME_COLUMN_NAME.`. The region in the figure below uses `&ENAME.` in the **Title** slot, `&ICON.` in the **Avatar Icon** slot, and in the **Description** slot:

```
&JOB. in &DNAME. (&LOC.) since &HIREDATE.
```

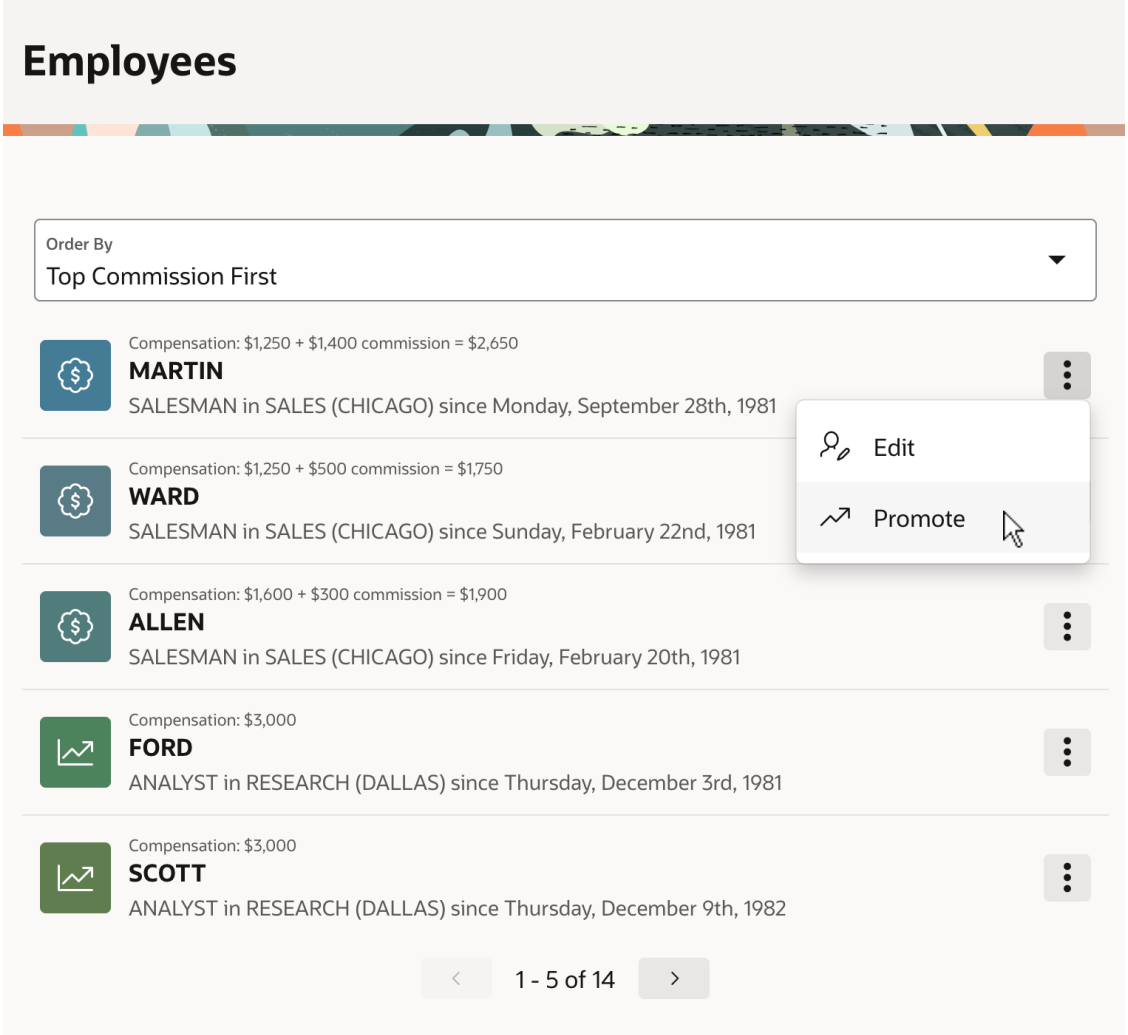
Its **Overline** slot contains the conditionally formatted expression:

```
Compensation: &SAL. {if COMM/} + &COMM. commission = &TOTAL_COMP.{endif/}
```

You can define actions like the *Edit* and *Promote* ones below to let end users take actions on any row. Due to the conditional formatting directive `{if COMM/}` above, notice that employees *FORD* and *SCOTT* who have no commission get formatted differently. Format masks for the `SAL`, `COMM`, `TOTAL_COMP`, and `HIREDATE` columns in the region's query are configured as needed in the column settings. In the figure, five employees appear per page with custom formatting and row-specific actions.

Copyright © 2026, Oracle and/or its affiliates.

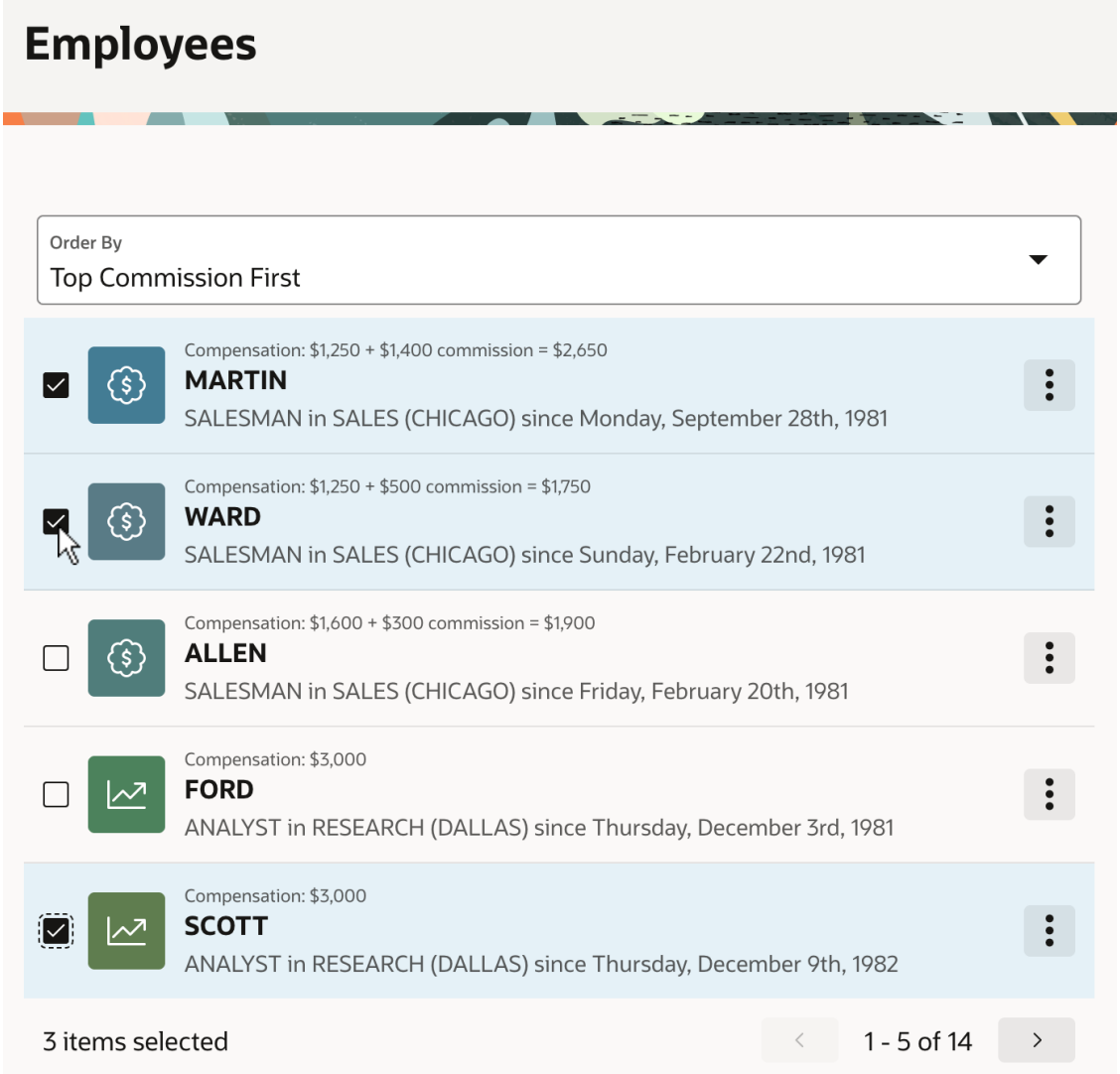
Figure 5-3 Content Row Region with Pagination and Action Menu Entries



When needed, you can enable single- or multi-row selection behavior and configure a hidden page item to contain the primary keys of the selected rows. The page is also using an Order By Item to let end users choose between a number of useful sorting orders. The figure below shows the same region after enabling these features with *MARTIN*, *WARD*, and *SCOTT* selected. For more information, see Managing Sorting with Order By Item in *Oracle APEX App Builder User's Guide*.

Copyright © 2026, Oracle and/or its affiliates.

Figure 5-4 Content Row Configured for Multiple Row Selection



5.4 Info Tiles Using Cards

To show results as information tiles, use a *Cards* region.

For each of various card slots, you can choose the column to appear or configure an HTML expression that mixes HTML tags, conditional formatting directives, and column values using `&SOME_COLUMN_NAME.` notation. The region in the figure below uses `ENAME` in the **Title** slot, `ICON` as the icon column, and in the **Body** slot:

```
&JOB. in &DNAME. (&LOC.)<br>
since &HIREDATE.
```

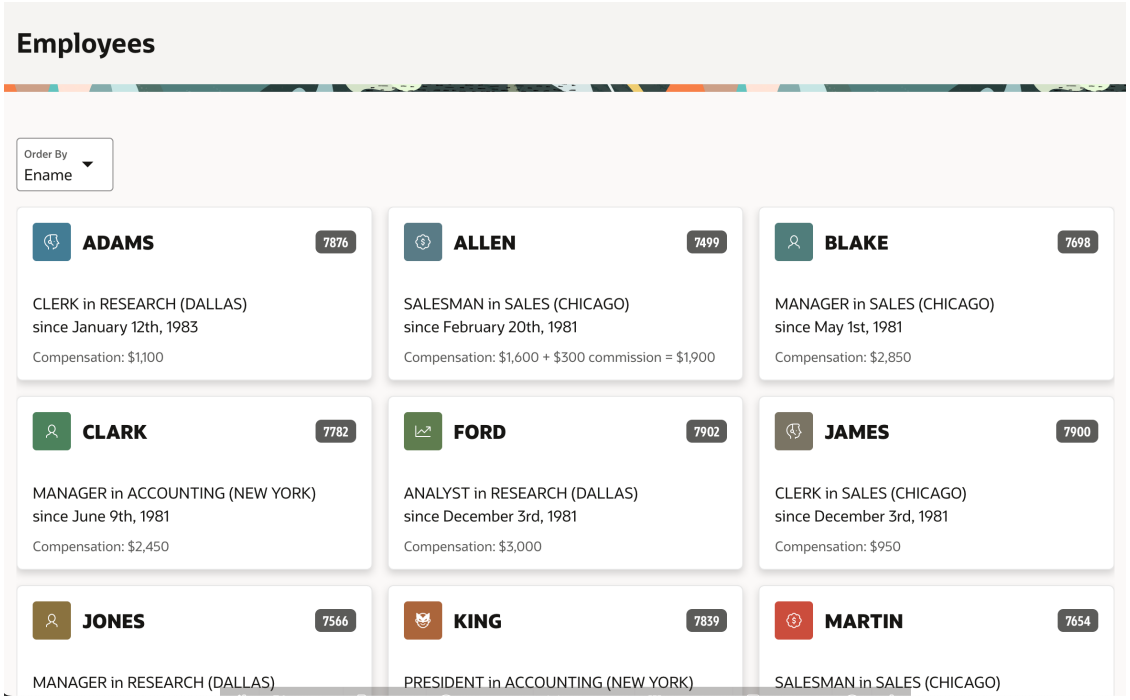
Its **Secondary Body** slot contains the conditionally formatted expression:

```
Compensation: &SAL. {if COMM/} + &COMM. commission = &TOTAL_COMP.{endif/}
```


Copyright © 2026, Oracle and/or its affiliates.

You can define actions to trigger when the user clicks on the full card, on title or subtitle, or to appear as buttons on each card to let end users take actions on any row. Due to the conditional formatting directive `{if COMM/}` above, notice that employees who have no commission get formatted differently. Format masks for the `SAL`, `COMM`, `TOTAL_COMP`, and `HIREDATE` columns in the region's query are configured as needed in the SQL statement you write to determine the cards region's data source.

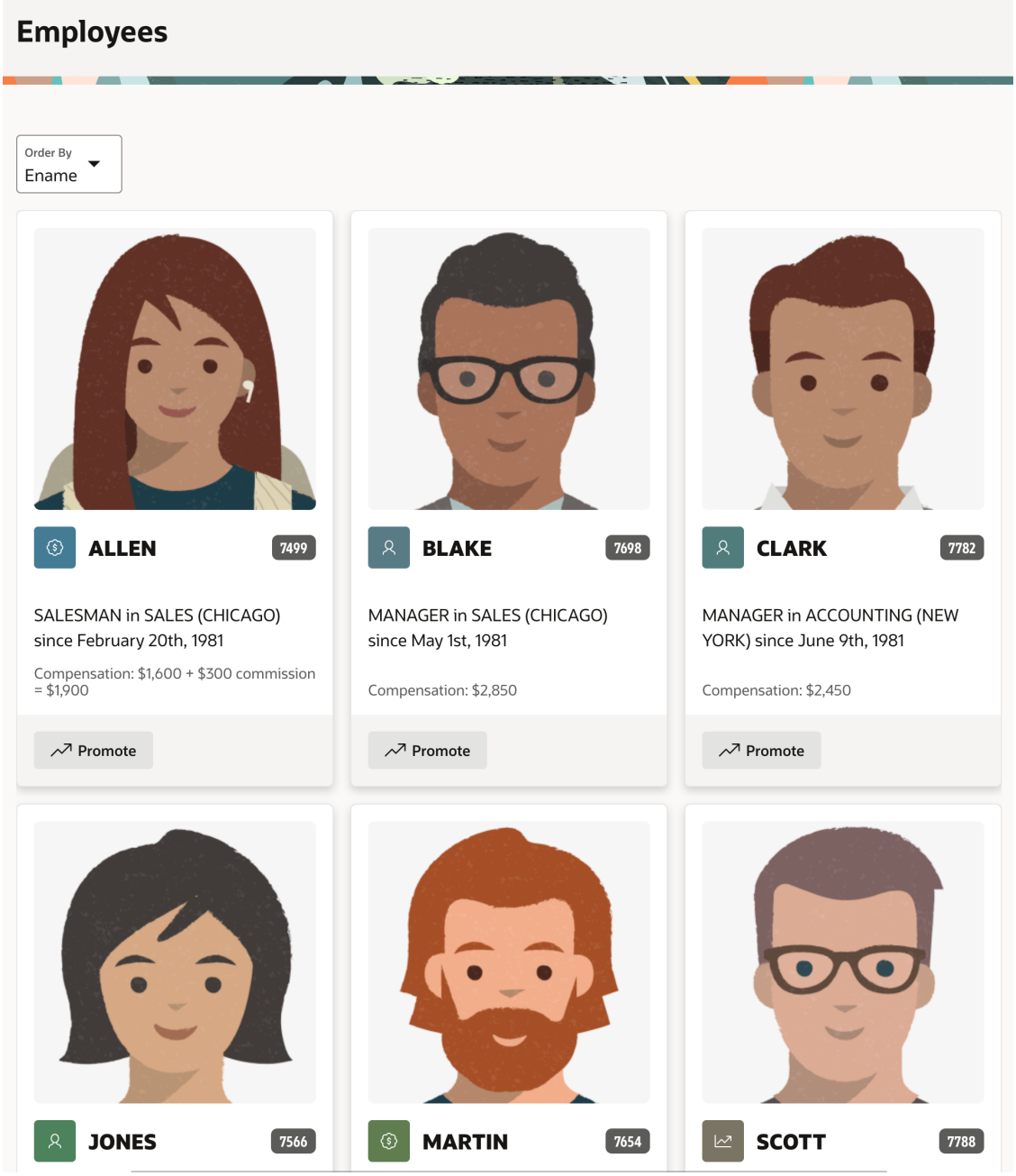
Figure 5-5 Employees Shown in a Cards Region



Your cards can also display images from `BLOB` columns or URLs, positioned and sized in various ways to complement the other content on the card, if any. When cards include media, actions can target a click on the image as well. If instead you configure them as buttons, they appear at the bottom of the card. The figure below shows results as a grid of information tiles, with a headshot image on each employee's card. Custom formatted data appears in multiple slots, and a **Promote** button shows on each tile. For more information, see Managing Cards in *Oracle APEX App Builder User's Guide*.

Copyright © 2026, Oracle and/or its affiliates.

Figure 5-6 Cards with Media and an Action Button Configured



5.5 Info at a Glance with Charts & Dashboards

Use a **Chart** region to present data visually using one of 18 different styles. You can add multiple charts on a page to create a dashboard. Most chart types can show multiple series of data.

- Choosing a Chart Type
  When creating a chart region, choose the chart type from the list shown in the Create Page wizard.

Copyright © 2026, Oracle and/or its affiliates.

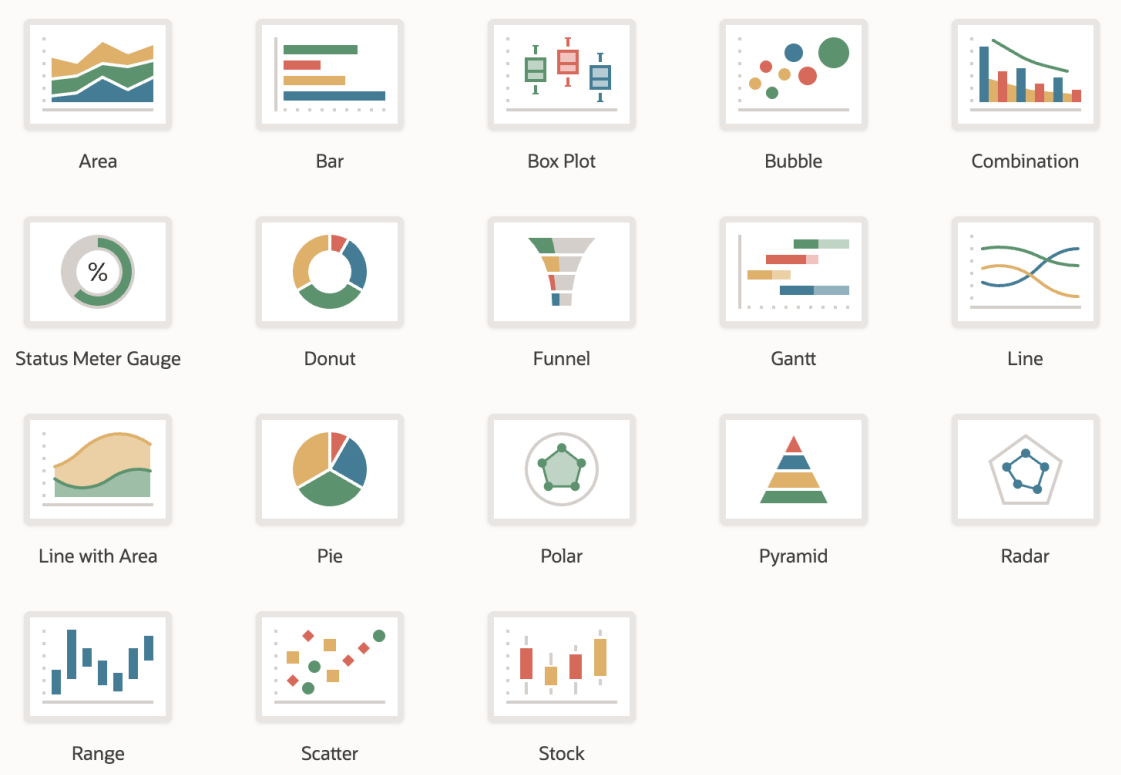
- Defining Chart Series
  Configure a table name, SQL query, or other data source to define a chart **series**. While *Gantt*, *Status Meter Gauge*, and *Stock* charts have a single series, the other types all support showing multiple series on the same chart.
- Including Related Data on a Chart
  Include related data on charts by using SQL.

5.5.1 Choosing a Chart Type

When creating a chart region, choose the chart type from the list shown in the Create Page wizard.

Supported chart types include Area, Bar, Box Plot, Bubble, Combination, Status Meter Gauge, Donut, Funnel, Gantt, Line, Line with Area, Pie, Polar, Pyramid, Radar, Range, Scatter, and Stock. To build a dashboard, just add multiple charts to a page and configure their grid layout like any other region. The figure shows the page of the Create Page wizard where you choose the type of chart to create, featuring a visual representation of every type.

Figure 5-7 Native Chart Types to Choose From for a Chart Region



5.5.2 Defining Chart Series

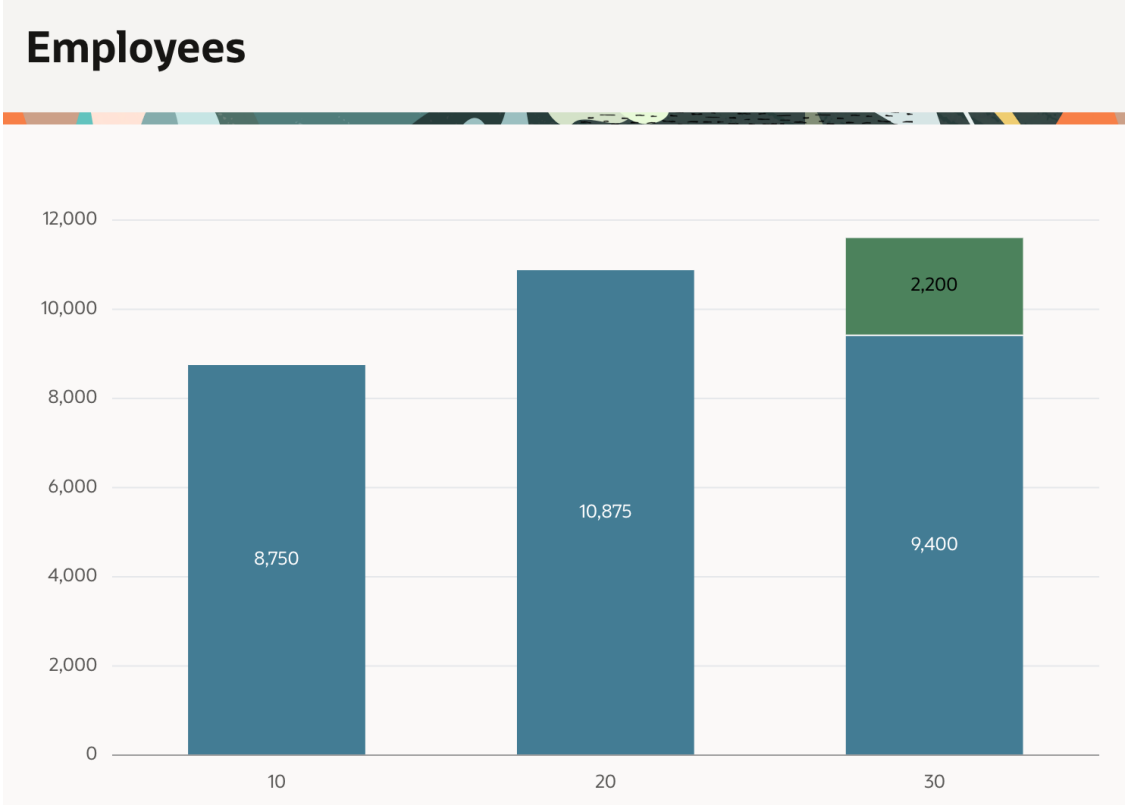
Configure a table name, SQL query, or other data source to define a chart **series**. While *Gantt*, *Status Meter Gauge*, and *Stock* charts have a single series, the other types all support showing multiple series on the same chart.

The figure below shows a bar chart showing the sum of employee salaries and commissions by department number (e.g. 10, 20, 30). By configuring the chart to **stack** its series, the bar

Copyright © 2026, Oracle and/or its affiliates.

representing department 30 commissions is placed atop the one for salaries instead of appearing next to it.

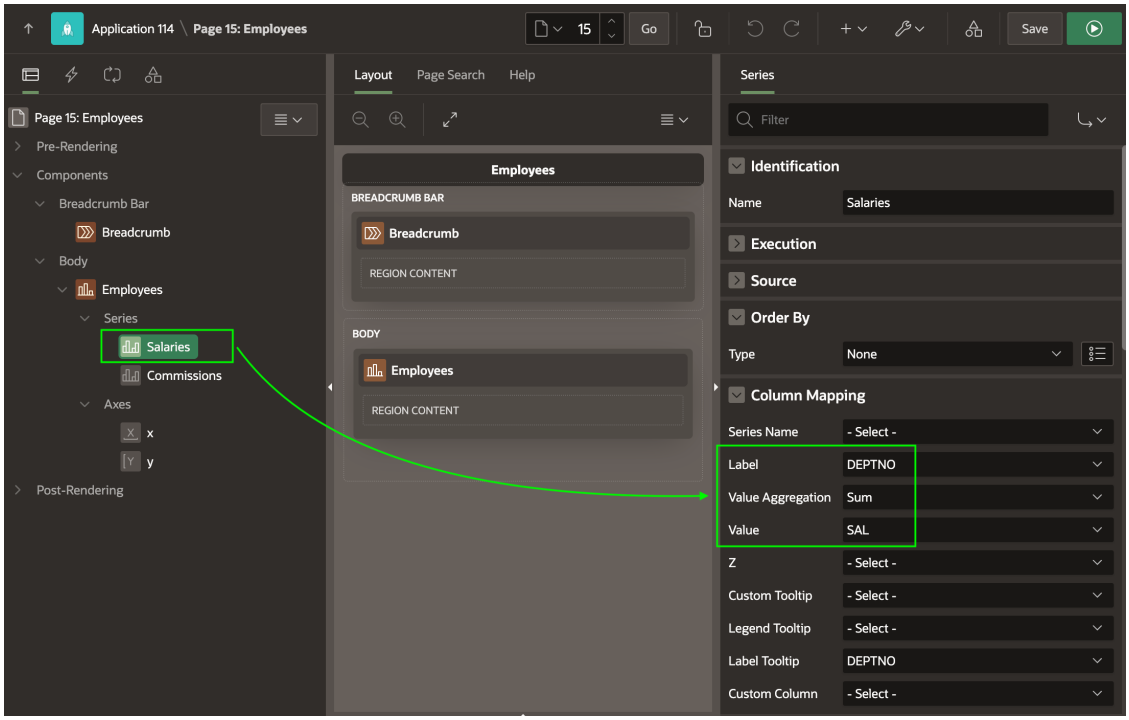
Figure 5-8 Sum of Employees Salaries and Commissions by Department



The figure below shows Page Designer with the *Salaries* series selected in the component tree. Both series use the `EMP` table as their data source. Notice in the **Column Mapping** section of the Property Editor that you configure label, value, and aggregations by choosing appropriate column names.

Copyright © 2026, Oracle and/or its affiliates.

Figure 5-9 Stacked Bar Chart in Page Designer



5.5.3 Including Related Data on a Chart

Include related data on charts by using SQL.

Showing the *name* of the department instead of the department number requires using a SQL Query to join the `EMP` and `DEPT` tables. You can switch the *Salaries* series to use a SQL Query like the following that joins the two tables on their related key value, and computes the `SUM` of the `SAL` values, grouping by department number:

```
SELECT E.DEPTNO,
       D.DNAME,
       SUM(E.SAL) AS TOTAL_SALARIES
  FROM EMP E
  JOIN DEPT D
    ON D.DEPTNO = E.DEPTNO
GROUP BY E.DEPTNO, D.DNAME
```

The *Commissions* series requires a similar query, but since `COMM` can be *null*, you need the `NVL()`function to provide a default value of zero (0) when the commission is null. Notice the SQL language is case-insensitive, so you can use a query in all lowercase if you find that easier to read:

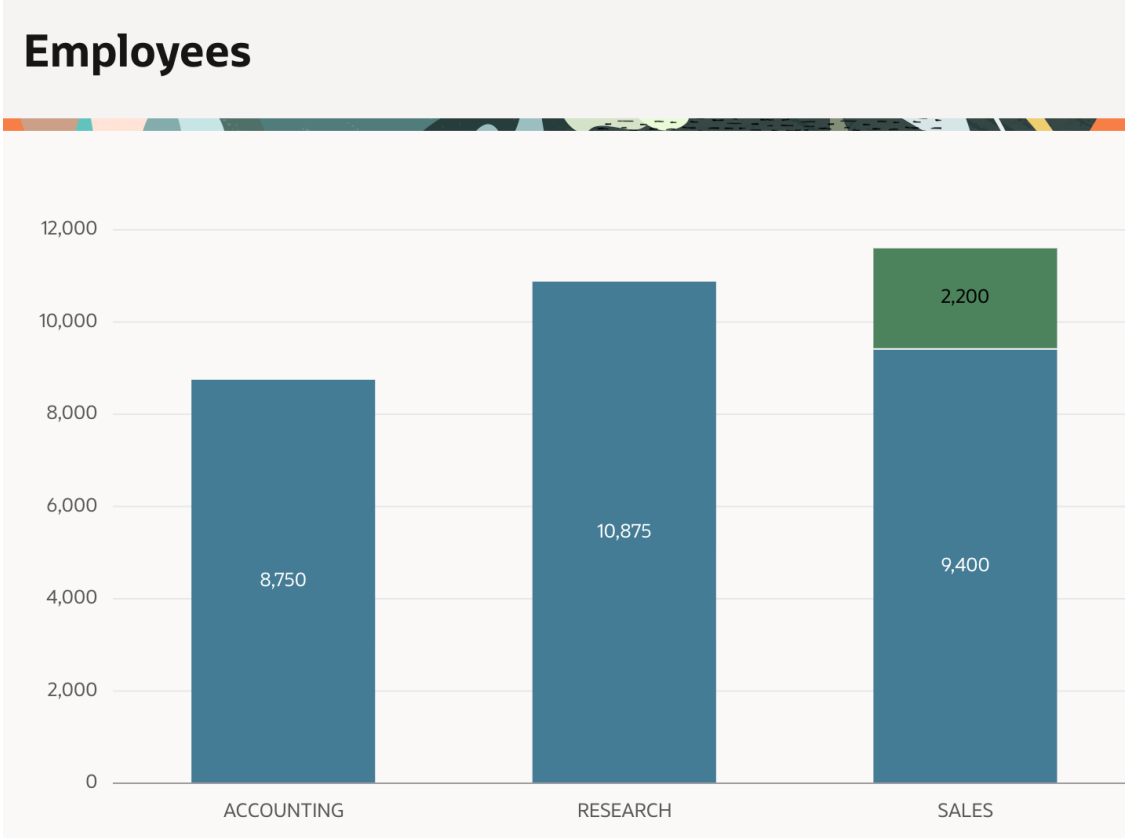
```
select e.deptno,
       d.dname,
       sum(nvl(e.comm,0)) as total_commissions
  from emp e
  join dept d
```


Copyright © 2026, Oracle and/or its affiliates.

```
    on d.deptno = e.deptno
group by e.deptno, d.dname
```

After adjusting the *Salaries* and *Commissions* series' *Column Mapping* sections to pick `DNAME` for the label and the `TOTAL_SALARIES` and `TOTAL_COMMISSIONS` for the respective value column, you can configure the series to order by the `DEPTNO` column. These changes produce the chart shown below with the department name now showing along the x-axis.

Figure 5-10 Stacked Bar Chart Now Showing Joined Department Names



For more information, see Creating Charts in *Oracle APEX App Builder User's Guide*. Consider installing the *Sample Charts* app from the *Gallery* to study additional useful chart examples.

5.6 Showing Data on Maps

Use a **Map** region to display data related to specific locations or areas on the earth. One map can show multiple layers when needed. For each layer you configure a table, SQL query, or other data source, and choose a layer type.

- Choosing a Map Layer Type
  A map can show multiple layers. For each one, configure a table, SQL query, or other data source, and choose a layer type.

Copyright © 2026, Oracle and/or its affiliates.

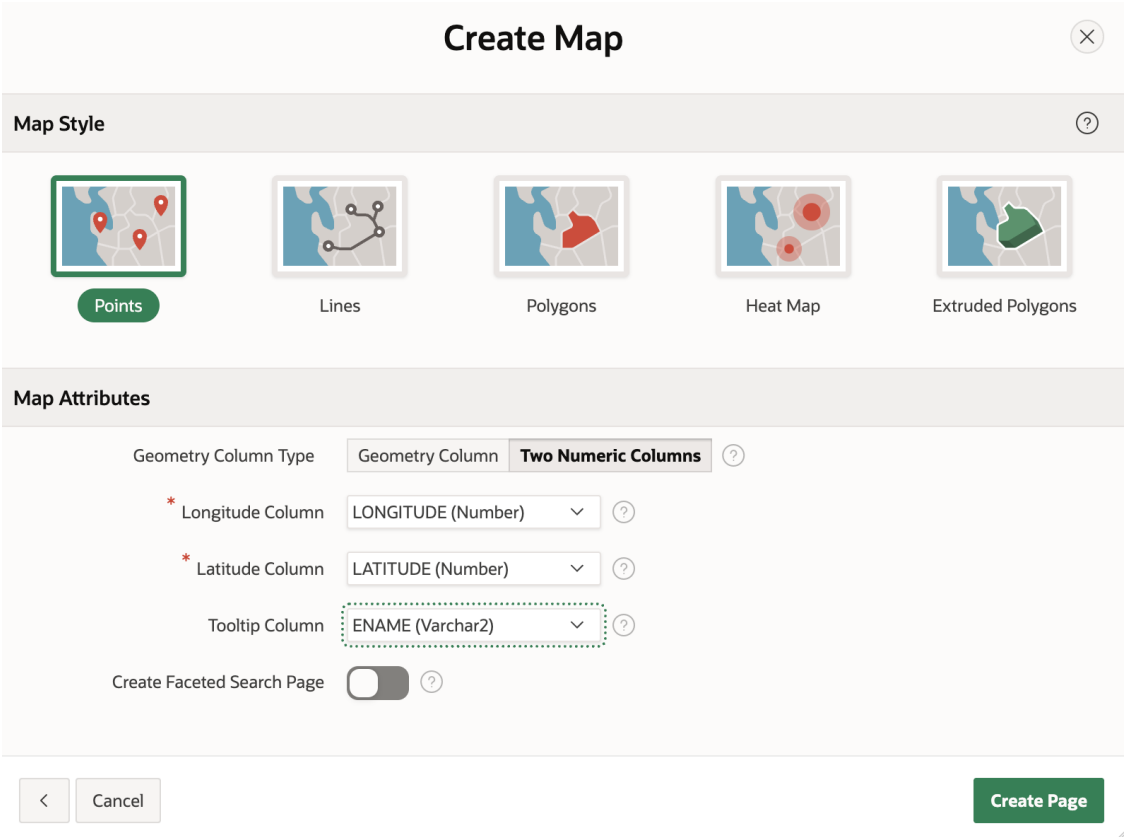
- Adding a Layer of Points
  Consider a map layer that shows each employee's home address as a point by joining `EMP` with a separate `EMP_ADDRESSES` table. You can configure a single data source at the map region level, then reference it in the layers. Or each layer can define its own data source.
- Conveying Quantities with a Heat Map
  A *Heat Map* layer associates an additional quantity to a point to convey how that value differs at different points on the map.
- Simplifying Points and Distance
  The `SDO_GEOMETRY` type lets your applications store, search on, and manipulate spatial data.
- Distance from a Point of Reference
  Using a database view, you can simplify pages that present the distance between points using helper functions.

5.6.1 Choosing a Map Layer Type

A map can show multiple layers. For each one, configure a table, SQL query, or other data source, and choose a layer type.

A layer can display points using a map marker, lines to indicate paths like navigation directions, or polygons to represent any geographic area. *Heat Map* and *Extruded Polygon* layers visualize an *additional* value associated with each point or shape respectively. The figure below shows the five available layer types as visual thumbnails in the Create Page wizard.

Figure 5-11 Five Different Types of Map Layer in the Create Page Wizard



Copyright © 2026, Oracle and/or its affiliates.

Your spatial data can be a pair of latitude and longitude numbers for *Points* and *Heat Map* layers, or all layer types can use SDO_GEOMETRY or GeoJSON. You decide which map tools end users see, including ones for zooming, measuring distance, and identifying a search radius. If one of the built-in map backgrounds doesn't meet your needs, you can define a custom one. When your data includes many points that might appear in close proximity, consider enabling point clustering for a cleaner look at lower zoom levels. Using dynamic actions, you can easily react to map changes and object selection.

5.6.2 Adding a Layer of Points

Consider a map layer that shows each employee's home address as a point by joining EMP with a separate EMP_ADDRESSES table. You can configure a single data source at the map region level, then reference it in the layers. Or each layer can define its own data source.

To simplify maintenance, consider creating a database view to encapsulate the region's SQL query. Then use it as the region's data source and pick column names to use in the layer. For example, consider the following EMP_WITH_ADDRESSES view that joins EMP and EMP_ADDRESSES and includes a computed TOTAL_COMPENSATION column:

```
create or replace view emp_with_addresses as
select e.empno,
       e.ename,
       a.address,
       a.latitude,
       a.longitude,
       e.sal + nvl(e.comm,0) as total_compensation
  from emp e
  join emp_addresses a
    on a.empno = e.empno;
```

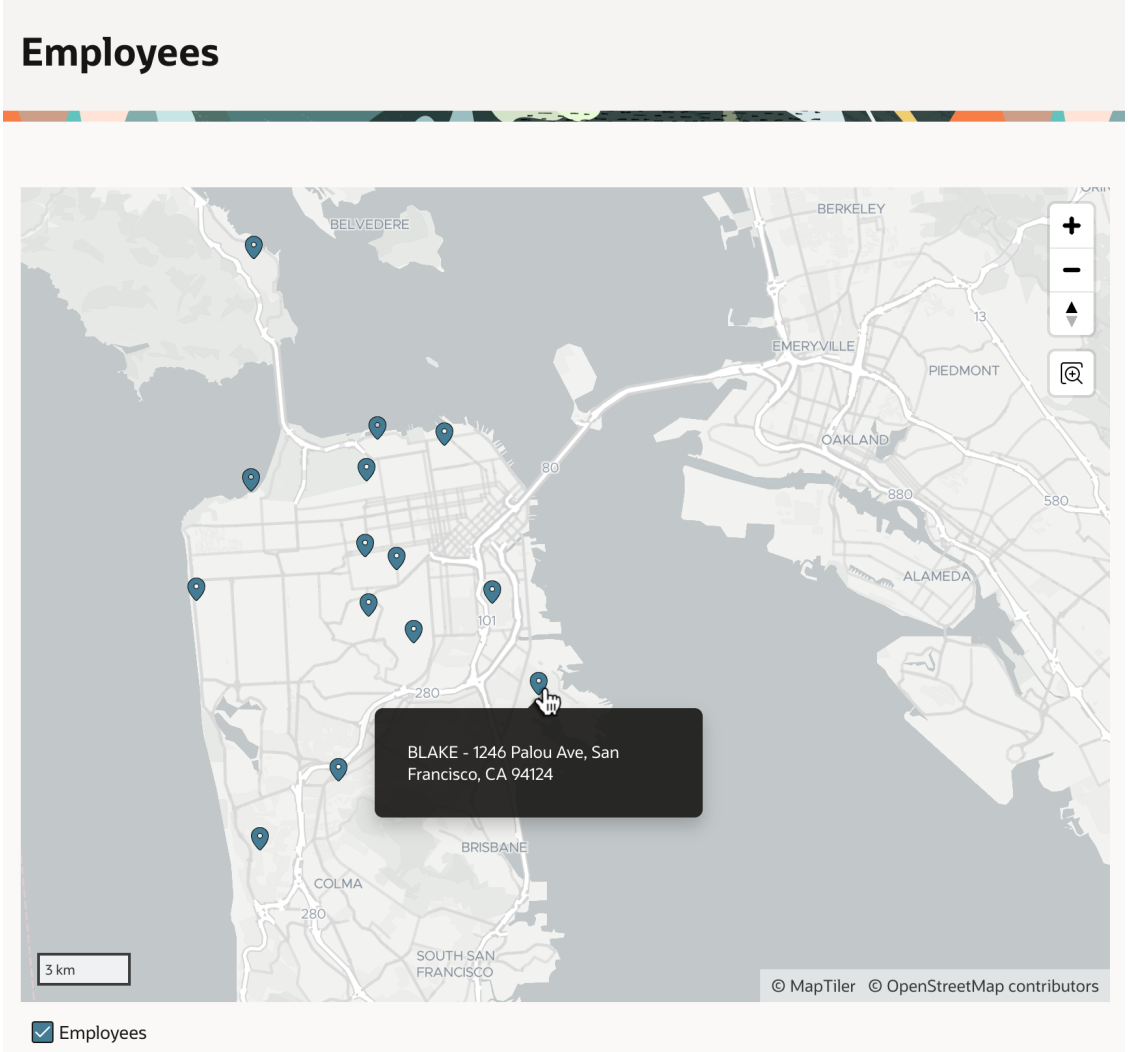
Then the map uses this view, configures its points using the LATITUDE and LONGITUDE columns, and defines the tooltip using the HTML Expression:

```
&ENAME. - &ADDRESS.
```

The figure below shows the resulting map, with its single point layer showing employee addresses around San Francisco, California and Marin County. Notice the user is hovering the mouse over BLAKE's point marker, and the tooltip displays "BLAKE - 1246 Palou Ave, San Francisco, CA 94124".

Copyright © 2026, Oracle and/or its affiliates.

Figure 5-12 Maps Showing Employees Addresses



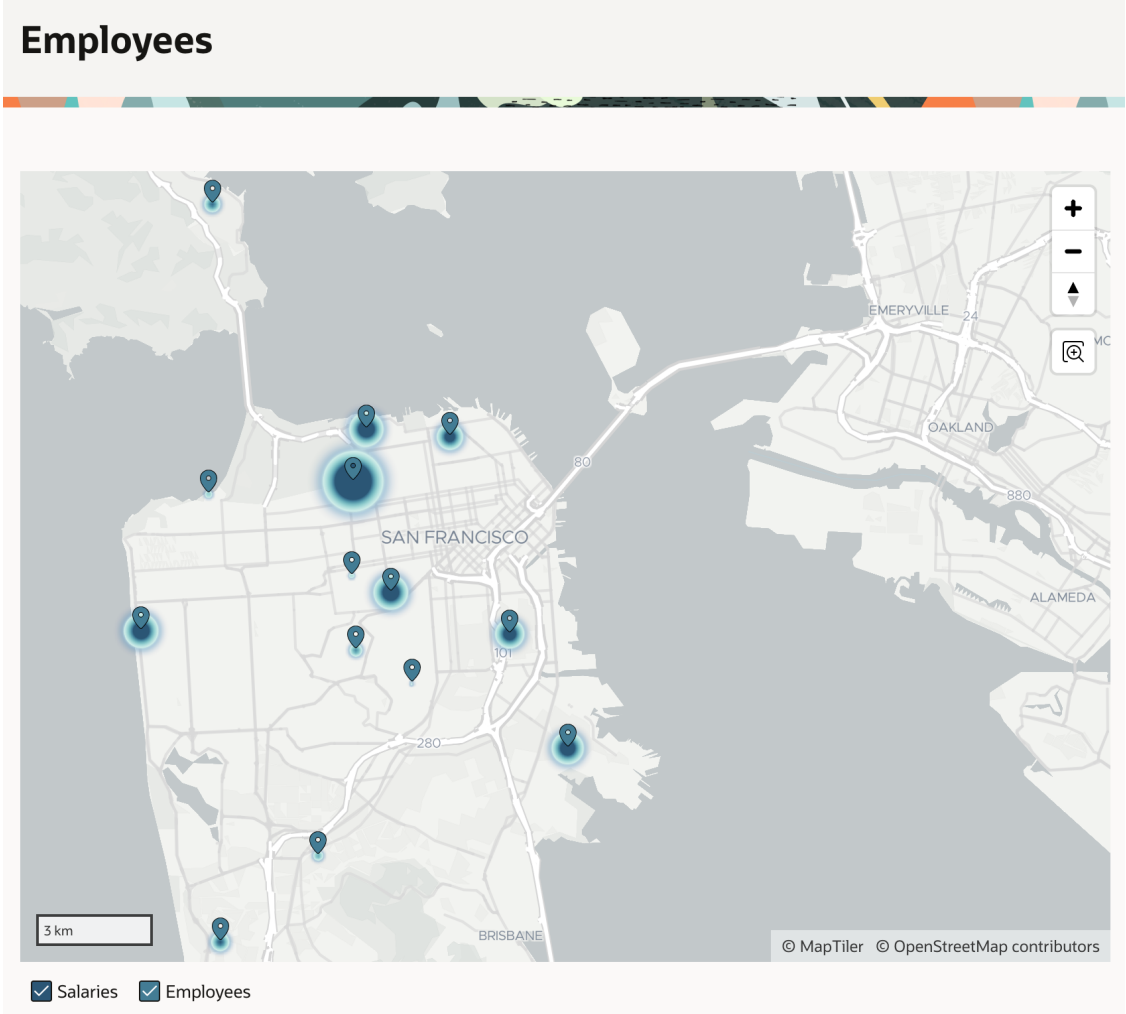
5.6.3 Conveying Quantities with a Heat Map

A *Heat Map* layer associates an additional quantity to a point to convey how that value differs at different points on the map.

After adding an additional *Salaries* heat map layer, using `TOTAL_COMPENSATION` as the value column, the page looks like the figure below when both layers are displayed. Each employee's address marker now shows a circle sized proportionally to their total pay relative to their peers.

Copyright © 2026, Oracle and/or its affiliates.

Figure 5-13 Point Layer and Heat Map Layer on a Map



5.6.4 Simplifying Points and Distance

The `SDO_GEOMETRY` type lets your applications store, search on, and manipulate spatial data.

A point is the simplest information an `SDO_GEOMETRY` value can hold. It can represent addresses and GPS locations. However, this data type also can handle:

- **lines** – for driving or walking routes or property boundaries,
- **polygons** – for city limits, sales territories, store footprints, lakes, or forests
- **circles** – for service areas, or delivery or impact zones.

Focusing on the points you need for the current task, in Oracle AI Database 26ai, you create an `SDO_GEOMETRY` for a point using the simple expression:

```
SDO_GEOMETRY(longitude,latitude)
```


Copyright © 2026, Oracle and/or its affiliates.

However, in earlier database versions, the following helper functions can simplify your work. The `MAKE_POINT` function returns an `SDO_GEOMETRY` value for longitude and latitude:

```
function make_point(
    p_longitude in number,
    p_latitude  in number)
return sdo_geometry
deterministic
is
    geom_type_2d_point   constant number := 2001;
    spatial_ref_type_gps constant number := 4326;
begin
    return case
              when     p_longitude is not null
                   and p_latitude  is not null
              then
                 sdo_geometry(
                    geom_type_2d_point,
                    spatial_ref_type_gps,
                    sdo_point_type(p_longitude, p_latitude,
                    null), null, null)
           end;
end;
```

The `DISTANCE_IN_KM` function returns the distance between two `SDO_GEOMETRY` points:

```
function distance_in_km(
    p_origin      in sdo_geometry,
    p_destination in sdo_geometry
)
return number
deterministic
as
begin
    return sdo_geom.sdo_distance(
            p_origin,
            p_destination,
            tol=>1,
            unit=>'unit=KM');
end distance_in_km;
```

Tip

For any `SDO_GEOMETRY` object, use the `TO_GEOJSON` function in the `SDO_UTIL` package to get its GeoJSON representation.

5.6.5 Distance from a Point of Reference

Using a database view, you can simplify pages that present the distance between points using helper functions.

Copyright © 2026, Oracle and/or its affiliates.

Suppose the office is located in the famous "Pyramid Building" at longitude `-122.402768` and latitude `37.795270`. To show how many kilometers away each employee lives from the office, update the `EMP_WITH_ADDRESSES` view to include a `DISTANCE_FROM_OFFICE_KM` column. Using the `MAKE_POINT` and `DISTANCE_IN_KM` functions from the previous section, the view becomes:

```
create or replace view emp_with_addresses as
select e.empno,
       e.ename,
       a.address,
       a.latitude,
       a.longitude,
       e.sal + nvl(e.comm,0) as total_compensation,
       round(distance_in_km(
                make_point(a.longitude,a.latitude),
                make_point(-122.402768,37.795270)),1)
       as distance_from_office_km
  from emp e
  join emp_addresses a
    on a.empno = e.empno
```

Then, you could enhance the example map from to include a third layer to represent the office building's location using a query like this:

```
select   37.795270 as latitude,
       -122.402768 as longitude
  from dual
```

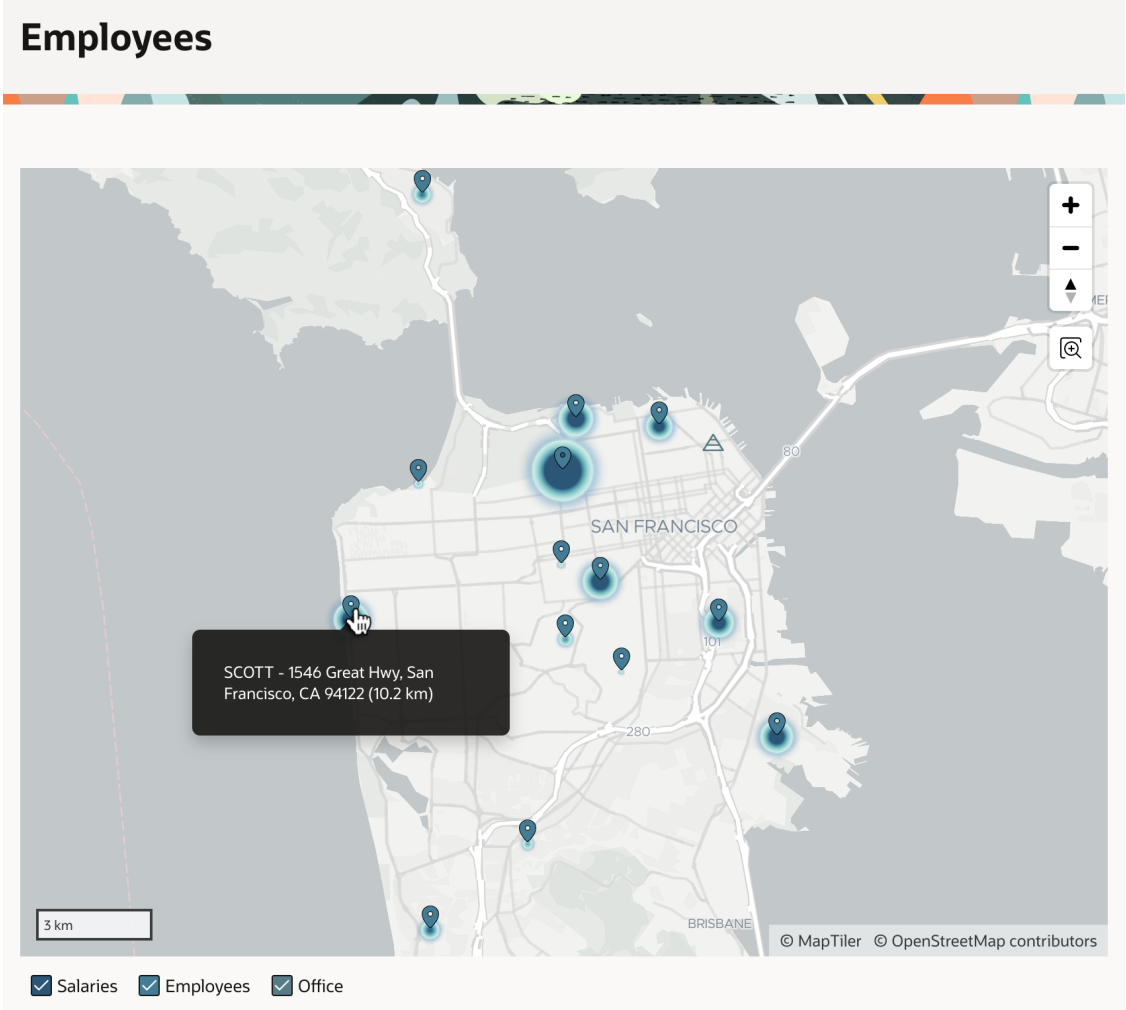
Updating the *Employees* layer tooltip HTML expression to include the distance, you can include that information when end users hover over an employee's house:

```
&ENAME. - &ADDRESS. (&DISTANCE_FROM_OFFICE_KM. km)
```

These changes produce the map shown below. Notice the user's mouse hovering over SCOTT's map marker. The tooltip now reveals she lives 10.2km from the office. For more information, see Creating Maps in *Oracle APEX App Builder User's Guide*, and consider installing the *Sample Maps* app from the *Gallery* to study additional useful map examples.

Copyright © 2026, Oracle and/or its affiliates.

Figure 5-14 Showing the Office and Employees Distance Away



5.7 Placing Events on a Calendar

Use a *Calendar* region to display scheduled events with a month, week, day, or list view.

If your data source has only an event title and a start date, then the events show on the day they occur. If you enable the *Show Time* switch, they display as a one-hour event. If you also configure an end date, then the calendar shows the event duration as well.

- Configuring a Basic Calendar
  After adding a calendar, configure its key settings to identifying the events it will display. After choosing a data source, set properties like the event **Title**, **Start Date**, and **End Date**.
- Adding Create and Edit Links
  Use the Property Editor to configure create and edit links that navigate to appropriate pages to enter or modify an event.
- Changing Events with Drag and Drop
  If you enable drag and drop support, then the user can drag the event to a new starting day or time and they can drag its top edge or bottom edge to change the duration.

Copyright © 2026, Oracle and/or its affiliates.

- Joining In Additional Info Using a View
  Customize additional information in a calendar using a view.
- Enhancing Event Display
  To further customize calendar event display, you can use a custom formatting notation, paired with an event content formatter function.

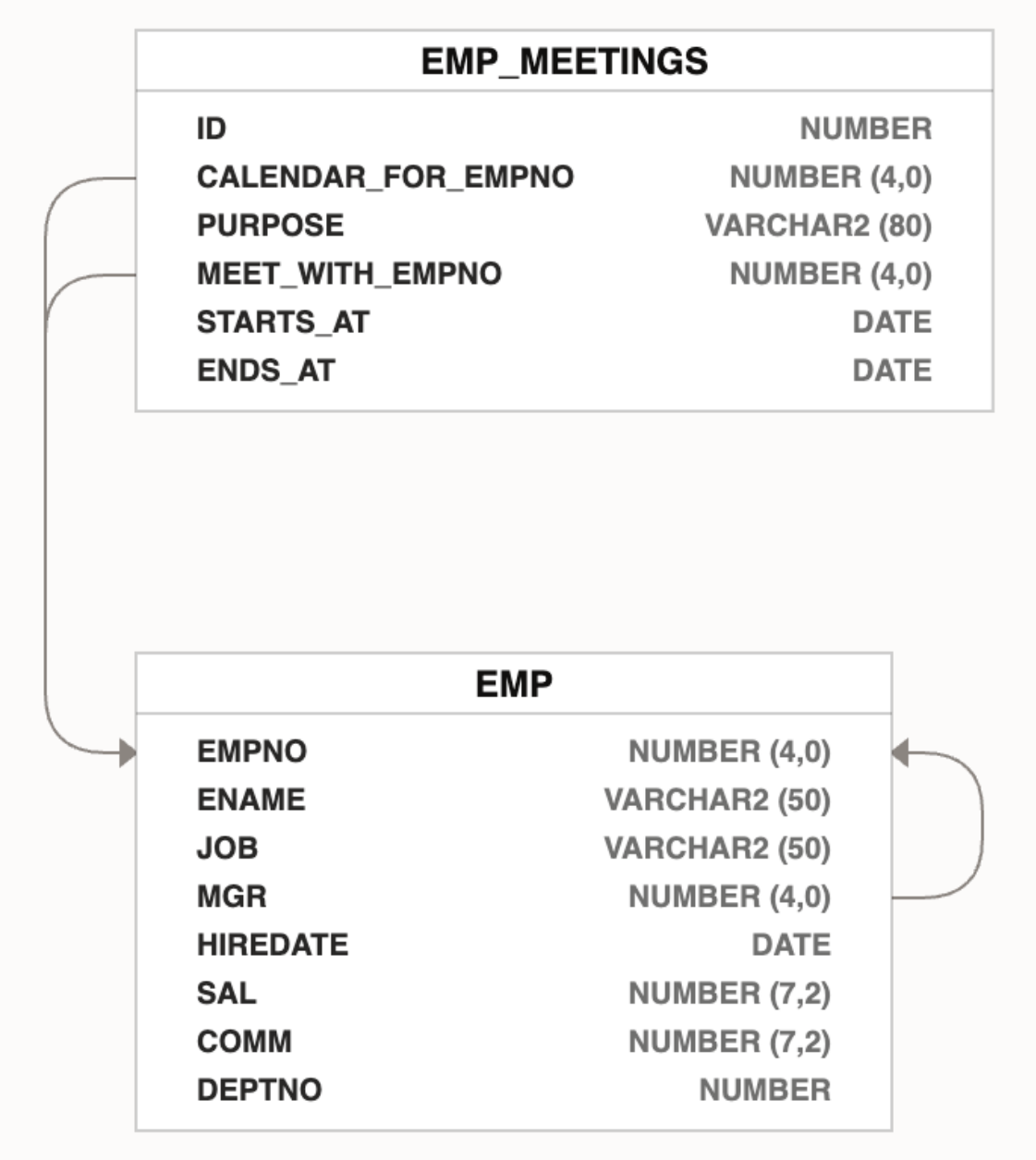
5.7.1 Configuring a Basic Calendar

After adding a calendar, configure its key settings to identifying the events it will display. After choosing a data source, set properties like the event **Title**, **Start Date**, and **End Date**.

Imagine an `EMP_MEETINGS` table that stores scheduled events between one employee and another. The diagram in the figure below shows how the `EMP` table and `EMP_MEETINGS` relate. This new table contains a `CALENDAR_FOR_EMPNO` column for the *inviting* employee and a `MEET_WITH_EMPNO` column for the invitee.

Copyright © 2026, Oracle and/or its affiliates.

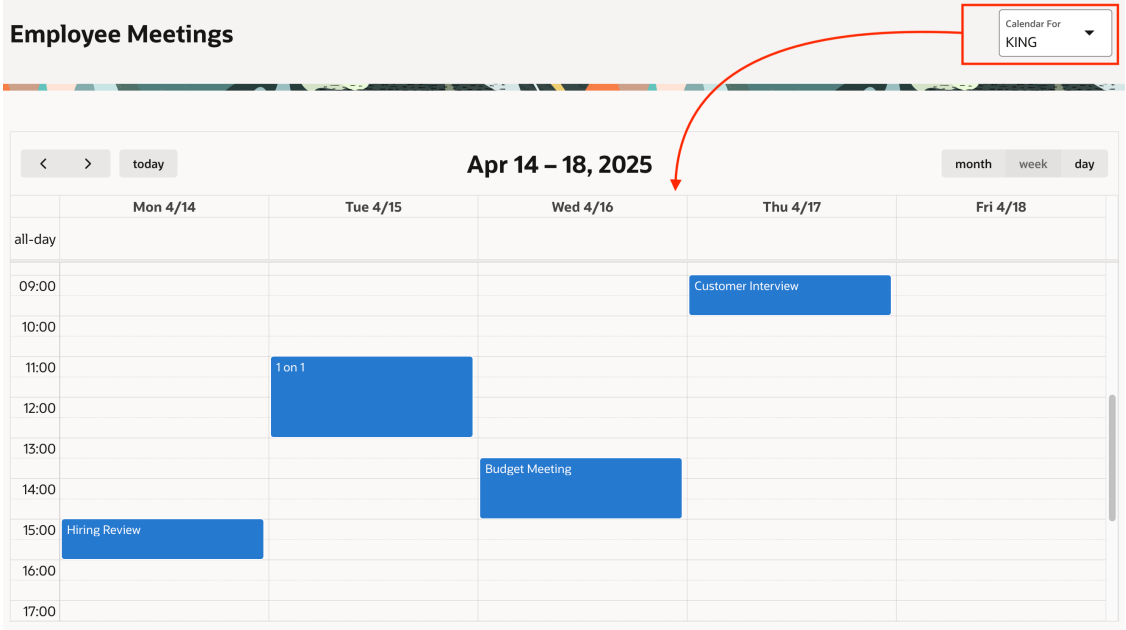
Figure 5-15 EMP_MEETINGS Table Records for EMP to EMP Scheduled Events



The figure below shows the week view of a calendar region based on the `EMP_MEETINGS` table after configuring the `PURPOSE` column as the event title, `STARTS_AT` as the start date, and `ENDS_AT` as end date. You can configure the calendar to *not* show weekends, to allow month, week, and day views, and to show the navigation buttons to let users change the current month, week, or day.

Copyright © 2026, Oracle and/or its affiliates.

Figure 5-16 Four Meetings in Employee KING's Schedule with Dynamic Action on Select List



Notice the *Calendar For* select list page item (`P19_CALENDAR_FOR`) in the upper right. It determines which employee's events the calendar displays by referencing its value as a bind variable in the region's `WHERE` clause. When set to employee like `KING`, you aim to display events where she invited someone as well as those where she is invited to attend. You can accomplish that with a filter like this:

```
   CALENDAR_FOR_EMPNO = :P19_CALENDAR_FOR
OR MEET_WITH_EMPNO    = :P19_CALENDAR_FOR
```

To automatically refresh the calendar region when the value of *Calendar For* changes, add a dynamic action that reacts to the select list's change event. This combination gives a flexible page to review and manage any employee's meetings. It is important to mention `P19_CALENDAR_FOR` in the **Page Items to Submit** region property so any data refresh uses the latest value of this page item from the browser.

5.7.2 Adding Create and Edit Links

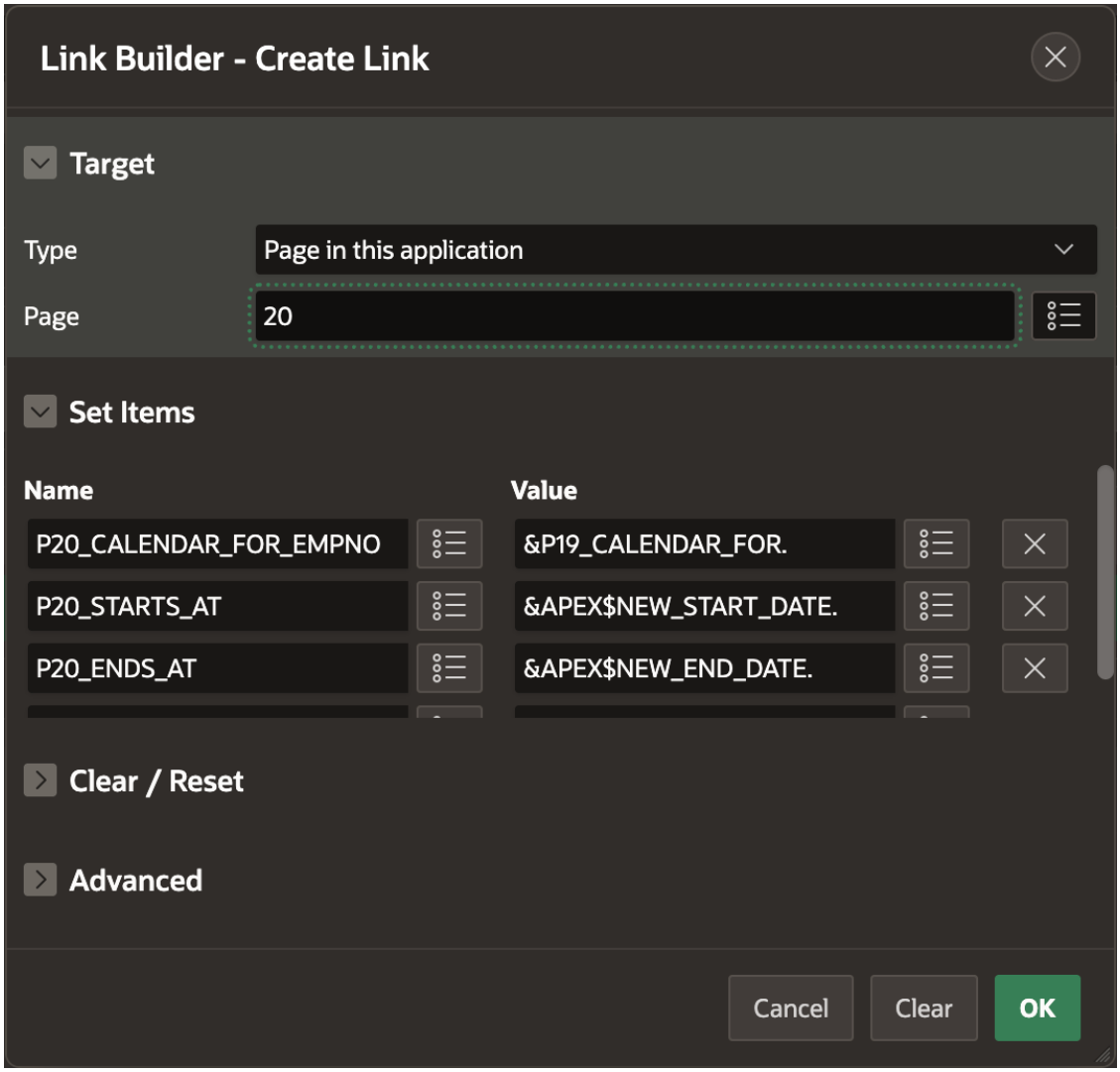
Use the Property Editor to configure create and edit links that navigate to appropriate pages to enter or modify an event.

To define the create link, use the **Link Builder** dialog to define the target page. Once configured, the user can click on the calendar or click-and-drag to create a new event. The figure shows configuring the create link to target page 20 and pass the value of `P19_CALENDAR_FOR` as the value of the target page's `P20_CALENDAR_FOR_EMPNO` page item.

Notice you can easily pass the `APEX$NEW_START_DATE` and `APEX$NEW_END_DATE` values to your event creation page using the `&NAME.` syntax. In a similar way, after configuring a target link for event *editing*, then a user can click on an existing event in the calendar to opens a page to edit the details.

Copyright © 2026, Oracle and/or its affiliates.

Figure 5-17 Configuring Create Link to Pass Start and End Date



5.7.3 Changing Events with Drag and Drop

If you enable drag and drop support, then the user can drag the event to a new starting day or time and they can drag its top edge or bottom edge to change the duration.

You only need to write the logic to update your event with the new start and end dates like this. APEX formats the start and end date values as a canonical 12-digit number representing a date like **April 19th, 2025 at 9:30am** as `202504190930`. You reference `APEX$NEW_START_DATE` and `APEX$NEW_END_DATE` as bind variables along with `APEX$PK_VALUE` for the primary key of the meeting that the user modified.

```
update emp_meetings
   set starts_at = to_date(:APEX$NEW_START_DATE, 'YYYYMMDDHH24MISS'),
         ends_at = to_date(:APEX$NEW_END_DATE,   'YYYYMMDDHH24MISS')
 where id = :APEX$PK_VALUE;
```


Copyright © 2026, Oracle and/or its affiliates.

5.7.4 Joining In Additional Info Using a View

Customize additional information in a calendar using a view.

To show *additional* information in the calendar for each event like the name of the employee invited to the meeting, you could create an `EMP_MEETINGS_V` view to join the `EMP_MEETINGS` table with `EMP` to retrieve the inviting employee name and the invited employee name like this:

```
create or replace view emp_meetings_v as
select m.id,
       m.purpose,
       m.starts_at,
       m.ends_at,
       m.calendar_for_empno,
       cal_e.ename  as calendar_for_ename,
       m.meet_with_empno,
       meet_e.ename as meet_with_ename
  from emp_meetings m
  join emp cal_e  /* inviting emp */
    on cal_e.empno = m.calendar_for_empno
  join emp meet_e /* invited emp  */
    on meet_e.empno = m.meet_with_empno
```

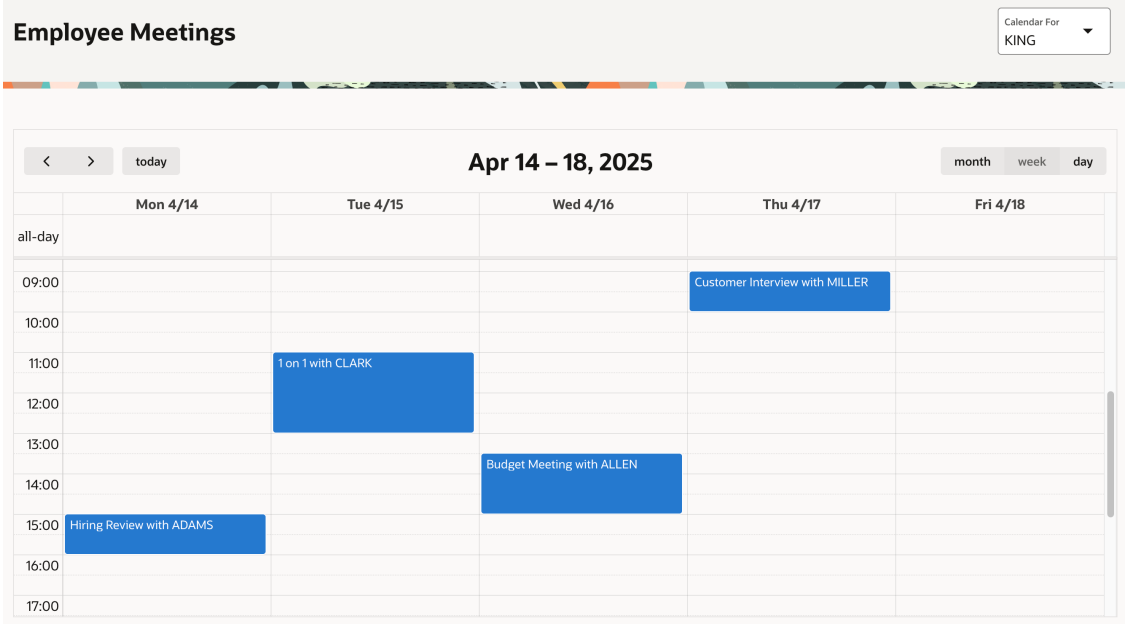
You could then change the region to be based on the SQL Query below that selects from this new view. It uses a CASE expression as part of a `FORMATTED_TITLE` column to show either the `MEET_WITH_ENAME` or the `CALENDAR_FOR_ENAME` value, depending on the value of the `P19_CALENDAR_FOR` page item.

```
select id,
       purpose||' with '||
          case to_number(:P19_CALENDAR_FOR)
             when calendar_for_empno then meet_with_ename
             else calendar_for_ename
          end as formatted_title,
       starts_at,
       ends_at
  from emp_meetings_v
where calendar_for_empno = :P19_CALENDAR_FOR
   or meet_with_empno    = :P19_CALENDAR_FOR
```

After changing the event title to use the new `FORMATTED_TITLE` column, the calendar looks like the figure below showing the name of the person KING is meeting with.

Copyright © 2026, Oracle and/or its affiliates.

Figure 5-18 Events Showing Purpose and Meeting With Employee Name



5.7.5 Enhancing Event Display

To further customize calendar event display, you can use a custom formatting notation, paired with an event content formatter function.

The SQL below updates the existing query to use the `FORMAT` function in the `APEX_STRING` package to inject two values into a string template: `[[%s]]|with %s`. You supply the `PURPOSE` column value for the first `%s` placeholder. To conditionally return either the `CALENDAR_FOR_ENAME` or the `MEET_WITH_ENAME` depending on the value of the `P19_CALENDAR_FOR` select list you use a `CASE` statement.

```
select id,
       apex_string.format('[[%s]]|with %s',
          purpose,
          case to_number(:P19_CALENDAR_FOR)
             when calendar_for_empno then meet_with_ename
             else calendar_for_ename
          end) as formatted_title,
       starts_at,
       ends_at
  from emp_meetings_v
where calendar_for_empno = :P19_CALENDAR_FOR
   or meet_with_empno    = :P19_CALENDAR_FOR
```

The string template value ends up looking like the following for one example `EMP_MEETINGS` row:

```
[[Hiring Review]]|with ADAMS
```


Copyright © 2026, Oracle and/or its affiliates.

You further customize the calendar event behavior and display by using a *JavaScript Initialization Function* like the one below. This function configures calendar settings on the `pOptions` object it is passed as an argument, then returns the modified `pOptions` object as its result. The function can use any of the features of the open-source *FullCalendar* library to tailor how the calendar looks and behaves.

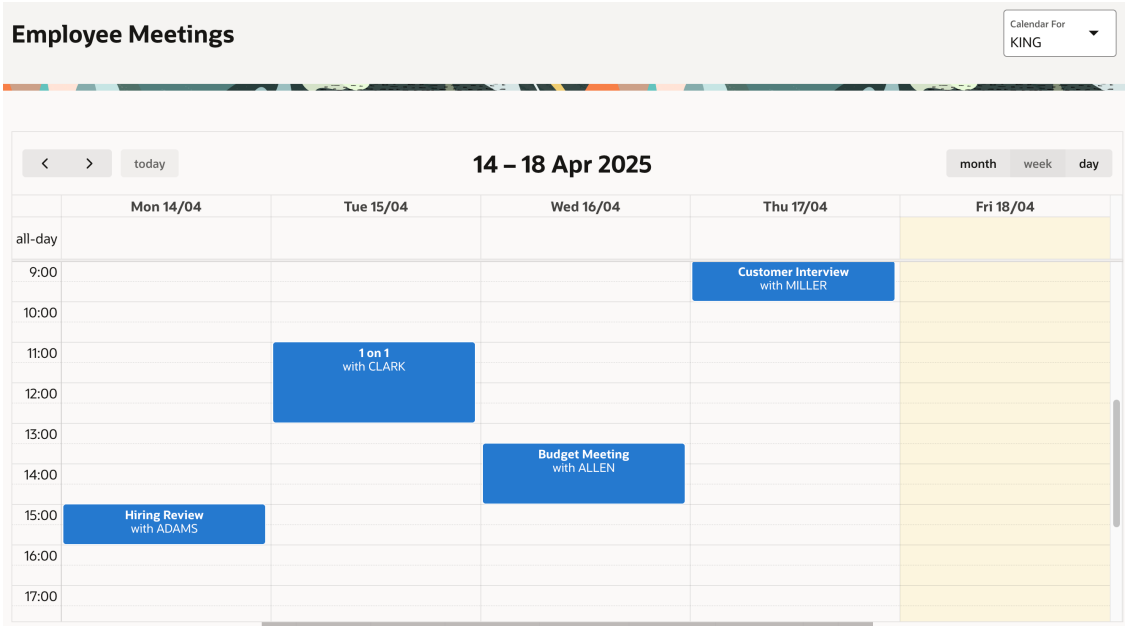
The custom event rendering function below replaces the `[[`, `]]`, and `|` with appropriate HTML tags to bold the title and add a line break before the additional detail text. You can't directly use HTML tags in the original string because for security reasons the APEX engine would escape them and it wouldn't have the desired effect. It also sets the calendar `locale` to `en-gb` for a UK date format. Notice it configures an `eventContent` function that returns custom `html` formatting for the event display. It references the event title using `arg.event.title` and uses the JavaScript `replace()` function to replace `|` by `<br>`, `[[` by `<b>`, and `]]` by `</b>`. It also wraps the event title content with in an HTML `<center>` tag.

```
function (pOptions) {
    pOptions.locale       = "en-gb"; // UK date format
    pOptions.eventContent = function(arg) {
      return {
        html: '<center>' +
              arg.event.title.replace(/\|/g, '<br>')
                             .replace(/\[\[/g,'<b>')
                             .replace(/\]\]/g,'</b>') +
              '</center>'
      }
    };
    return pOptions;
}
```

This combination of using the `EMP_MEETINGS_V` view to join in employee names and custom event title formatting produces the updated calendar you see below. Each event's text is centered, and shows the purpose of each meeting on its own line in a bold font, and the person the meeting is with on a second line. You can customize the calendar further using CSS styling. Consider installing the *Sample Calendar* app from the gallery to explore additional examples of what the calendar region can do. For more information, see Creating Calendars in *Oracle APEX App Builder User's Guide*.

Copyright © 2026, Oracle and/or its affiliates.

Figure 5-19 Employee KING's Week of Meetings



5.8 Exploring Hierarchical Data in a Tree

Use the *Tree* region to show hierarchical data like the management chain of employees in the `EMP` table. APEX can manage the hierarchy so you only need to configure the data to display in each tree node.

- Configuring a Basic Tree Display
  After selecting a table like `EMP`, configure at least a *Node Id* column, a *Node Label* column, and a *Parent Key* column. The latter value references another node id as its parent to form the hierarchy.
- Tracking and Remembering Selection
  Tracking and remembering the tree selection requires configuring a hidden page item with its *Value Protected* property set to false. You'll use this to store the node value of the selected node. If this hidden page item's value is set during page rendering the tree uses it to restore the selected node.
- Adding Icons to the Tree
  You can use more meaningful node icons to communicate additional information to end users. To control the icon of every node in the tree, provide an additional column whose value indicates the Font APEX icon name to use.
- Parent/Child Data in a Tree
  You can show parent/child data in a tree by using a `UNION ALL` query and cleverly adjusting the foreign key value.

5.8.1 Configuring a Basic Tree Display

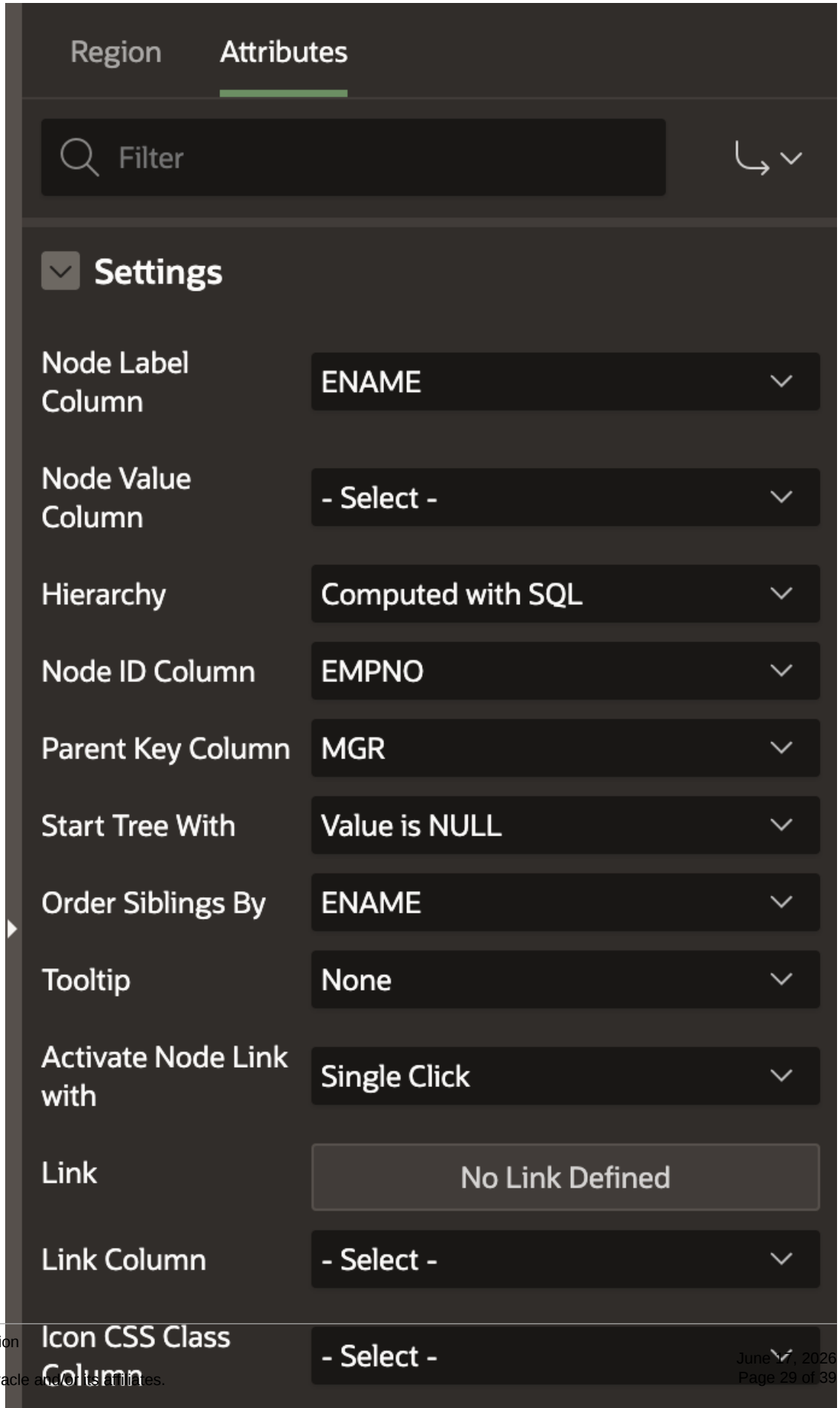
After selecting a table like `EMP`, configure at least a *Node Id* column, a *Node Label* column, and a *Parent Key* column. The latter value references another node id as its parent to form the hierarchy.

Copyright © 2026, Oracle and/or its affiliates.

As shown below, you can use the `ENAME` column for the node label, `EMPNO` as the node ID, and `MGR` as the parent key. Notice the *Hierarchy* property is set to *Computed with SQL*. This choice means, "Let the APEX engine compute the hierarchy with SQL that it automatically generates at runtime." The *Order Siblings By* property sets `ENAME` as the sorting order for nodes in the tree at the same level, while the *Start Tree With* property setting of *Value is NULL* causes the top level of the tree to be nodes where the Parent Key Column `MGR` is null.

Copyright © 2026, Oracle and/or its affiliates.

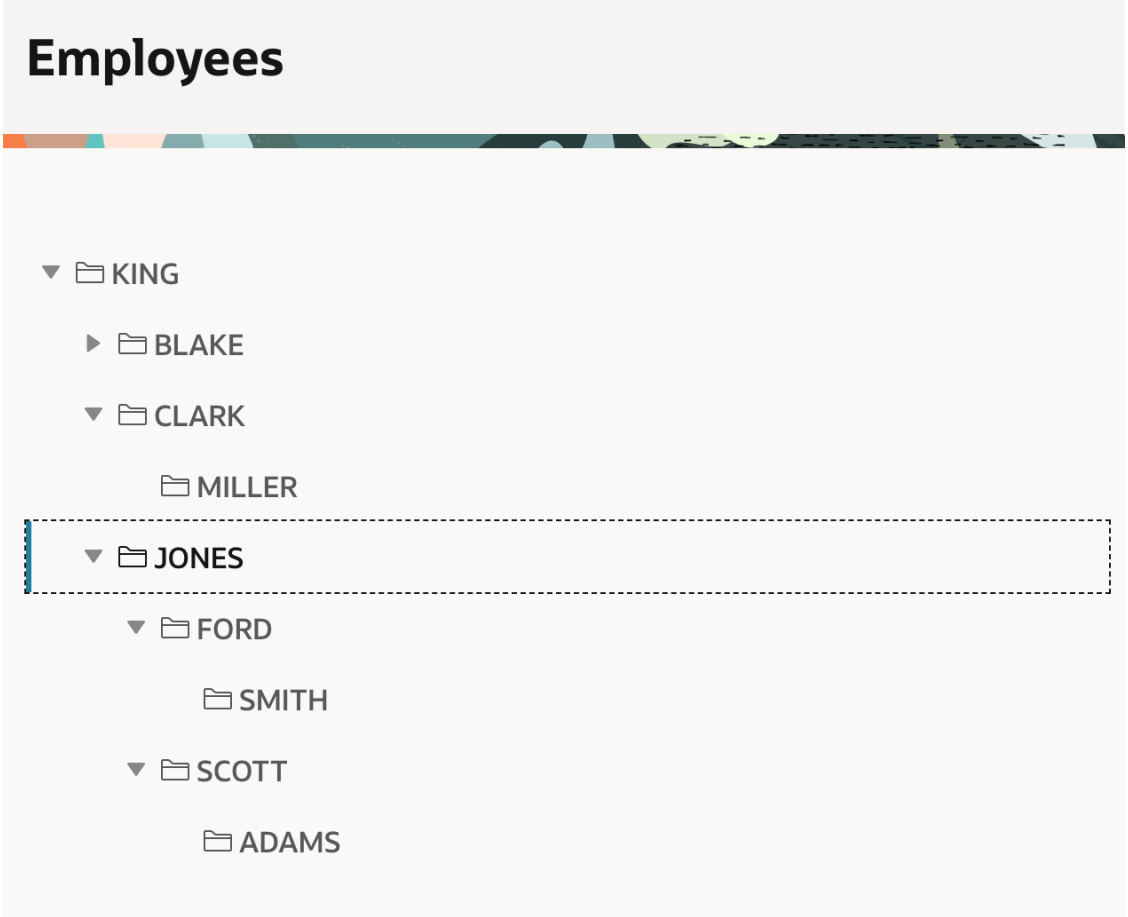
Figure 5-20 Minimal Settings for a Working Tree Region



Copyright © 2026, Oracle and/or its affiliates.

These settings produce the tree shown below. **KING** has a `null` value for her `MGR` column, so she displays as the root node of the tree. APEX computed the hierarchical query at runtime, including the configured colums, to retrieve all the tree's data in a single query. In the figure, the **CLARK** and **JONES** levels are expanded to reveal the direct reports to multiple levels.

Figure 5-21 Basic Tree Region Showing EMP Management Hierarchy



5.8.2 Tracking and Remembering Selection

Tracking and remembering the tree selection requires configuring a hidden page item with its *Value Protected* property set to false. You'll use this to store the node value of the selected node. If this hidden page item's value is set during page rendering the tree uses it to restore the selected node.

Start by defining an item like `P17_SELECTED_EMPNO` and confirming that its *Session State* > *Storage* property is set to *Per Session (Persistent)*. Then you can configure the tree region's *Node Value* to use the `EMPNO` column and set the *Selected Node Page Item* to `P17_SELECTED_EMPNO`. This takes care of restoring the selected node when the hidden page item is populated.

Completing the picture requires two additional steps. Begin by setting the *Static ID* property of the tree region, for example, to `emptree`. Next, define a dynamic action on the tree region to react to the *Selection Changed [Tree]* event. You can title it *When Node Selected*, and configure its first dynamic action step to be of type *Set Value* with *Affected Elements* of *Item* `P17_SELECTED_EMPNO`. Set the *Fire on Initialization* switch to false and configure the *Set Type* to

Copyright © 2026, Oracle and/or its affiliates.

be *JavaScript Expression* with the line below. It returns the node value of the *first* selected node in the region with *Static ID* `emptree`.

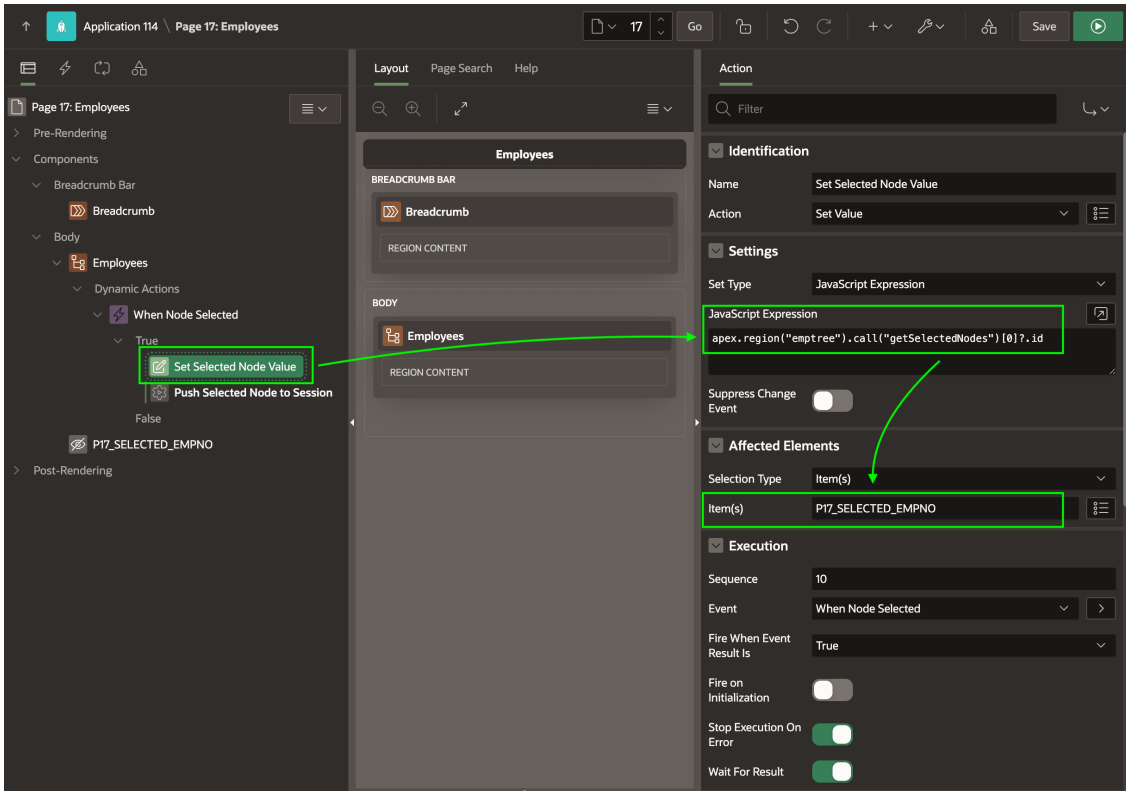
```
apex.region("emptree").call("getSelectedNodes")[0]?.id
```

If remembering the selected employee number when the user submits the page is adequate, then this is the only additional step required. If instead you want the user's tree selection to *immediately* be stored in the session, configure an additional dynamic action step of type *Execute Server-side Code* with its *Page Items to Submit* property set to `P17_SELECTED_EMPNO`. The PL/SQL code in this action step can simply be:

```
null; -- Push Items to Submit into Session State
```

With these settings in place as shown in the figure below, the tree tracks and remembers the user's selection throughout their session.

Figure 5-22 Configuring Dynamic Action Step to Set Selected Node Id Into Hidden Page Item



5.8.3 Adding Icons to the Tree

You can use more meaningful node icons to communicate additional information to end users. To control the icon of every node in the tree, provide an additional column whose value indicates the Font APEX icon name to use.

You can review the available icon names in your browser in *Icons* page of the *Universal Theme Reference App* at oracleapex.com/ut.

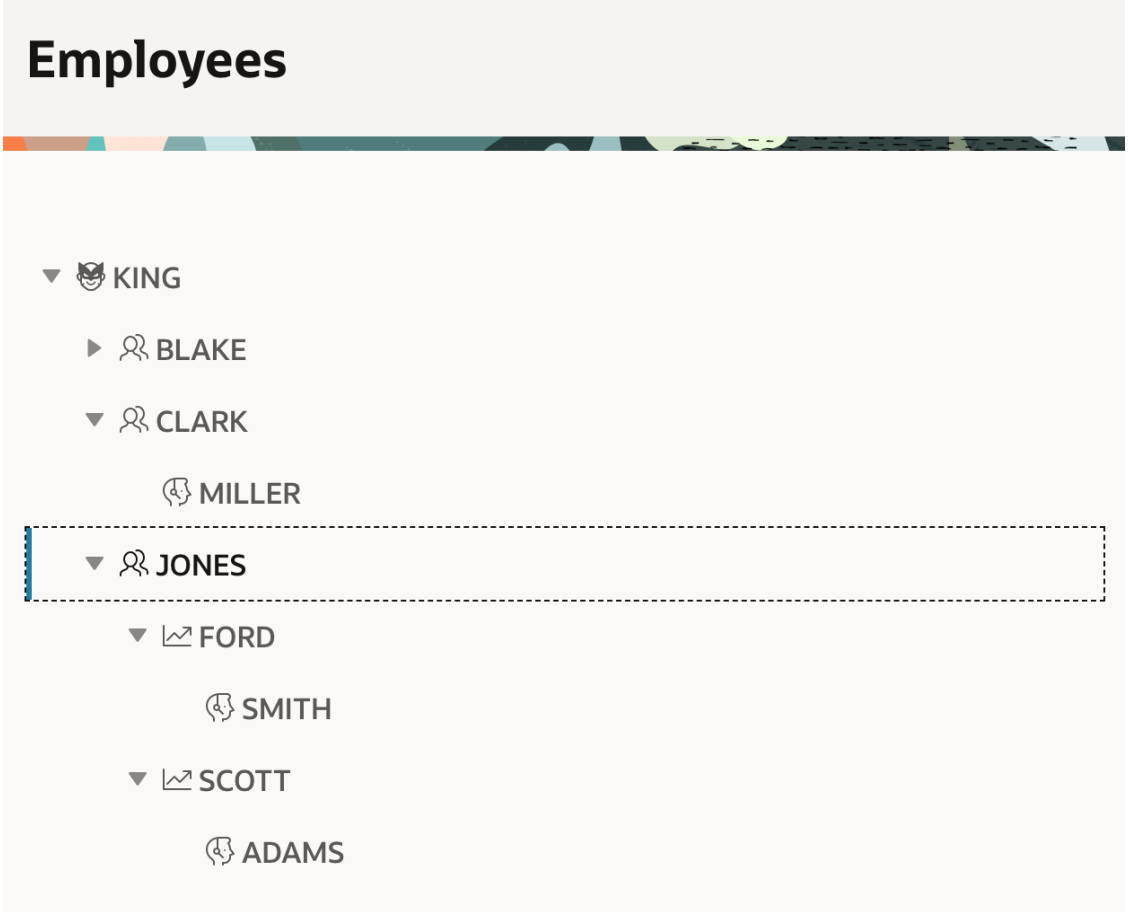
Copyright © 2026, Oracle and/or its affiliates.

For example, if you define the following `EMP_WITH_ICONS_V` view, it uses a `CASE` statement to return a different icon name based on the `JOB` column value of each employee:

```
create or replace view emp_with_icons_v as
select empno,
       ename,
       job,
       deptno,
       mgr,
       hiredate,
       sal,
       comm,
       'fa '||case job
            when 'PRESIDENT' then 'fa-badgerine'
            when 'ANALYST'   then 'fa-line-chart'
            when 'CLERK'     then 'fa-user-headset'
            when 'SALESMAN'  then 'fa-badge-dollar'
            when 'MANAGER'   then 'fa-users'
       end as icon
from emp
```

After changing the tree region to be based on `EMP_WITH_ICONS_V` and setting its *Icon CSS Class Column* property to `ICON`, the tree changes to look like the figure below. Each node in the tree renders with a job-appropriate icon.

Figure 5-23 Employee Hierarchy with Icons Based on Job



Copyright © 2026, Oracle and/or its affiliates.

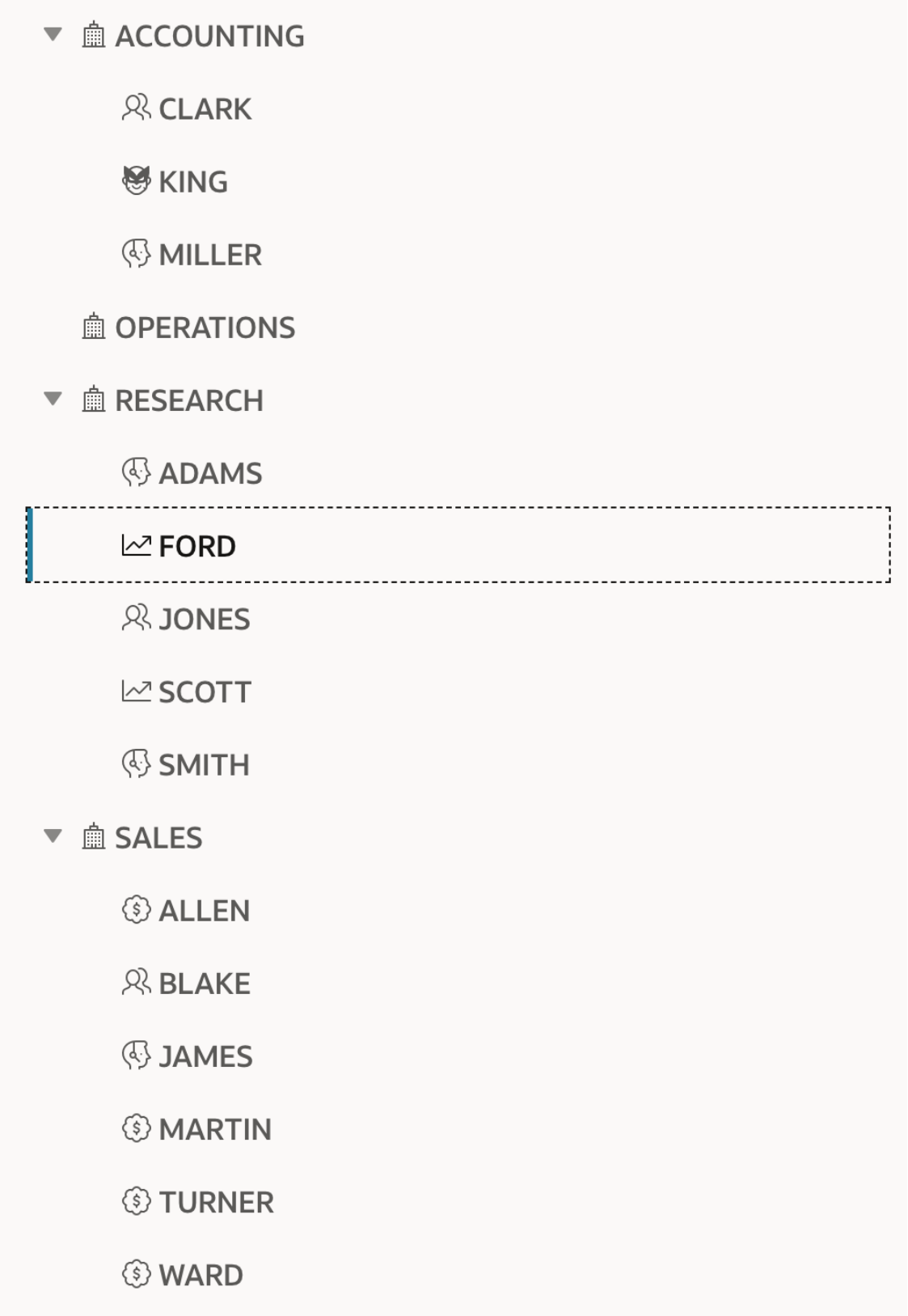
5.8.4 Parent/Child Data in a Tree

You can show parent/child data in a tree by using a `UNION ALL` query and cleverly adjusting the foreign key value.

For example, consider the task of creating the tree shown below. Employees appear in alphabetical order indented under a tree node representing the department they work in. The task is simple, but requires thinking like the APEX tree region to make it happen.

Copyright © 2026, Oracle and/or its affiliates.

Figure 5-24 Exploring Parent/Child Data in a Tree



Copyright © 2026, Oracle and/or its affiliates.

The tree region needs three basic elements: a node label, a node id, and a parent id. When querying hierarchical data from a single table like `EMP`, you know that the node id `EMPNO` and the parent id `MGR` are both employee IDs. However, when creating a hierarchy combining department and employee numbers, it's important to ensure that the node id values don't overlap. You can achieve this by performing a math operation on the department numbers to be sure they cannot clash with the employee numbers. For example, you could multiply the `DEPTNO` column value by `-1` so that the department numbers are negative values, while the employee numbers remain positive values.

The SQL query below uses this technique while combining rows from the `EMP_WITH_ICONS_V` view and the `DEPT` table using the `UNION ALL` set operator. Notice the department rows alias the expression `DEPTNO * -1` as the value of the `NODE_ID` column, and use a `NULL` value for the `PARENT_KEY`. Then, in the employee query the same expression is used as the value of the `PARENT_KEY`.

```
create or replace view dept_emp_tree_v as
select -1 * deptno        as node_id,
       dname              as node_label,
       null               as parent_key,
       'fa fa-building-o' as icon
 from dept
union all
select empno              as node_id,
       ename              as node_label,
       -1 * deptno        as parent_key,
       icon
 from emp_with_icons_v
```

Then, you simply need to configure the tree region to be based on this SQL query, configuring the appropriately-named columns as Node ID, Node Label, and Parent Key. Using the *Hierarchy* setting of *Computed in SQL* and the *Start Tree With* property set to *Value is NULL*, you get a tree whose first level are the department nodes, and whose first level beneath those are the employees who work in the respective department. By setting *Order Siblings By* to the `NODE_LABEL` column, you ensure both the departments and employees nodes get sorted alphabetically.

5.9 Dynamically Formatting Data

When none of APEX's other region types meet your requirements, the *Dynamic Content* region lets you programmatically generate all its HTML markup. Your code constructs a `CLOB` containing any mix of tags and data and returns it for the APEX engine to include in the appropriate place in the page.

- Generating HTML Tags to Present Data
  When requirements demand total control over HTML markup, use a *Dynamic Content* region.
- Using CSS for Styling
  Your Dynamic Content region's generated HTML markup can reference Cascading Style Sheets (CSS) classes to fine-tune its styling.
- Referencing Page Item Values
  As with other region types, you can use a dynamic action to interactively refresh the data a Dynamic Content region displays.

Copyright © 2026, Oracle and/or its affiliates.

5.9.1 Generating HTML Tags to Present Data

When requirements demand total control over HTML markup, use a *Dynamic Content* region.

Harkening back to a time when enterprise applications used character mode displays, consider a simple example of an "ASCII art" chart of employee salaries. Of course using an APEX *Chart* region would look more *polished*, but this is a fun way to reminisce while learning the basics of the Dynamic Content region. The goal is producing an old-school chart that looks like a *Dec VT220* terminal of yore like in the figure below.

Figure 5-25 VT220-Style Chart of Employee Salaries Using Dynamic Content Region



After setting the region type to *Dynamic Content*, you write a PL/SQL block to generate the HTML markup to include in the page. The example below is a function *body* that returns a `CLOB` containing the data and HTML tags the chart requires. First you select the maximum salary used for scaling the salary bars, then loop over the employee rows in descending order by salary. For each employee, you calculate the number of block characters for a 40-character-wide result. The block count is the rounded result of dividing the current employee's salary by the maximum salary to get a percentage, multiplied by 40. Using the `rpad()` function, you produce a string of `l_blocks` bar characters. Then you include this "bar" in the contents of an HTML `<pre>` tag for *preformatted* text, along with the current employee's name and a vertical

Copyright © 2026, Oracle and/or its affiliates.

bar symbol. The code relies on the `p()` local helper function to incrementally "print" each line of HTML tags and data to the `l_content` variable. The PL/SQL block returns this variable as its result at the end.

```
declare
    l_max_sal number;
    l_bar     varchar2(1000);
    l_blocks  number;
    l_content clob;

    procedure p(p_text varchar2)
    is
    begin
        l_content := l_content || p_text;
    end p;
begin
    select max(sal)
      into l_max_sal
      from emp;

    for j in (select ename, sal from emp order by sal desc) loop
      l_blocks := round(j.sal / l_max_sal * 40);
      l_bar := rpad('▒', l_blocks, '▒');
      p('<pre>' || rpad(j.ename, 10) || ' | ' || l_bar
                || ' ' || j.sal || '</pre>');
    end loop;
    return l_content;
end;
```

5.9.2 Using CSS for Styling

Your Dynamic Content region's generated HTML markup can reference Cascading Style Sheets (CSS) classes to fine-tune its styling.

Since you want the result to look like a Dec VT220 terminal, you can define a CSS class named `vt220` in your page's *CSS* > *Inline* section in the Property Editor. Its name begins with a dot and contains the appropriate style property name/value pairs to give a green, monospace font on a black background with a glowing box shadow. To override the *Universal Theme* styling of a `<pre>` tag in the contents of an element with the `vt220` class assigned to it, you include a second CSS rule for the `.vt220 pre` selector to get a tighter line spacing as well.

To assign the `vt220` CSS class to the dynamic content region, simply mention it in the *Appearance* > *CSS Classes* property of the region *without* the dot in front of the name. This combination produces the desired result.

```
.vt220 {
  background-color: black;
  color: #00FF00;
  font-family: 'Courier New', monospace;
  font-size: 14px;
  padding: 1.5em;
  border-radius: 16px;
  border: 3px solid #333;
  box-shadow: 0 0 20px #00ff00aa;
  width: fit-content;
```


Copyright © 2026, Oracle and/or its affiliates.

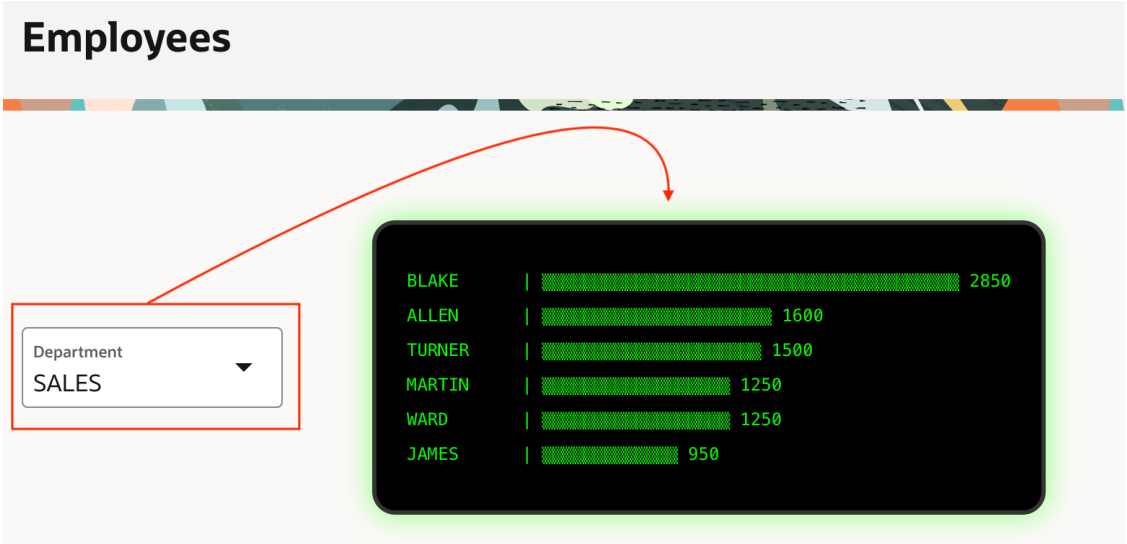
```
  margin: 2em auto;
}
.vt220 pre {
  line-height: 1.0;
}
```

5.9.3 Referencing Page Item Values

As with other region types, you can use a dynamic action to interactively refresh the data a Dynamic Content region displays.

Consider the page shown below having a `P22_DEPARTMENT` select list using 3 of the 12 columns in the layout grid and an old-school ASCII art chart dynamic content region in the same row. Configure a dynamic action on the select list to react to the *Change* event with an action step to *Refresh* the dynamic content region. Then the chart refreshes to show the latest data any time the end user changes the selected department.

Figure 5-26 Dynamic Content Region Using a Page Item Value



To complete the implementation, adjust the dynamic content region's PL/SQL block to reference the `:P22_DEPARTMENT` page item value in the `select max(sal)` query as well as in the query in the `for` loop as shown below:

```
-- Adjusted two queries to reference page item value as bind variable
select max(sal)
  into l_max_sal
  from emp
 where deptno = :P22_DEPARTMENT;

for j in (select ename, sal from emp
           where deptno = :P22_DEPARTMENT
           order by sal desc)
loop
  l_blocks := round(j.sal / l_max_sal * 40);
  l_bar := rpad('▒', l_blocks, '▒');
  p('<pre>' || rpad(j.ename, 10) || ' | ' || l_bar
```


Copyright © 2026, Oracle and/or its affiliates.

```
                || ' ' || j.sal || '</pre>');
    end loop;
```

With these changes in place, the last step is mentioning `P22_DEPARTMENT` in the dynamic content region's ***Page Items to Submit*** property so the APEX engine gets the latest value from the browser automatically whenever the region refreshes.

Copyright © 2026, Oracle and/or its affiliates.

6

Enhancing Search for End-Users

Searching for data is a fundamental part of any enterprise app. Oracle APEX has multiple native components you can use to deliver compelling search experiences tailored to the task at hand.

When end users need to narrow results based on one or more categories of filter criteria, the *Faceted Search* region presents a familiar paradigm to get the job done. The *Smart Filters* region offers similar features in a more compact design, leaving even more space for search results. The *Search* region cooperates with the search configurations you define to format results from across multiple sources in a consistent way. For ultimate control over how end users can specify targeted search criteria, create a query by example page using any of APEX's page item types.

- Narrowing a Search Using Facet Filters
  Use a *Faceted Search* region to let users narrow results by choosing options from groups called *facets*.
- More Space for Results with Smart Filters
  The *Smart Filters* region gives users faceted search functionality in a compact experience.
- Unified Search Results Across Sources
  End users find things on the web by typing a few words into a search bar, reviewing top results, and clicking on promising links to learn more. You can mirror this intuitive search experience in your APEX apps using a *Search* region that references *Search Configurations*.
- Showing Search Results in Other Regions
  You can use the dedicated `SEARCH` API to present results from search configurations in any region. However, when the user filters data with a Faceted Search, Smart Filters, Interactive Report, or Interactive Grid, showing the filtered results in another region requires a different approach.
- Using Page Items for Query by Example
  Page items, including list-driven ones, are useful not *only* for data entry, but also for query by example use cases. The user can enter a value or pick a choice from a list, and another region on the same page can filter *its* data based on their choice.

6.1 Narrowing a Search Using Facet Filters

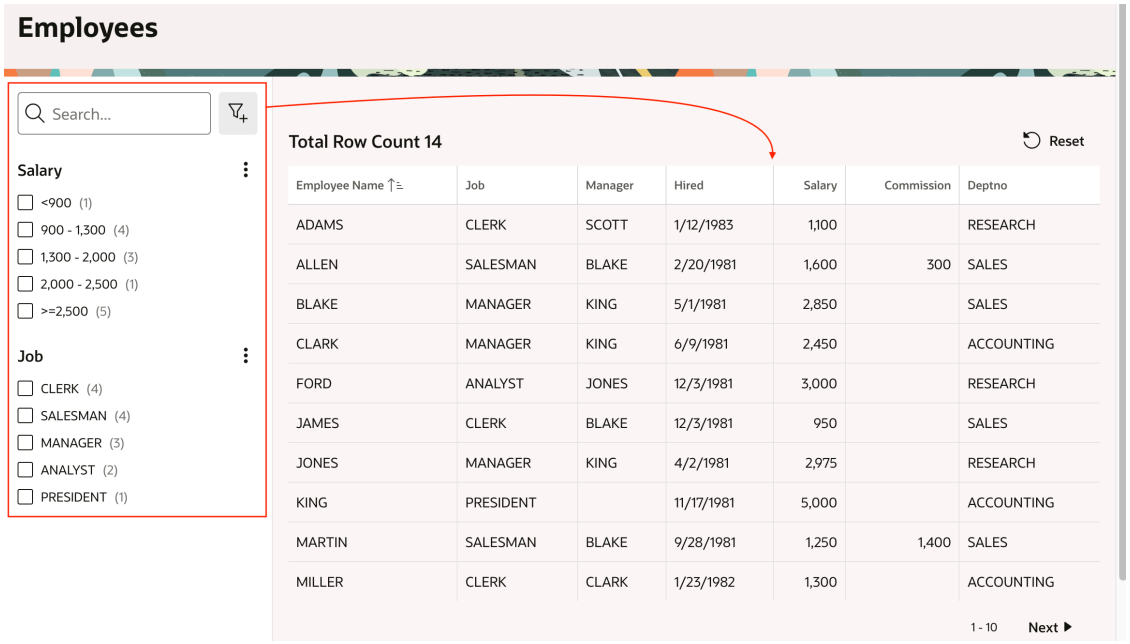
Use a *Faceted Search* region to let users narrow results by choosing options from groups called *facets*.

Each facet reflects specific values of an interesting characteristic on which the user might want to filter. Users familiar with internet shopping sites find the experience intuitive. Just as they search for a perfect laptop by indicating preferences in categories like *Memory*, *Screen Size*, or *Price Range*, they can use the same paradigm to find any kind of data in your APEX app.

As shown below, you typically place a faceted search region in the *Left Column* slot of a page using the *Left Side Column* page template. It can filter a *Classic Report*, *Cards*, or *Map* region as well as *Content Row*, template components, and custom plug-ins that support facets. As a user selects criteria, the results refresh automatically. You control all aspects of how the facet behaves at runtime by configuring appropriate settings in the Property Editor.

Copyright © 2026, Oracle and/or its affiliates.

Figure 6-1 Faceted Search in the Left Column Filtering Classic Report in the Body



- Defining Your Facets
  For each facet, you configure a *Label*, a *Type*, and a *Database Column* to filter on.
- Configuring Range Facets
  For *Range* facets, ensure the source column's data type is `NUMBER`, `DATE`, or `TIMESTAMP`. The return values in the list for the different range intervals uses the vertical bar (`|`) character to separate a low value and high value.
- Behavior of Multiple Selections
  When using a *Checkbox Group* or *Range* facet, if users select multiple options, end users see results that match any of the selected choices as shown below.
- Counts, Dependency, and Charts
  You can enhance the user experience by showing how many rows match each facet option value, optionally sorting by this count, and hiding options with no matches if useful. Facets can depend on other facets, and you can enable a chart view of a facet to help the user see a distribution of results.
- Working with Multi-Value Columns
  If a facet's filtered column potentially stores multiple values in each row, then set the *Multiple Values* property to either *Delimiter* or *JSON Array*. Once selected, additional options appear.
- Displaying Facets in Add Filter Dialog
  When you configure a facet, in the *Appearance* section of the Property Editor you decide whether it will display *inline* or appear in the *Add Filter* dialog.
- Finding Text Across Columns
  By default, every *Faceted Search* region has a *Search* facet that lets users find text across columns. It can appear inline in the region, or reference a page item placed elsewhere on the page.
- Showing Filter Chips and Total Row Count
  Offer users additional filtering feedback with "chips" showing applied filters and a total row count.

Copyright © 2026, Oracle and/or its affiliates.

6.1.1 Defining Your Facets

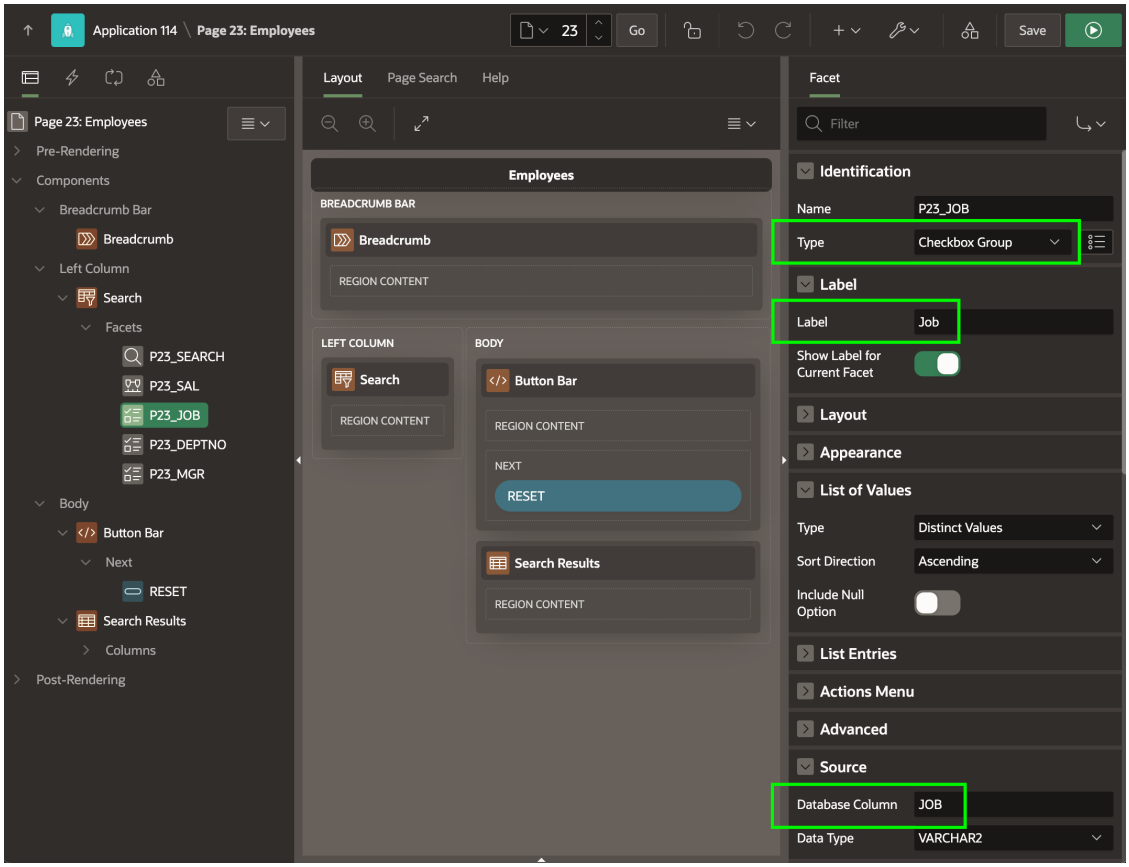
For each facet, you configure a *Label*, a *Type*, and a *Database Column* to filter on.

As shown in the Page Designer below, when using the *Left Side Column* page template, the *Layout* pane shows the *Left Column* and *Body* slots. Notice the *Faceted Search* region in the *Left Column* has five facets defined, with the `P23_JOB` facet selected in the component tree. Facet names look like page item names since they behave identically. At runtime, their values reflect the choices the end user makes in each facet group.

For facets of type *Checkbox Group*, *Radio Group*, and *Select List* you provide a list of values to determine the choices the user sees. You specify the list in the same ways as for a list-valued page item, with one handy additional option: *Distinct Values*. When selected, APEX computes filter options using the distinct values of the underlying column in rows meeting the currently applied filters.

The figure highlights the **Type**, **Label**, and **Database Column** properties of the selected `P23_JOB` facet in Page Designer.

Figure 6-2 Search Facets in Page Designer



6.1.2 Configuring Range Facets

For *Range* facets, ensure the source column's data type is `NUMBER`, `DATE`, or `TIMESTAMP`. The return values in the list for the different range intervals uses the vertical bar (`|`) character to separate a low value and high value.

Copyright © 2026, Oracle and/or its affiliates.

For example, the figure below shows the *Static Values* list for the `P23_SAL` range facet. The *Display Value* can use the substitution string `#G#` for the group (thousands) separator and `#D#` for the decimal separator. Notice how the *Return Value* uses the format:

`LowValue``|`*`HighValue`*.

Omit either a low value or a high value to indicate an interval that is "unbounded" on that side. Notice the option with display value "<900" has a corresponding return value of `|900` with no low value.

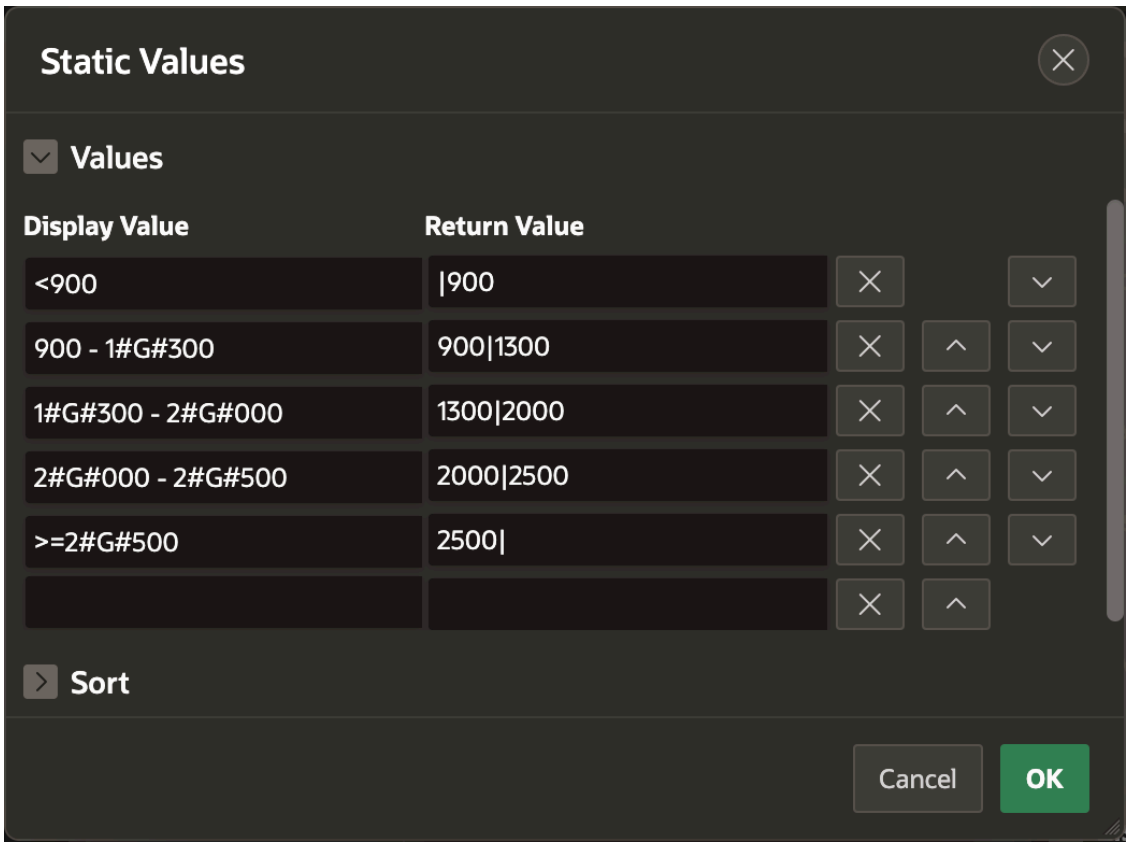
For date and timestamp columns, the low and high values use the canonical format `YYYYMMDDHH24MISS`. For example, a date range option with display value "Before 2020" would have a return value with no low value and a high value representing December 31st, 2019 at 23:59:59 as follows:

```
|20191231235959
```

An option with display value "2020 to 2024" would have a corresponding return value with a low value of January 1st, 2020 at 00:00:00 and a high value of December 31st, 2024 at 23:59:59 like this:

```
20200101000000|20241231235959
```

Figure 6-3 Specifying Range Facet Interval Low and High Values



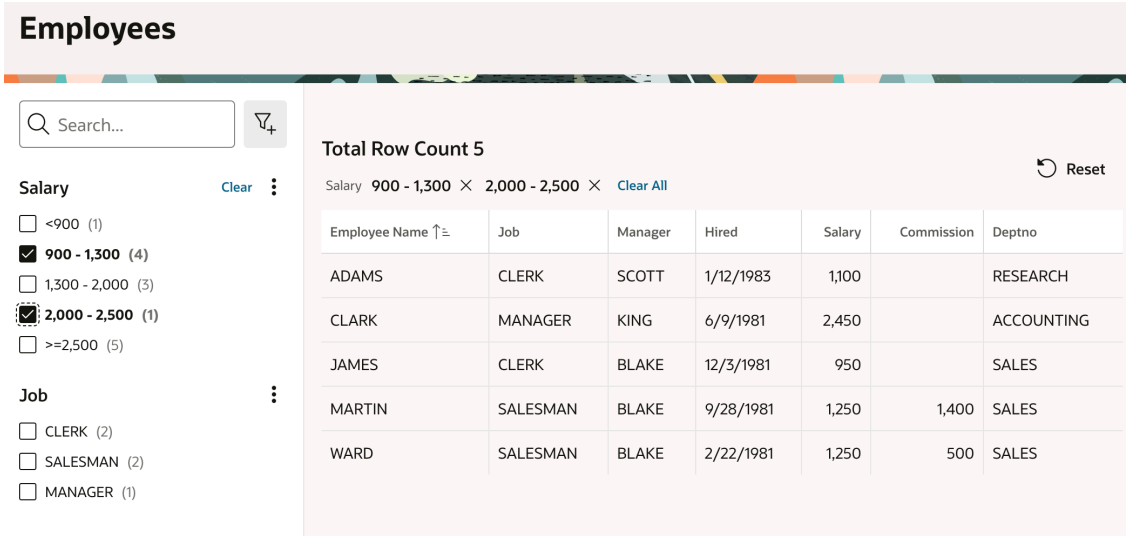
6.1.3 Behavior of Multiple Selections

When using a *Checkbox Group* or *Range* facet, if users select multiple options, end users see results that match any of the selected choices as shown below.

Copyright © 2026, Oracle and/or its affiliates.

The figure shows an *Employees* faceted search page with **Salary** range facet options *"900-1300"* and *"2000-2500"* selected. Employees with salaries in either range display in the results.

Figure 6-4 Results Include Rows Matching Any Choice in Multi-Select Facets



6.1.4 Counts, Dependency, and Charts

You can enhance the user experience by showing how many rows match each facet option value, optionally sorting by this count, and hiding options with no matches if useful. Facets can depend on other facets, and you can enable a chart view of a facet to help the user see a distribution of results.

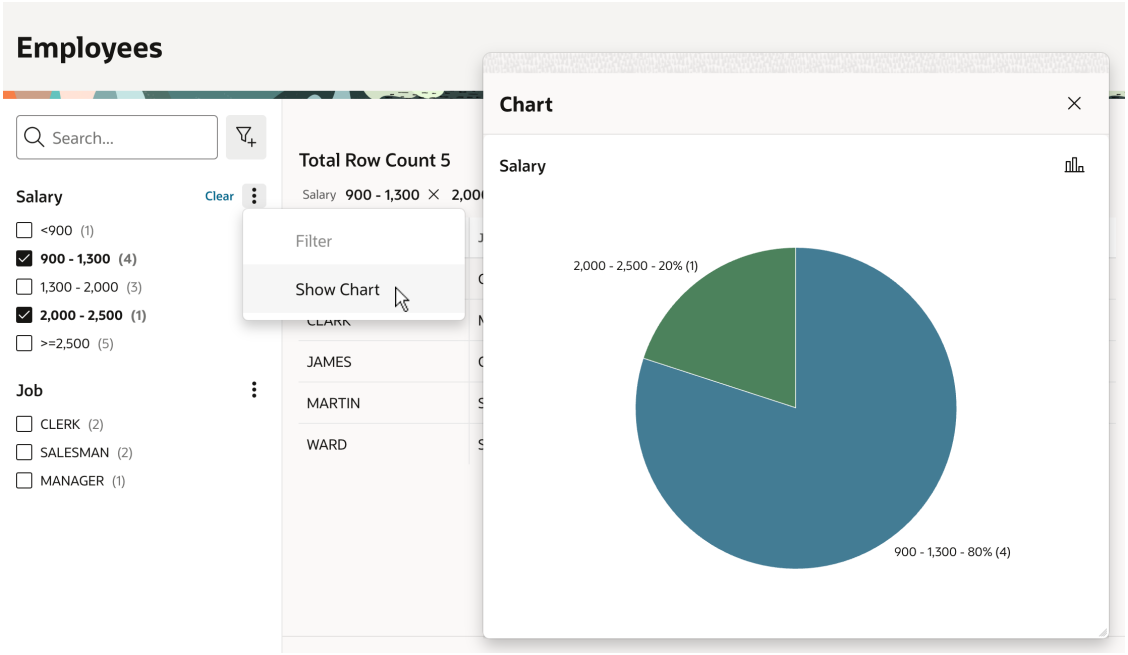
You can include counts to show how many rows match each option and sort options by match count. If you show the counts, you can decide whether zero-count options are hidden. You can also choose to show the facet options in descending order by top counts for *Checkbox Group* and *Radio Group* facets.

One facet can depend on another, so it appears only if the user picks a value from the other one.

As shown below, you can also configure a facet to let end users review a pie or bar chart to understand how filtered rows are distributed across the facet's option values. The figure shows the **Chart** modal dialog that appears when the user chooses **Show Chart** from the **Salary** facet group's kebab menu. Users can toggle between a pie and bar chart to better understand how matching results are distributed across facet values.

Copyright © 2026, Oracle and/or its affiliates.

Figure 6-5 Comparing Facet Value Distribution with Pie or Bar Chart



6.1.5 Working with Multi-Value Columns

If a facet's filtered column potentially stores multiple values in each row, then set the *Multiple Values* property to either *Delimiter* or *JSON Array*. Once selected, additional options appear.

The *Trim Whitespace* setting removes space, tab, or newline characters from the beginning and end of the values. The *Filter Combination* property decides if a row matches when:

- *any* contained value matches some filter option – *OR (Union)*, or
- only when the contained values match *all* filter choices – *AND (Intersect)*.

For example, imagine the `EMP` table had an additional column `INTERESTS` of type `VARCHAR2(1000)` storing multiple values using the colon character as a delimiter. Assume employee `SMITH` has the following value for this column:

```
Art:Animation:Urban Sketching:Math
```

If you configure a `P23_INTERESTS` facet with colon as the multi-value delimiter and a filter combination setting of *AND (Intersect)*, then `SMITH` will match when the end user selects both "Animation" and "Math" group filter options, but employees with `INTERESTS` containing only `Art` or only `Math` will not match.

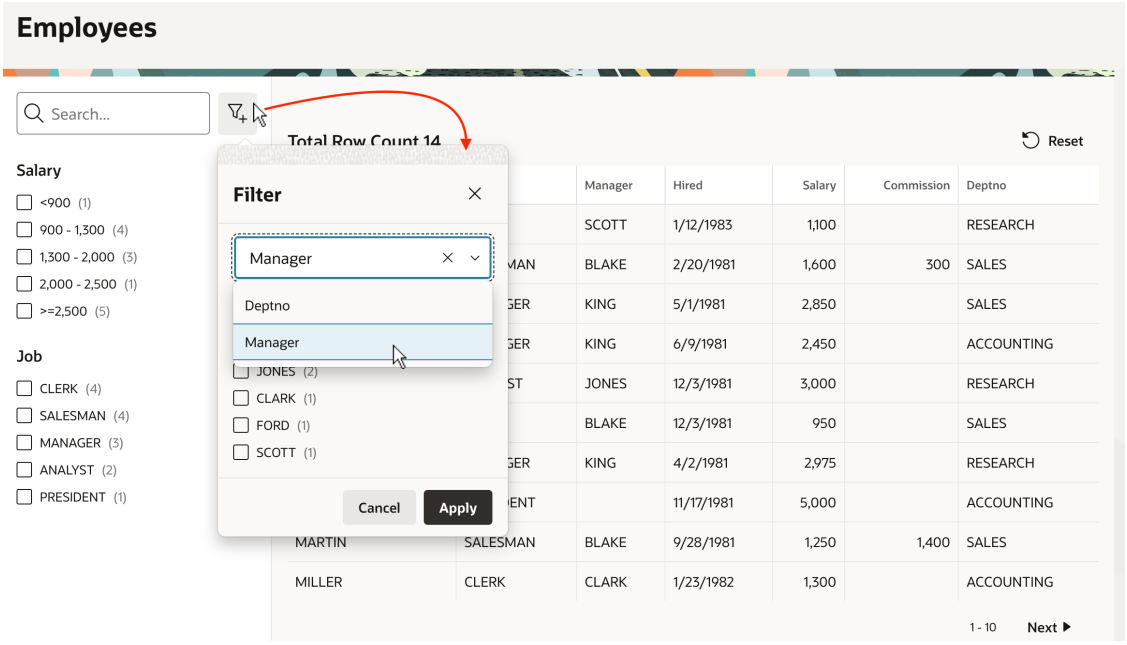
6.1.6 Displaying Facets in Add Filter Dialog

When you configure a facet, in the *Appearance* section of the Property Editor you decide whether it will display *inline* or appear in the *Add Filter* dialog.

As shown below, the end user can click the *Add Filter* button to access a potentially large number of facets using a modal dialog. The *Select One* item at the top of the dialog lets users type to find the facet they want to use. Clicking the *(Apply)* button applies their filter options and dismisses the dialog.

Copyright © 2026, Oracle and/or its affiliates.

Figure 6-6 Applying Filters from the Add Filter Dialog



6.1.7 Finding Text Across Columns

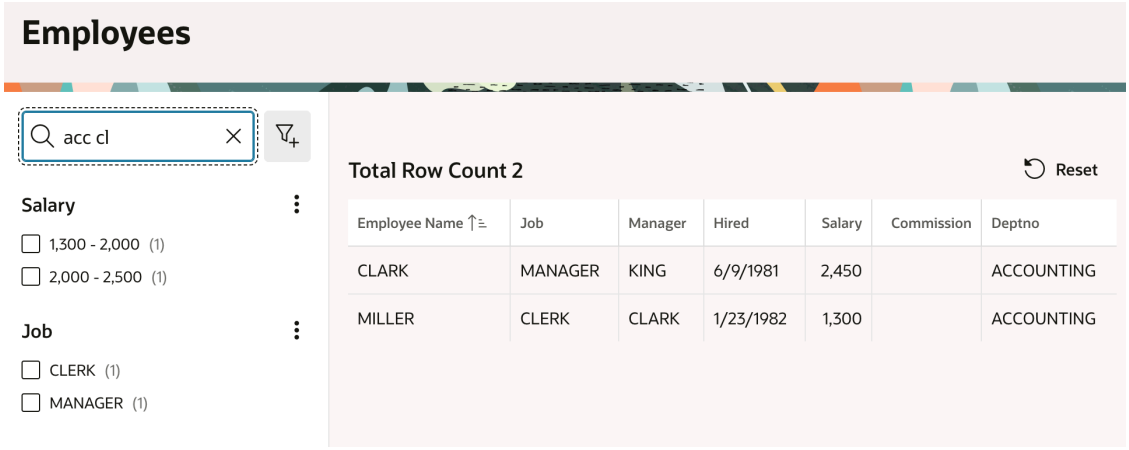
By default, every *Faceted Search* region has a *Search* facet that lets users find text across columns. It can appear inline in the region, or reference a page item placed elsewhere on the page.

Configure that columns the search facet will search over by setting the **Database Columns** property in the **Source** group of the *Search* facet. Use a comma-separated list of column names.

Users type one or more words they are looking for. When they press *[Enter]*, APEX finds rows containing *any* of the words in *any* column you've configured. If other facet filter options are applied, the results include only rows that match *all* of the filters. In the figure below, the user types `acc` followed by `cl` and the matches include one row with `CLARK` in *Employee Name* and `ACCOUNTING` in the *Deptno* column and another with `CLERK` in the *Job* column and `ACCOUNTING` in the *Deptno* column.

Copyright © 2026, Oracle and/or its affiliates.

Figure 6-7 Finding Text Across Columns with the Search Facet



6.1.8 Showing Filter Chips and Total Row Count

Offer users additional filtering feedback with "chips" showing applied filters and a total row count.

Since there can be many *inline* facets and more in the *Add Filter* dialog, users find it useful to see what filter criteria from which facets are applied without scrolling or opening dialogs. Use the ***Show Current Facets*** property of the *Faceted Search* region to show "filter chips" that display this information.

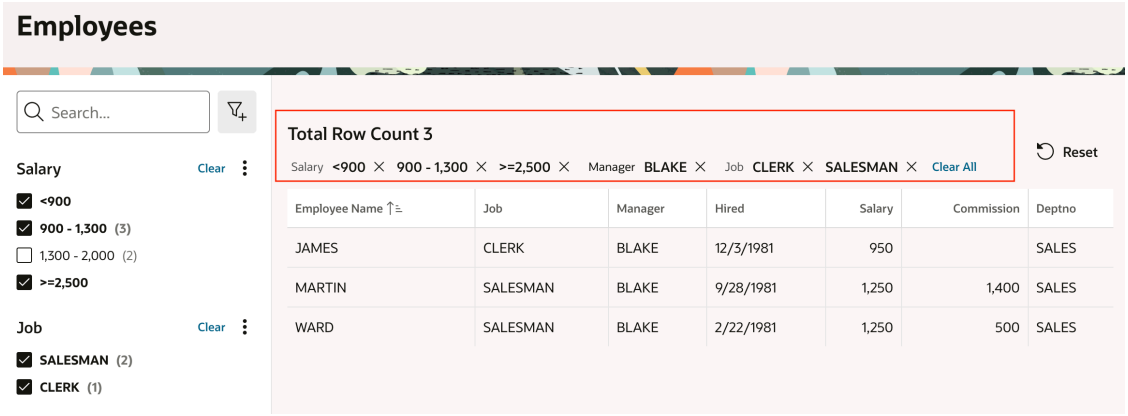
The chips display the facet label and applied filter criteria display value, along with an "x" icon users can click to remove a particular filter. A "Clear All" link lets them remove all filters in a single click as well.

Set ***Show Current Facets*** to *Yes* to show the chips at the top of the faceted search region itself. However, it's usually in the *Left Column* slot that is narrow by design. Instead, set the *HTML DOM ID* of a static content region in the *Body* slot to a unique name like `active_facets` and then set ***Show Current Facets*** to *Selector*. This setting causes the additional *Current Facets Selector* to appear in the Property Editor. After setting this to `#active_facets`, referencing the HTML DOM ID of the static content region, the filter chips will display wherever on the page this static content region is positioned.

Set the ***Show Total Row Count*** property of the faceted search region to *Yes* to have the total row count appear at the top of the current facets area. For more flexibility, you can set this property to *Selector* and provide a CSS selector like `#total_count_area` to show the count with no label in the static content region having the HTML DOM ID of `total_count_area`. The figure below shows a faceted search page with both options enabled.

Copyright © 2026, Oracle and/or its affiliates.

Figure 6-8 Row Count and Filter "Chips" Over Classic Report

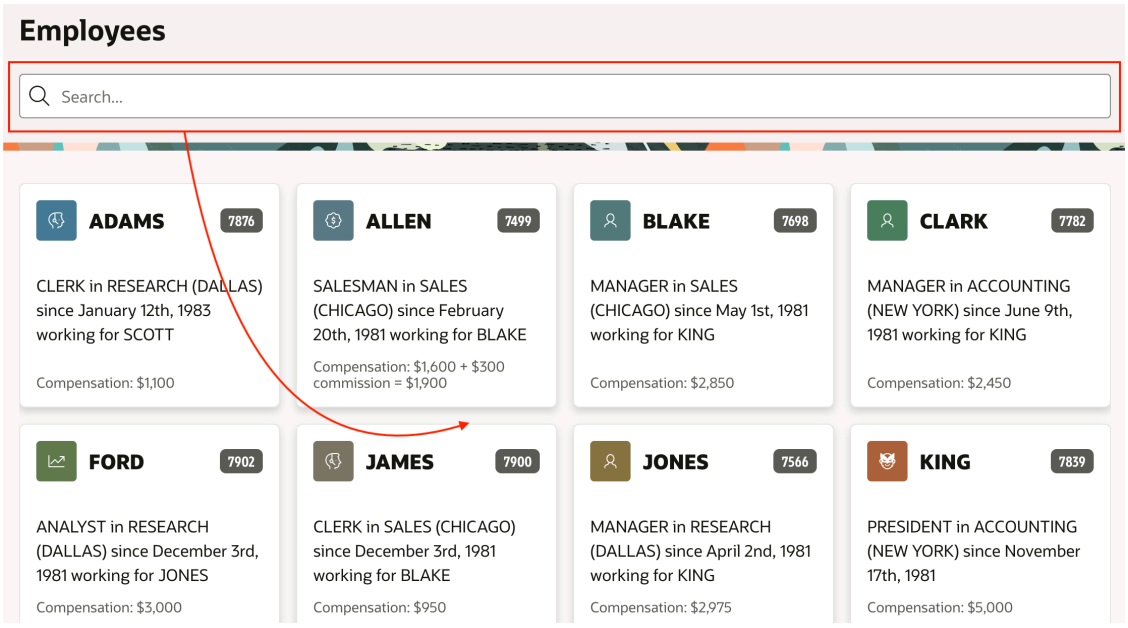


6.2 More Space for Results with Smart Filters

The *Smart Filters* region gives users faceted search functionality in a compact experience.

It's perfect when screen space is limited or when your search results need the full page width. It can filter a *Classic Report*, *Cards*, or *Map* region as well as *Content Row*, template components, and custom plug-ins that support facets. As shown below, it occupies only the space of a single search field on the page.

Figure 6-9 Compact Smart Filters Region is a Single Search Field



- Defining Your Filters
  For each filter, you configure a *Type*, a *Label*, and a *Database Column* to filter on.
- Smart Filters User Experience
  Learn about the user experience for Smart Filters.

Copyright © 2026, Oracle and/or its affiliates.

- Configuring Suggestions
  Sometimes end users appreciate a suggestion for common or popular filters to apply. When no other filters are applied, the Smart Filters region can display suggestion "chips" to help a user get started.
- Even More Space for Search Results
  When you want to dedicate even *more* space to the search results, you can move the Smart Filters region into the Navigation Bar.

6.2.1 Defining Your Filters

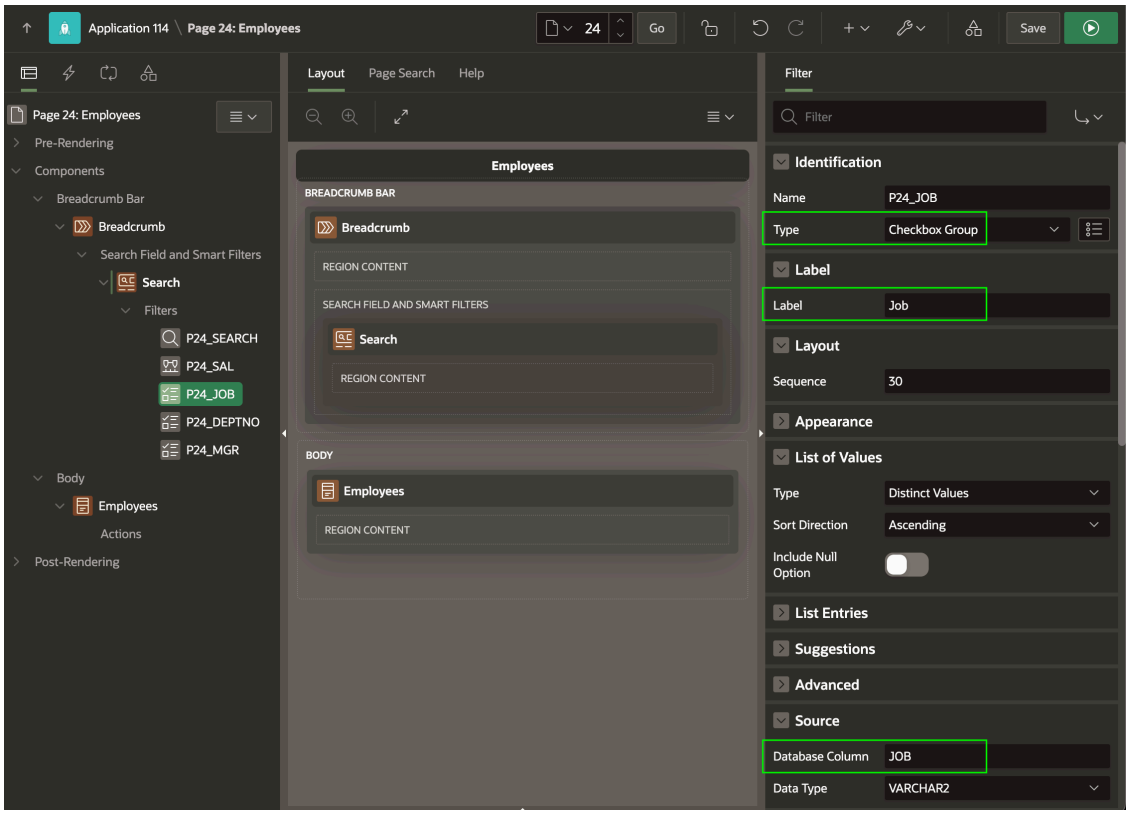
For each filter, you configure a *Type*, a *Label*, and a *Database Column* to filter on.

As shown below, the *Smart Filters* region in the breadcrumb's *Search Field and Smart Filters* slot has five filters defined, with the `P24_JOB` filter selected in the component tree. Filter names look like page item names since they behave the same. At runtime, their values reflect the choices the end user makes in each filter.

For filters of type *Checkbox Group* and *Radio Group* you define a list of values to provide the choices the user sees. You configure the list in the same ways as for a list-valued page item, with one extra option: *Distinct Values*. When selected, APEX shows filter options using the distinct values of the underlying column in rows matching the currently applied filters.

Configuring range facets, counts, dependency, and multi-valued columns works the same way as for the *Faceted Search* region's facets described in the previous section. When computing counts, you can show the filter options in descending order by top counts for *Checkbox Group* and *Radio Group* filters.

Figure 6-10 Defining Multiple Filters in Page Designer



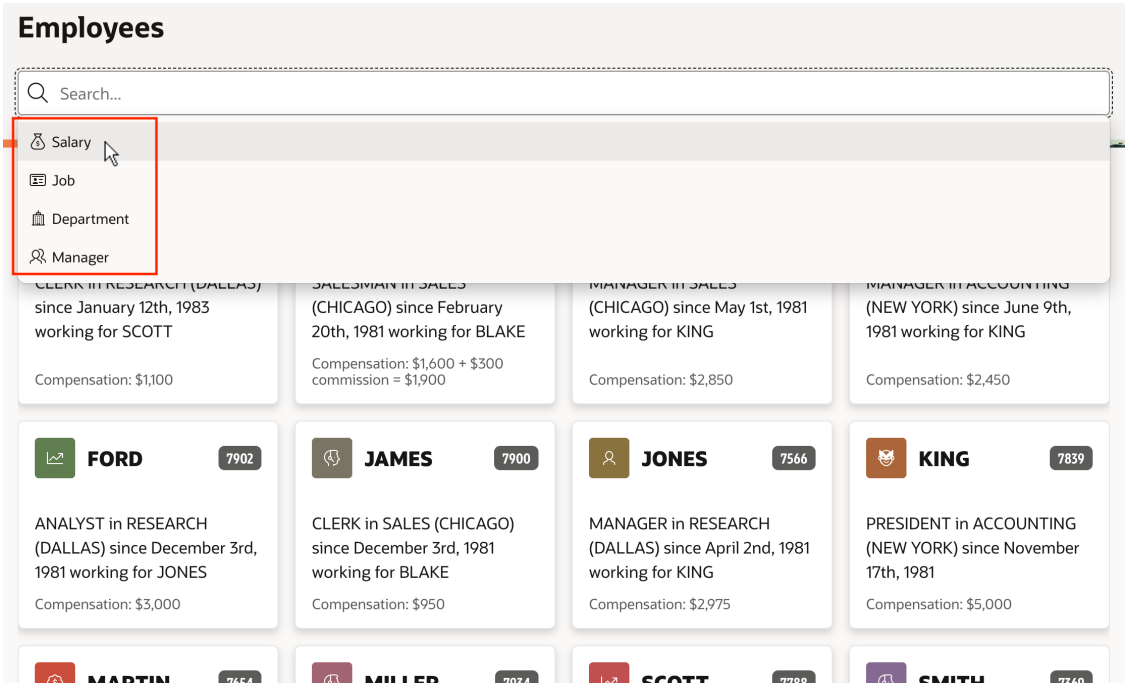
Copyright © 2026, Oracle and/or its affiliates.

6.2.2 Smart Filters User Experience

Learn about the user experience for Smart Filters.

When a user navigates to the Smart Filters search field, as shown below, they see the display name and icon of each filter you've defined.

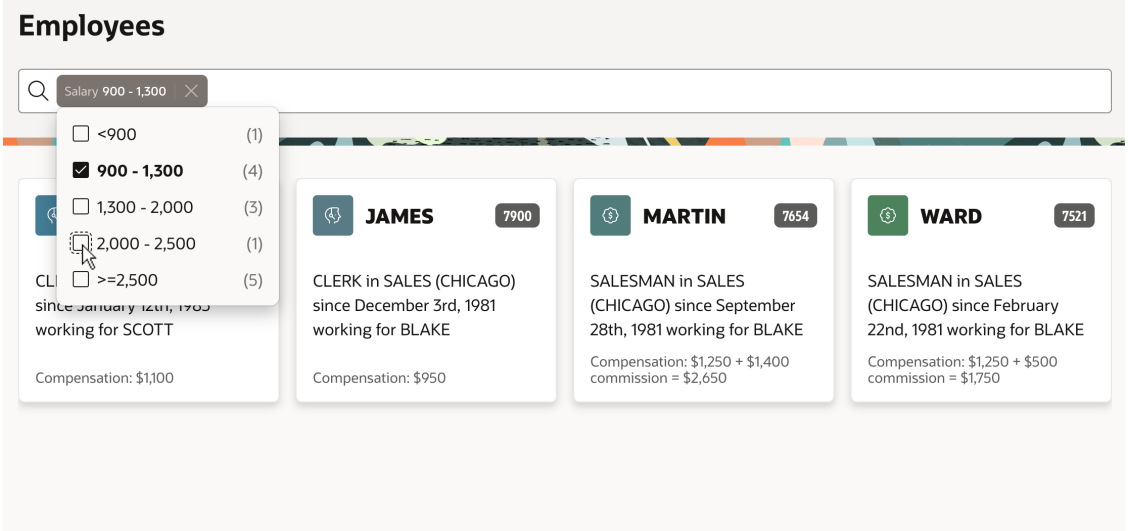
Figure 6-11 Filter Display Names and Icons Appear in Search Dropdown



When they select a particular filter like *Salary*, as shown below, a corresponding "chip" appears in the search field with the filter display name and the filter options show in a dropdown. If configured to allow multiple selections, the user can do so. As the user selects filter options, the filtered region immediately updates to reflect the narrowed results.

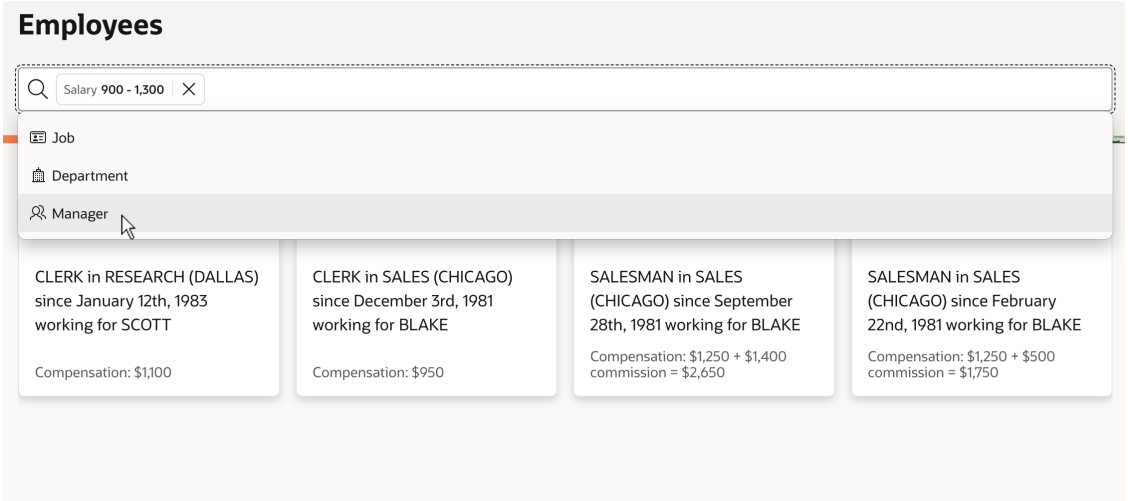
Copyright © 2026, Oracle and/or its affiliates.

Figure 6-12 Selecting One or More Options from a Filter



Once a filter is in use, as shown below, its chip stays visible in the search field to represent applied filters and the dropdown list shows remaining filters. A user can navigate to the *Salary* filter chip to revise their choices or click its "x" icon to remove it.

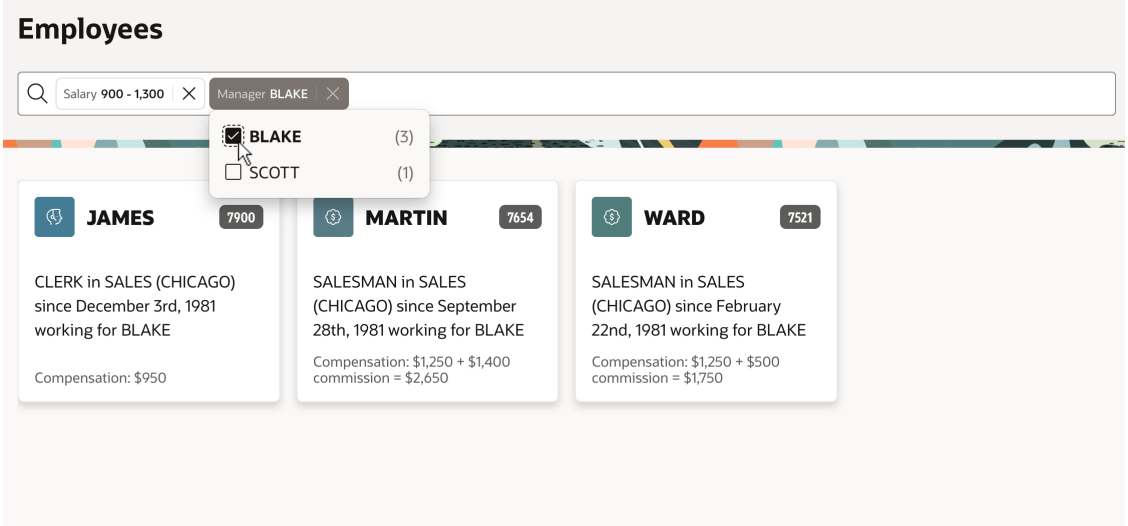
Figure 6-13 One Filter Used, Other Filters Remain Available



If the user picks the *Manager* filter and selects *BLAKE* an additional chip appears and results refresh to show employees *BLAKE* manages whose salary is between 900 and 1300.

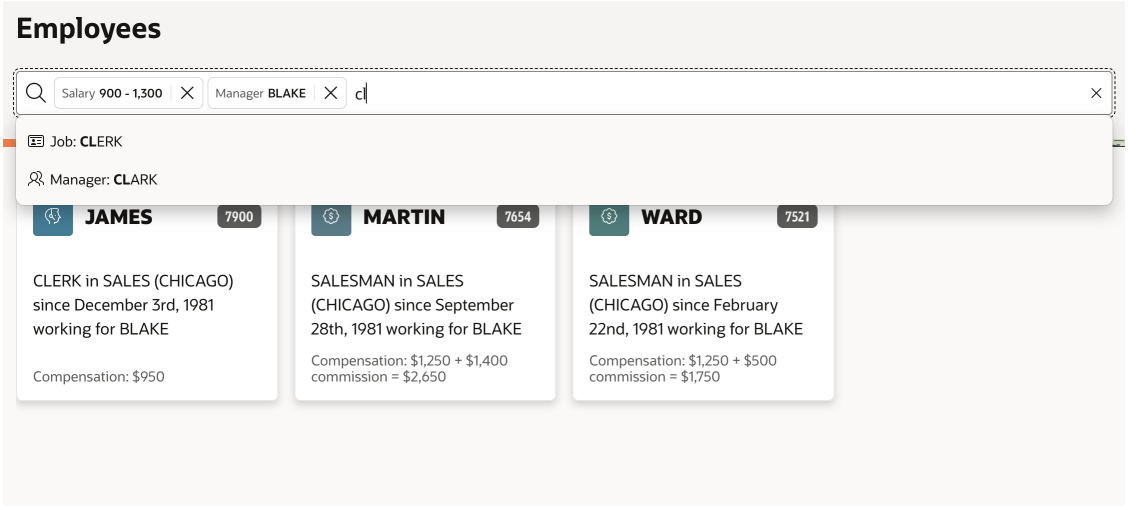
Copyright © 2026, Oracle and/or its affiliates.

Figure 6-14 Applying Multiple Filters Using a Smart Filters Region



At any time the user can type into the search field to enter one or more terms to find across columns. You configure *which* columns APEX searches using the *Database Columns* property of the *Search* filter. As shown below, as the user types into the search field any matching filter options appear in the dropdown as well. For example, after typing `cl`, suggestions show for *Job: CLERK* and *Manager: CLARK*. The user can keep typing to complete their search term and press *[Enter]* to see the results. Alternatively, they can pick one of the suggested filter options.

Figure 6-15 Typing Text to Search Across Columns or Find Filters



6.2.3 Configuring Suggestions

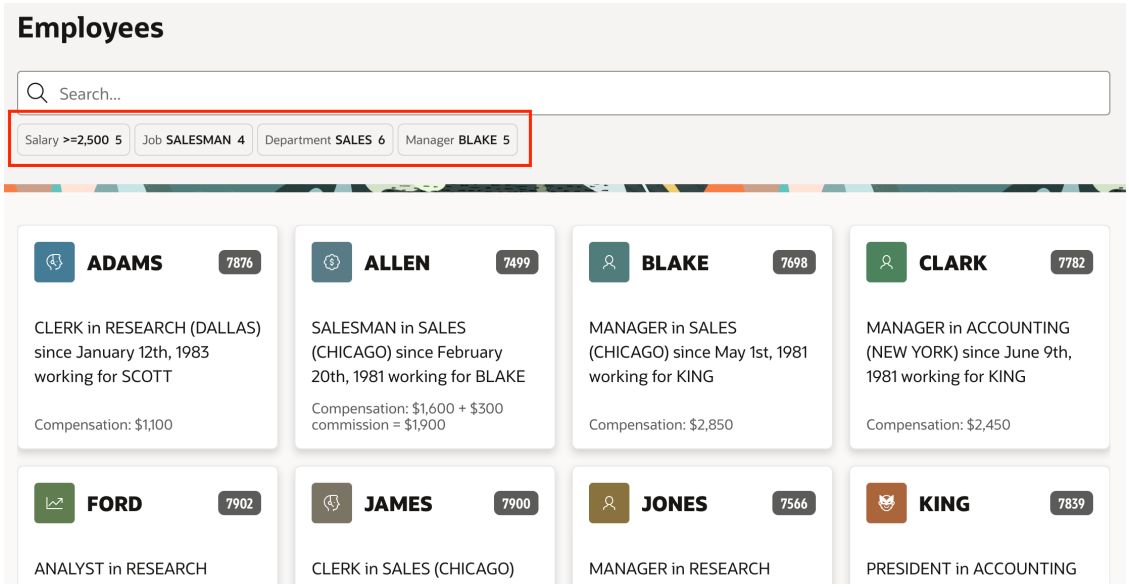
Sometimes end users appreciate a suggestion for common or popular filters to apply. When no other filters are applied, the Smart Filters region can display suggestion "chips" to help a user get started.

Copyright © 2026, Oracle and/or its affiliates.

For each additional filter you add beyond the required *Search* one, you can configure its suggestion chip behavior with its *Suggestions* > *Type* property. Set to *Dynamic* to show the first visible filter value. If the filter computes counts, then it will be the most frequently occurring filter option. Otherwise, it's the first option in sorted order. Use a *Type* of *Static Values* to set a hard-coded value. The *SQL Query* option gives you a flexible way to configure the suggestion based on a custom scheme, perhaps involving the current username (`:APP_USER`) or any other data-driven approach. In all cases, the suggestion value must be one of the return values defined in the filter's list of values. To have no suggestion for a particular filter, its its *Suggestions* > *Type* to *None*.

You can limit the number of suggestion chips using the *Maximum Suggestion Chips* region property. If left unset, then up to five suggestions can appear, one per suggestion-enabled filter. To suppress suggestions if they don't add value to a particular Smart Filters usage, set this property to zero (`0`).

Figure 6-16 Suggestion Chips For Initial Filter to Apply



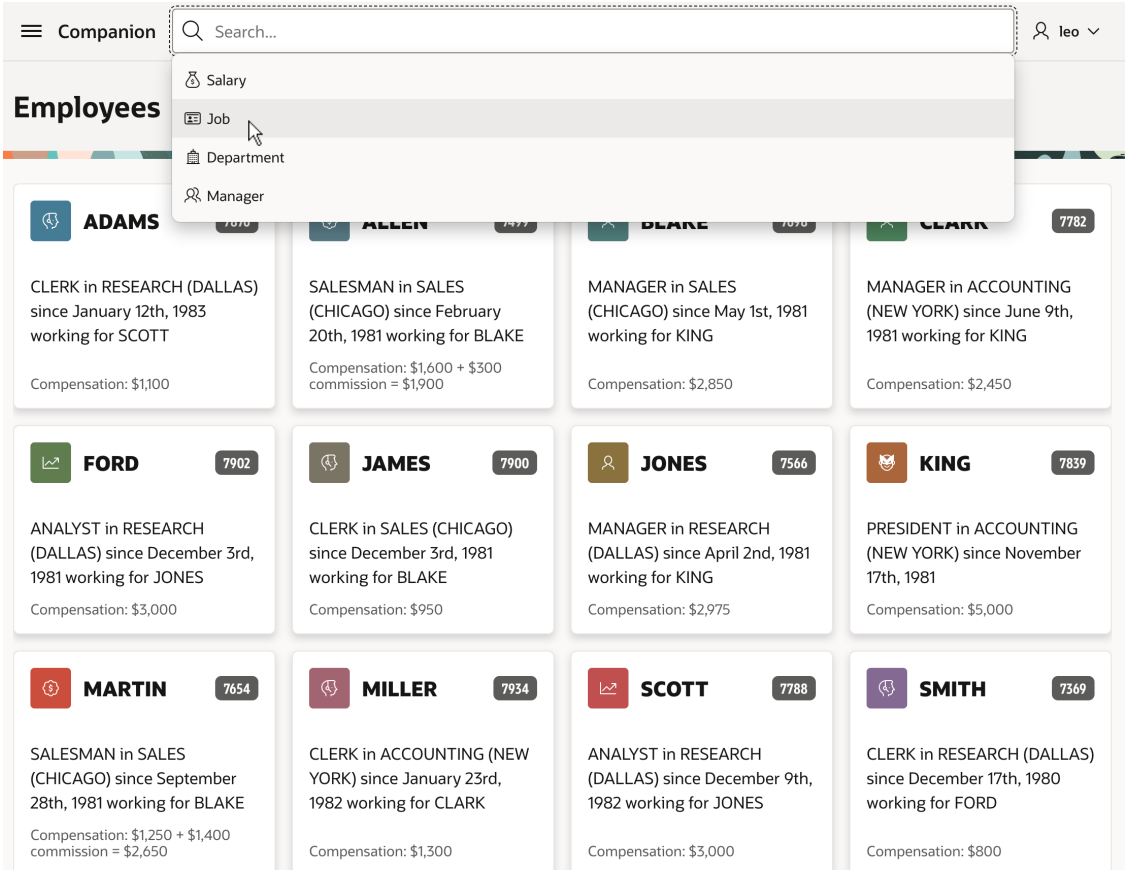
6.2.4 Even More Space for Search Results

When you want to dedicate even *more* space to the search results, you can move the Smart Filters region into the Navigation Bar.

By setting the Smart Filters region's *Parent Region* property to *No Parent*, choosing the *Before Navigation Bar* slot, and setting *CSS Classes* to `w100p`, you can make it occupy the empty space in the nav bar as shown below. This position works best with *Maximum Suggestion Chips* set to zero (`0`). The figure shows the Smart Filters search field tucked into the navigation bar between the **Companion** application name on the left and the *Navigation Bar* showing the logged-in username **leo** on the right.

Copyright © 2026, Oracle and/or its affiliates.

Figure 6-17 Smart Filters Region in the Navigation Bar



6.3 Unified Search Results Across Sources

End users find things on the web by typing a few words into a search bar, reviewing top results, and clicking on promising links to learn more. You can mirror this intuitive search experience in your APEX apps using a *Search* region that references *Search Configurations*.

- Defining a Search Configuration
  Define a *Search Configuration* shared component for each kind of *source* data users need to search. Each one identifies a local or remote data source and its searchable columns. You also choose which columns play standard roles like primary key, title, description, and icon.
- Using Search Configurations
  After adding a *Search* region to a page, reference one or more search configurations as search sources.
- Search Results Formatting Options
  You can customize how unified search results appear in three ways.

6.3.1 Defining a Search Configuration

Define a *Search Configuration* shared component for each kind of *source* data users need to search. Each one identifies a local or remote data source and its searchable columns. You also choose which columns play standard roles like primary key, title, description, and icon.

Copyright © 2026, Oracle and/or its affiliates.

To let users drill-down on a result from a particular search configuration, configure a link target on it. Users will navigate to the indicated page when they click a search result that comes from that source.

Depending on the search configuration *type*, APEX performs:

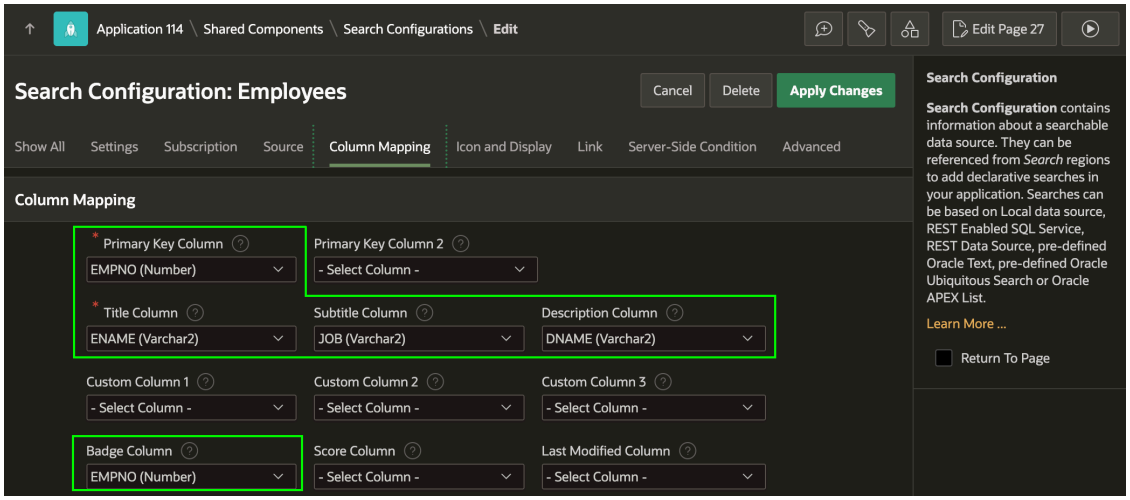
- Standard `LIKE` filtering over local tables, queries, APEX list entries, or REST Data Sources
- Full text searching using Oracle Text or Oracle AI Database 26ai *Ubiquitous Search*, or
- Semantic similarity searching using Oracle AI Database 26ai *Vector Search*.

For example, the *Employees* search configuration shown in the editor below uses the following SQL query as its *Source* and identifies searchable columns: `ENAME`, `JOB`, `DNAME`, `EMPNO`, and `WORKS_FOR_ENAME`.

```
select  e.empno,
        e.ename,
        m.ename as works_for_ename,
        d.dname,
        e.job,
        case e.job
           when 'PRESIDENT' then 'fa-badgerine'
           when 'ANALYST'   then 'fa-line-chart'
           when 'CLERK'     then 'fa-user-headset'
           when 'SALESMAN'  then 'fa-badge-dollar'
           when 'MANAGER'   then 'fa-user'
        end as icon
  from  emp e
  join  dept d
    on  e.deptno = d.deptno
  left join emp m
         on m.empno = e.mgr
```

As shown below, it maps `EMPNO` as primary key, `ENAME` as the title, `JOB` as subtitle, `DNAME` as description, and shows a badge using the `EMPNO` column value.

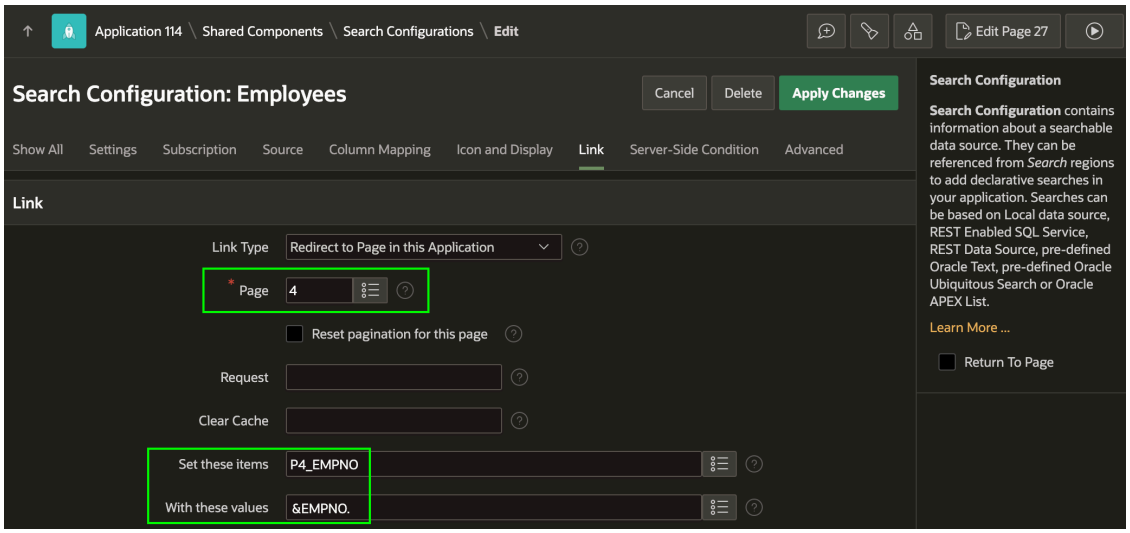
Figure 6-18 Mapping Search Configuration Columns to Standard Slots



Copyright © 2026, Oracle and/or its affiliates.

After configuring that the search result icon name comes from the `ICON` column on the *Icon and Display* tab, you define the link information. As shown below, when the user clicks on a search result from this *Employees* search source, they should navigate to a modal *Employee* edit page number 4, passing the value of the result's `EMPNO` column as the `P4_EMPNO` parameter. The figure shows the *Link* section of the *Search Configuration* edit page where your set up these options.

Figure 6-19 Defining a Search Configuration's Link Target Page



In a similar way, you could define a *Departments* search configuration based directly on the `DEPT` table using the `fa-building-o` icon for every row, mapping columns appropriately, and setting the link information to open a *Department* edit page.

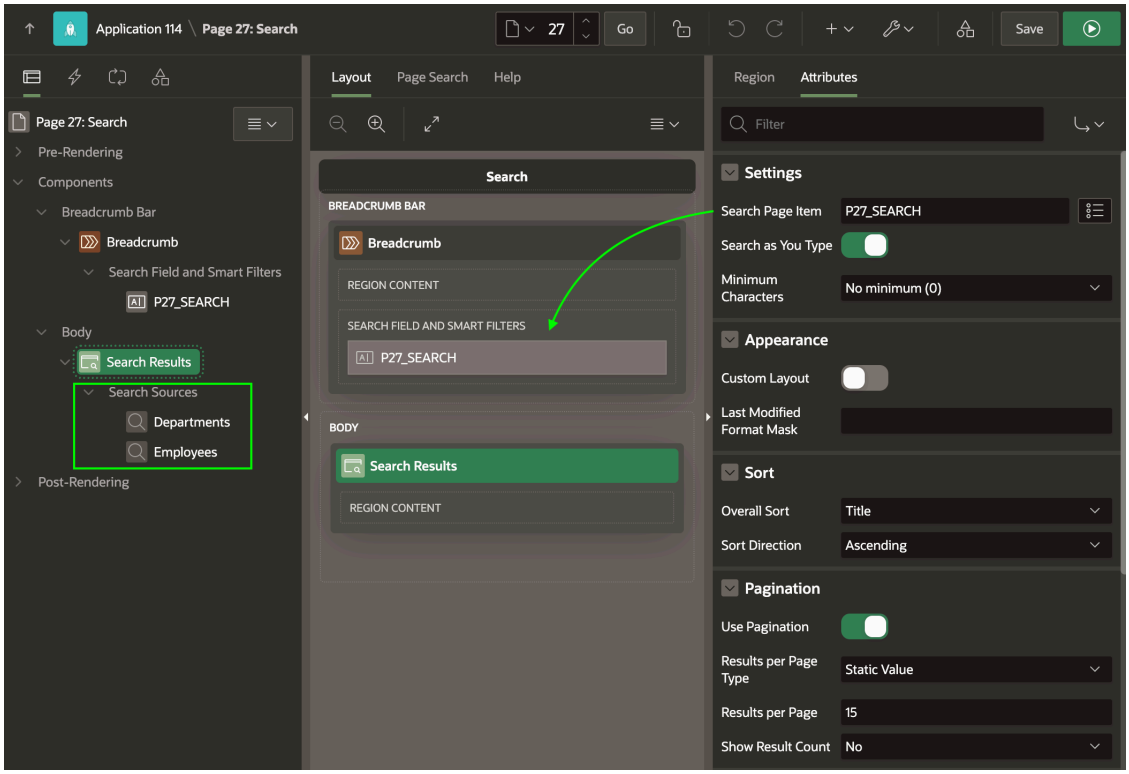
6.3.2 Using Search Configurations

After adding a *Search* region to a page, reference one or more search configurations as search sources.

For example, the figure below shows a Search region in Page Designer, with *Employees* and *Departments* search configurations added as sources. By default a search region shows an integrated search field above the results area. For more layout flexibility, as shown below you can add your own `P27_SEARCH` page item that you position where you want and reference it in the region's *Search Page Item* property. Using other settings, you control how sorting and pagination of search results behave. If needed, you can configure a *Custom Layout* in the *Appearance* section to format results from all sources in a consistent, custom way. When doing so, your HTML can reference template directives for conditional output as well as use substitutions with names like `TITLE`, `SUBTITLE`, `DESCRIPTION`, `ICON`, `LINK`, and more. See the **Help** tab in Page Designer for the *Result Row Template* property for the complete list.

Copyright © 2026, Oracle and/or its affiliates.

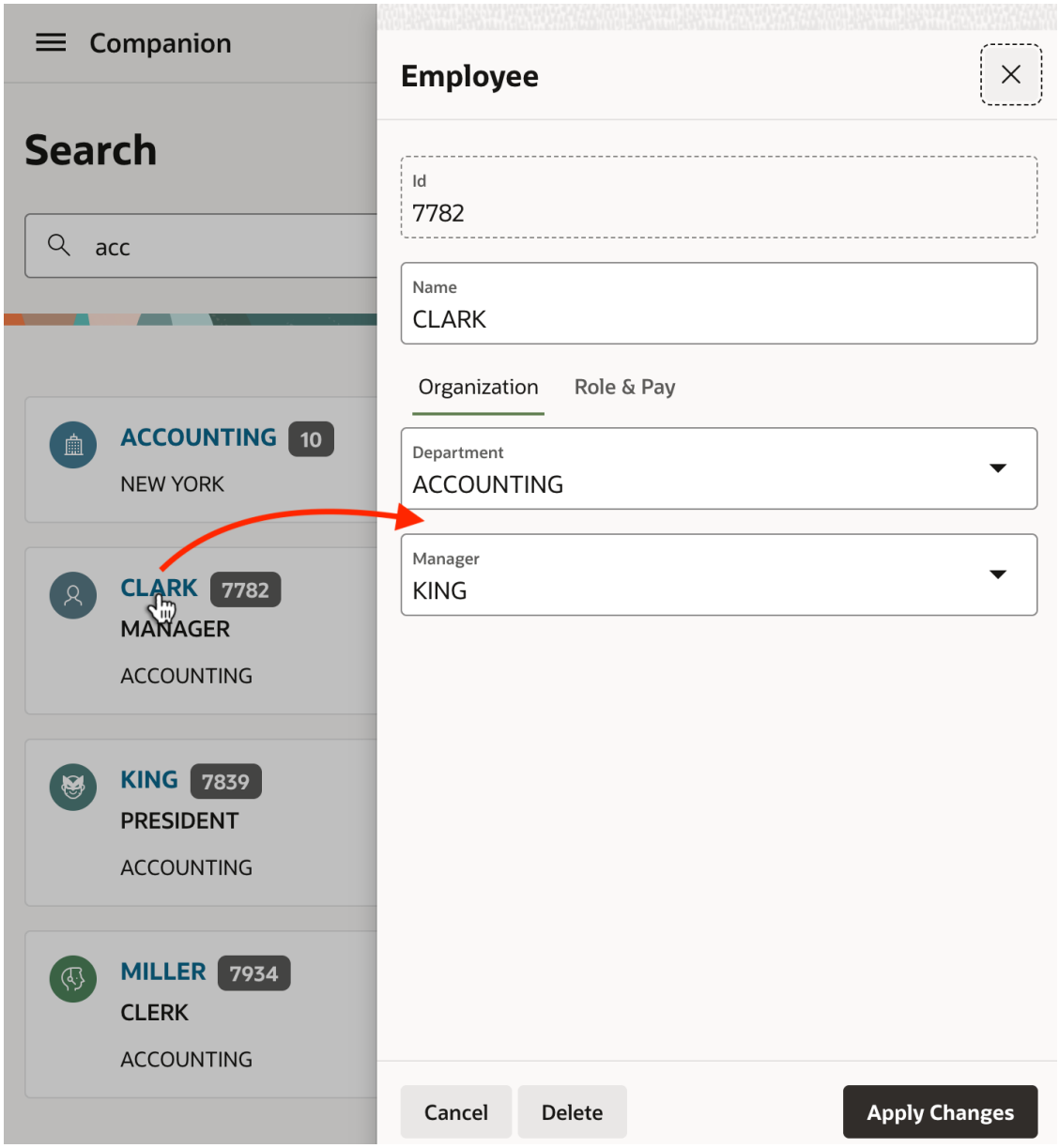
Figure 6-20 Configuring a Search Region with Search Configurations



At runtime, as shown below, when the user types a term into the search field like acc matching results from both the *Departments* and *Employees* sources appear in the unified list of results. Clicking on a result like *CLARK* who works in *ACCOUNTING*, the appropriate employee-related target page opens to show the details.

Copyright © 2026, Oracle and/or its affiliates.

Figure 6-21 Finding and Drilling Down on Search Results



6.3.3 Search Results Formatting Options

You can customize how unified search results appear in three ways.

Using a combination of a *Search* region with one more *Search Configurations*, you can customize how your unified search results appear by:

1. Defining custom CSS rules to change how the default search results appear,
2. Providing a custom *Result Row Template* on the search region,
3. Configuring a *Default Result Row Template* on some or all of your search configurations

The CSS approach is good for simpler changes to spacing, fonts, and colors. The other two options give you full formatting control. Define a *Default Result Row Template* on a search

Copyright © 2026, Oracle and/or its affiliates.

configuration to style its result row. It can reference column names from the data source as substitutions. This template gets used automatically unless you configure the *Result Row Template* on the *Search* region to format results of all sources in a consistent way.

- Custom Search Region Row Template
  By default, the search region formats results using a *built-in* result row template when a search configuration does not specify its own. This "blueprint" combines HTML tags with `&NAME.` references and HTML directives for conditional formatting.
- Custom Search Configuration Template
  To format a *specific* search configuration's results in a custom way, use a *Default Result Row Template*. Find this setting in the *Icon and Display* section of the shared component's edit page.

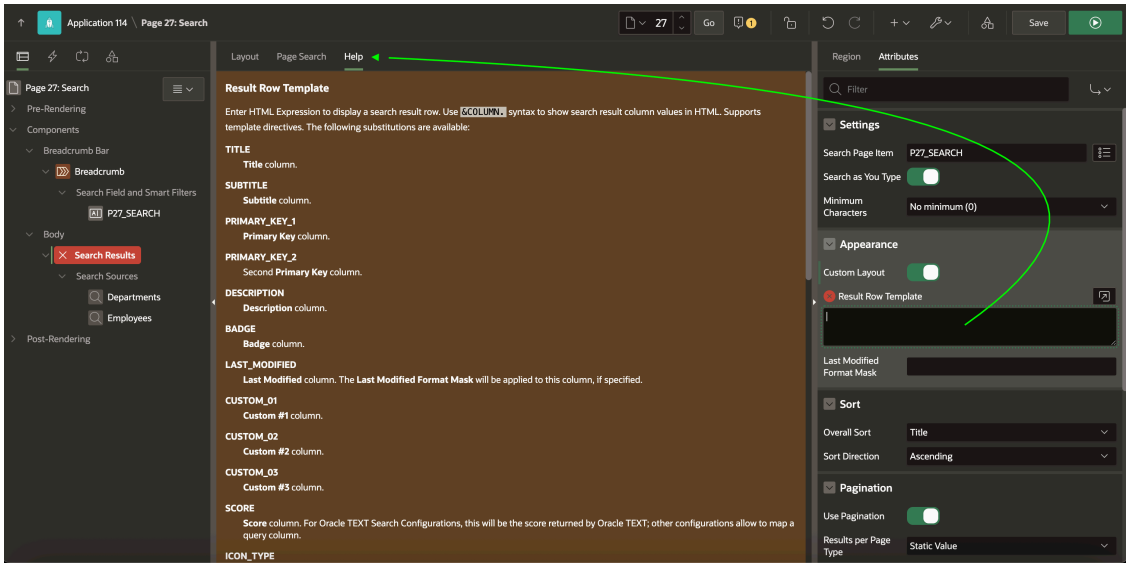
6.3.3.1 Custom Search Region Row Template

By default, the search region formats results using a *built-in* result row template when a search configuration does not specify its own. This "blueprint" combines HTML tags with `&NAME.` references and HTML directives for conditional formatting.

For example, if a search configuration provides a badge column, then certain HTML tags will appear in the rendered output to properly display the badge value. Otherwise, the badge markup is omitted.

To change how *all* search results appear, specify a *Result Row Template* on the search region. As shown below, after enabling the *Custom Layout* switch, the *Help* tab for the *Result Row Template* property lists all of the standard substitution strings you can reference using `&NAME.` syntax. Scrolling the *Help* tab page, you can also find the built-in result row template.

Figure 6-22 Consulting Help Tab for Search Region Result Row Template



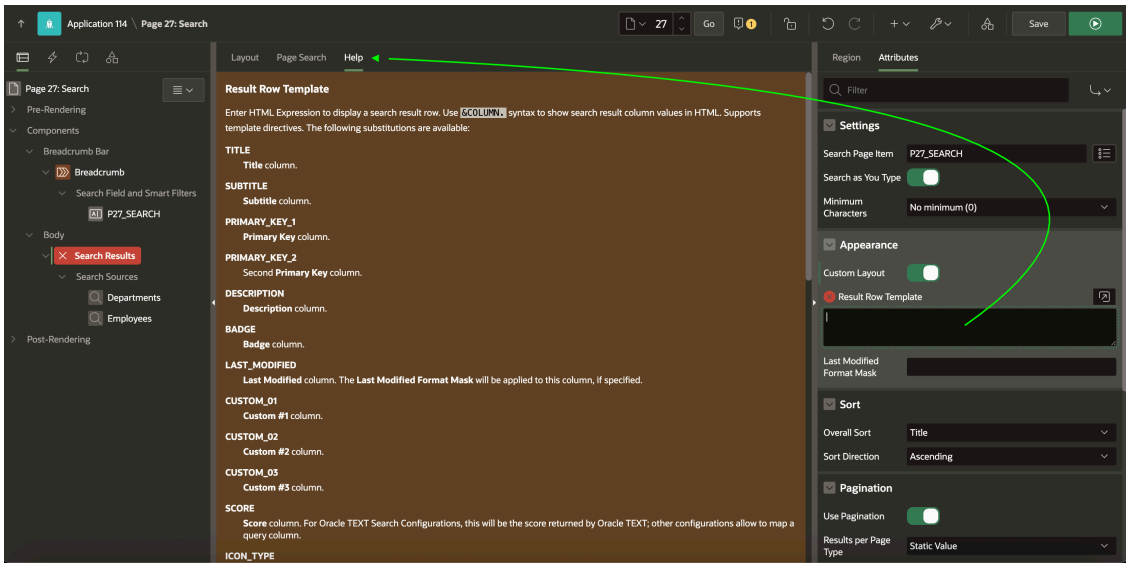
6.3.3.2 Custom Search Configuration Template

To format a *specific* search configuration's results in a custom way, use a *Default Result Row Template*. Find this setting in the *Icon and Display* section of the shared component's edit page.

Copyright © 2026, Oracle and/or its affiliates.

Your template can use HTML tags, data source `&COLUMNNAME.` substitutions, and HTML template directives for conditional formatting. As shown below, the inline help dialog mentions additional substitution strings available and provides an example template for inspiration. APEX uses your custom search configuration's default result row template unless a search region provides one that overrides it to format all results in the same way.

Figure 6-23 Consulting Help Tab for Search Region Result Row Template



6.4 Showing Search Results in Other Regions

You can use the dedicated `SEARCH` API to present results from search configurations in any region. However, when the user filters data with a Faceted Search, Smart Filters, Interactive Report, or Interactive Grid, showing the filtered results in another region requires a different approach.

- Custom Unified Search Experience
  The *Search* region manages a search field and displays results using result row templates. When this region's user interface doesn't meet your requirements, you can build a custom search experience using the `SEARCH` function in the `APEX_SEARCH` package.
- Showing Filtered Data in Another Region
  To show one region's data in another region, you can reference the primary keys of a source region's search results in another region's query or `WHERE` clause. Whenever the source region's results change, a dynamic action can automatically refresh the target region.

6.4.1 Custom Unified Search Experience

The *Search* region manages a search field and displays results using result row templates. When this region's user interface doesn't meet your requirements, you can build a custom search experience using the `SEARCH` function in the `APEX_SEARCH` package.

This function accepts a search term and list of search configuration static IDs. It returns a table of search results you can use in the `FROM` clause of a SQL query source in any APEX region. Simply wrap the function call with the `table()` operator. Each result row contains standard

Copyright © 2026, Oracle and/or its affiliates.

column names that match the substitutions available in the Search region's *Result Row Template*. This example query returns the results of searching the regions with static ID `employees` and `departments` using a search term from the `P31_SEARCH` page item:

```
select *
  from table(
         apex_search.search(
           p_search_expression => :P31_SEARCH,
           p_search_static_ids => apex_t_varchar2('employees',
                                                  'departments')))
```

If you include an `ORDER BY` clause in the query, then pass one additional parameter to avoid unnecessary sorting as follows:

```
select *
  from table(
         apex_search.search(
           p_search_expression => :P31_SEARCH,
           p_search_static_ids => apex_t_varchar2('employees',
                                                  'departments'),
           p_apply_order_bys   => 'N'))
 order by title
```

This approach lets you collect the search term in any way necessary and display results in *Cards*, *Map*, *Chart*, *Classic Report*, template component, or any other kind of APEX region.

See the *Oracle APEX API Reference* APEX_SEARCH for the full list of standard columns you can reference. The list includes columns like `PRIMARY_KEY_1` and `PRIMARY_KEY_2`, `TITLE`, `SUBTITLE`, `DESCRIPTION`, and `BADGE`. You can also map three custom columns in your search configuration that show up in the results. For example, if your search configuration contains a `LATITUDE` and `LONGITUDE` column, you can map those to *Custom Column 1* and *Custom Column 2* and then refer to their values in the search results using columns `CUSTOM_01` and `CUSTOM_02`.

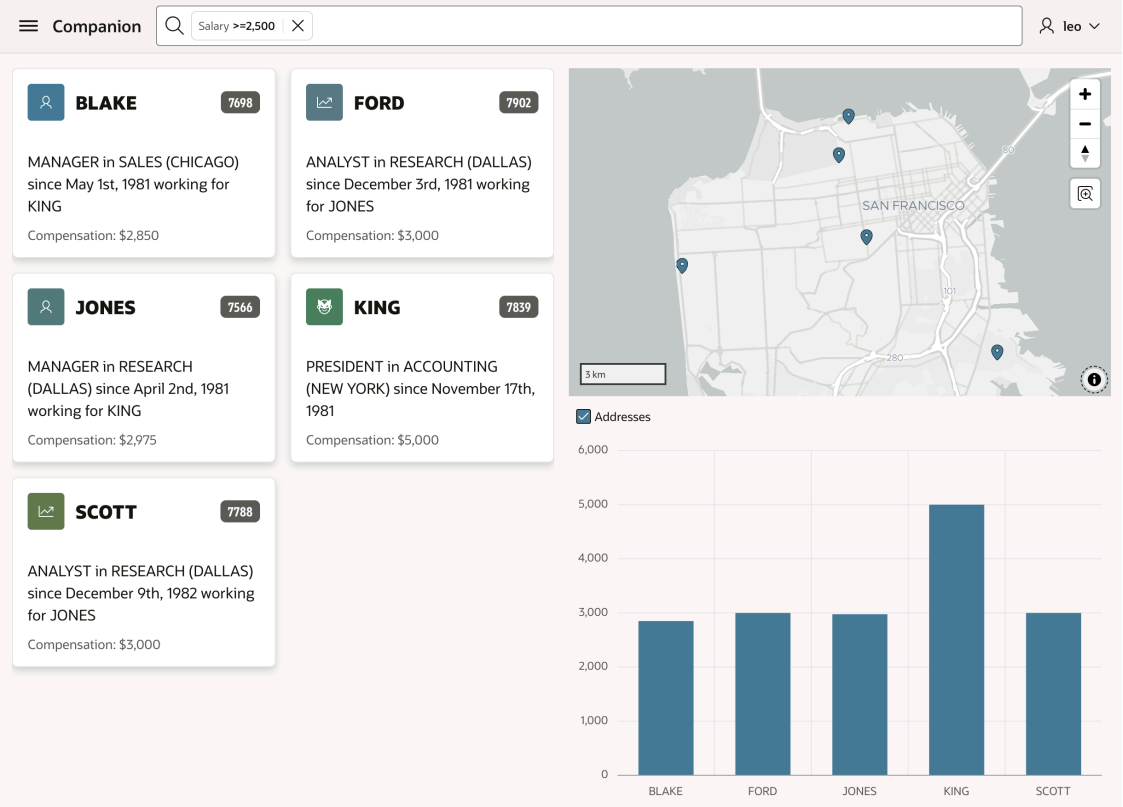
6.4.2 Showing Filtered Data in Another Region

To show one region's data in another region, you can reference the primary keys of a source region's search results in another region's query or `WHERE` clause. Whenever the source region's results change, a dynamic action can automatically refresh the target region.

Consider the *Smart Filters* region shown below that searches `EMP` data in a *Cards* region. As a user narrows the results, seeing the same data on a chart or a map can be useful. The figure below shows a Smart Filters region positioned in the navigation bar. In the body of the page, a Cards region occupies the first six columns, while a Map and Chart region show one above the other in the right six columns.

Copyright © 2026, Oracle and/or its affiliates.

Figure 6-24 Show Smart Filtered Cards Region Data in Chart and Map



- Getting Filtered Data Primary Keys
  A helper function can return the primary keys of a filtered region's results.
- Using Filtered Data Primary Keys
  Use the SQL `table()` operator to use rows from a `pipelined` function in the data source query of any APEX region.
- Refreshing the Target Region
  To keep the source region and the target region's data in sync, define a dynamic action on the source region that triggers when the source region's results are refreshed.

6.4.2.1 Getting Filtered Data Primary Keys

A helper function can return the primary keys of a filtered region's results.

The `region_pks` function below accepts the static ID of the filtered region as a parameter. It represents the *source* whose filtered results you want to present in an alternative way using another region. If `ID` is the primary key, then only the first parameter is required. If not, pass its name in the optional `p_pk_column_name` parameter. The function returns a `pipelined` list of numbers representing the primary keys of the indicated region's filtered results. It uses the `OPEN_QUERY_CONTEXT()` function in the `APEX_REGION` package to reference the source region's filtered results. Then it iterates over them with `NEXT_ROW()` and `GET_NUMBER()` in the `APEX_EXEC` package. The `pipe row` returns the primary key of each result.

```
create or replace function region_pks(
    p_region_static_id in varchar2,
    p_pk_column_name   in varchar2 default 'ID')
```


Copyright © 2026, Oracle and/or its affiliates.

```
return apex_t_number
pipelined
is
    c_page_id constant number := V('APP_PAGE_ID');
    l_context          apex_exec.t_context;
begin
    l_context := apex_region.open_query_context(
                    p_page_id   => c_page_id,
                    p_region_id => apex_region.get_id(
                                       p_page_id       => c_page_id,
                                       p_dom_static_id => p_region_static_id));
    while apex_exec.next_row(l_context) loop
        pipe row (apex_exec.get_number(l_context, p_pk_column_name));
    end loop;
    apex_exec.close(l_context);
    return;
exception
    when no_data_needed then
        apex_exec.close(l_context);
        return;
end region_pks;
```

6.4.2.2 Using Filtered Data Primary Keys

Use the SQL `table()` operator to use rows from a `pipelined` function in the data source query of any APEX region.

Using the SQL `WITH` clause as shown below, you can *name* the filtered region's list of primary keys returned by the `region_pks()` function. Then, your query can `SELECT` from the `result_pks` *common table expression* joined with the `emp` table on their respective `empno` column values. The join condition between left and right tables uses a common column name, so the shorthand `using (empno)` is easier to type than `on emp.empno = result_pks.empno`.

```
with result_pks as (
    select column_value as empno
      from table(region_pks('employees','EMPNO'))
)
select ename,
       sal
from result_pks
join emp e using (empno)
```

You can also involve the filtered results' primary keys in a `WHERE` clause when needed like this:

```
empno in (select column_value
            from table(region_pks('employees','EMPNO')))
```

6.4.2.3 Refreshing the Target Region

To keep the source region and the target region's data in sync, define a dynamic action on the source region that triggers when the source region's results are refreshed.

Copyright © 2026, Oracle and/or its affiliates.

For a Classic Report region, the event is *After Refresh*. For a Cards region, use *Page Change (Cards)* instead. In the list of *True* actions, add a *Refresh* dynamic action step that refreshes the target region.

6.5 Using Page Items for Query by Example

Page items, including list-driven ones, are useful not *only* for data entry, but also for query by example use cases. The user can enter a value or pick a choice from a list, and another region on the same page can filter *its* data based on their choice.

You accomplish this by *referencing* page item values in the other region's `WHERE` clause. When you use page items values as query by example filters, you can also easily *disable* the default warning the user receives when leaving the page with unsaved pending changes.

- Using Null Display Value
  When using the *Select List*, *Radio Group*, and *Popup LOV* you can optionally include an item in the list that represents the `null` value, and if included it can have a custom label.
- Filtering on a Page Item's Value
  You can use any page item's value in a query filter in another region. Most page item types, including *single-selection* list-driven items, store just one value that can also be `null`.
- Filtering on a Multi-Selection Page Item
  When a query-by-example item allows multiple selection, filtering by its value requires an `IN` clause with a subquery.
- Optimal Approach to Query by Example
  Two approaches exist for query-by-example `WHERE` clause filters. One requires less typing; the other performs better.
- Disabling Warn on Unsaved Changes
  Query-by-example items change values by design. Disable the unsaved changes warning so users aren't confused when navigating away.

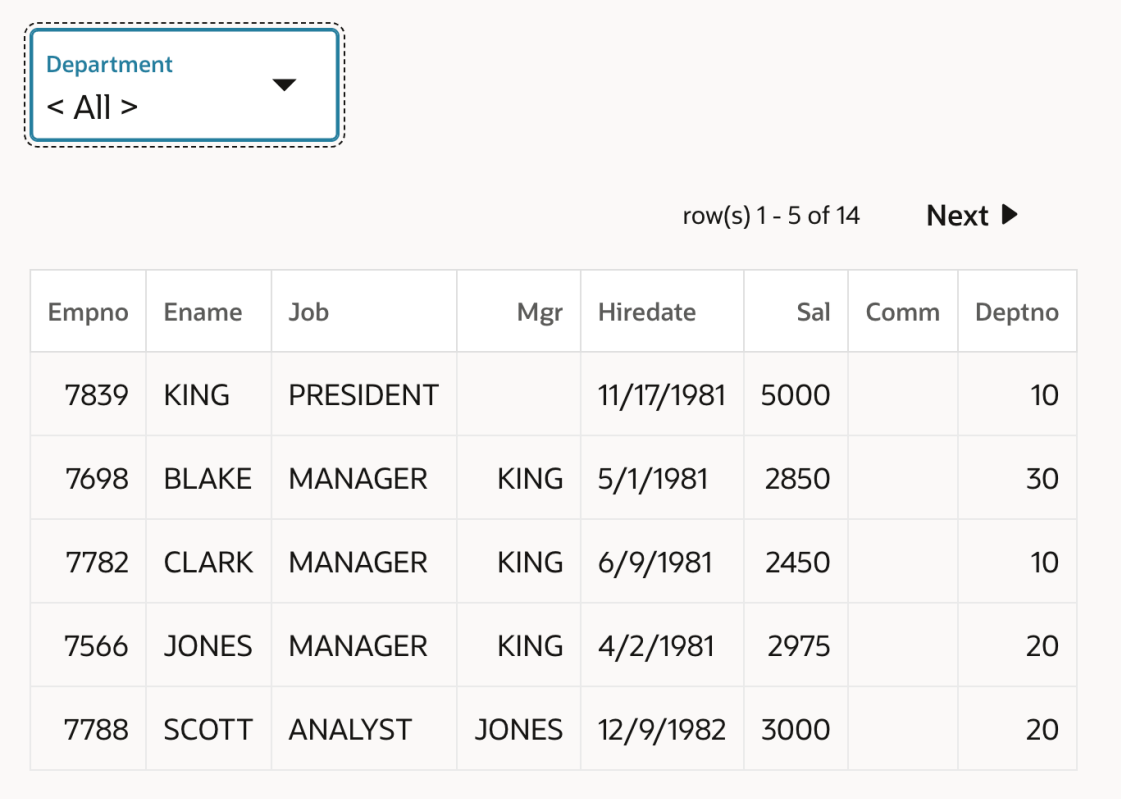
6.5.1 Using Null Display Value

When using the *Select List*, *Radio Group*, and *Popup LOV* you can optionally include an item in the list that represents the `null` value, and if included it can have a custom label.

This can be useful to help an end user understand what *not* selecting a value means. For example, consider the Department *Select List* page item below used as an optional query by example criteria field. The *Null Display Value* of "<*All*>" suggests that employees from *all* departments will be shown if they don't select a specific department.

Copyright © 2026, Oracle and/or its affiliates.

Figure 6-25 Custom Label Representing the Null Value in a Select List



6.5.2 Filtering on a Page Item's Value

You can use any page item's value in a query filter in another region. Most page item types, including *single-selection* list-driven items, store just one value that can also be `null`.

Assuming your query by example page item is a *Select List* named `P5_SELECTED_DEPARTMENT`, you can reference its value in the `WHERE` clause of another region on the page using the following syntax:

```
DEPTNO = :P5_SELECTED_DEPARTMENT
```

If the page item's value can be `null`, then your `WHERE` clause must anticipate this accordingly using this modified version of the filter clause:

```
/*  Either no department is selected,
 or match the selected department */
(   :P5_SELECTED_DEPARTMENT IS NULL
 OR DEPTNO = :P5_SELECTED_DEPARTMENT)
```


Copyright © 2026, Oracle and/or its affiliates.

> **⚠ Caution**
>
> When a region data source references a page item as a bind variable, list that item in the region's **Page Items to Submit** property. Omit the colon and comma-separate multiple item names. If you forget this step, the bind variable evaluates to `null` on refresh and the query may return no rows.

6.5.3 Filtering on a Multi-Selection Page Item

When a query-by-example item allows multiple selection, filtering by its value requires an `IN` clause with a subquery.

By default, the value of a multi-select page item will contain the colon-delimited (`:`) return values of the user's selections. Assuming your item is named `P5_SELECTED_DEPARTMENTS`, you can reference its value in the `WHERE` clause of another region on the page using the syntax below. If you use a different delimiter, pass it instead of the colon in `APEX_STRING.SPLIT()`'s second parameter.

```
DEPTNO IN (SELECT COLUMN_VALUE
             FROM APEX_STRING.SPLIT(:P5_SELECTED_DEPARTMENTS,':'))
```

If the page item is *optional*, then your `WHERE` clause must anticipate that with this modified version of the filter clause:

```
/* Either no department is selected,
or match the selected departments */
(   :P5_SELECTED_DEPARTMENTS IS NULL
 OR DEPTNO IN (SELECT COLUMN_VALUE
                 FROM APEX_STRING.SPLIT(:P5_SELECTED_DEPARTMENTS,':')))
```

> **⚠ Caution**
>
> When a region data source references a page item as a bind variable, list that item in the region's **Page Items to Submit** property. Omit the colon and comma-separate multiple item names. If you forget this step, the bind variable evaluates to `null` on refresh and the query may return no rows.

6.5.4 Optimal Approach to Query by Example

Two approaches exist for query-by-example `WHERE` clause filters. One requires less typing; the other performs better.

The simpler-to-understand technique shown below uses an `OR` clause that *first* checks if the page item value `IS NULL` and *then* provides a filter on the desired column involving the page item bind variable.

```
/* Either no department is selected,
or match the selected department */
```


Copyright © 2026, Oracle and/or its affiliates.

```
(   :P5_SELECTED_DEPARTMENT IS NULL
 OR DEPTNO = :P5_SELECTED_DEPARTMENT)
```

When processing a query for the first time, the database considers bind variable values when deciding which indexes to use for fast data retrieval. Then, it reuses this "query plan" on subsequent executions of the same SQL statement, even if the bind variable values have changed. This means that using the simpler approach above can result in not always using the best index for the filtering job at hand, especially when your query might contain many such clauses.

For the best performance, include a query by example filter clause *only* when it is relevant. In other words, if the corresponding page item value is *not null*, then include the clause that filters a column based on its value. Otherwise, if the page item value is null, leave that filtering clause out of the query.

Employing this strategy requires a dynamic approach to constructing the SQL query. APEX lets you write a *Function Body Returning SQL* to easily implement this technique. Instead of specifying the SQL statement verbatim, you use PL/SQL to build it up as needed and return the resulting SQL statement. For example, the code below returns a SQL statement that includes the filter predicate if the `P5_SELECTED_DEPARTMENT` page item value is *not null*, or leaves it out otherwise.

```
return q'~
SELECT EMPNO, ENAME, SAL, DEPTNO, HIREDATE
  FROM EMP
 WHERE 1=1
~'
|| CASE WHEN :P5_SELECTED_DEPARTMENT IS NOT NULL THEN
q'~
AND DEPTNO = :P5_SELECTED_DEPARTMENT
~' END;
```

The code above is what you would type into the Property Editor for a region's data source after selecting the *Function Body Returning SQL* option. Notice it's using the PL/SQL syntax for a multi-line string literal using `q'~` to begin the string and `~'` to end it. This makes the parts of your SQL statement being concatenated more readable and avoids the need to handle single quotes in a special way. The double vertical bar symbol is the Oracle SQL string concatenation operator. Finally, recall that the SQL `CASE` expression returns `null` if none of its `WHEN` or optional `ELSE` conditions evaluates to true, and that concatenating a `null` value to a string leaves the original string unchanged.

At runtime, whenever your region retrieves its data the APEX engine "wraps" your code with an enclosing "envelope" shown below. Then it evaluates the function to compute the effective text of the SQL statement it asks the database to execute.

```
function x
return varchar2
is
begin
   /* Your Function Body Here */
end x;
```


Copyright © 2026, Oracle and/or its affiliates.

Using this technique, when P5_SELECTED_DEPARTMENT is null the database sees the query:

```
SELECT EMPNO, ENAME, SAL, DEPTNO, HIREDATE
  FROM EMP
 WHERE 1=1
```

In contrast, when P5_SELECTED_DEPARTMENT has a non-null value the query executed is:

```
SELECT EMPNO, ENAME, SAL, DEPTNO, HIREDATE
  FROM EMP
 WHERE 1=1
AND DEPTNO = :P5_SELECTED_DEPARTMENT
```

You can adopt the same technique to conditionally include *multiple* filter clauses. For example, the following function body evaluates three different query by example page items to include the relevant filter clauses. Using the always true predicate WHERE 1=1 ensures a syntactically correct query for any number of added filters. It also lets each conditionally-included predicate uniformly include the leading AND conjunction.

```
return q'~
SELECT EMPNO, ENAME, SAL, DEPTNO, HIREDATE
  FROM EMP
 WHERE 1=1
~'
|| CASE WHEN :P5_SELECTED_DEPARTMENT IS NOT NULL THEN
q'~
AND DEPTNO = :P5_SELECTED_DEPARTMENT
~' END
|| CASE WHEN :P5_ENAME_CONTAINS IS NOT NULL THEN
q'~
AND INSTR(UPPER(ENAME), UPPER(:P5_ENAME_CONTAINS)) > 0
~' END
|| CASE WHEN :P5_MAX_SALARY IS NOT NULL THEN
q'~
AND SAL <= :P5_MAX_SALARY
~' END;
```

To maximize maintainability, you can put your query-building code into a PL/SQL package function. Imagine you have a UTILS package, and added to its specification and body the p5_employees_query function shown below. In that case, your region's function body returning SQL becomes:

```
return utils.p5_employees_query(:P5_SELECTED_DEPARTMENT,
                                 :P5_ENAME_CONTAINS,
                                 :P5_MAX_SALARY);
```

The p5_employees_query function would look like this:

```
function p5_employees_query(
   p_selected_deptno in number,
   p_ename_contains  in varchar2,
   p_max_salary      in number)
   return              varchar2
```


Copyright © 2026, Oracle and/or its affiliates.

```
is
begin
   return q'~
   SELECT EMPNO, ENAME, SAL, DEPTNO, HIREDATE
     FROM EMP
    WHERE 1=1
   ~'
   || CASE WHEN p_selected_deptno IS NOT NULL THEN
   q'~
   AND DEPTNO = :P5_SELECTED_DEPARTMENT
   ~' END
   || CASE WHEN p_ename_contains IS NOT NULL THEN
   q'~
   AND INSTR(UPPER(ENAME), UPPER(:P5_ENAME_CONTAINS)) > 0
   ~' END
   || CASE WHEN p_max_salary IS NOT NULL THEN
   q'~
   AND SAL <= :P5_MAX_SALARY
   ~' END;
end p5_employees_query;
```

Whenever your function body *might* include a reference to a page item value as a bind variable in a region data source query or `WHERE` clause, list that item in the region's **Page Items to Submit** property. Omit the colon and comma-separate multiple item names. If you forget this step, the bind variable evaluates to null on refresh and the query may return no rows.

> ⚠ **Caution**
>
> When App Builder checks the syntax of a function body returning SQL in Page Designer, all of the referenced bind variables will evaluate to `null`. Therefore, it's important that you *always* return a syntactically-valid `SELECT` statement even in this situation.

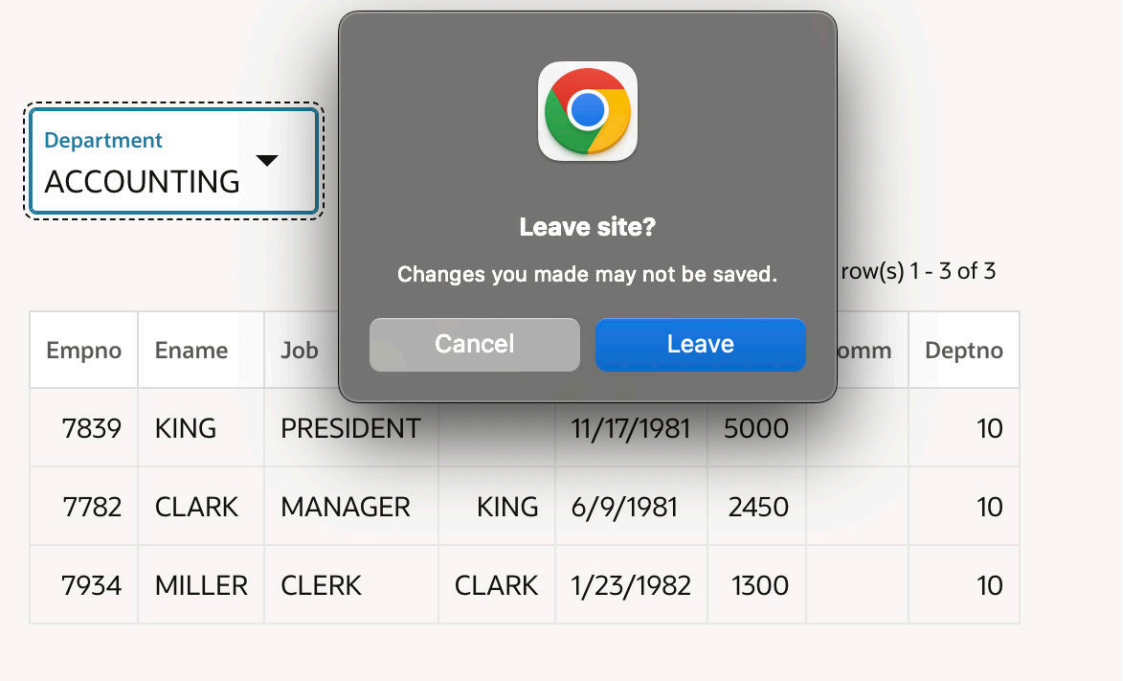
6.5.5 Disabling Warn on Unsaved Changes

Query-by-example items change values by design. Disable the unsaved changes warning so users aren't confused when navigating away.

As shown below, by default APEX warns about unsaved changes when navigating to another page. This can alarm the user if they were not performing a data entry task. When using a page item for query by example, consider *disabling* its *Warn on Unsaved Changes* setting. You can configure this setting once at page level, and also set it per page item as appropriate for your use case.

Copyright © 2026, Oracle and/or its affiliates.

Figure 6-26 Warn on Unsaved Changes for Query by Example Confuses Users



Copyright © 2026, Oracle and/or its affiliates.

7

Adding Functionality with Plug-ins

When native APEX features don't fully meet your needs, you can easily extend the platform with plug-ins. These shared components add new functionality to your apps like a new kind of region, page item, dynamic action, page process, AI tool, workflow activity, data source, or scheme for authentication or authorization.

Developers with basic web design skills can create template components with no code, while other plug-in types require coding skills. Once a plug-in exists, the rest of the team can use it declaratively just like a built-in feature. However, before creating one yourself, search the APEX community "app store" site https://apex.world. The extended functionality you need might already be just a download away.

When your application incorporates plug-ins, you can review where each is used on the *Utilization* tab of the *Shared Components* > *Plug-ins* list page.

- Reusable Components Using HTML Skills
  A template component is a reusable user interface element you create without coding.
- Using a Plug-in from apex.world
  By using a existing plug-in from the apex.world "app store", you can quickly add new functionality that another developer or team has made available to the APEX community.

7.1 Reusable Components Using HTML Skills

A template component is a reusable user interface element you create without coding.

Developers familiar with basic HTML, placeholder `#NAME#` substitution, and HTML directives can create sophisticated, reusable user interface controls. Other team members use template components just like native ones in Page Designer, configuring their settings in the Property Editor.

- Template Component Overview
  You can create a reusable UI component using templates with HTML tags, `#NAME#` placeholder substitutions, and template directives for conditional formatting.
- Custom Attributes for Data and Settings
  The custom attributes you define are named placeholders developers configure in the Property Editor to provide the data the component displays and any settings it requires.
- Slots Enable Component Composition
  It's easy to define a component that can compose other regions, page items, and buttons. Just define a named *slot* and include its contents in your template using the `#SLOTNAME#` syntax.
- Applying Template Using Directives
  Developers can use a template component with internal name `TEMPLATE_NAME` anywhere *directives* are allowed.
- Controlling Component Availability
  Using the `{with/}…{apply/}` directive, developers can *always* apply your template component by name, but *you* decide if they can use it in Page Designer by setting its *Available as* options.

Copyright © 2026, Oracle and/or its affiliates.

- Templates Comprising the Component
  A template component always has a primary template called the *Partial*. This name evokes the role it plays as a *part* of the web page at runtime. Your *Partial* template includes the main markup that determines how the component's attributes appear on the page.
- Presenting Grouped Data
  When you make your template component available as a *Multiple (Report)* region, it can present the rows from its data source grouped by common values. Define the HTML markup that should appear for each distinct group using the *Report Group* template that includes the rows in the group using `#APEX$ROWS#`.
- Configuring Pagination Support
  For a template component available as a *Multiple (Report)* region, you configure whether users view additional results beyond the first screenful by scrolling or paging. This works both for normal and grouped results.
- Enabling Row Selection
  If you make your template component available as a *Multiple (Report)* region, enable row selection by checking *Has Row Selection Support* in the *Standard Attributes* section of the edit page.
- Attribute Scopes and Availability to Templates
  You assign a scope of *Component*, *Report*, or *Report Group* to each custom attribute.
- Action Positions and Action Templates
  If your template component presents a single row or multiple rows of data, it can also define action positions. These are row-specific slots where teammates using the component can define buttons or menus to perform actions on a specific row.
- Defining Files and File URLs to Load
  If your component uses Cascading Style Sheets (CSS) or JavaScript, store them in separate component files to improve maintainability.

7.1.1 Template Component Overview

You can create a reusable UI component using templates with HTML tags, `#NAME#` placeholder substitutions, and template directives for conditional formatting.

Custom attributes let teammates use Page Designer to configure the data your component displays and the settings that influence its output. Your component can include *slots* developers see in Page Designer as areas where they can add other components of the kinds you allow.

Developers can apply your component using the `{with/}…{apply/}` directive, and optionally use it in Page Designer:

- as a custom region that can contain other components,
- to format a single row of data, or
- to render multiple rows.

When used as a multi-row report, it can automatically support grouping, row selection, and pagination as well. If your component uses Cascading Style Sheets (CSS) rules or JavaScript, you can include those files as part of the component's definition.

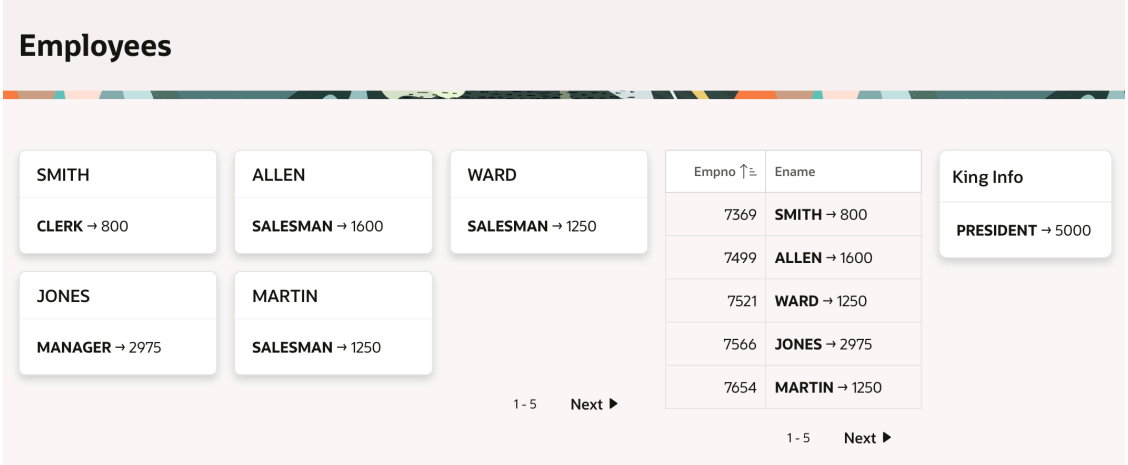
Imagine creating a simple template component to display `NAME` and `VALUE` attributes using a template like:

```
<strong>#NAME#</strong> → #VALUE#
```


Copyright © 2026, Oracle and/or its affiliates.

The figure below shows a page that uses this component in the body of a *Cards* region and in the column of a classic report to display employee names and salaries. It also uses the component as a single-row partial to display employee *KING*'s job and salary.

Figure 7-1 Reusing a Template Component Three Different Ways



7.1.2 Custom Attributes for Data and Settings

The custom attributes you define are named placeholders developers configure in the Property Editor to provide the data the component displays and any settings it requires.

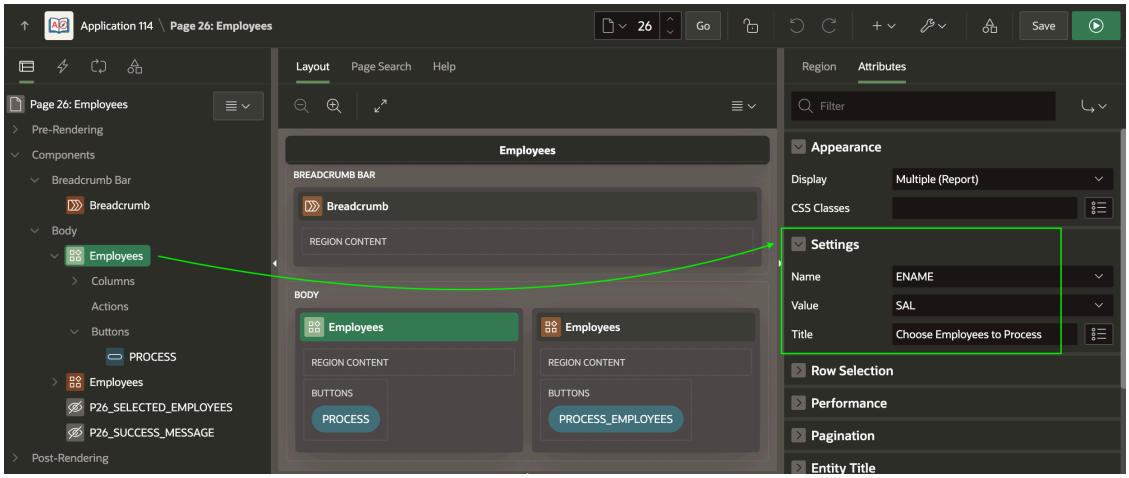
Your templates use directives referencing custom attribute names to conditionally format data as required and `#ATTRNAME#` syntax to include an attribute's value.

In addition to a static ID and name, each attribute definition has other details that inform the Property Editor experience in Page Designer. For example, attributes of type *Session State Value* show developers a select list to choose a data source column name. A *Yes/No* type attribute displays an on/off switch, and a *Select List* lets developers choose a pre-defined option from a set you specify. Use other attribute types to show an icon picker, a color picker, an HTML code editor, a link builder, and more. You can define help text and examples for each attribute that a developer can consult in the *Help* tab in Page Designer to better understand how each setting affects the component's output.

As shown below, by default, your custom attributes display in the Property Editor's *Attributes* tab in the general *Settings* section in the display sequence you configure. If multiple attributes contribute to a common purpose, create an attribute group and assign related attributes to that group. The Property Editor displays the grouped attributes in a titled section to clarify how multiple properties affect one aspect of a component's features.

Copyright © 2026, Oracle and/or its affiliates.

Figure 7-2 Custom Attributes in the Settings Section



7.1.3 Slots Enable Component Composition

It's easy to define a component that can compose other regions, page items, and buttons. Just define a named *slot* and include its contents in your template using the `#SLOTNAME#` syntax.

Page Designer helps developers add the buttons, page items, or regions you allow into these slots and enforces any constraints on the slot contents that you configure. In short, if you say a particular slot can only contain buttons, then Page Designer makes sure teammates follow your rules.

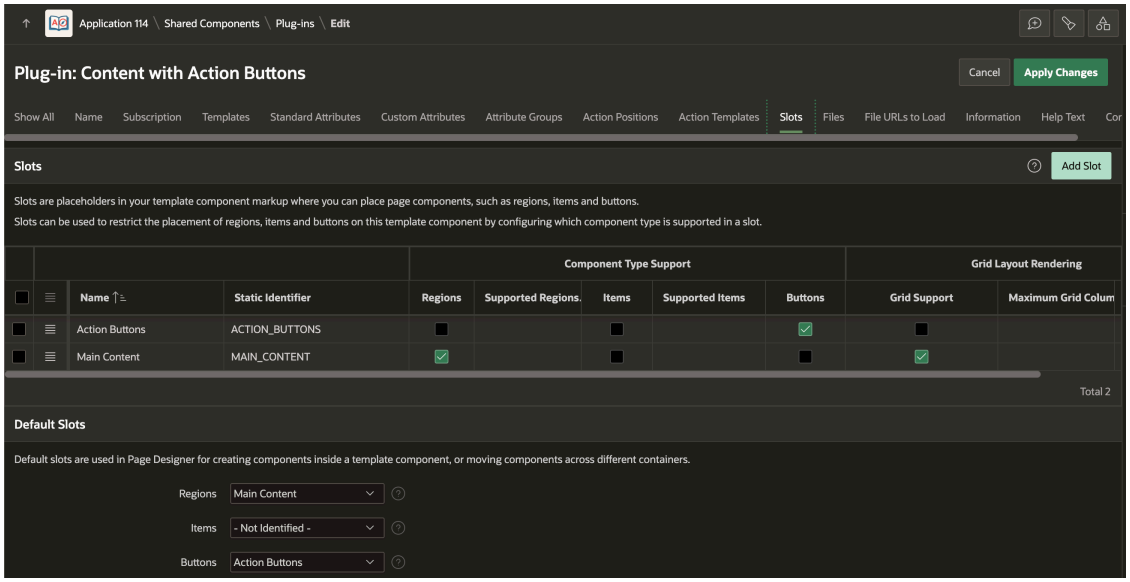
For example, consider a template component named *Content with Action Buttons* with two slots named `MAIN_CONTENT` and `ACTION_BUTTONS`. Its simple partial template can use `{if/}` directives to conditionally format the contents of these two slots as follows. Notice the formatting of the `MAIN_CONTENT` slot changes depending on whether the developer has provided a value for the `HEIGHT` custom attribute.

```
{if ACTION_BUTTONS/}
  <div class="action-buttons">
    #ACTION_BUTTONS#
  </div>
{endif/}
{if HEIGHT/}
  <div style="height:#HEIGHT#">
    #MAIN_CONTENT#
  </div>
{else/}
  #MAIN_CONTENT#
{endif/}
```

When defining the slots, you can indicate, as shown below, that the `ACTION_BUTTONS` slot should only allow buttons and that the `MAIN_CONTENT` slot should only allow regions and support the 12-column grid layout. In the *Default Slots* section below, notice that you can indicate that the *Main Content* slot is the default slot for subregions and that *Action Buttons* is the default slot for buttons. These additional details further simplify the Page Designer experience for developers using the component.

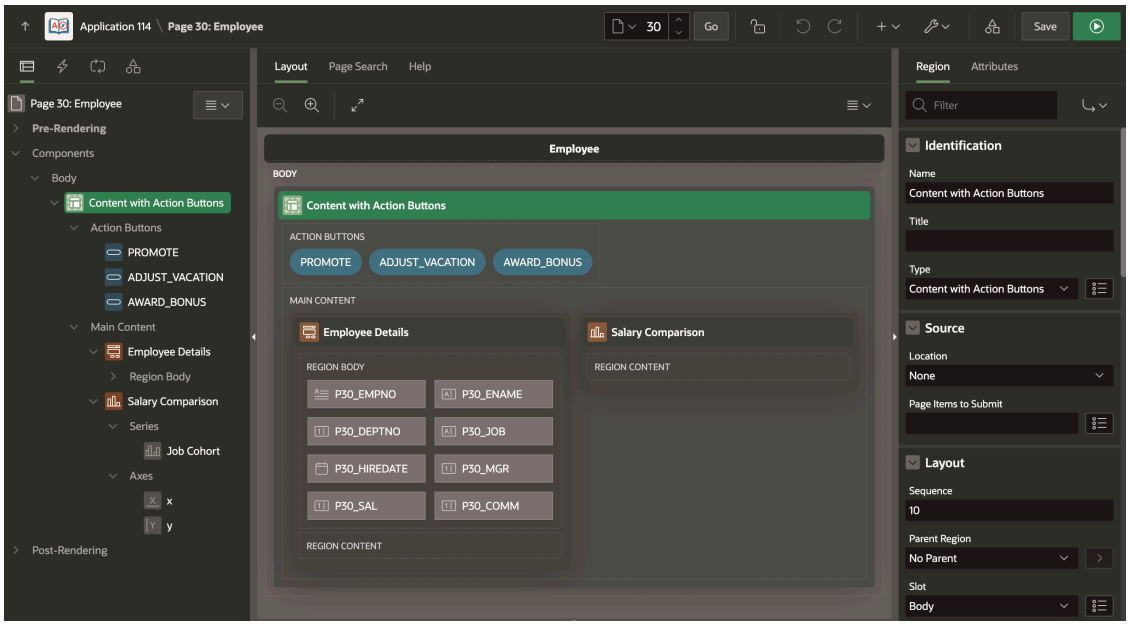
Copyright © 2026, Oracle and/or its affiliates.

Figure 7-3 Configuring Slot Behavior in a Template Component



When a developer uses the *Content with Action Buttons* component on a page, as shown below, Page Designer displays the *Action Buttons* slot and ensures they can only add buttons into it. Similarly, the developer can only add regions into the *Main Content* slot.

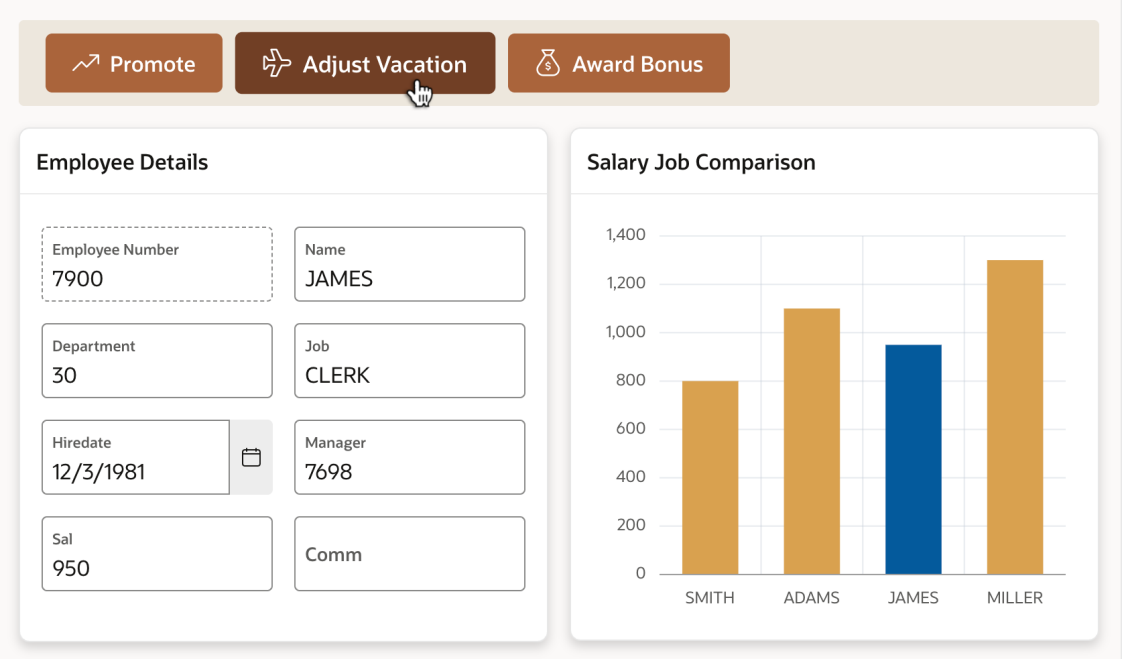
Figure 7-4 Using a Template Component's Slots in Page Designer



Running the page, you see below how the *Content with Action Buttons* component styles the buttons contained in the `ACTION_BUTTONS` slot and contains two regions each using 6 columns of the 12-column grid row in the `MAIN_CONTENT` slot.

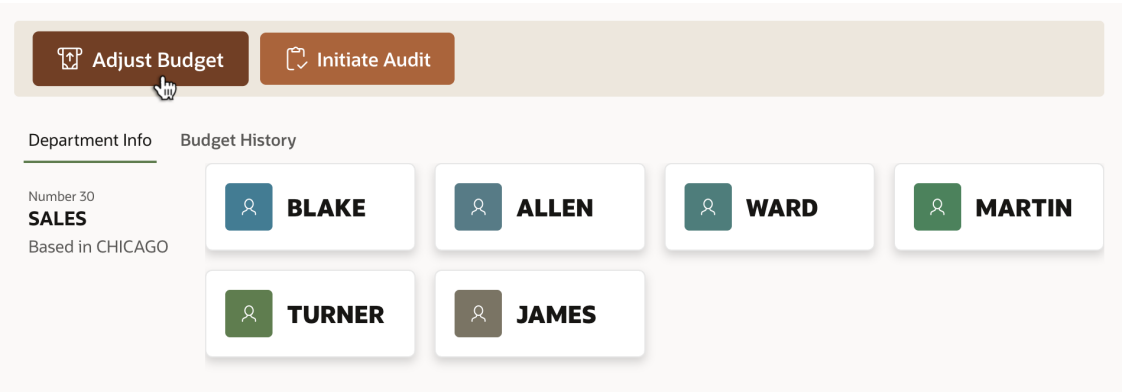
Copyright © 2026, Oracle and/or its affiliates.

Figure 7-5 Template Component Positions and Styles Composed Buttons and Regions



In another page, you can use the *Content with Action Buttons* component with different buttons and regions to create a tabbed display of department information. The figure below shows **Adjust Budget** and **Initiate Audit** buttons in the *Action Buttons* component slot, and in the *Main Content* slot, **Department Info** and **Budget History** tabs appear with their respective contents.

Figure 7-6 Different Usage of Content with Action Buttons Template Component



7.1.4 Applying Template Using Directives

Developers can use a template component with internal name *`TEMPLATE_NAME`* anywhere *directives* are allowed.

Copyright © 2026, Oracle and/or its affiliates.

The developer supplies any necessary attribute values and applies the template using the syntax:

```
{with/}
   ATTR1:=VALUE1
   ATTR2:=VALUE2
    ▒
{apply TEMPLATE_NAME/}
```

7.1.5 Controlling Component Availability

Using the `{with/}…{apply/}` directive, developers can *always* apply your template component by name, but *you* decide if they can use it in Page Designer by setting its *Available as* options.

If you select any of these options, Page Designer displays your component in the region palette and the list of region types. The particular option you choose controls how the Property Editor presents the component's features.

A *Region-only* component displays your custom attributes and slots, but offers no additional built-in settings. Choosing *Single (Partial)* or *Multiple (Report)* causes the Property Editor to show the *Source* section so a developer can configure a data source. Its columns are then available to reference in custom attributes. The data source for a *Single (Partial)* template component must return at most one row, or a runtime error results. Given this guarantee, the component renders only its *Partial* template with no additional space used for pagination controls.

When you make your template component available as a *Multiple (Report)* component, additional settings related to pagination and grouping appear. If you checked *Has Row Selection Support* when defining it, further settings appear for this capability.

7.1.6 Templates Comprising the Component

A template component always has a primary template called the *Partial*. This name evokes the role it plays as a *part* of the web page at runtime. Your *Partial* template includes the main markup that determines how the component's attributes appear on the page.

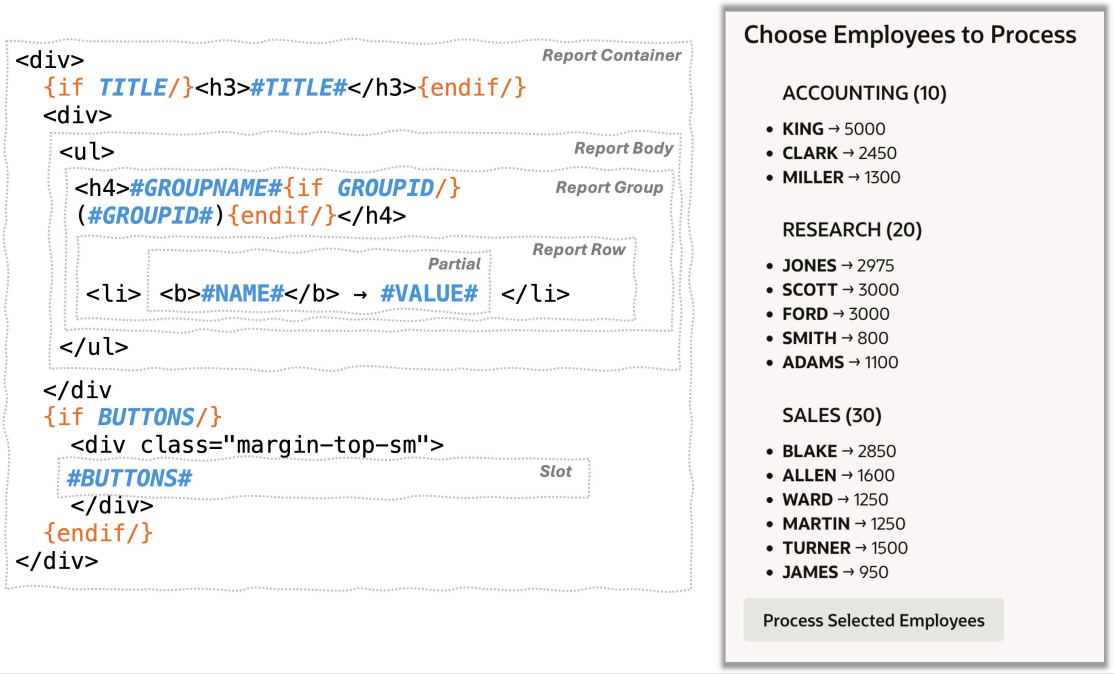
If you make your component available as a *Multiple (Report)* type region, then you can define additional templates:

- *Report Row* wraps *each row* of data and includes the partial with `#APEX$PARTIAL#`
- *Report Group* wraps each distinct group of rows and includes them with `#APEX$ROWS#`
- *Report Body* wraps the current page of rows and includes them with `#APEX$ROWS#`
- *Report Container* wraps the report body and includes it with `#APEX$REPORT_BODY#`

Consider a template component *Grouped Name/Value Pairs* that defines a report attribute `TITLE`, component attributes `NAME` and `VALUE`, group attributes named `GROUPNAME` and `GROUPID`, as well as a `BUTTONS` slot. The figure below shows how the different templates cooperate to produce the result on the right. The developer has configured the title, chosen `DNAME` and `DEPTNO` for the group name and id, used `ENAME` and `SAL` for the name and value attributes, and added a single *(Process Selected Employees)* button into the *Buttons* slot.

Copyright © 2026, Oracle and/or its affiliates.

Figure 7-7 Templates Involved to Format a Multiple (Report) Component



7.1.7 Presenting Grouped Data

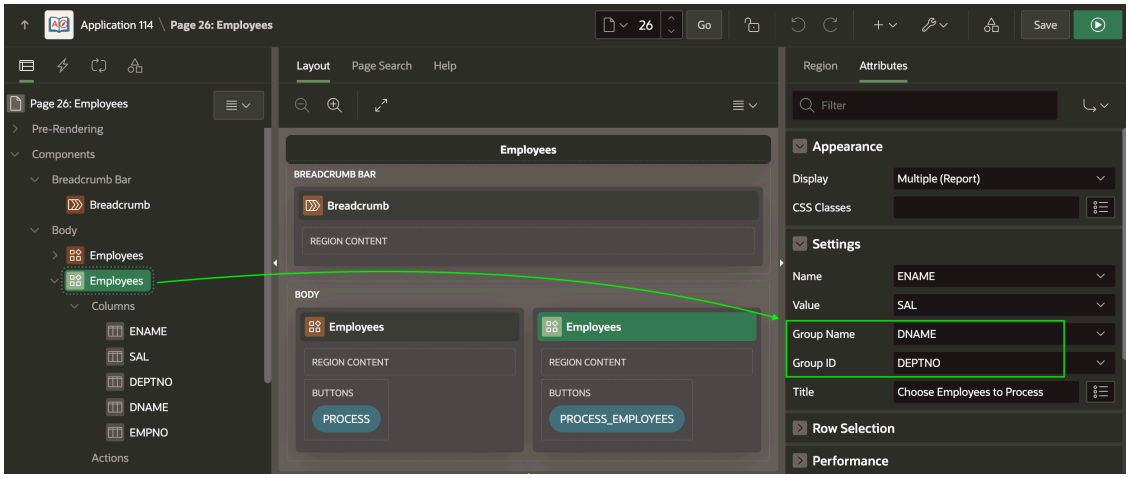
When you make your template component available as a *Multiple (Report)* region, it can present the rows from its data source grouped by common values. Define the HTML markup that should appear for each distinct group using the *Report Group* template that includes the rows in the group using `#APEX$ROWS#`.

When using the template component, configure the data source and enable the *Group* switch on one or more columns based on whose values you want the component to group its rows. All columns for which you enable the *Group* switch must appear in the region data source's ORDER BY clause. Your component will format a new group for each distinct combination of the grouping columns.

Once at least one column has the *Group* switch enabled, as shown below, additional *Report Group* attributes display in the Property Editor.

Copyright © 2026, Oracle and/or its affiliates.

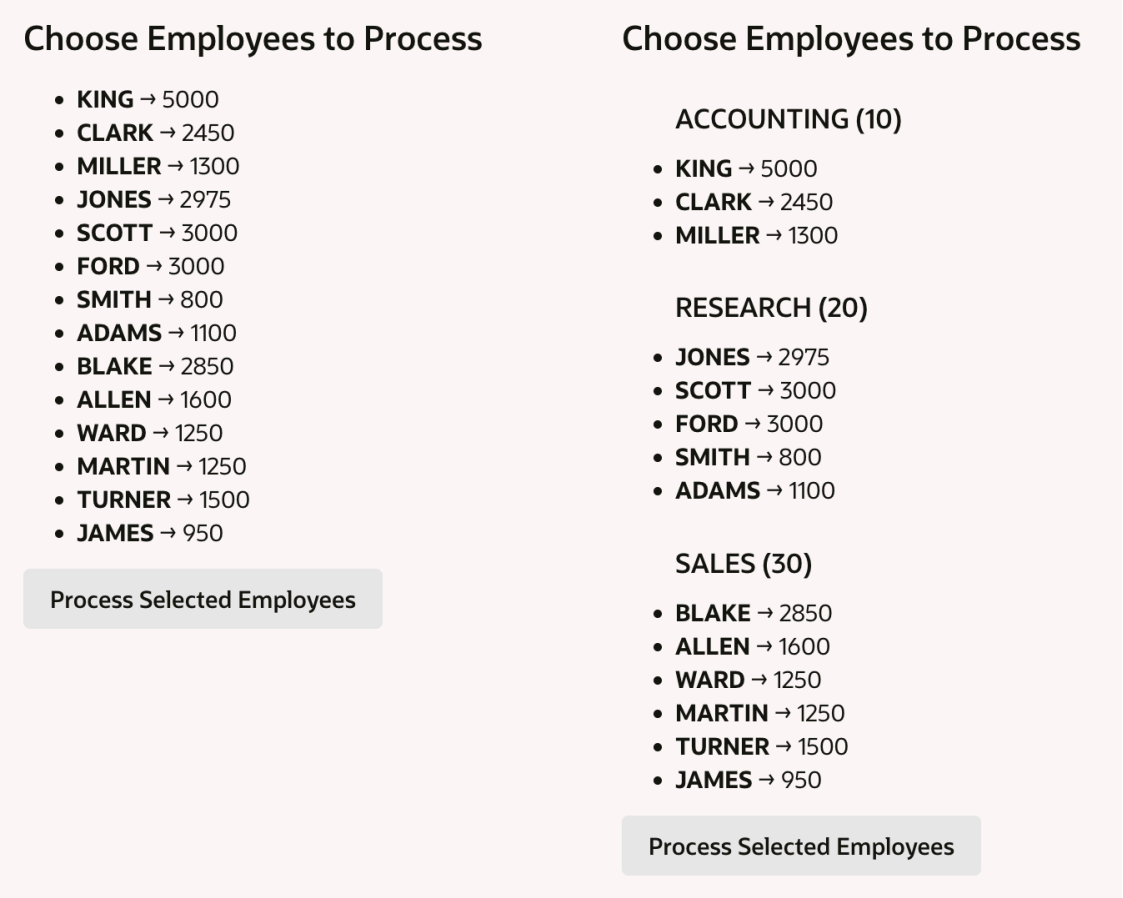
Figure 7-8 Report Group Attributes in the Property Editor



As shown below, the same component can present both normal or *grouped* results. When developers configure *Group* columns in a particular usage, the *Report Group* template automatically participates to format the output. The figure shows side-by-side usages of the same template component. On the left, no columns in the data source have been configured for grouping. On the right, the *Group* setting on the `DNAME` and `DEPTNO` columns has been enabled, and their values used for the component's group attributes *Group Name* and *Group ID*.

Copyright © 2026, Oracle and/or its affiliates.

Figure 7-9 Template Component Without and With Grouped Data



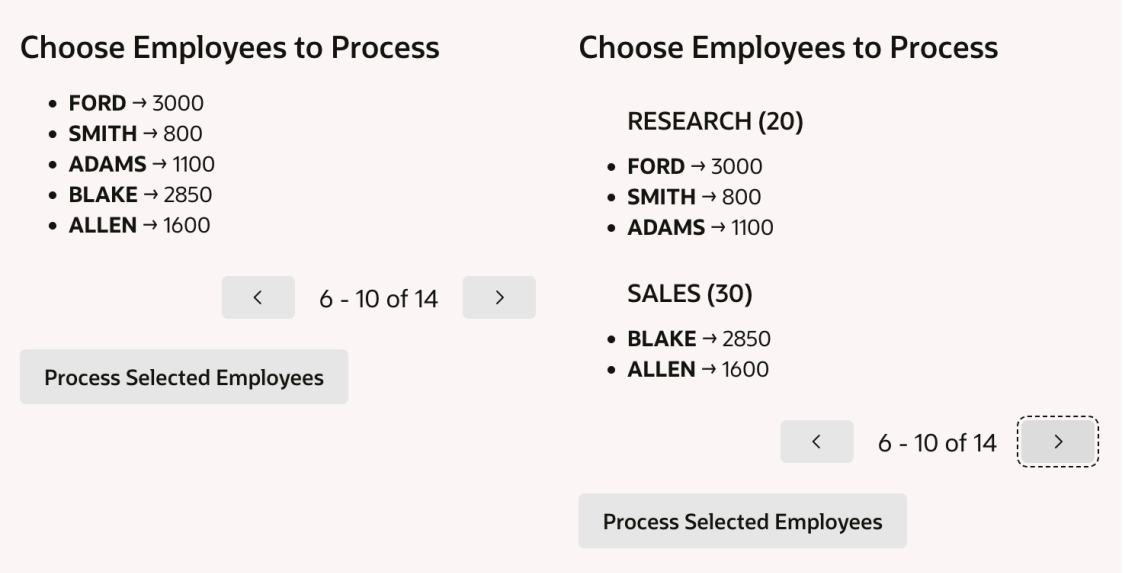
7.1.8 Configuring Pagination Support

For a template component available as a *Multiple (Report)* region, you configure whether users view additional results beyond the first screenful by scrolling or paging. This works both for normal and grouped results.

When you choose to page through results, users see a page at a time showing the number of rows you configure. The figure below shows results from the `EMP` table when configured to show five (5) rows per page. The usage on the right is configured to group the data by departments.

Copyright © 2026, Oracle and/or its affiliates.

Figure 7-10 Normal and Grouped Data with Pagination

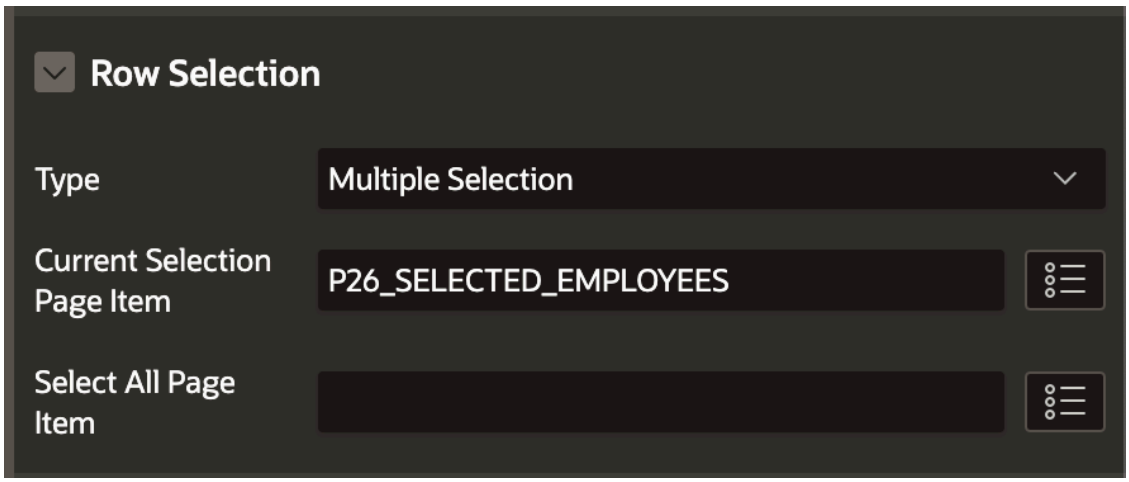


7.1.9 Enabling Row Selection

If you make your template component available as a *Multiple (Report)* region, enable row selection by checking *Has Row Selection Support* in the *Standard Attributes* section of the edit page.

When this setting is on, the Property Editor displays the *Row Selection* section shown below. A developer using your component can enable single row selection, multiple selection, or focus-only selection, and optionally provide the name of a page item to store the selected row primary keys. For multi-row selection, the value stored in the *Current Selection Page Item* uses a colon (:) to delimit multiple values.

Figure 7-11 Row Selection Settings on Template Component Available as Multiple (Report)

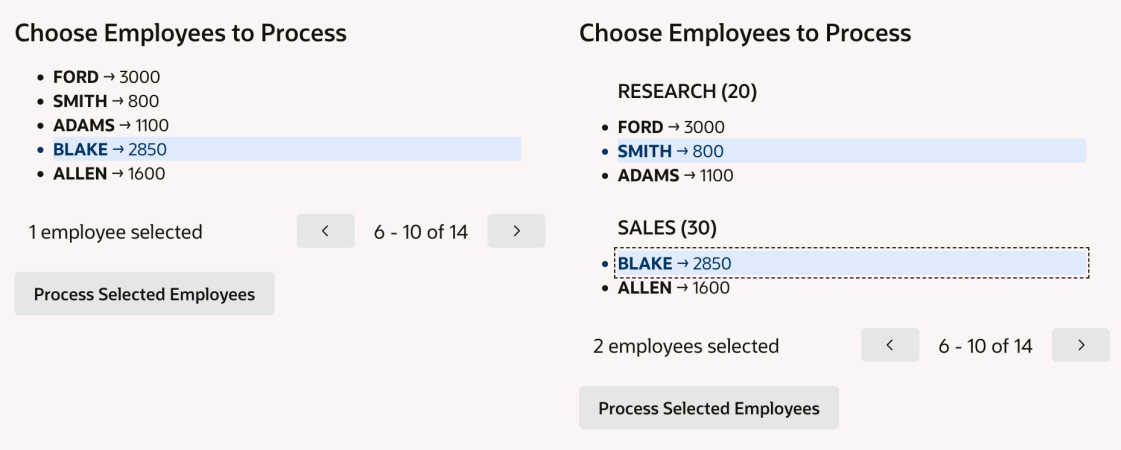


In the figure below, the *Grouped Name/Value Pairs* template component appears twice. Both instances are configured to show `ENAME` and `SAL` and to use pagination with five rows per page.

Copyright © 2026, Oracle and/or its affiliates.

The instance on the right has also configured `DNAME` and `DEPTNO` as the group name and group id. The end user has selected a single row in the template component on the left, and multiple employees in the one on the right. Notice the messages "*1 employee selected*" and "*2 employees selected*". The developer configured the component's *Entity Title* settings to use "employee" for the *Singular* and "employees" for the *Plural* to make the selection message more user-friendly.

Figure 7-12 Single & Multiple Selection Showing Entity Title



The page contains a *Handle Selected Emps* page process using the PL/SQL code below to iterate over the selected employees by "splitting" the colon-delimited primary keys in the `P26_SELECTED_EMPLOYEES` page item. If no employees were selected, it adds a translatable error message using the text message key `SELECT_AT_LEAST_ONE_EMP`. If some employee primary keys were processed, then it uses the `GET_MESSAGE()` function in the `APEX_LANG` package to retrieve a translatable text message to assign to the hidden `P26_SUCCESS_MESSAGE` page item. If a single row was processed, it uses the message with key `PROCESSED_ONE` otherwise it uses the key `PROCESSED_MANY`. The page process references this dynamically computed success message as `&P26_SUCCESS_MESSAGE.` in its *Success Message* property.

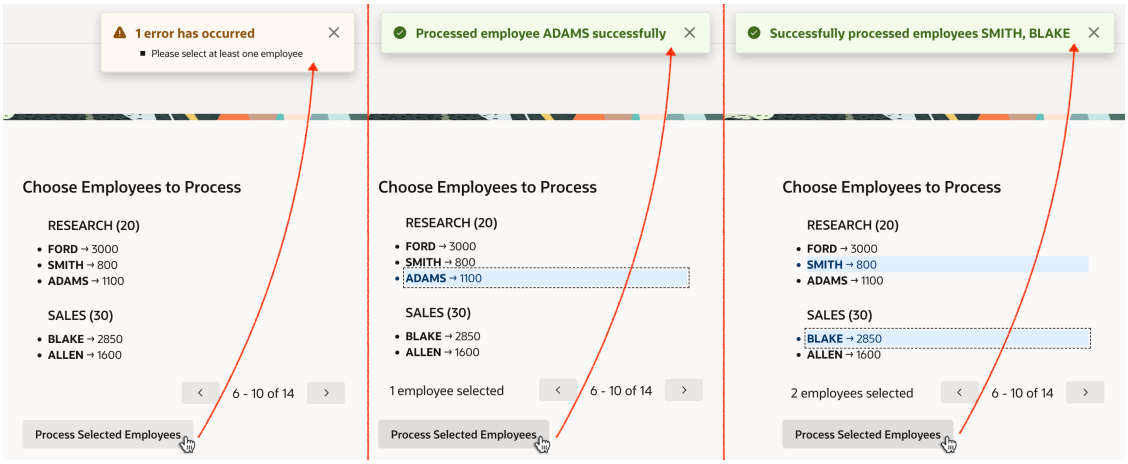
```
declare
    l_processed apex_t_varchar2;
begin
    for cur_emp in
       (select ename, empno
          from emp
         where empno in
            (select column_value
               from apex_string.split_numbers(
                  :P26_SELECTED_EMPLOYEES,':')))
    loop
        apex_string.push(l_processed,cur_emp.ename);
        -- Process cur_emp.empno employee here
    end loop;
    -- SELECT_AT_LEAST_ONE_EMP => "Please select at least one employee"
    if l_processed is null then
        apex_error.add_error(p_error_code       => 'SELECT_AT_LEAST_ONE_EMP',
                             p_display_location => apex_error.c_on_error_page,
                             p_page_item_name   => null);
    else
```


Copyright © 2026, Oracle and/or its affiliates.

```
        -- PROCESSED_ONE  => "Processed employee %0 successfully"
        -- PROCESSED_MANY => "Successfully processed employees %0"
        :P26_SUCCESS_MESSAGE := apex_lang.get_message(
                                  'PROCESSED_'||
                                  case l_processed.count
                                  when 1 then 'ONE' else 'MANY' end,
                                   apex_t_varchar2('0',
                                     apex_string.join(l_processed,', ')));
    end if;
end;
```

The figure below shows the three different situations the end user might encounter upon clicking the Process Selected Employees button. In the first example, with no employees selected, the user sees an error requesting that they select at least one. In the second example, after selecting a single employee they see a message containing the singular name that was processed. In the third example, they see a message listing all the employees processed.

Figure 7-13 Processing Zero, One, and Multiple Selected Rows



7.1.10 Attribute Scopes and Availability to Templates

You assign a scope of *Component*, *Report*, or *Report Group* to each custom attribute.

The scope determines the templates in which you can reference its value. The table below lists which templates can use attributes of each scope.

Table 7-1 Attribute Scopes and Availability in Templates

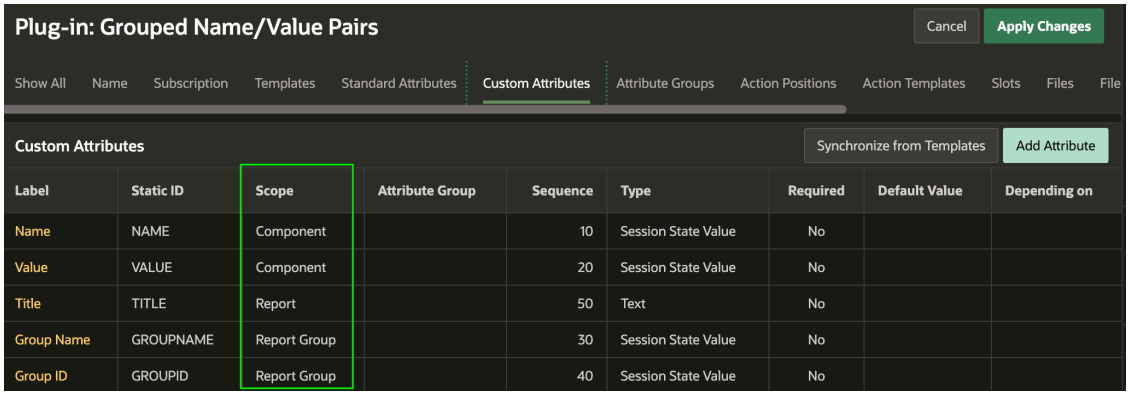
Scope	Available in Templates
Component	*Partial*
Report	*Partial*, *Report Row*, *Report Body*, *Report Group*, *Report Container*
Report Group	*Report Group*

The figure below shows the custom attributes of a *Grouped Name/Value Pairs* template component, with attributes of each scope type. The *Name* and *Value* attributes are

Copyright © 2026, Oracle and/or its affiliates.

Component-scoped. The *Title* attribute is Report-scoped, while *Group Name* and *Group ID* are scoped to the Report Group.

Figure 7-14 Attributes of Different Scopes in Grouped Name/Value Pairs Component



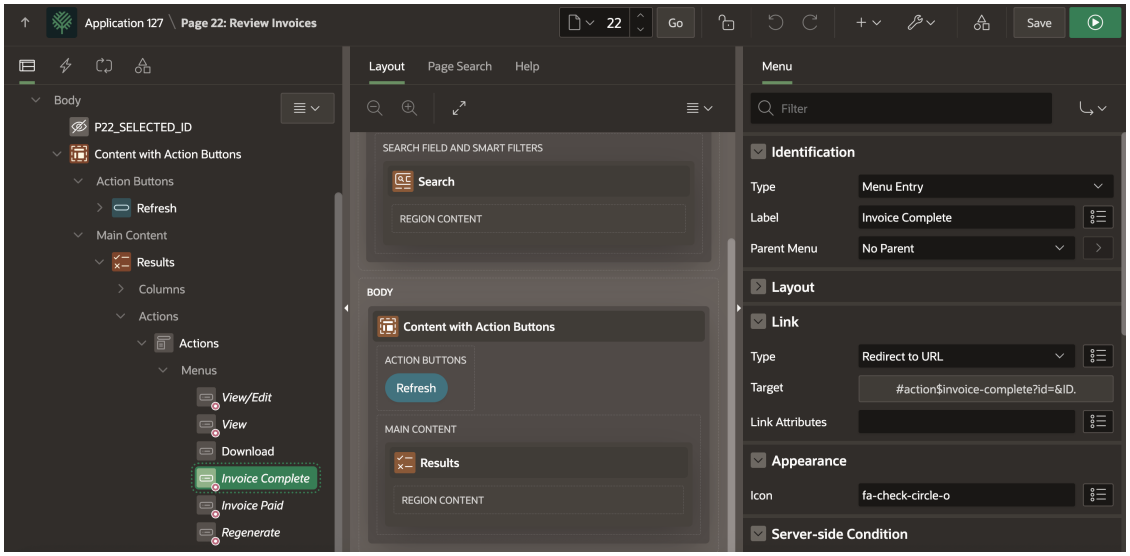
7.1.11 Action Positions and Action Templates

If your template component presents a single row or multiple rows of data, it can also define action positions. These are row-specific slots where teammates using the component can define buttons or menus to perform actions on a specific row.

After naming your action position, you include it in your *Partial* or *Report Row* template with the `#ACTIONPOSITIONNAME#` syntax. Each action position references a corresponding action template that defines the HTML markup for the button or menu option.

The figure below shows the *Rendering* tab for the native *Content Row* template component after defining an *Actions* menu in the *Primary Actions* action position. The menu defines several submenu options, including the *Invoice Complete* one selected.

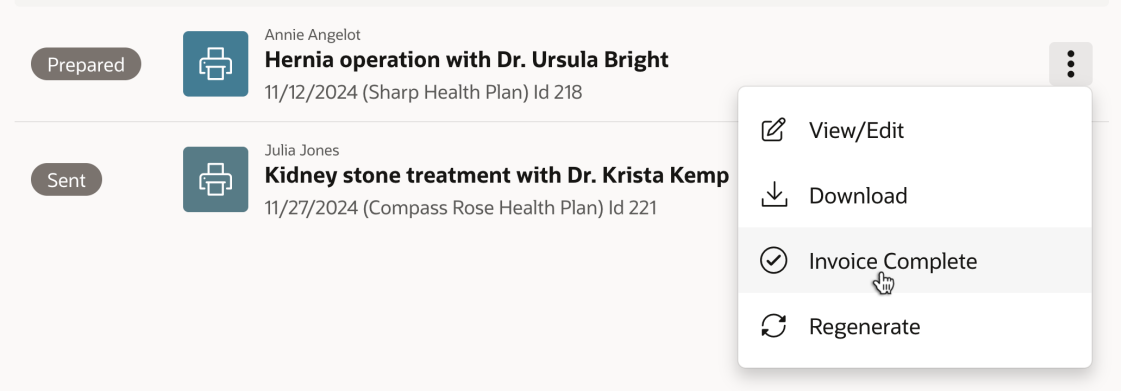
Figure 7-15 Defining Menu and Submenu Options in an Action Position



Copyright © 2026, Oracle and/or its affiliates.

At runtime, end users see the actions in the action position for each row rendered. In the figure below, two invoices appear in the *Content Row* region and the end user is selecting the *Invoice Complete* menu option on the first invoice displayed.

Figure 7-16 End User Choosing a Template Component Row Action



7.1.12 Defining Files and File URLs to Load

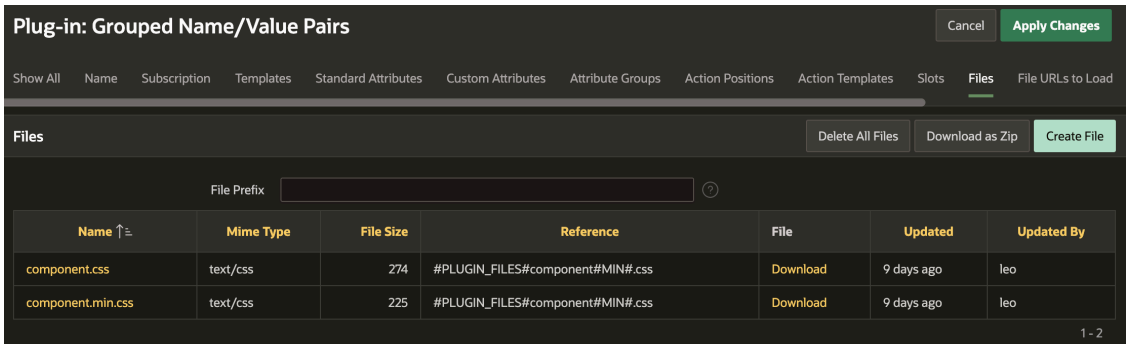
If your component uses Cascading Style Sheets (CSS) or JavaScript, store them in separate component files to improve maintainability.

For example, consider a *Grouped Name/Value Pairs* component that defines the following CSS rule in a file named `component.css`:

```
li.is-selected {
    background-color: #e0f0ff;
    color: #003366;
}
```

You create the file in the *Files* tab of the template component edit page shown below. As shown below, APEX automatically creates a smaller "minified" version of CSS and JavaScript files whenever you save or modify them. The *Reference* column in the file list provides the filename to reference in the *File URLs to Load* tab, where you explicitly list which component files should be loaded when the component is used on a page. List them using the *Reference* string for the file, one per line.

Figure 7-17 Files List for a Template Component



Copyright © 2026, Oracle and/or its affiliates.

7.2 Using a Plug-in from apex.world

By using a existing plug-in from the apex.world "app store", you can quickly add new functionality that another developer or team has made available to the APEX community.

The apex.world site is accessible at https://apex.world

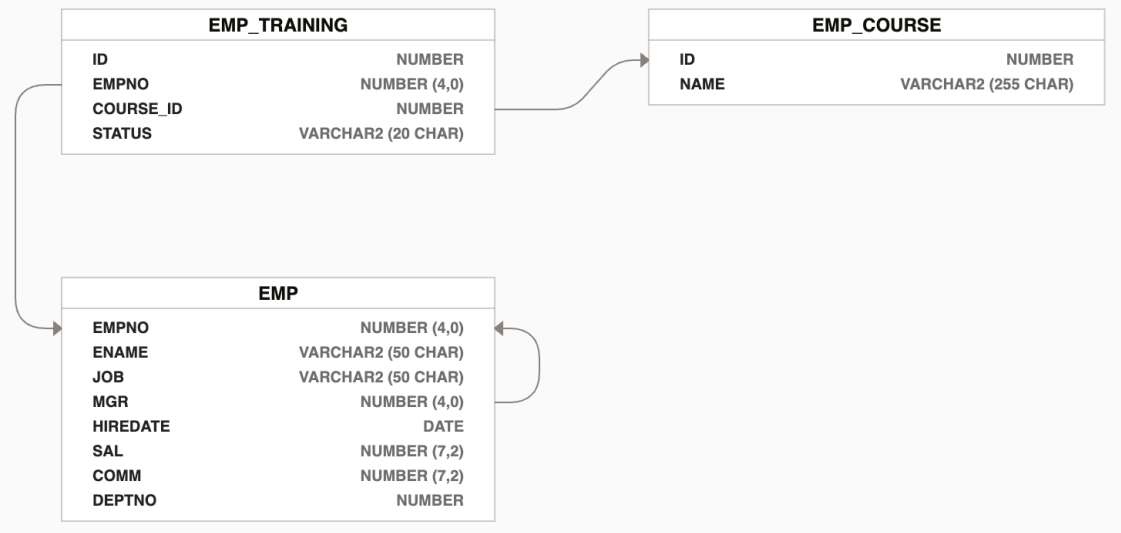
- Tracking Employee Training Status
  Suppose your application users need to track the status of an employee's training that progresses from *Enrolled* to *In Progress* to *Completed* to *Certified*.
- Downloading and Importing a Plug-in
  You can download and import plug-ins to add specific functionality to your application.
- Configuring the Kanban Plug-in
  Once you install the plug-in, it shows in the list of region types you can create in Page Designer.

7.2.1 Tracking Employee Training Status

Suppose your application users need to track the status of an employee's training that progresses from *Enrolled* to *In Progress* to *Completed* to *Certified*.

The data model shown below stores employees who enroll in a course, including the status of their training that can be `ENROLLED`, `IN_PROGRESS`, `COMPLETED`, and `CERTIFIED`. It shows the familiar `EMP` table with a related `EMP_TRAINING` table for trainings an employee undertakes, and an `EMP_COURSE` table related to each training.

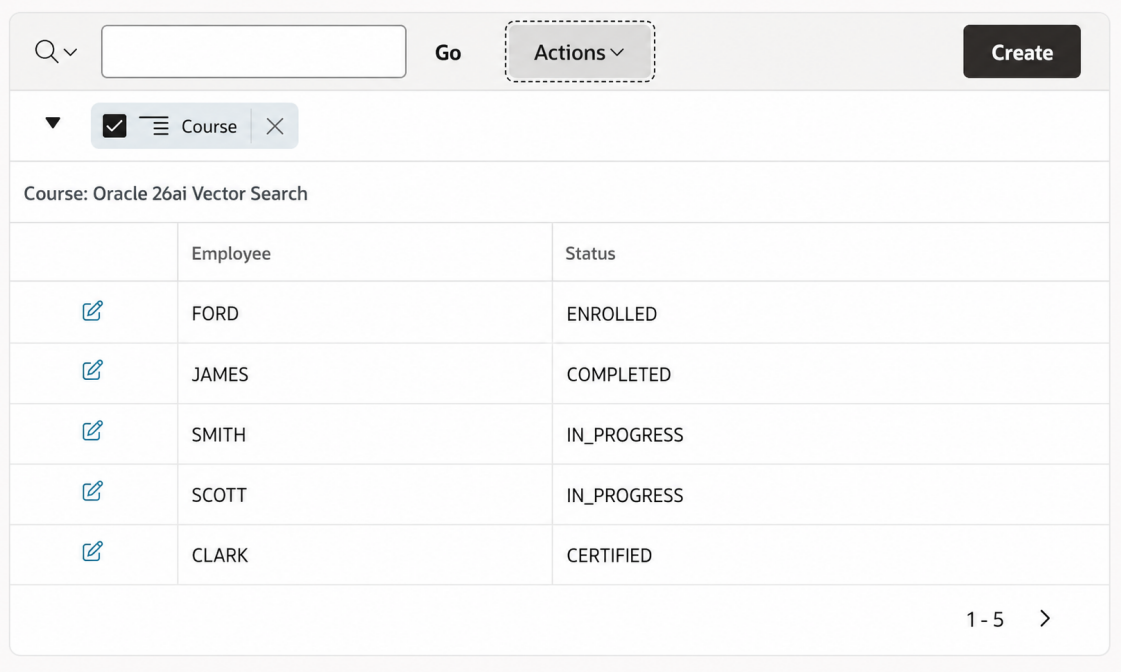
Figure 7-18 Additional EMP Tables Tracking Training Course Progress



You could use an Interactive Report region like the one below to show the status of employee training, using the *Action > Format > Control Break* to visually group the data into separate courses. The figure shows a *Vector Search* source with five employees in various stages of completion.

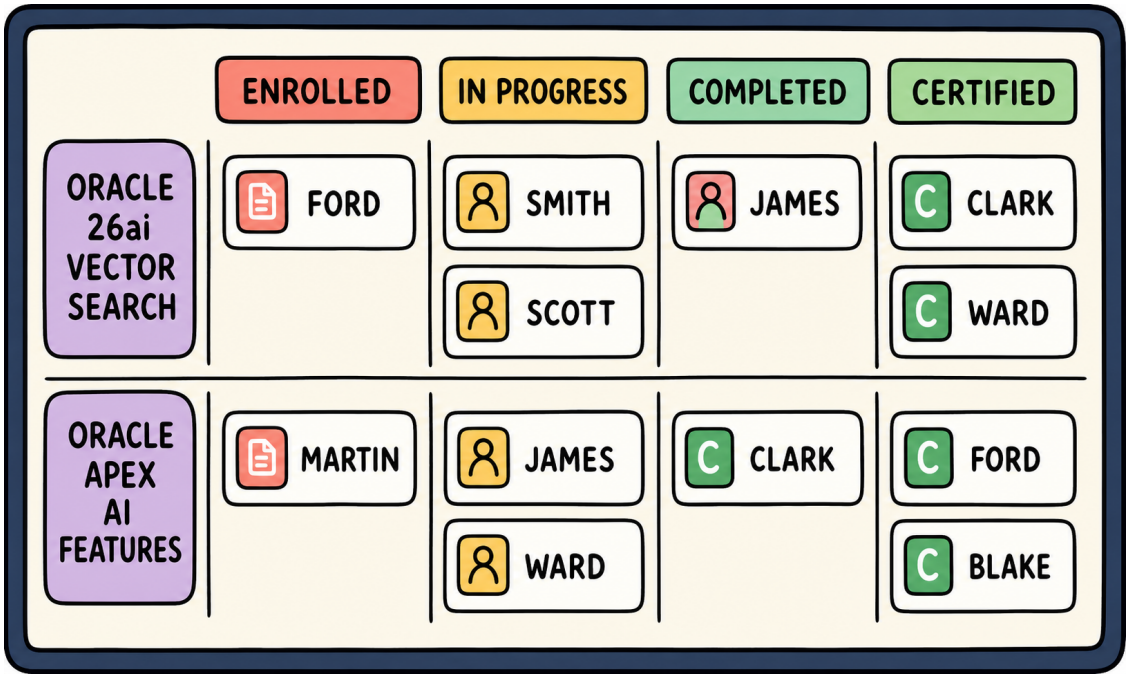
Copyright © 2026, Oracle and/or its affiliates.

Figure 7-19 Employee Training Status Using Interactive Report



However, showing the status of ongoing tasks is something a *Kanban Board* would be great for. Ideally, you could find an existing plug-in that would let you show your data like the UI mockup below. The diagram shows four vertical "columns" representing the *Enrolled*, *In Progress*, *Completed*, and *Certified* training statuses, and two horizontal "lanes" representing different *Vector Search* and *AI Features* courses. Employee tiles appear in each cell of the grid, representing the status of the training they must complete.

Figure 7-20 Kanban Board Mockup Showing Employee Training Status



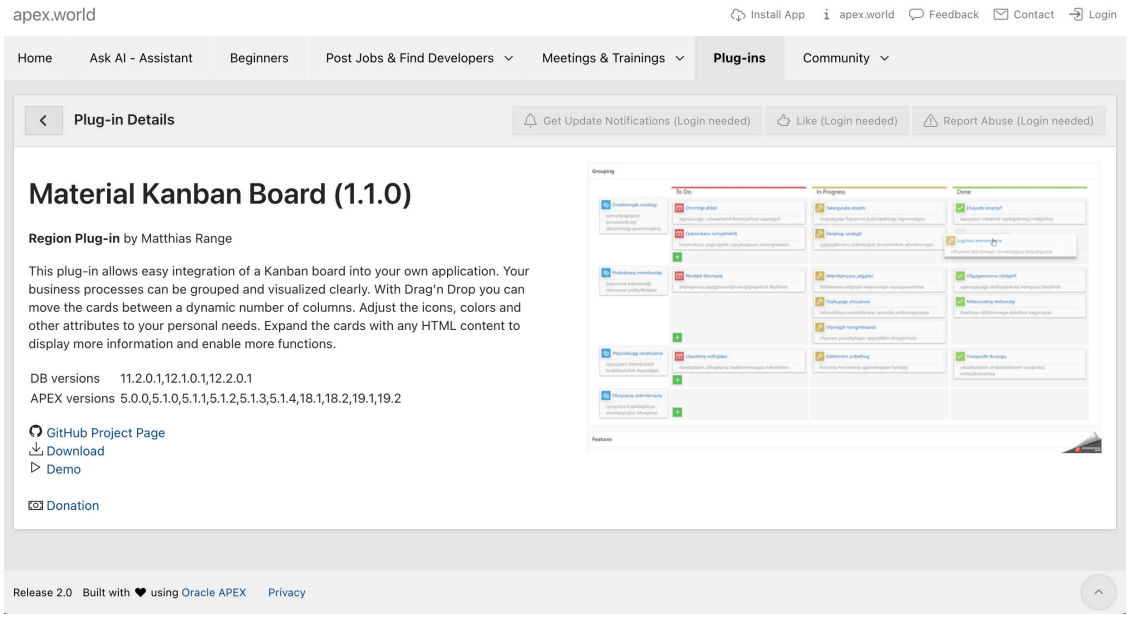
Copyright © 2026, Oracle and/or its affiliates.

7.2.2 Downloading and Importing a Plug-in

You can download and import plug-ins to add specific functionality to your application.

Searching the **Plug-ins** tab on https://apex.world for *kanban*, the *Material Kanban Board* plug-in shown below appears. Clicking its download link and unzipping the archive, you find its `region_type_plugin_material_kanban.sql` export file. The figure shows the plug-in details page on the apex.world site for the Material Kanban Board plug-in.

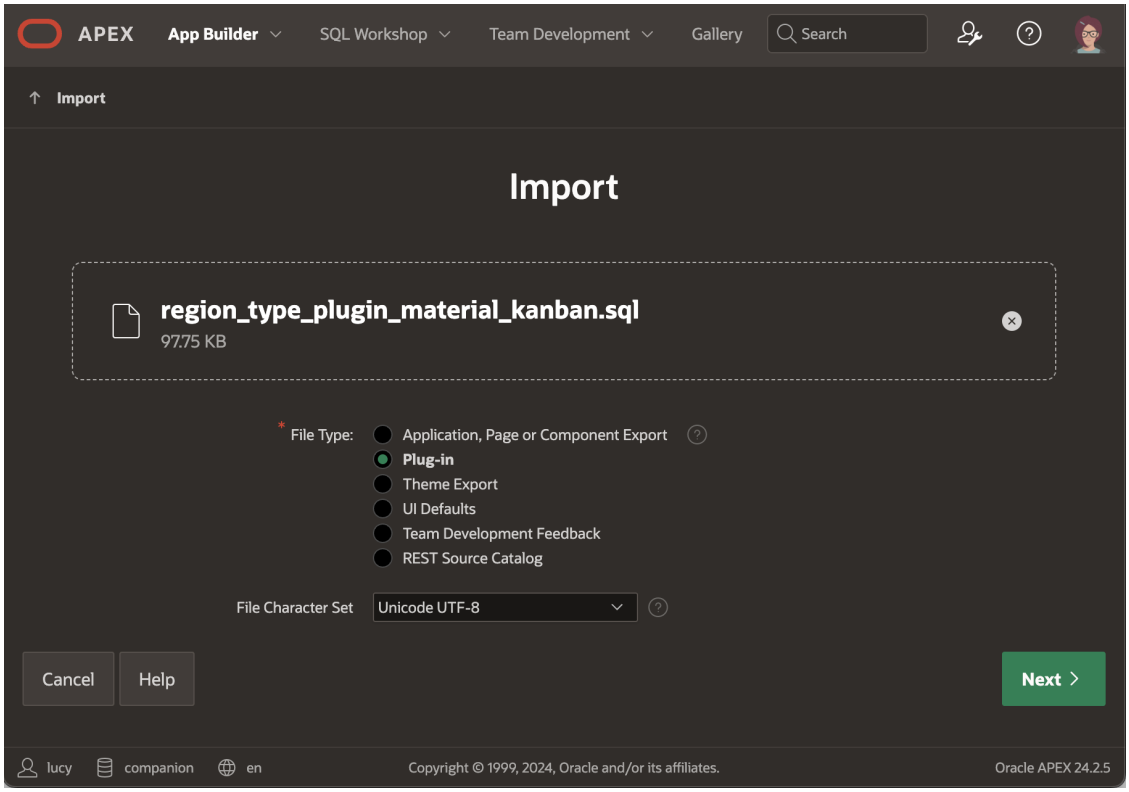
Figure 7-21 Kanban Board Plug-in on apex.world



To import the plug-in into your app, visit the *Shared Components* > *Plug-ins* list page and click the *(Import)* button. Choose the `region_type_plugin_material_kanban.sql` export file you downloaded, and click *(Next)* to complete the import. The figure shows the **Import** dialog after selecting the plug-in SQL file. Notice the Plug-in option is selected in the **File Type** radio group.

Copyright © 2026, Oracle and/or its affiliates.

Figure 7-22 Importing a Plug-in Into Your Application



7.2.3 Configuring the Kanban Plug-in

Once you install the plug-in, it shows in the list of region types you can create in Page Designer.

The figure below shows the new *Material Kanban Board* region on the page and the plug-in's *ConfigJSON* settings. Following the directions in the plug-in's documentation, you define four columns with appropriate status title and icon using the JSON format. They reflect the *Enrolled*, *In Progress*, *Completed*, and *Certified* statuses your application needs. Since you plan to use the grouping feature, you also set `groupExtension` to `true`.

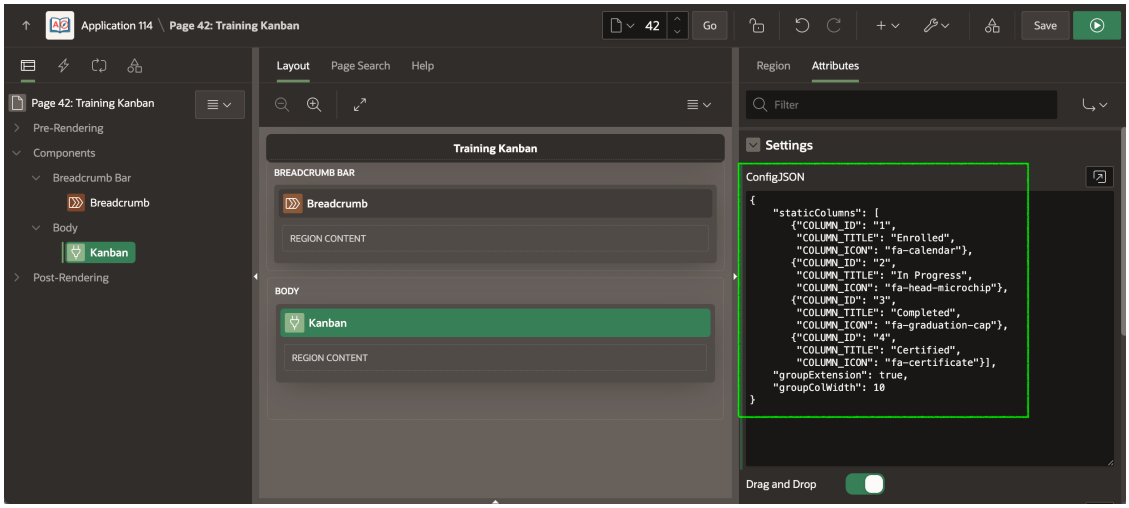
```
{
    "staticColumns": [
       {"COLUMN_ID": "1",
        "COLUMN_TITLE": "Enrolled",
        "COLUMN_ICON": "fa-calendar"},
       {"COLUMN_ID": "2",
        "COLUMN_TITLE": "In Progress",
        "COLUMN_ICON": "fa-head-microchip"},
       {"COLUMN_ID": "3",
        "COLUMN_TITLE": "Completed",
        "COLUMN_ICON": "fa-graduation-cap"},
       {"COLUMN_ID": "4",
        "COLUMN_TITLE": "Certified",
        "COLUMN_ICON": "fa-certificate"}],
    "groupExtension": true,
```


Copyright © 2026, Oracle and/or its affiliates.

```
    "groupColWidth": 10
}
```

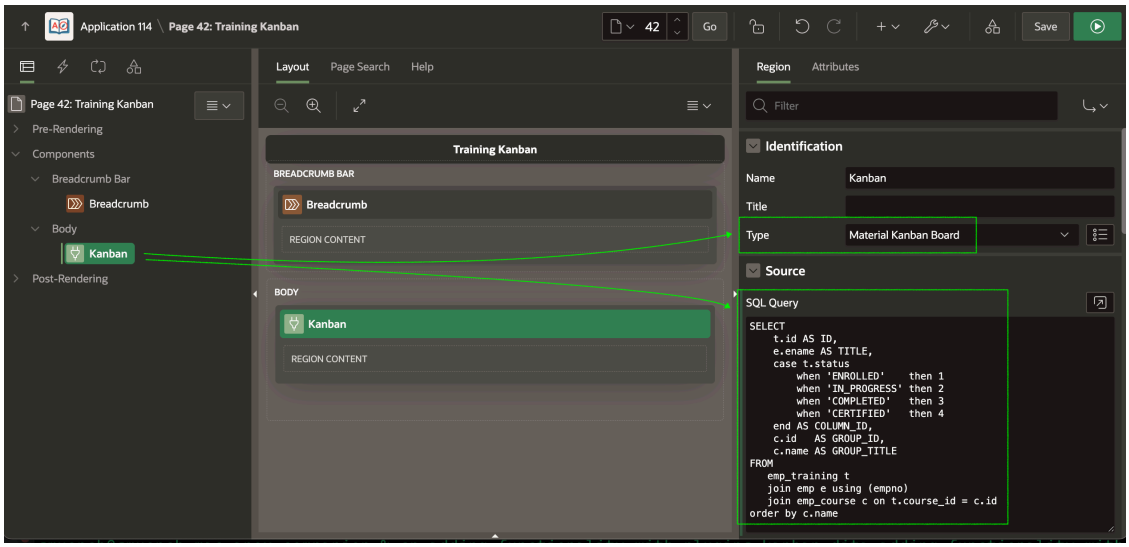
The figure highlights the **ConfigJSON** setting of the *Material Kanban Board* region selected in the component tree in Page Designer.

Figure 7-23 Defining Kanban Board Status Columns and Icons



Then as shown below, you write a SQL Query that defines the information the Kanban board will display. The SQL query joins the `EMP_TRAINING`, `EMP_COURSE`, and `EMP` tables to retrieve key details about the courses and employees in training.

Figure 7-24 Configuring the Kanban Region's SQL Query



This particular plug-in uses the column names in the SQL Query you configure to identify the role each column plays in the Kanban board display. The query appears below. Notice the column aliases for the `TITLE` of the elements to display and the `GROUP_TITLE` that defines

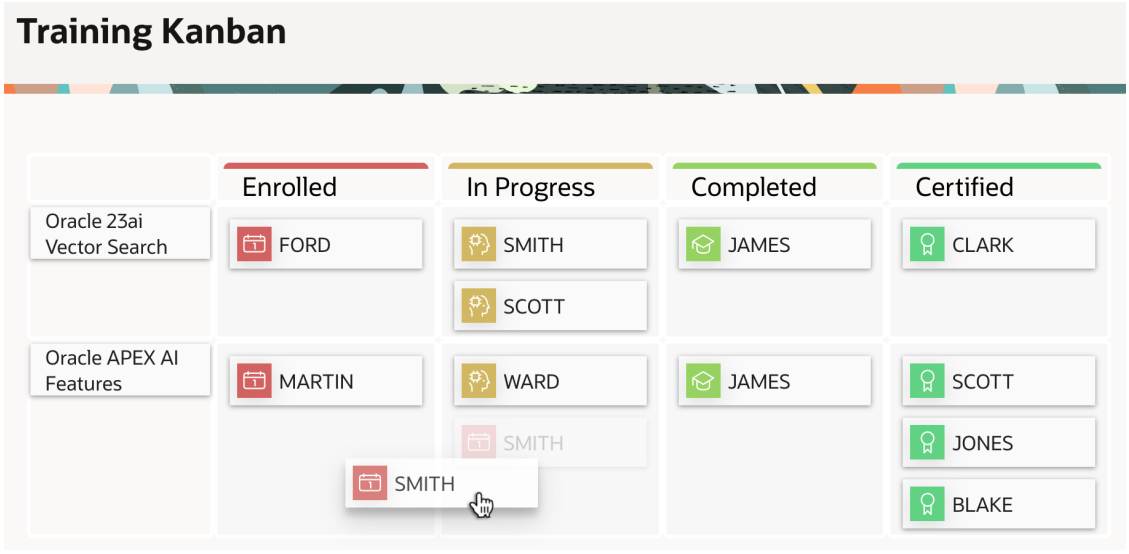
Copyright © 2026, Oracle and/or its affiliates.

different "swim lanes" of related elements. Here, the items being displayed are `ENAME` values of each employee taking a course, and the course name provides the group title. Finally, `COLUMN_ID` in the `SELECT` list is the sequential column number you defined in the JSON configuration above corresponding to the different status values an employee training can have.

```
SELECT
    t.id AS ID,
    e.ename AS TITLE,
    case t.status
        when 'ENROLLED'    then 1
        when 'IN_PROGRESS' then 2
        when 'COMPLETED'   then 3
        when 'CERTIFIED'   then 4
    end AS COLUMN_ID,
    c.id   AS GROUP_ID,
    c.name AS GROUP_TITLE
FROM
   emp_training t
   join emp e using (empno)
   join emp_course c on t.course_id = c.id
order by c.name
```

With these two configurations done, you can run the page to see the employee training status at a glance in your new Kanban board. With a small amount of additional PL/SQL code, you can enable end users to change the status of an employee's training with drag and drop. The figure shows the employees in various stages of their assigned trainings, split into a different "lane" for each distinct course.

Figure 7-25 Kanban Board Region Plug-in Shows Employee Training Status



Copyright © 2026, Oracle and/or its affiliates.

8

Understanding Page Flow & Session State

Your app helps end users get their jobs done, and they expect easy access to all tasks they can perform. Your *navigation menu* shows them what business functions are available, and your *navigation bar list* lets them access app options.

Often a task they initiate requires visiting multiple pages to complete. You define this "flow" of pages using *branches* that link one page to another. *Breadcrumbs* or a *Wizard Progress* list help users understand where they are in a flow and how to go back. Modal dialogs provide another option to perform related actions and appear on top of the page that branches to them. In addition to these facilities to define and access page flows, APEX also dramatically simplifies their development by managing user sessions and the temporary storage of data values across pages. To cement your understanding of these fundamental APEX concepts, this section explores a three-step wizard flow and a typical list/edit flow.

- Navigating to Business Functions
  After end users log in, they expect to see a list of tasks they can do.
- App Options in the Navigation Bar
  Users look for settings and sign out options in a dropdown menu under their username. Your *Navigation Bar List* entries define the items that appear there.
- Configuring a Wizard Page Flow
  A wizard is a multi-step UI pattern that helps users complete a task intuitively.
- Additional Page Flow Features
  In addition to getting rendered and submitted, APEX pages also uses background data requests and support redirects to navigate without submitting data. Breadcrumbs help visualize the navigation path of drill down page flows. Modal dialogs offer an alternative to present related information without losing the context of the calling page.
- Understanding Session State Management
  When an end user submits a page, sometimes you save their input to application tables immediately. Other times, to improve usability or manage file uploads, you need to store values *temporarily* and save them later.
- Studying a List and Edit Flow
  The Create Page wizard builds list and edit pages for any data source in a few clicks. Understanding how those pages work together helps you use them well.

8.1 Navigating to Business Functions

After end users log in, they expect to see a list of tasks they can do.

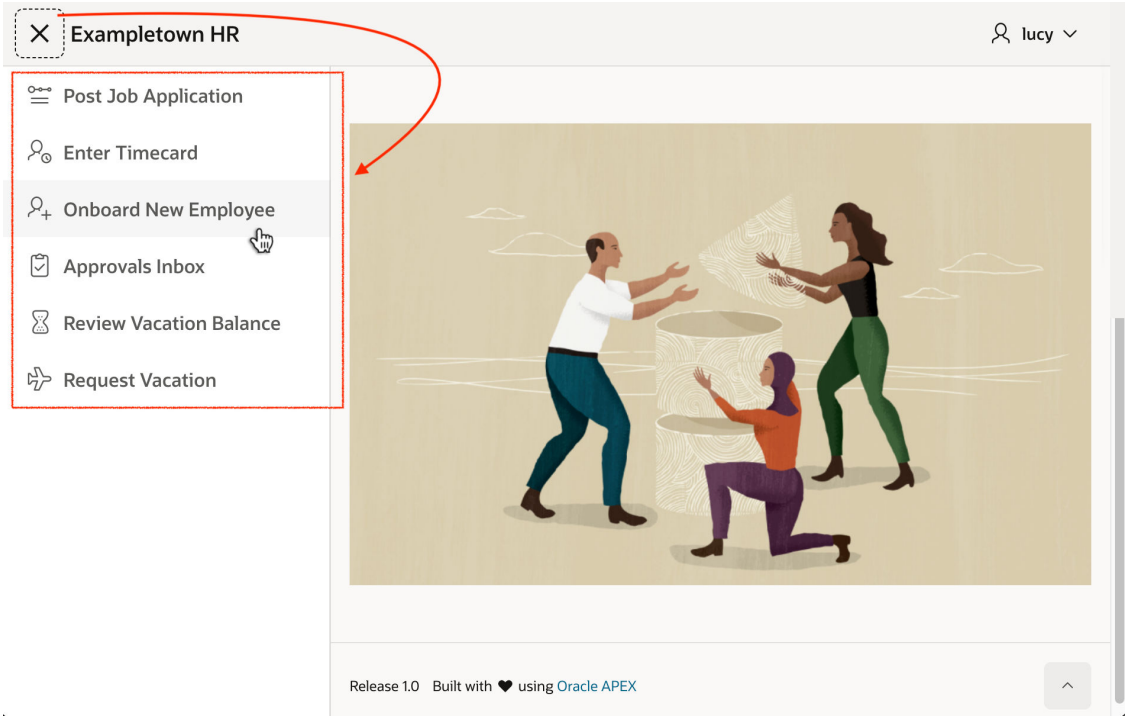
You define *Navigation Menu* list entries to give users this task-focused guidance. The figure below shows six entries on the collapsible *Navigation Menu* on the left. Clicking any of the choices navigates the user to the page that initiates the selected business function. For example, the business functions listed for this Exampletown HR application are *Post Job Application*, *Enter Timecard*, *Onboard New Employee*, *Approvals Inbox*, *Review Vacation Balance*, and *Request Vacation*.

When you use the *Create Page* wizard, you choose whether to add a navigation menu entry for your new page. You can also define a new entry any time in *Shared Components* >

Copyright © 2026, Oracle and/or its affiliates.

Navigation Menu. For more information, see Managing Navigation Menus in *Oracle APEX App Builder User's Guide*.

Figure 8-1 Navigation Menu Lets Users Initiate Business Functionality



8.2 App Options in the Navigation Bar

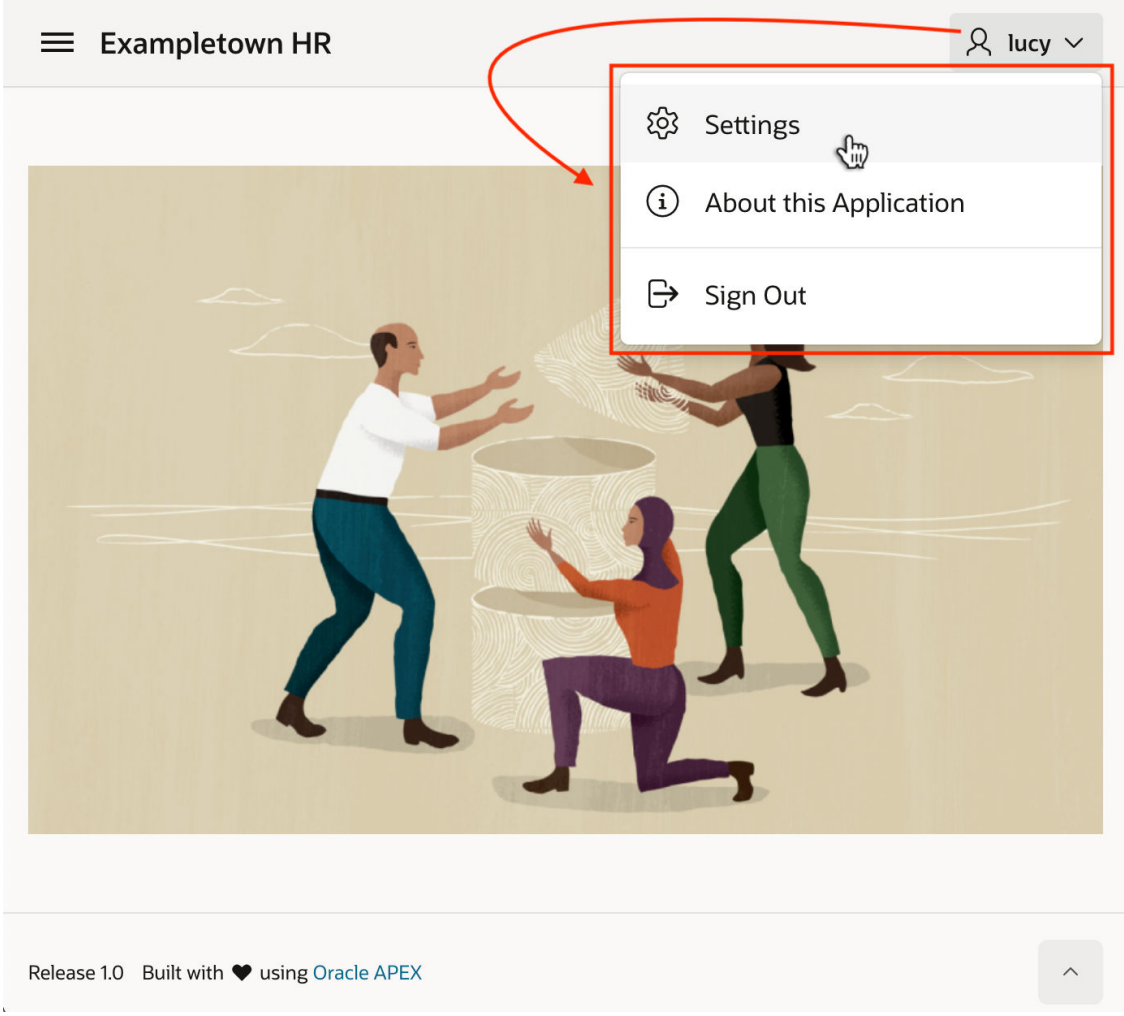
Users look for settings and sign out options in a dropdown menu under their username. Your *Navigation Bar List* entries define the items that appear there.

The figure below shows three entries under the logged-in user's name: *Settings*, *About this Application*, and *Sign Out*.

The *Create Application* wizard sets up an initial *Sign Out* entry, and other features like *About Page*, and *Push Notification* add entries, too. You can define a new entry in *Shared Components* > *Navigation Bar List*. For more information on this shared component, see Managing Navigation Bars in *Oracle APEX App Builder User's Guide*.

Copyright © 2026, Oracle and/or its affiliates.

Figure 8-2 Navigation Bar List for Application Options and Logout



8.3 Configuring a Wizard Page Flow

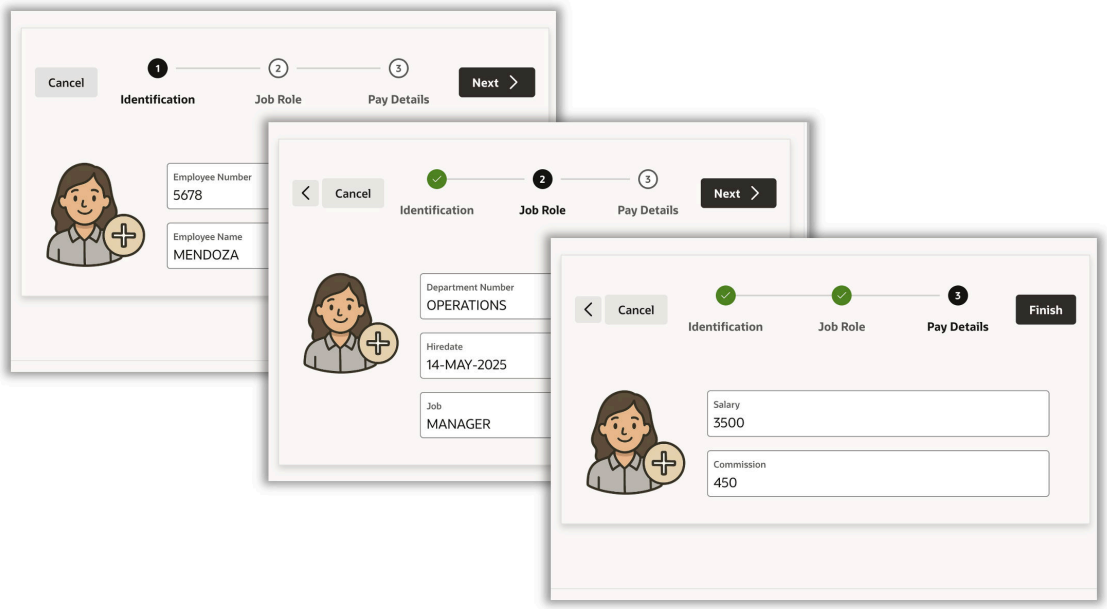
A wizard is a multi-step UI pattern that helps users complete a task intuitively.

A wizard guides users through a sequence of smaller, structured actions in a logical order, one page at a time. The user sees a "route map" at the top. It identifies the "stops" along the way to the destination. For example, onboarding an new employee involves entering their identifying info, job role, and pay details. The figure below shows a simple example of a three-page employee onboarding wizard. Users expect to enter data across the steps and go back to review data in a previous step if needed before completing the task by clicking (Finish) on the last step.

Your application's multi-step wizard contains multiple *pages* linked by *branches* for forward and back navigation. Progress indicator *list entries* define the wizard step titles.

Copyright © 2026, Oracle and/or its affiliates.

Figure 8-3 Three Step Wizard for Onboarding an Employee



- Creating a Multi-Step Wizard
  The *Create Page* wizard automates making your app's wizard. It generates the wizard's multiple pages linked by branches, with progress indicator list entries that define the wizard step titles.
- Multi-Step Wizard in Page Designer
  After the wizard creates the initial pages, use Page Designer to add step-specific content to each one.
- Shared List Defines Wizard Pages
  The page entries in a shared component list define the wizard "train" stop titles and their sequence.
- Branches Link Pages to Define Flows
  You establish the page *navigation* flow an end user can follow by defining *branches*.
- Render, Submit, Branch, Repeat
  At runtime, the model-driven APEX engine breathes life into your page and component definitions using a simple lifecycle: render, submit, branch, and repeat.

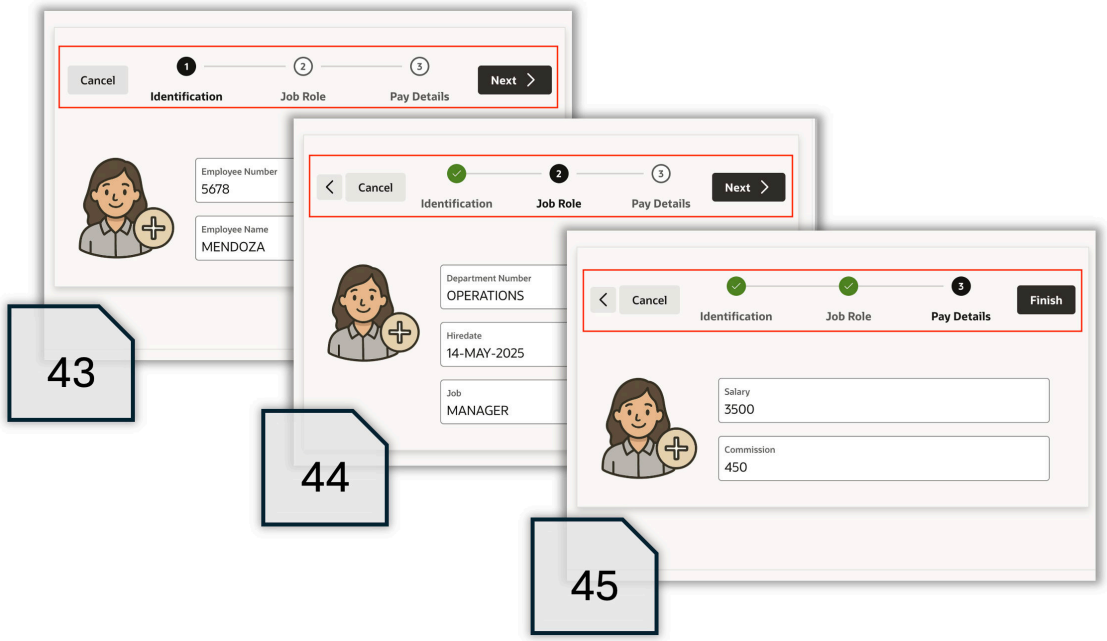
8.3.1 Creating a Multi-Step Wizard

The *Create Page* wizard automates making your app's wizard. It generates the wizard's multiple pages linked by branches, with progress indicator list entries that define the wizard step titles.

As shown below, you need to create three pages for this *Onboard New Employee* wizard. Here they are numbered 43, 44, and 45. Each page includes a wizard progress indicator at the top showing the steps involved and the current step. This area includes appropriate buttons to proceed to the next step, return to the previous step, finish the task, and cancel the flow. Each page also contains page items for entering data appropriate to the step at hand.

Copyright © 2026, Oracle and/or its affiliates.

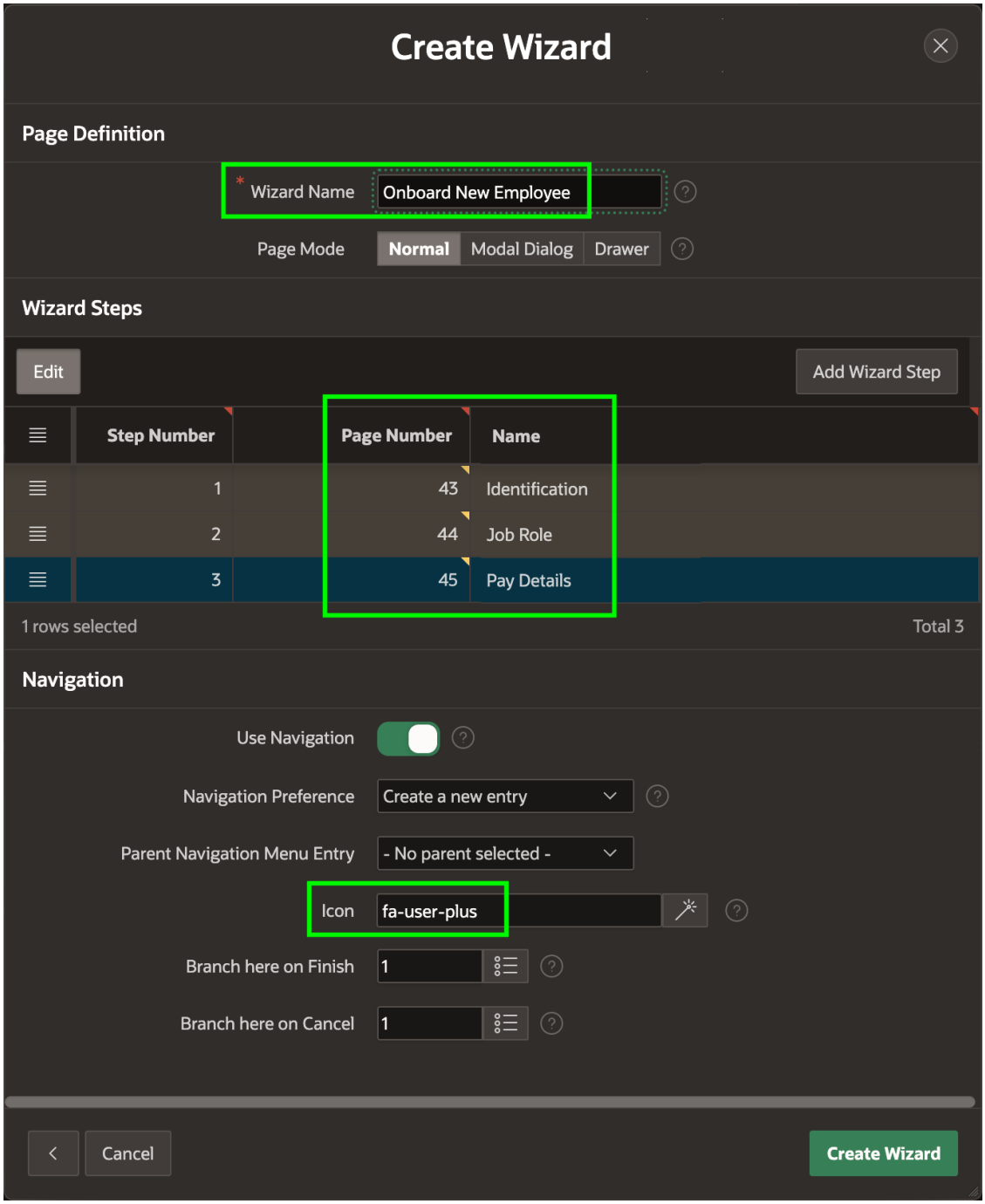
Figure 8-4 Wizard Container on Each Page Shows Steps in a Flow



The Create Page wizard sets up this multi-step page flow. Choosing the *Wizard* tile in the Create Page dialog and clicking *(Next >)* shows the *Create Wizard* step shown below. There you define the name of the wizard, the step labels and page numbers, and pick a navigation menu icon. Clicking *(Create Wizard)* creates the pages and opens the first page in Page Designer.

Copyright © 2026, Oracle and/or its affiliates.

Figure 8-5 Defining the Wizard Steps in the Create Wizard Dialog



8.3.2 Multi-Step Wizard in Page Designer

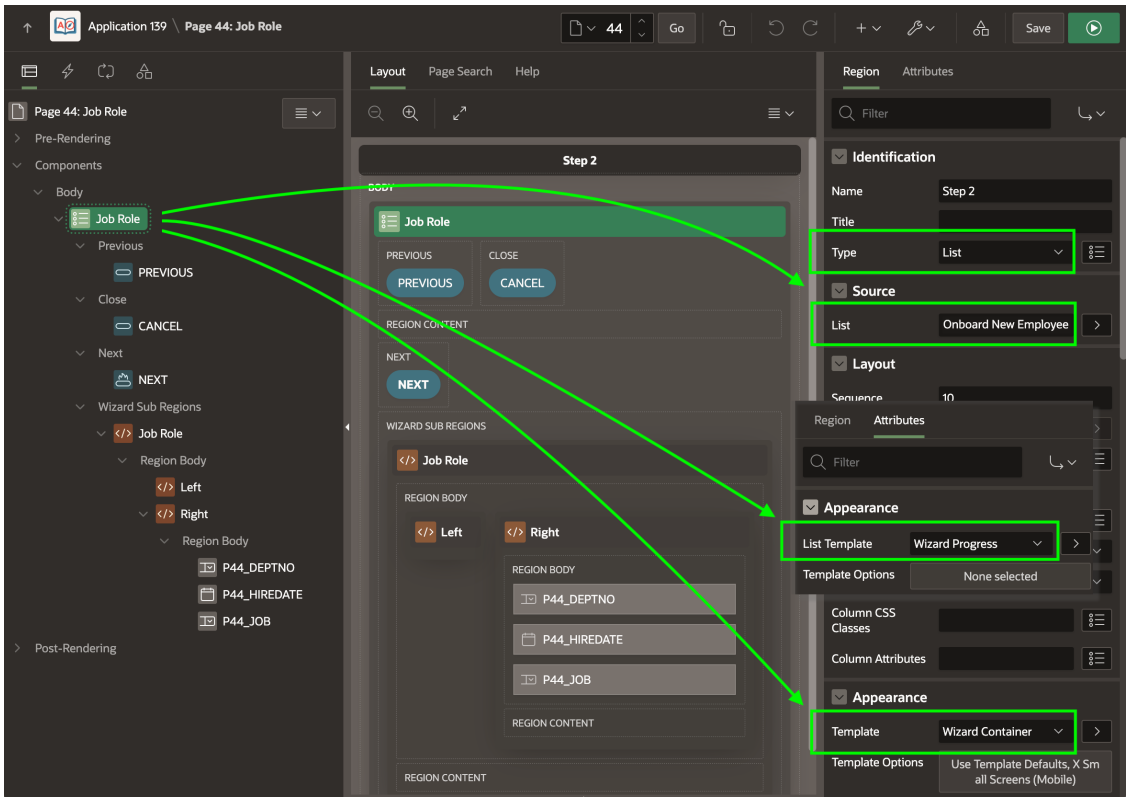
After the wizard creates the initial pages, use Page Designer to add step-specific content to each one.

The figure below shows page 44 in Page Designer after adding the three page items into the *Job Role* region the wizard created. Notice this region is selected in the *Rendering* tab and that the Property Editor shows it's a *List* region. The *Job Role* list *region* uses the *Wizard Container* template option and the *Wizard Progress* list template. This choice renders the list as a

Copyright © 2026, Oracle and/or its affiliates.

sequence of "train" stops. It is one of 13 useful ways a list region can display a list of pages. In the example below, the shared List component named *Onboard New Employee* provides the page list. The list region's *Previous*, *Close*, and *Next* slots contain buttons that the *Wizard Container* template shows in line with the steps.

Figure 8-6 Wizard List Region Shows List of Pages as "Train" Stops



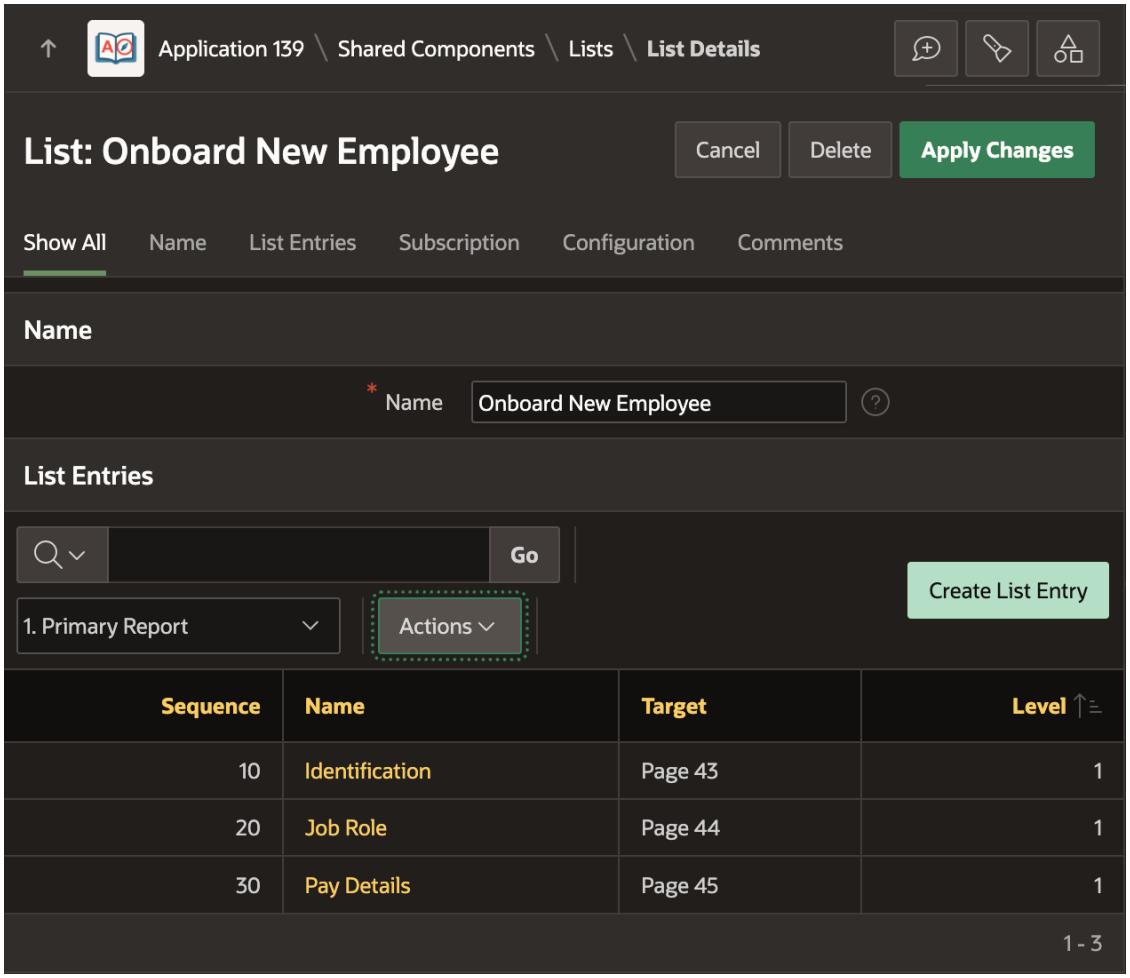
8.3.3 Shared List Defines Wizard Pages

The page entries in a shared component list define the wizard "train" stop titles and their sequence.

The figure below shows the definition of the *Onboard New Employee* shared component list. It contains the three page entries for pages 43, 44, and 45, with appropriate names that the *List* region displays as step titles of the "train" stops. The *Wizard Progress* list template configured on the list region in each wizard page determines how the list entries render.

Copyright © 2026, Oracle and/or its affiliates.

Figure 8-7 List Shared Component Defining Wizard Step Pages and Titles



8.3.4 Branches Link Pages to Define Flows

You establish the page *navigation* flow an end user can follow by defining *branches*.

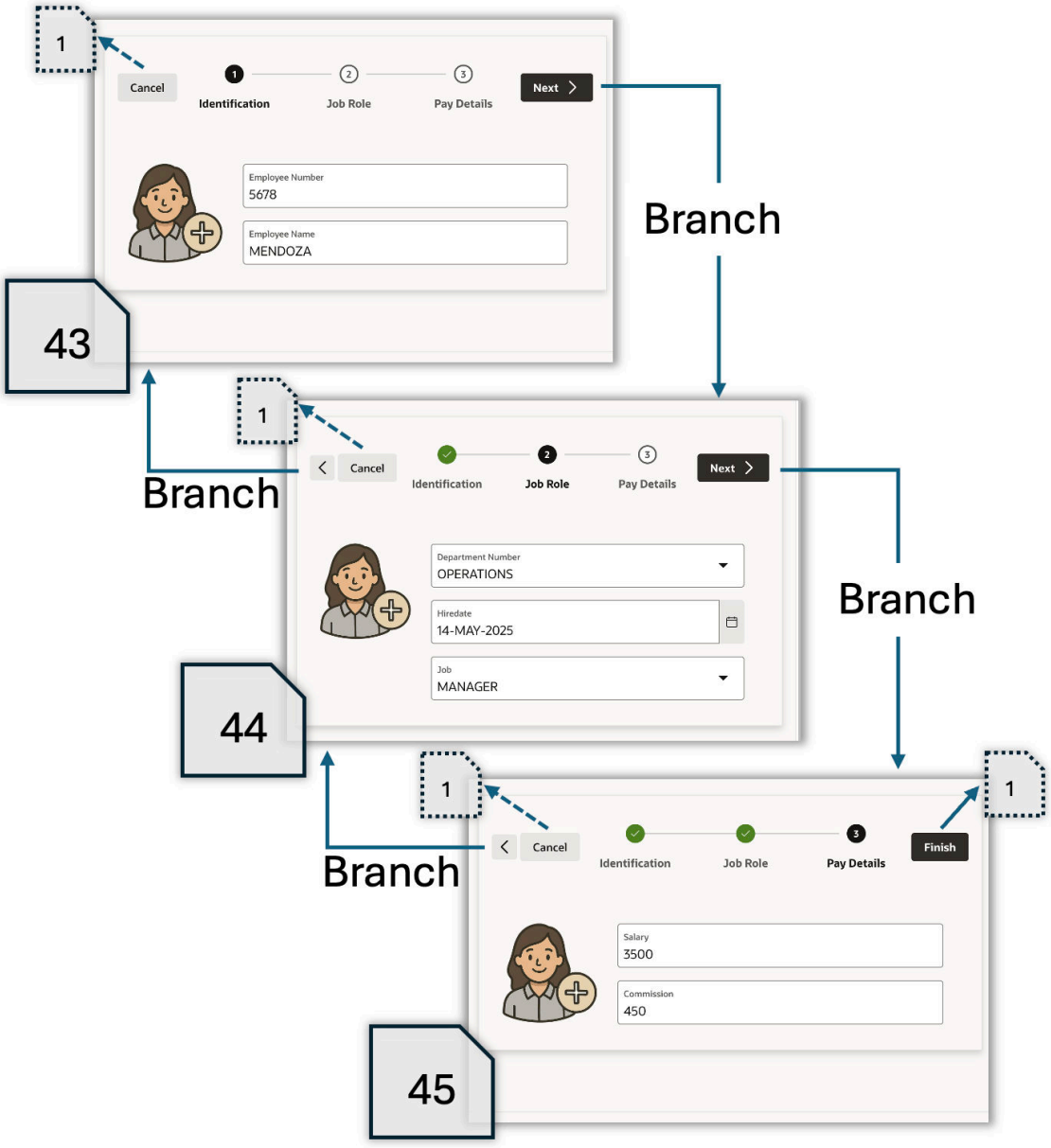
In Page Designer's *Processing* tab, you create a branch to identify a target page the user should see next, optionally passing along parameter values. By setting the *When Button Pressed* property on the branch to the name of a button, you indicate the branch should happen *only* when the user clicks *that* button. As shown below, branches connect the three pages of the *Onboard New Employee* wizard:

- Page 43 has a branch for the
  - *(Next)* button that targets page 44.
- Page 44 has a branch for the
  - *(<)* previous button that targets page 43, and
  - *(Next)* button that targets page 45.
- Page 45 has a branch for the
  - *(<)* button that targets page 44, and
  - *(Finish)* button that targets a home page (1).

Copyright © 2026, Oracle and/or its affiliates.

Each of the three pages configures its *(Cancel)* button to *redirect* to the home page (1) and this button's *Warn on Unsaved Changes* property to *Do Not Check*. A redirect acts like a hyperlink. It navigates the user to another page *without* submitting any data from the current page for processing. Later sections explain how branches and redirects differ. The figure shows the three wizard pages and the branches their buttons trigger to navigate between pages.

Figure 8-8 Branches Connect Three Pages in Onboard New Employee Flow



8.3.5 Render, Submit, Branch, Repeat

At runtime, the model-driven APEX engine breathes life into your page and component definitions using a simple lifecycle: render, submit, branch, and repeat.

Copyright © 2026, Oracle and/or its affiliates.

- Wizard Example Page Lifecycle
  When a user clicks *Onboard New Employee* in your app's navigation menu, the APEX engine renders page 43 based on its page definition. As the user enters data and clicks buttons in the page, branches control what page to show next. Optional computations, validations, and processing let you create flows of pages for any business process.
- Computing Values on Render or Submit
  Define *computations* to set the value of a page item or application item.
- Validating Submitted Data
  Define *validations* to enforce the correctness of submitted data before saving it. You create them in the *Validating* execution point of the *Processing* tab in your page.
- Custom Processing on Render or Submit
  Create page *processes* for custom behavior in the *Pre-Rendering* phase of the *Rendering* tab, or in the *After Submit* or *Processing* phases of the *Processing* tab. The *Execute Code* process type lets you include custom code to performs any business logic required, while you can use other native process types to handle other common tasks declaratively.
- Best Practice Use of Invoke API
  For easier maintenance and testing of your custom code, Oracle recommends writing most business logic in functions and procedures of a PL/SQL package. You can then engage these packaged program units from one-line statements or expressions in your pages, or invoke them declaratively using an *Invoke API* page process.
- Making Almost Any Element Conditional
  Add a Server-side Condition to almost any element in your APEX app to control whether APEX renders or runs it.

8.3.5.1 Wizard Example Page Lifecycle

When a user clicks *Onboard New Employee* in your app's navigation menu, the APEX engine renders page 43 based on its page definition. As the user enters data and clicks buttons in the page, branches control what page to show next. Optional computations, validations, and processing let you create flows of pages for any business process.

It sends appropriate data, HTML markup, CSS style rules, and JavaScript to the user's browser based on the regions, template options, page items, and buttons the page contains.

Before rendering the page, however, it executes any *computations* or *processes* you define in the page's *Pre-Rendering* section of the Page Designer *Rendering* tab. These optional elements influence the data initially shown to the user. The net result: the user sees the first page of the *Onboard New Employee* page flow in their browser. The diagram below illustrates the *Onboard New Employee* wizard pages lifecyle.

After entering the employee's identifying information, when the user clicks on the *(Next)* button in page 43, as shown below, the browser submits the page to the APEX engine for processing. It includes the values of all page items. When APEX engine receives this page submission, if you defined any *computations*, *validations*, *processes*, or *branches* in the *Processing* tab in Page Designer, then it executes these elements in the order you sequenced them. You can optionally configure a *server-side condition* on any of these elements, or a *When Button Pressed* condition. This lets you control *exactly when* each element executes.

Page 43 defines a *branch* for the *(Next)* button the user clicked. This determines page 44 shows next, and the process repeats itself. Page 44 renders to the browser.

From the user's perspective, they:

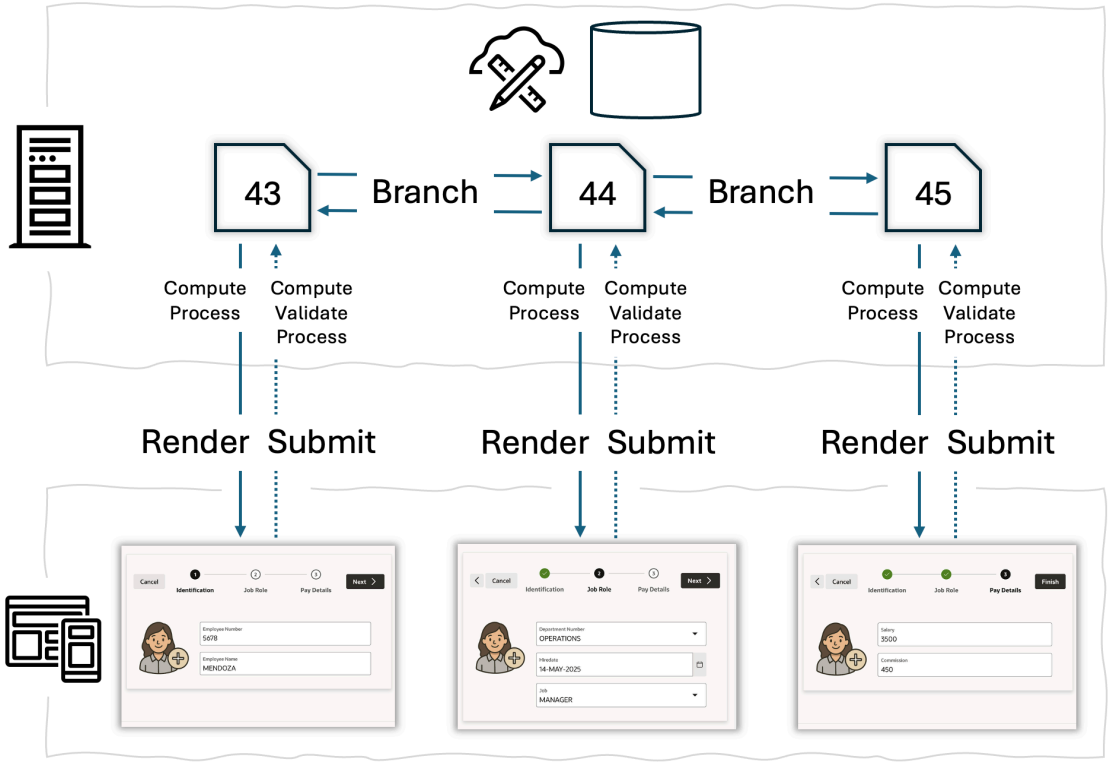
- click a navigation menu link to start the wizard,
- enter some data on the *Identification* step of the wizard and click *(Next)*, and

Copyright © 2026, Oracle and/or its affiliates.

- see the *Job Role* step of the wizard.

If *there* they click the *(<)* previous button, they return to the *Identification* step.

Figure 8-9 Page Renders to Browser, then User Submits Page for Processing



For more information, see Controlling Navigation Using Branches in *Oracle APEX App Builder User's Guide*.

8.3.5.2 Computing Values on Render or Submit

Define *computations* to set the value of a page item or application item.

If the calculated value influences what the page initially displays, then set its *Execution Point* to a stage in the *Pre-Rendering* phase. If, instead, the computation affects the processing of submitted data, then set that property to the *After Submit* stage in the *Processing* tab.

You can assign the target item:

- a static value
- the value of another item
- the result of a PL/SQL expression
- the value of a single-column SQL query, or
- a user preference.

If the data item you set has its *Multiple Values* property set, then a multi-row query result gets formatted appropriately either using a delimiter or as a JSON Array.

Copyright © 2026, Oracle and/or its affiliates.

Computations in the *Pre-Rendering* section of your page set up data the user will initially see on the page. Ones in the *Processing* section are good for computing hidden fields, or adjusting user input, before saving their values to a system of record table.

For more information, see Understanding Page Computations in *Oracle APEX App Builder User's Guide*.

8.3.5.3 Validating Submitted Data

Define *validations* to enforce the correctness of submitted data before saving it. You create them in the *Validating* execution point of the *Processing* tab in your page.

Validations can check the value of a single page item or test a more complex condition involving multiple values. They can consult system of records tables as needed.

For convenience, you can also create one for a particular page item by choosing *Create Validation* from the item's context menu in the *Rendering* tab. In this case, Page Designer defaults its *Associated Item* property as expected.

Each validation defines a condition that determines if the value or values being checked are considered valid. You can pick from a list of declarative conditions, use a PL/SQL expression or boolean returned from a PL/SQL function body, or use a SQL statement that returns a row or not. If the condition is true, the validation succeeds. If it is false, the end user sees your configured error message. If any validations fail, the user sees *all* of their failure messages and the APEX engine does not execute the elements in the *Processing* phase of the *Submit* lifecycle.

For more information, see Understanding Validations in *Oracle APEX App Builder User's Guide*.

> **Tip**
>
> To validate columns in an Interactive Grid, set the *Editable Region* property of the validation to the name of the related grid and then reference the grid's columns by name when defining the validity condition.

8.3.5.4 Custom Processing on Render or Submit

Create page *processes* for custom behavior in the *Pre-Rendering* phase of the *Rendering* tab, or in the *After Submit* or *Processing* phases of the *Processing* tab. The *Execute Code* process type lets you include custom code to performs any business logic required, while you can use other native process types to handle other common tasks declaratively.

Built-in page process types include starting a workflow, sending an email or push notification, printing a report, loading data, invoking a PL/SQL or REST API, storing user preferences, saving form or grid changes, and more. You can find other page process types created by the APEX developer community "app store" on https://apex.world.

For example, in the last page of the *Onboard a New Employee* wizard, as shown below, an *Execute Code* page process with *When Button Pressed* set to the `FINISH` button inserts a new `EMP` row using the page item values from page 43, 44, and 45 to add a new employee. Notice the *Success Message* is configured to display a message like "New employee MENDOZA added successfully" when the process is successful. The code looks like this:

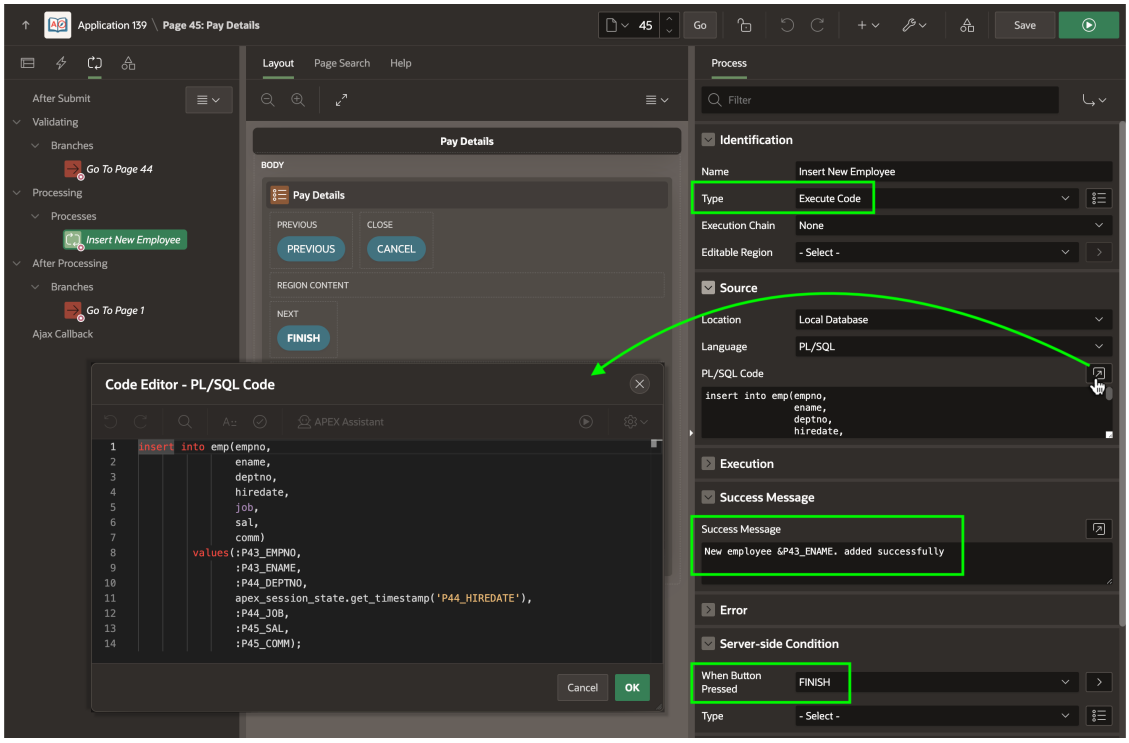
```
insert into emp(empno,
                ename,
```


Copyright © 2026, Oracle and/or its affiliates.

```
            deptno,
            hiredate,
            job,
            sal,
            comm)
    values (:P43_EMPNO,
            :P43_ENAME,
            :P44_DEPTNO,
            apex_session_state.get_timestamp('P44_HIREDATE'),
            :P45_JOB,
            :P45_SAL,
            :P45_COMM);
```

For more information, see Understanding Page Processes in *Oracle APEX App Builder User's Guide*.

Figure 8-10 Execute Code Page Process to Insert New Employee



8.3.5.5 Best Practice Use of Invoke API

For easier maintenance and testing of your custom code, Oracle recommends writing most business logic in functions and procedures of a PL/SQL package. You can then engage these packaged program units from one-line statements or expressions in your pages, or invoke them declaratively using an *Invoke API* page process.

For example, you can move the `INSERT` statement that adds a new employee into an `ADD_NEW_EMPLOYEE` procedure in an `HR_APP` package like the following:

```
create or replace package body hr_app as
    procedure add_new_employee(
```

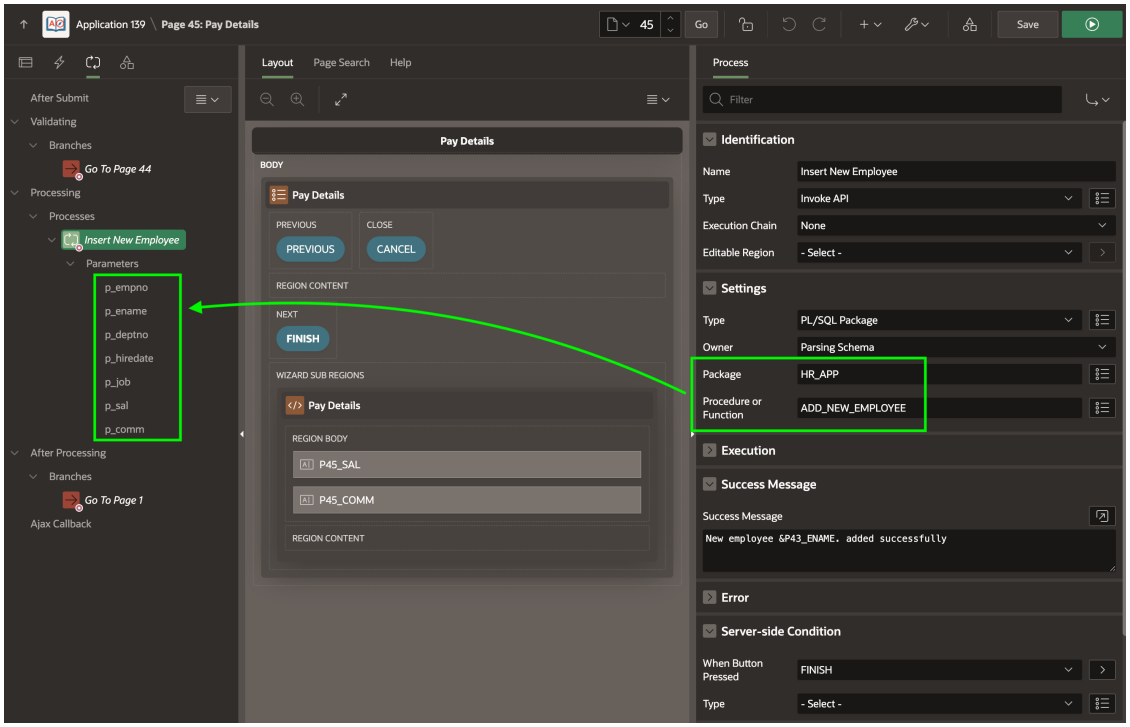

Copyright © 2026, Oracle and/or its affiliates.

```
        p_empno    in emp.empno%type,
        p_ename    in emp.ename%type,
        p_deptno   in emp.deptno%type,
        p_hiredate in emp.hiredate%type,
        p_job      in emp.job%type,
        p_sal      in emp.sal%type,
        p_comm     in emp.comm%type)
    is
    begin
        insert into emp(empno,
                        ename,
                        deptno,
                        hiredate,
                        job,
                        sal,
                        comm)
                values (p_empno,
                        p_ename,
                        p_deptno,
                        p_hiredate,
                        p_job,
                        p_sal,
                        p_comm);
    end add_new_employee;
end hr_app;
```

Then the *Insert New Employee* page process in the *Onboard New Employee* wizard can change to use the *Invoke API* native page process, configuring the package and procedure name as shown below. After setting the *Package* and *Procedure* name, the *Processing* tab updates to show the procedure's parameters under the *Parameters* section of the page process.

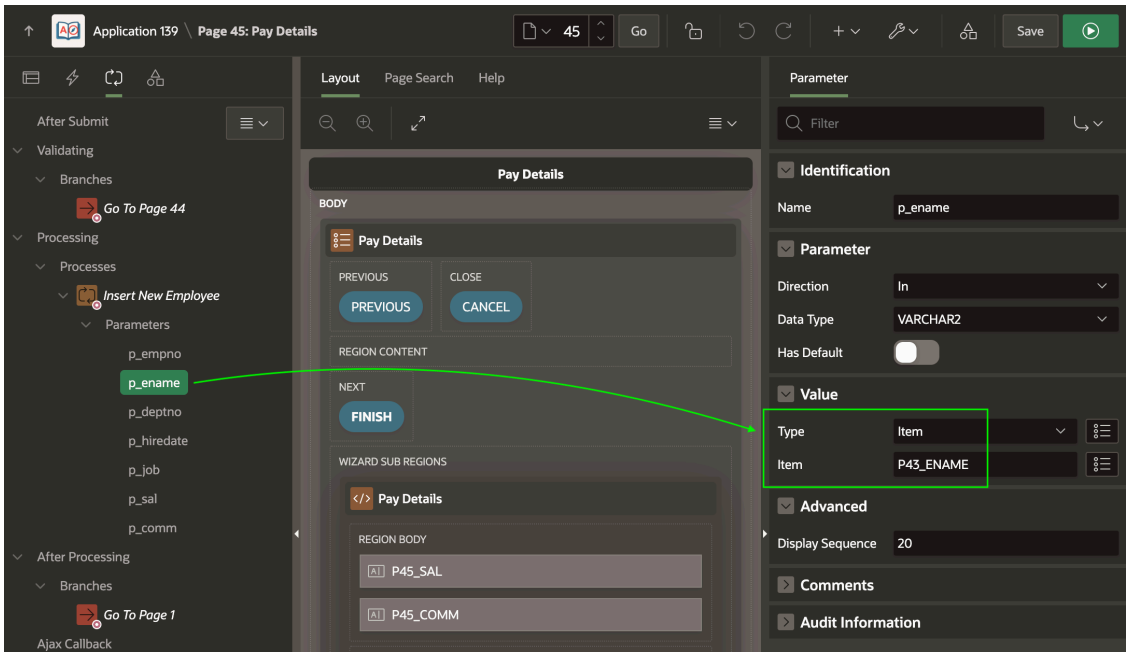
Copyright © 2026, Oracle and/or its affiliates.

Figure 8-11 Configuring Invoke API Process to Call Package Business Logic



Where possible, Page Designer automatically maps matching page item names in the current page to provide appropriate parameter values. You can inspect and configure any of the parameters using the Property Editor as shown below.

Figure 8-12 Configuring Parameter Values for Invoke API



Copyright © 2026, Oracle and/or its affiliates.

8.3.5.6 Making Almost Any Element Conditional

Add a Server-side Condition to almost any element in your APEX app to control whether APEX renders or runs it.

After configuring this rule on a component or page element, the APEX engine includes or executes it only when the test evaluates to true. Choose a declarative option, a PL/SQL expression, a Boolean returned by a PL/SQL function body, or a SQL statement that returns a row. You can apply the check to many elements, including regions, page items, columns, buttons, validations, computations, processes, and branches.

The *When Button Pressed* property of validations, processes, and branches limits its execution to one button on the page. To run the element for *multiple* buttons, use the declarative condition *Request is contained in value* and provide a value that includes a delimited list of button names (e.g. `SAVE,UPDATE,INCREMENT`).

This technique works because when a button named `SOME_BUTTON` triggers a page submit, the APEX engine sets the value of the `REQUEST` substitution string to the name of the button. Since you can reference substitution strings as bind variables in SQL or PL/SQL, you can also express this multiple buttons condition using a PL/SQL expression like:

```
:REQUEST in ('SAVE','UPDATE','INCREMENT')
```

Tip

Use the `:REQUEST` expression with an `IN` clause when a longer name of a page button (e.g. `SAVE_ADDRESS`) includes the *shorter* name of another button (e.g. `SAVE`). This avoids unintentionally running logic when the `SAVE` button is pressed, that happens to be part of the other buttons' names.

```
:REQUEST in ('SAVE_ADDRESS','SAVE_ASSIGNMENT') /* SAVE alone doesn't
match */
```

For more information, see Available Conditions in *Oracle APEX App Builder User's Guide*.

8.4 Additional Page Flow Features

In addition to getting rendered and submitted, APEX pages also uses background data requests and support redirects to navigate without submitting data. Breadcrumbs help visualize the navigation path of drill down page flows. Modal dialogs offer an alternative to present related information without losing the context of the calling page.

- Understanding AJAX and Redirect
  In addition to the page render and submit interactions with the APEX engine, your page can also run business logic on the server and refresh regions to show the latest data without re-rendering the entire page.
- Breadcrumbs Visualize Drilldown Path
  Users often drill down from a list row to view or edit details. If that page shows related records, they can keep following those relationships. *Breadcrumbs* show their place in the flow and let them return to higher levels.

Copyright © 2026, Oracle and/or its affiliates.

- Modal Dialogs and Drawers
  A modal dialog appears centered above the page that opens it. By default, it sizes to its content, but you can set its width and height or make it fill the available space. Configured as a drawer, it slides in from the start or end edge.

8.4.1 Understanding AJAX and Redirect

In addition to the page render and submit interactions with the APEX engine, your page can also run business logic on the server and refresh regions to show the latest data without re-rendering the entire page.

This involves a web application technique called an "AJAX" request, where your page quietly talks to the server in the background to get or send information without refreshing the whole page. APEX makes the process a no code experience. When appropriate, your page can also navigate to another page *without submitting user changes* to the server. This is known as redirecting to another page, instead of submitting and branching to another page.

As shown below, during an AJAX request APEX sends data from the page to the server and it receives a response containing data to update the page. Two common features that use AJAX requests are executing server-side business logic and refreshing a region. APEX handles the lower-level communications with the server while you focus on what your application needs to accomplish.

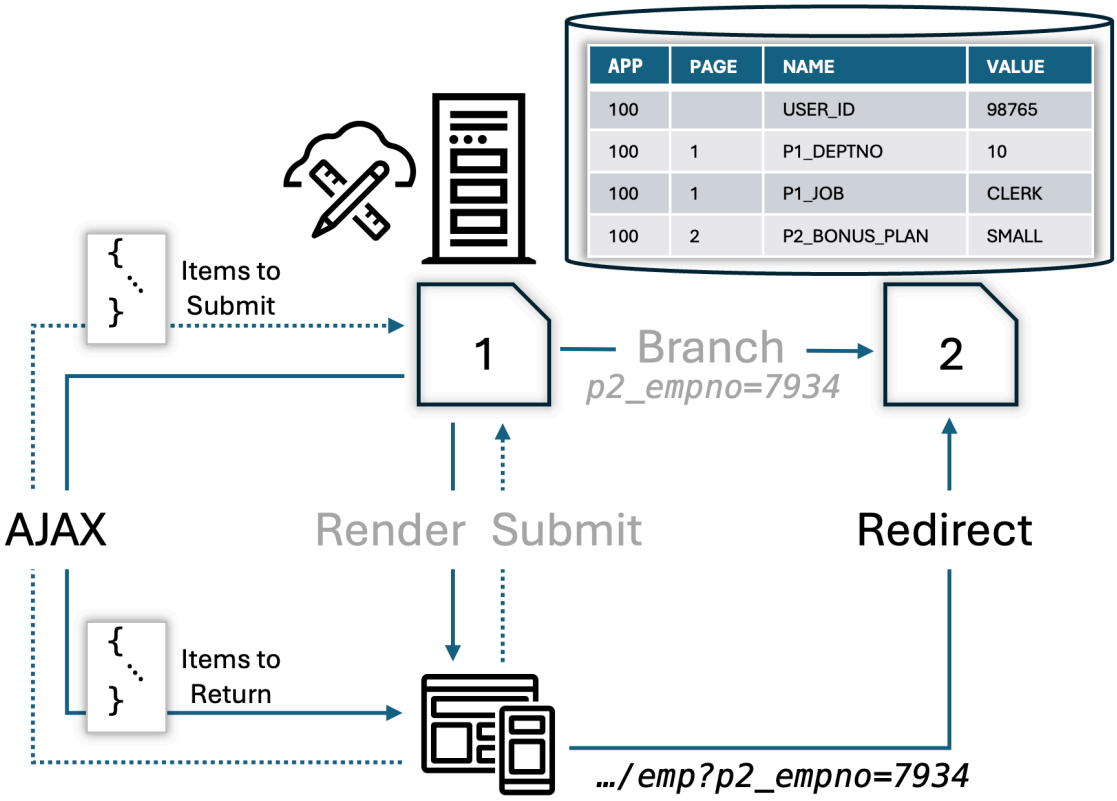
The *Execute Server-side Code* dynamic action lets you invoke server-side PL/SQL or JavaScript. You configure the ***Items to Submit*** and ***Items to Return*** properties that inform APEX what to include in the data sent to the server and what to return back to the page. You author only the application-specific logic that *only you* could write.

When you use a *Refresh* dynamic action to make a region show the latest data, APEX performs an AJAX request to do the job. In this case, the region's ***Page Items to Submit*** property determines the data to send. The response includes the HTML markup and data for the partial page update that APEX applies automatically to refresh just that area of the page.

When one of your pages *redirects* to another, it can pass along parameter values like a branch does. The key difference is that any end user data changes on the current page are ***not submitted*** to the server, so they ***do not update*** user session state values. By default, if there are unsaved changes on a page, the user sees a warning before your app redirects to another page. You can control this warning using the *Warn on Unsaved Changes* property at page level and on individual page items as needed. For example, if a page item is used as a query-by-example filter and you've configured it to have session state storage of *Per Request (Memory Only)*, then the fact that the end user changed the filter value does not matter. The diagram below illustrates the difference between the default render, submit, and branch lifecycle and the AJAX and Redirect mechanisms. The database table in the figure shows the session state values the APEX engine manages automatically.

Copyright © 2026, Oracle and/or its affiliates.

Figure 8-13 AJAX Requests and Redirects Complement Render, Submit, and Branch



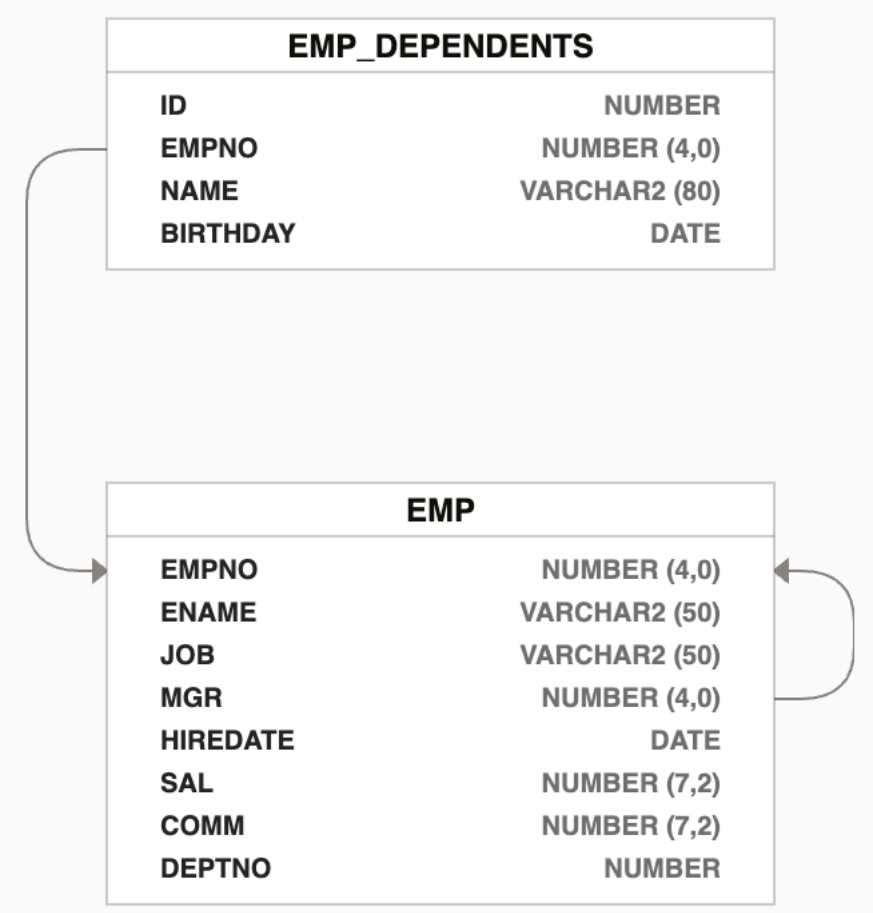
8.4.2 Breadcrumbs Visualize Drilldown Path

Users often drill down from a list row to view or edit details. If that page shows related records, they can keep following those relationships. *Breadcrumbs* show their place in the flow and let them return to higher levels.

Consider the data model shown below with employees having one or more dependents. The diagram shows the familiar `EMP` table and a related `EMP_DEPENDENTS` table that stores the name and birthday of an employee's family members.

Copyright © 2026, Oracle and/or its affiliates.

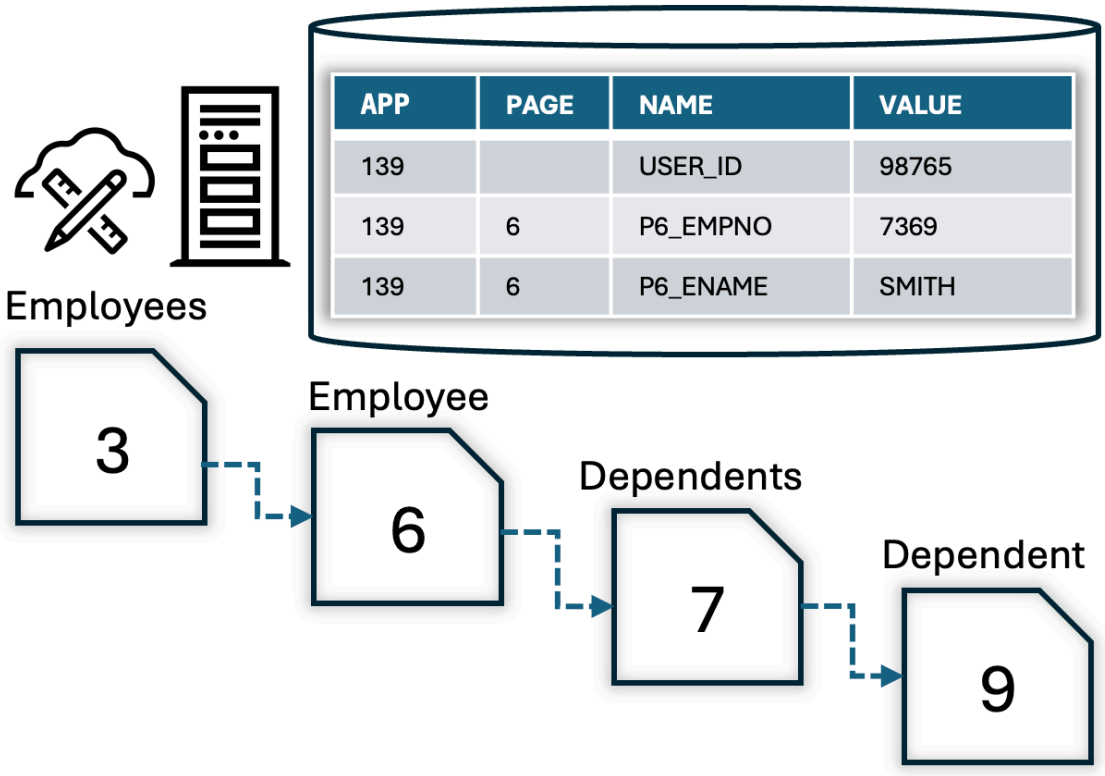
Figure 8-14 Data Model for Employees Having One or More Dependents



The figure below shows how four pages are linked by redirects to create a multi-level drilldown page flow. The user starts in the *Manage Employees* interactive report page 3. Then they can drill down to view the details of a particular employee like *SMITH* in *Employee* form page 6. There they can click a *(Dependents)* button to redirect to the list of *SMITH*'s dependents in the *Dependents* page 7. Finally, the user can drill down to view the details of one of *SMITH*'s children *Janie* in the *Dependent* form page 9.

Copyright © 2026, Oracle and/or its affiliates.

Figure 8-15 Four-Level Drill Down Page Flow for Employees and Dependents

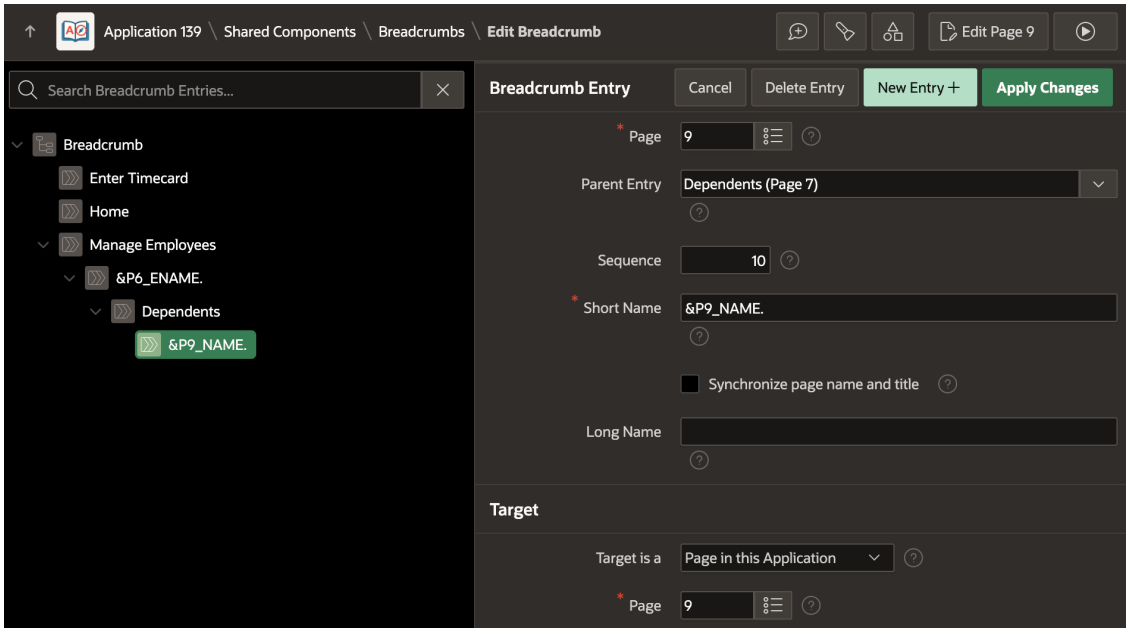


Every page includes a *breadcrumb* region that is based on the same *breadcrumb* shared component. It defines a hierarchy of page *breadcrumb entries* that the breadcrumb region uses to render the navigation path based on the current page in the hierarchy. As shown below, the breadcrumb entries reflect the four level page hierarchy using the *Parent Entry* setting in each entry definition. Notice an entry like *Employee* page 6's can reference `&P6_ENAME.` to include the employee name in the breadcrumb short name, and the *Dependent* form page 9 uses `&P9_NAME.` for the dependent name.

For the `P6_ENAME` to render correctly for subsequent pages in the flow, it's necessary to configure its storage to be *Per Session (Persistent)*. This is also how the `P6_EMPNO` is configured to ensure navigating back to page six uses the right employee number value. The diagram above reflects these session state values in the database table.

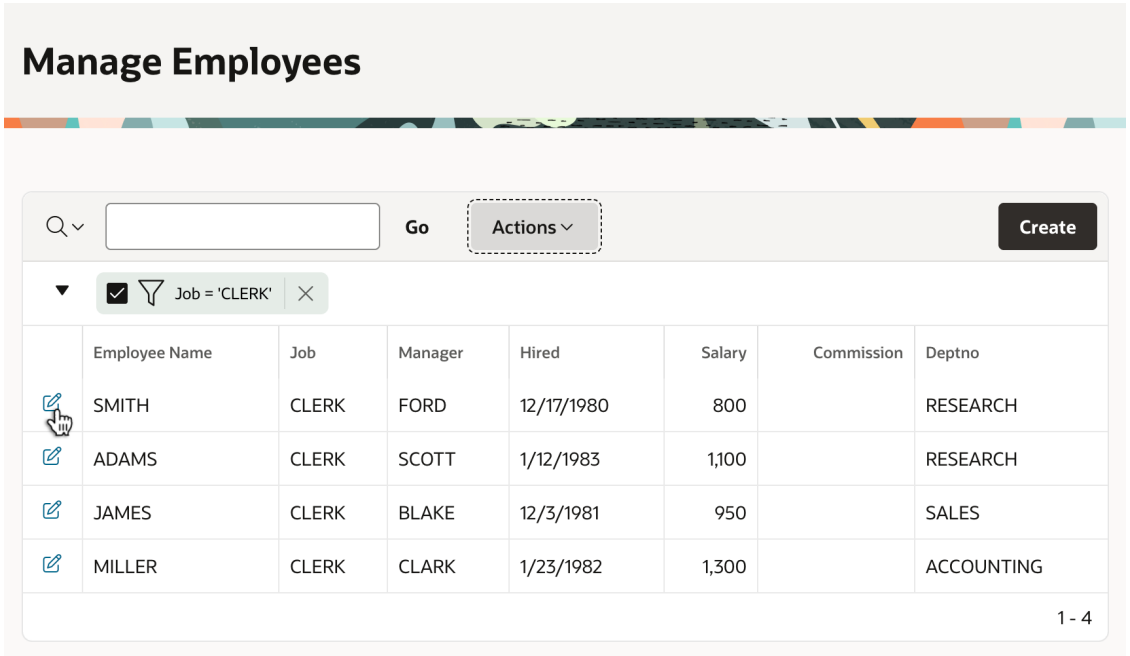
Copyright © 2026, Oracle and/or its affiliates.

Figure 8-16 Editing Breadcrumbs Shared Component to Reflect Navigation Hierarchy



At runtime, the end user selects *Manage Employees* from the navigation menu and sees page 3 as shown below. Notice the breadcrumb region renders the current page's breadcrumb short name *Manage Employees* in a large font. The interactive report region shows the clerks the user has filtered. Employees *SMITH* is one of these.

Figure 8-17 Manage Employees Page Redirects to Employee Form Page for Edit

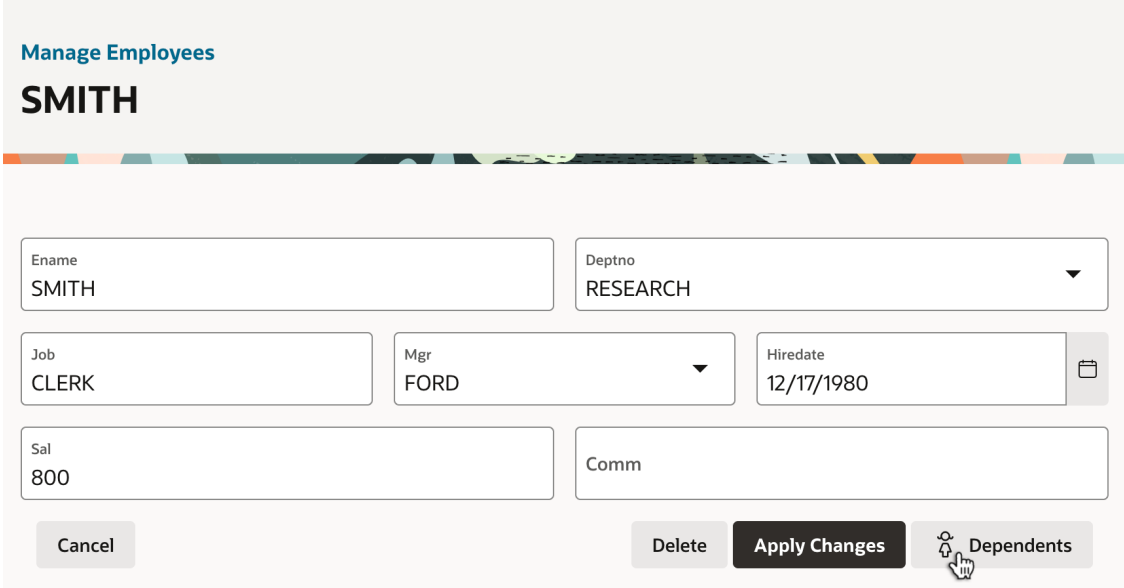


If the user clicks on the edit icon next to *SMITH*, they drill down to the *Employee* form page 6. The breadcrumb region renders a link for the *Manage Employees* page in a smaller font, and

Copyright © 2026, Oracle and/or its affiliates.

the current page's breadcrumb entry in a larger font. Recall that page 6's breadcrumb short name was `&P6_ENAME.` so this substitution renders as *SMITH* in the breadcrumb.

Figure 8-18 Drilling Down on Employee to See Their Dependents

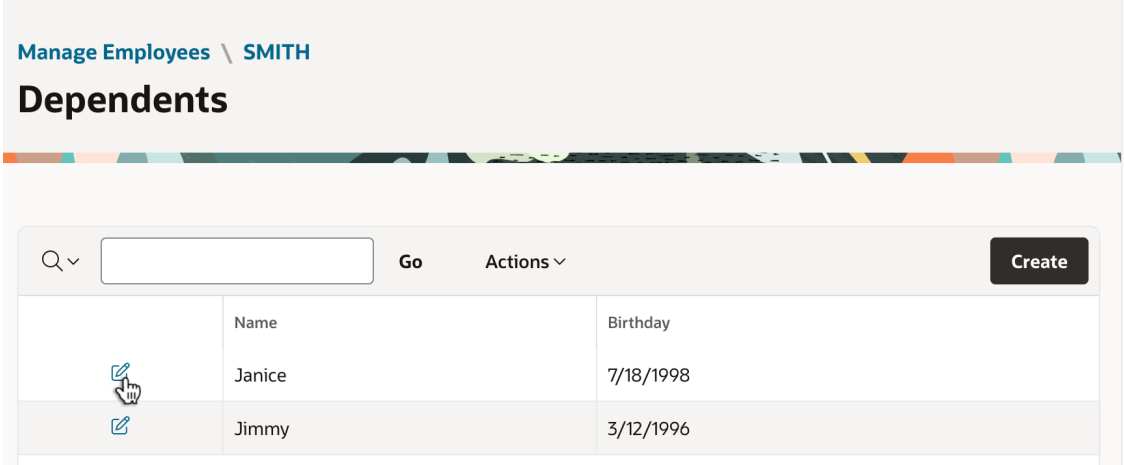


If the user clicks on the *(Dependents)* button for employee *SMITH* they drill down to the *Dependents* page 7. Since `P6_EMPNO` is in session state, and the user only visits page 7 after first going to page 6, you can reference the current employee number in the interactive report's `WHERE` clause:

```
EMPNO = :P6_EMPNO
```

This ensures the page only shows the dependents for the current employee being edited. Notice in the fugure below the breadcrumb now shows two levels of links to navigate back to, and again shows the *Dependents* current page breadcrumb shortname in a larger font. The table shows *SMITH*'s dependents *Janice* and *Jimmy* with their birthdays.

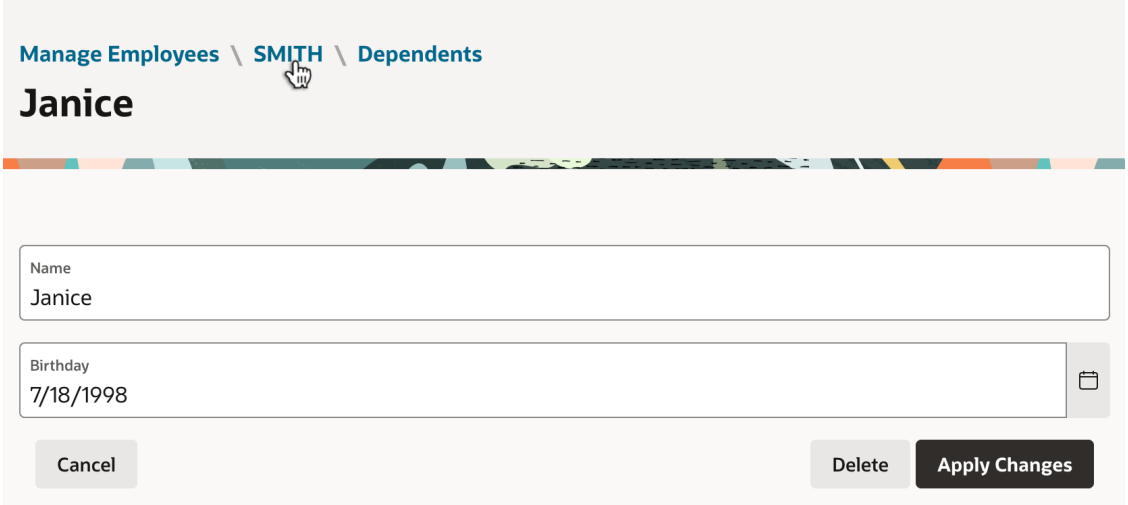
Figure 8-19 Viewing Employee SMITH's Dependents and Drilling Down Again



Copyright © 2026, Oracle and/or its affiliates.

Finally, when the user clicks the edit icon next to *SMITH*'s dependent *Janice*, they end up in the *Dependent* page 9. The *Dependents* page passes the `ID` value of the current dependent to the edit form. Notice the breadcrumb region now shows three levels of navigation links. The user can click on any of the links to navigate back to a previous level of the drill down page flow.

Figure 8-20 Viewing SMITH's Dependent Janie: Breadcrumb Shows Path "Home"



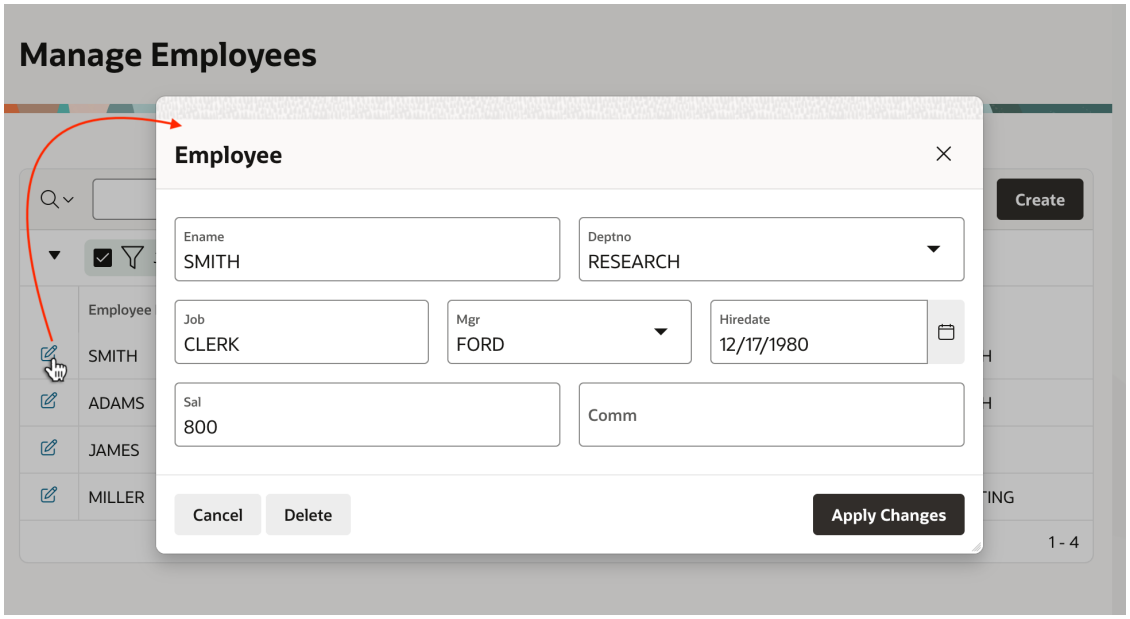
8.4.3 Modal Dialogs and Drawers

A modal dialog appears centered above the page that opens it. By default, it sizes to its content, but you can set its width and height or make it fill the available space. Configured as a drawer, it slides in from the start or end edge.

You create a modal dialog page by setting its *Page Mode* property to *Modal Dialog*. While it displays, the user can't interact with the calling page below. When the user cancels or closes the dialog, it disappears and the calling page becomes active again. The figure below shows an *Employee* edit page as a modal dialog above an *Employees* interactive report page.

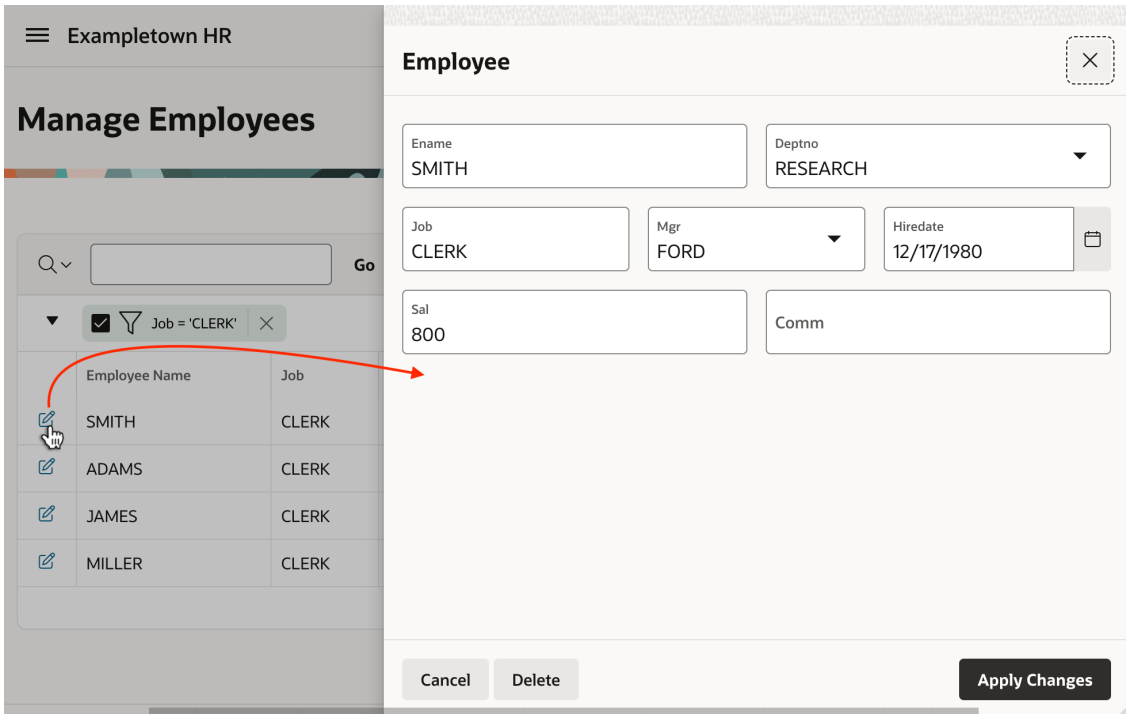
Copyright © 2026, Oracle and/or its affiliates.

Figure 8-21 Modal Dialog Form Page Called from a List Page



If you set the page's *Dialog Template* property to *Drawer*, you opt for a different behavior. The drawer shown below is an alternative way to present a modal page. It occupies the full height of the window and slides out from either the start or end edge of the window.

Figure 8-22 Modal Drawer Slides Out from the Start or End of the Window



In either case, you can close the dialog page using the *Close Dialog* dynamic action or the *Close Dialog* page process. These close dialog methods can optionally return one or more

Copyright © 2026, Oracle and/or its affiliates.

page item values to the calling page. A *Dialog Closed* dynamic action event handler can reference these *Dialog Return Items* back in the calling page. You can cancel the dialog using the *Cancel* dynamic action.

If your modal dialog branches or redirects to another modal dialog of the same kind, set the *Chained* property to *On* to have the called page *reuse* the current modal dialog's window. Setting this property to *Off* causes the called page to open in another modal dialog window on top of the current modal dialog.

8.5 Understanding Session State Management

When an end user submits a page, sometimes you save their input to application tables immediately. Other times, to improve usability or manage file uploads, you need to store values *temporarily* and save them later.

The APEX engine manages this temporary storage across each user session, including application values, value collections, and uploaded files. These stored values are collectively known as "session state".

- Use Cases for Temporary Storage
  Use temporary storage for multi-page tasks, query-by-example searches, temporary data collections, and file uploads.
- Session Context for Temporary Values
  A *user session* is the span of time during which a particular end user interacts with your web application.
- How APEX Manages User Session State
  For the *Onboard New Employee* wizard page flow shown below, you only want to insert the new employee row in the `EMP` table when the user clicks *(Finish)* on the last page.
- User-Specific Temporary Storage
  To store a page item's value as a user preference that spans *multiple* sessions, set its *Session State* > *Storage* property to *Per User (Persistent)*.
- Clearing Session State
  The APEX engine clears session state values for a user session when it ends.
- Application Items
  An *application item* is a named value holder that persists for the duration of the user session.
- Using Temporary Collections
  Collections let APEX store one or more rows of temporary information. At runtime, you create a named collection with the `APEX_COLLECTION` package. A session can contain multiple collections, and APEX clears them automatically when the session ends.
- Temporary Files
  Both the APEX *File Upload* and *Image Upload* page items provide a *Storage Type* setting of *Table APEX_APPLICATION_TEMP_FILES*. Both support uploading either single or multiple files.
- Inspecting Session State for Debugging
  When running your app from App Builder, the developer toolbar appears along the edge of the browser window and offers options useful for debugging.

8.5.1 Use Cases for Temporary Storage

Use temporary storage for multi-page tasks, query-by-example searches, temporary data collections, and file uploads.

Copyright © 2026, Oracle and/or its affiliates.

Four key use cases that require temporary storage include:

- ***Tasks Involving Multiple Pages***
  User-entered values need to "hang around" while a user visits multiple pages to work on a task. Your app saves these temporary values only once the user confirms they have completed the job. A three-page *Onboard New Employee* wizard is a perfect example.
- ***Query-by-Example Search Pages***
  Users may appreciate their filter criteria "staying sticky" across multiple uses of a particular business function during their session. Their criteria values are not meant to ever be saved in an application table, but their temporary storage during the user's session improves usability.
- ***Temporary Data Collections***
  Imagine a shopping cart page flow where the end user repeats a sequence of searching for an item, adding it to their cart, and adjusting the quantities they want to purchase. Until the user clicks *(Check Out)* to make a purchase, the set of product codes and quantities in their cart is a temporary collection. During order creation, your app iterates through the shopping cart items to create the permanent order lines of the new order.
- ***File Uploads***
  In a page flow where a user requests a medical insurance reimbursement, they may need to upload one or more attachments to support their claim. The uploaded files need temporary storage until your business logic can move their contents to appropriate application tables when a task completes.

8.5.2 Session Context for Temporary Values

A *user session* is the span of time during which a particular end user interacts with your web application.

It begins when the user first navigates to a page and logs in, and ends when they sign out or remain idle long enough to trigger a session timeout. At any moment, multiple users may be using your app, each working on different tasks. To streamline the experience for all concurrent users, APEX assigns a unique session ID to each one. This gives the context for each user’s interaction with the app and for the session state the APEX engine manages for each user session.

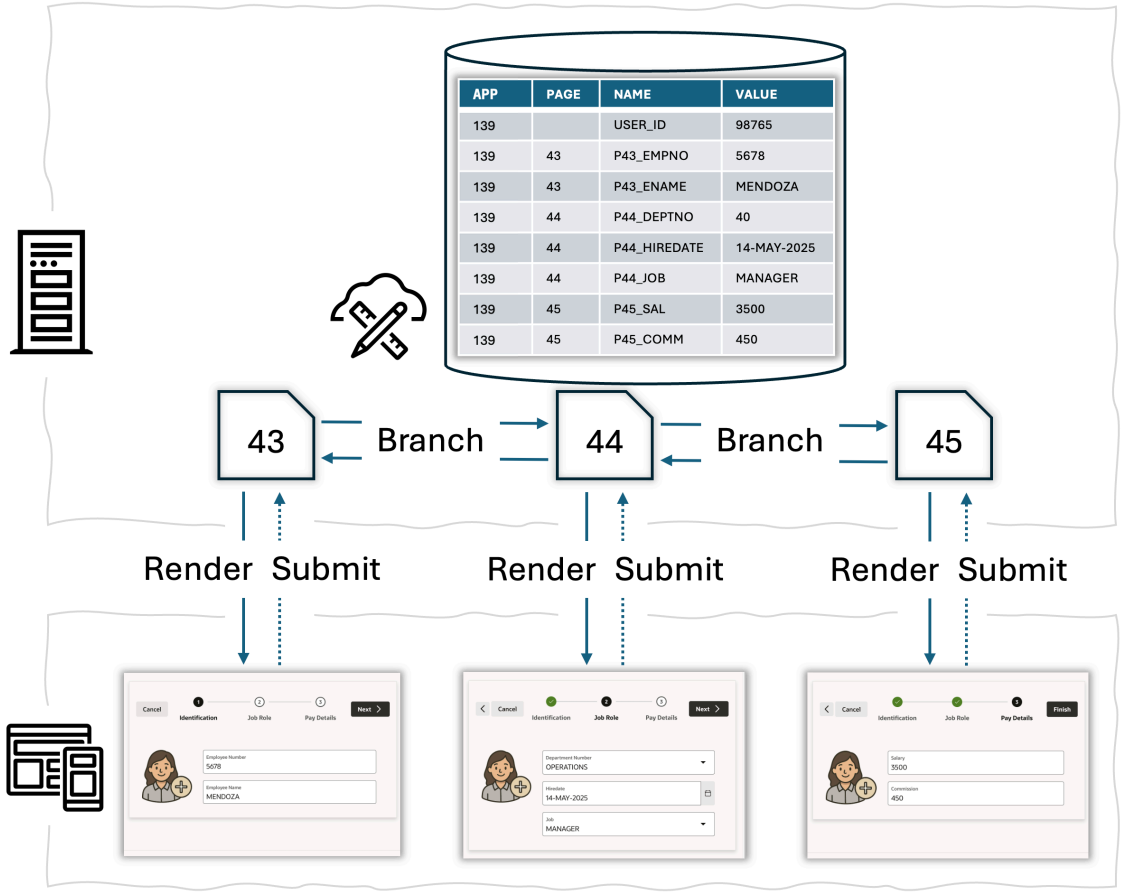
8.5.3 How APEX Manages User Session State

For the *Onboard New Employee* wizard page flow shown below, you only want to insert the new employee row in the `EMP` table when the user clicks *(Finish)* on the last page.

The user expects to click *(Next)* and *(<)* previous between the pages without losing any of their data entry when returning to review previous pages. You can configure the page items in pages 43, 44, and 45 to use APEX's automatic session state management to temporarily store the user's submitted values so *you* can decide when to save them permanently in your application tables. To do this, simply configure its *Session State > Storage* property to *Per Session (Persistent)*. The diagram illustrates the page render, submit, and branch flow of the three-page *Onboard New Employee* wizard flow. The session state values the APEX engine managed for the current user session display in the table, tagged with the application ID, page ID, and item name. For simplicity, the session ID of the current user is not shown in the figure, but all session state values are partitioned and secured based on that context value as well.

Copyright © 2026, Oracle and/or its affiliates.

Figure 8-23 Automatic Session State Management



When the APEX engine renders a page, it automatically uses any page item values from session state before your *Pre-Rendering* computations, processes, or branches execute. Then when rendering is complete, it stores the values of any page items whose *Session State* > *Storage* property is set to *Per Session (Persistent)* into session state.

When the user submits a page, the APEX engine lets your computations, processes, and branches reference the values of *all* page items. When submit processing is complete, the it saves the values of any *Per Session (Persistent)* page items from the current page. This occurs *before* it follows a branch to a new page or re-renders the existing page.

It "forgets" any page items whose *Session State* > *Storage* property is set to *Per Request (Memory Only)*. If your processes do not permanently store these page item values, or pass them along to the next page to render as parameters in the link, then they are no longer available.

The APEX engine stores the application id, page number, page item name, and value for every session state item. It also ensures complete isolation between user sessions so one user session can only see its own session state values.

The figure shows the contents of the user session state after a user has visited each of the three pages in the *Onboard New Employee* wizard and returned back to view previous steps as well. All page items in pages 43, 44, and 45 have *Per Session (Persistent)* storage. When the user clicks *(Finish)* on the last page, the page process related to that button passes the values from the relevant page items in pages 43, 44, and 45 to the `ADD_NEW_EMPLOYEE` procedure that permanently stores the new employee's data in the `EMP` table.

Copyright © 2026, Oracle and/or its affiliates.

8.5.4 User-Specific Temporary Storage

To store a page item's value as a user preference that spans *multiple* sessions, set its *Session State* > *Storage* property to *Per User (Persistent)*.

This is useful for query-by-example page item values or other options you want to remain exactly as the user left them the next time they visit your application. While the `APEX_UTIL` package has the `SET_PREFERENCE` procedure and the `GET_PREFERENCE` function, setting a page item's storage to *Per User (Persistent)* is an even simpler way to enable a user preference.

8.5.5 Clearing Session State

The APEX engine clears session state values for a user session when it ends.

This includes page items, collections, and temporary files. The cleanup happens when the user signs out of the app, or when they are idle long enough to trigger a session timeout. There are also two *declarative* ways to clear selective session state values during the session.

- Clearing State with Link Target Settings
  Anywhere in APEX, when you define a target link to a page you can optionally configure a list of one or more page numbers whose session state values you want to clear.
- Clearing State with a Page Process
  To clear specific session state at the end of your page's submit processing, use the native *Clear Session State* page process.

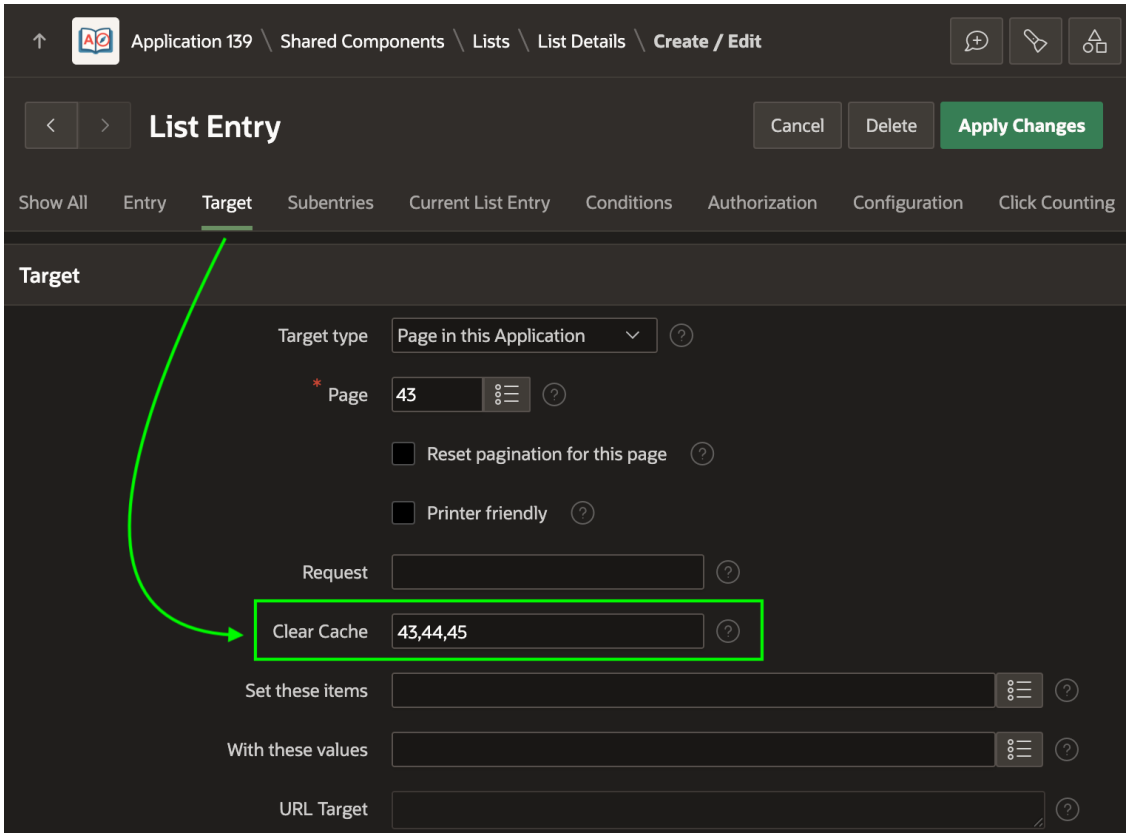
8.5.5.1 Clearing State with Link Target Settings

Anywhere in APEX, when you define a target link to a page you can optionally configure a list of one or more page numbers whose session state values you want to clear.

For example, the figure below shows the *Target* tab of *Onboard New Employee* navigation menu list entry. Notice its *Clear Cache* setting is set to the comma-separated list of page numbers `43,44,45`. In addition to page numbers, you can also list collection names to clear, as well as the special value `RP` to clear report pagination on the listed pages.

Copyright © 2026, Oracle and/or its affiliates.

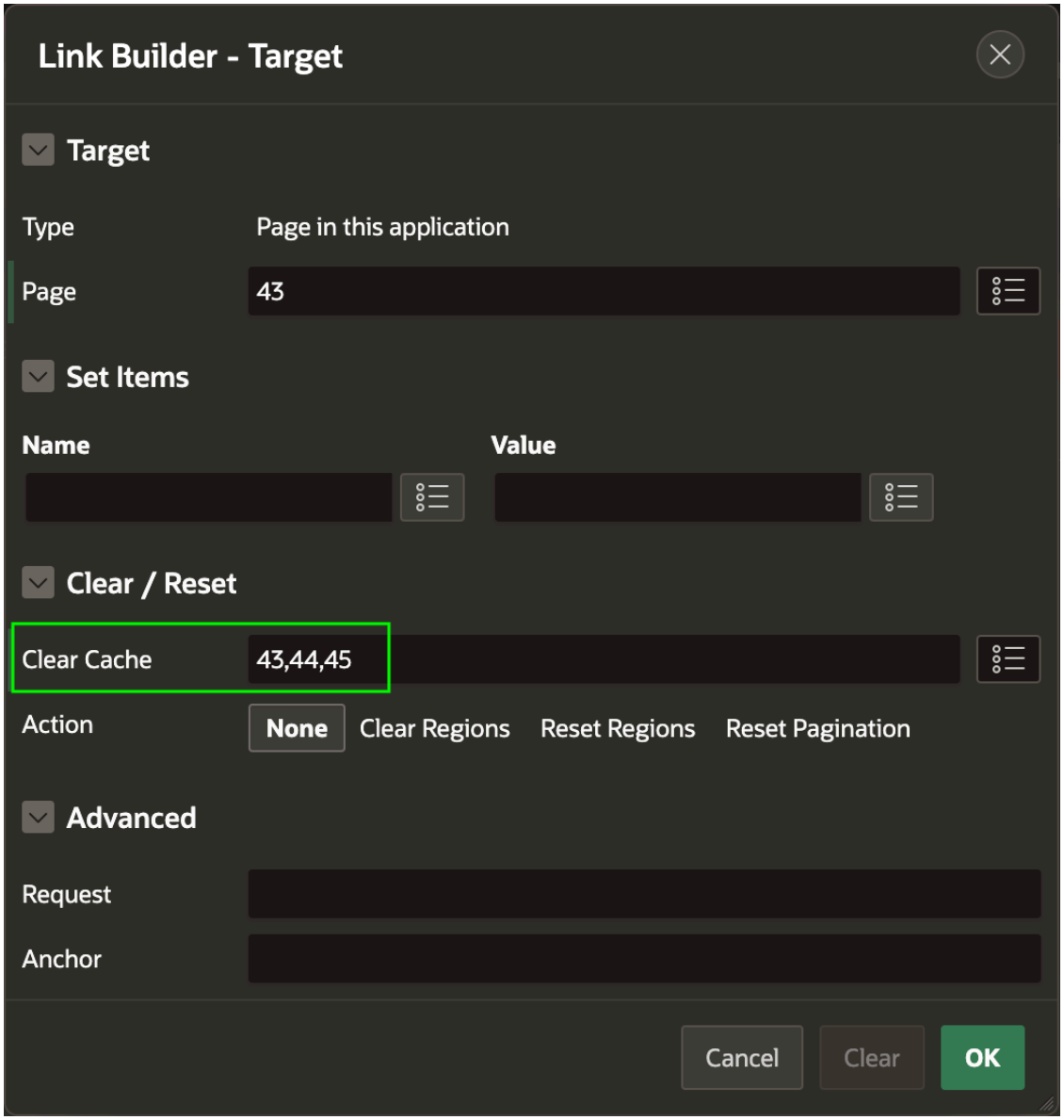
Figure 8-24 Clearing Session State in Link Target Settings



As shown below, the same settings are available in the *Link Builder* dialog when configuring the target of a button or link.

Copyright © 2026, Oracle and/or its affiliates.

Figure 8-25 Clear Cache Setting in Button Target Dialog



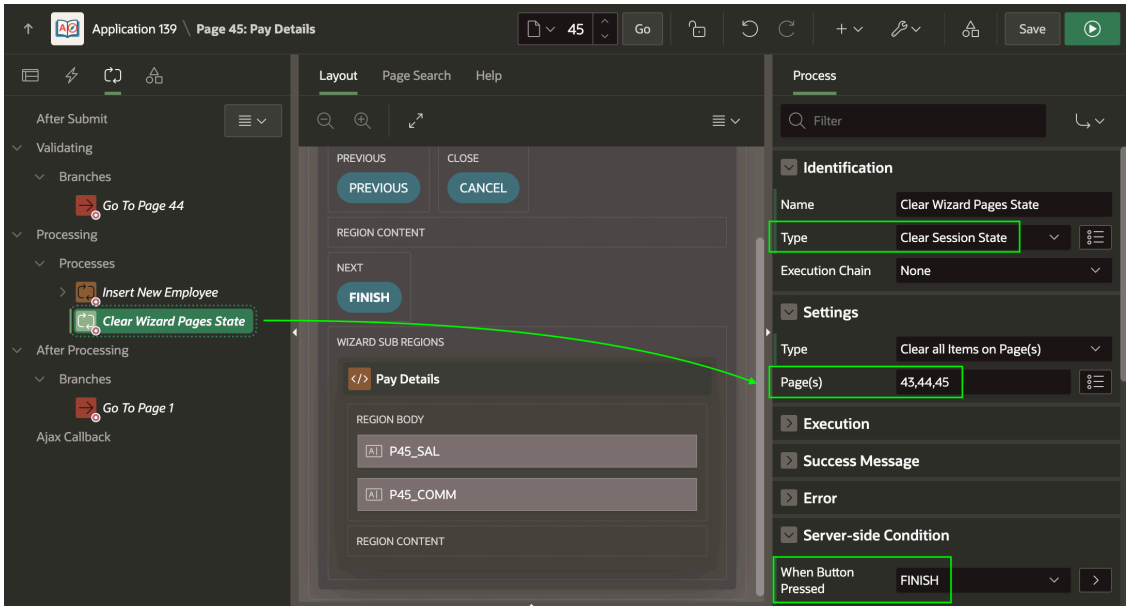
8.5.5.2 Clearing State with a Page Process

To clear specific session state at the end of your page's submit processing, use the native *Clear Session State* page process.

For example, the figure below shows how the last page of the *Onboard New Employee* wizard page flow can conditionally clear the page item session state on pages 43, 44, and 45 when the `FINISH` button is pressed, after it has successfully inserted the new employee. Notice the page process type is set to *Clear Session State*, the page numbers are specified as a comma-separated value, and the Server-side Condition's *When Button Pressed* property is set to `FINISH`.

Copyright © 2026, Oracle and/or its affiliates.

Figure 8-26 Using Clear Session State Page Process



8.5.6 Application Items

An *application item* is a named value holder that persists for the duration of the user session.

You define them in *Shared Components* > *Application Items*. A typical use case is a `USER_ID` item storing the numerical id of the currently logged-in user.

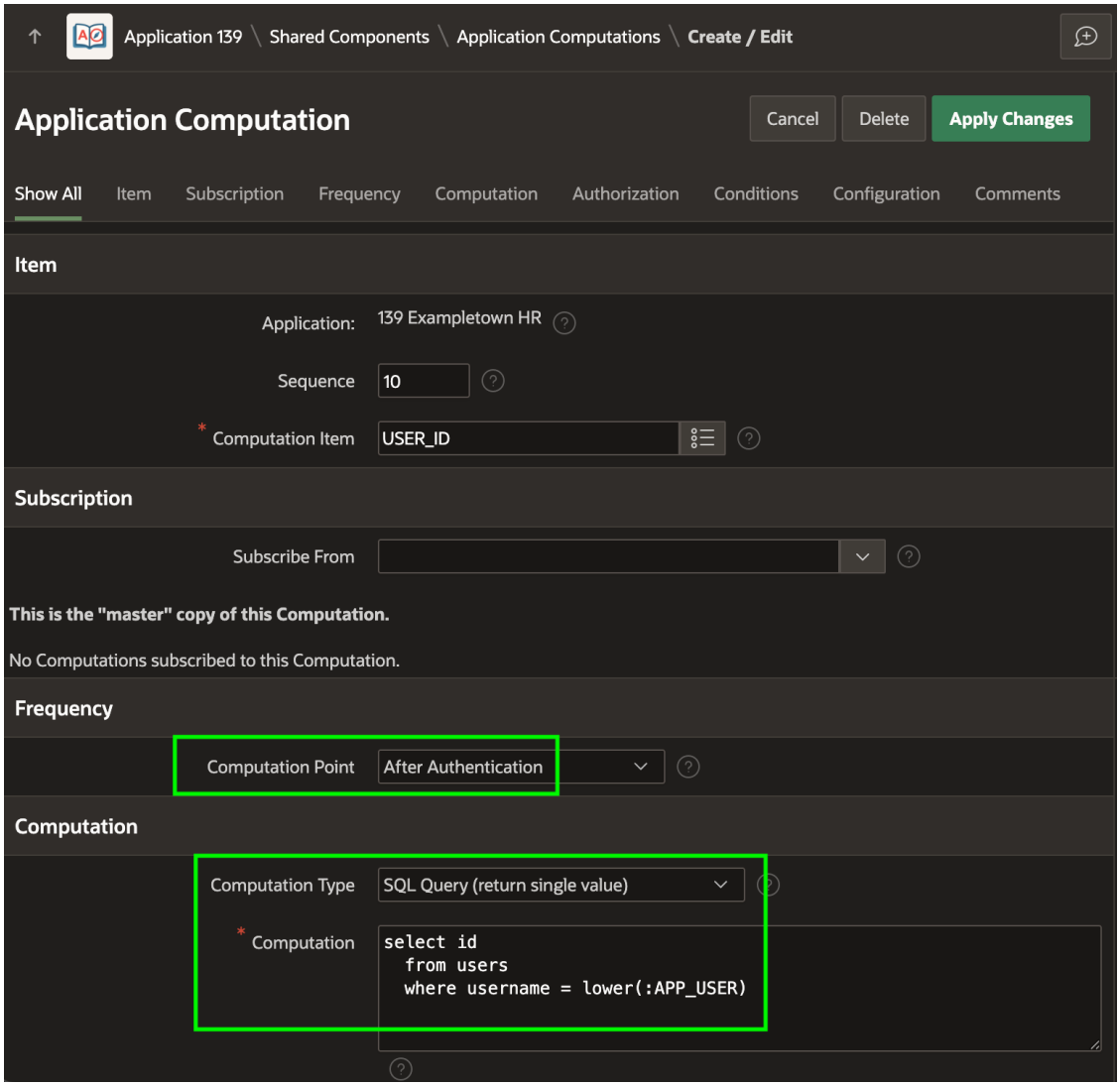
Many enterprise apps have some form of a `USERS` table storing the username, email address and other information about users. Other tables in the app data model reference the `ID` primary key of this `USERS` table. However, users authenticate to the application using their username. To avoid having to lookup the `ID` value from the `USERS` table using the logged-in username over and over throughout the app, define a `USER_ID` application item and lookup its value at login time. Then any page in the application can reference the application item's value using `#USER_ID#`, `&USER_ID.`, or `:USER_ID` depending on the context.

The application item is the value holder. To compute its value after the user authenticates, use a companion *application computation* shared component. As shown below, the app computation references the `USER_ID` app item name, chooses a *Computation Point* of *After Authentication*, and sets a value using one of various supported approaches. Here a SQL query computes the value by referencing the `:APP_USER` bind variable that provides the logged-in username:

```
select id
  from users
 where username = lower(:APP_USER)
```


Copyright © 2026, Oracle and/or its affiliates.

Figure 8-27 App Computation Sets the Value of App Item After Authentication



Any SQL or PL/SQL throughout your app can reference the value of an application item using an `:ITEM_NAME` bind variable and PL/SQL code you write in a page can assign a new value to it. To assign a new value from PL/SQL functions or procedures, use the `SET_VALUE` procedure in the `APEX_SESSION_STATE` package.

8.5.7 Using Temporary Collections

Collections let APEX store one or more rows of temporary information. At runtime, you create a named collection with the `APEX_COLLECTION` package. A session can contain multiple collections, and APEX clears them automatically when the session ends.

- Key Collections Concepts
  A collection requires no formal definition. Instead, you use its predefined generic columns, such as `N001`, `N002`, `C001`, `C002`, and `D001`, according to the number and type of values you need to store.

Copyright © 2026, Oracle and/or its affiliates.

- Tips for Maintainable Collection Use
  For additional ease of maintenance, create a view for the collection query so your pages can work with the meaningful column names anywhere they need to.

8.5.7.1 Key Collections Concepts

A collection requires no formal definition. Instead, you use its predefined generic columns, such as `N001`, `N002`, `C001`, `C002`, and `D001`, according to the number and type of values you need to store.

Collections support a fixed number of typed columns. For a given collection, just use the columns you need in each row.

Table 8-1 Collection Columns

Data Type	Column Name(s)	Quantity
VARCHAR2	c001 to c050	50
NUMBER	n001 to n005	5
DATE	d001 to d005	5
CLOB	clob001	1
BLOB	blob001	1
XMLTYPE	xmltype001	1

You add rows – also known as *members* – to a collection using the `ADD_MEMBER` procedure or function. Each member is assigned a sequence id (`SEQ_ID`) that is one larger than the current maximum sequence id in the collection. Using that sequence id, you can delete a member or update a member using `DELETE_MEMBER` or `UPDATE_MEMBER` procedures, respectively.

To access the data in a collection, you query the `APEX_COLLECTIONS` view using a `WHERE` clause of:

```
WHERE collection_name = 'COLLECTION_NAME'
```

For example, imagine creating a collection named `CART` to represent a shopping cart. You need to store:

- Item code (`VARCHAR2`)
- Quantity (`NUMBER`)
- Need by date (`DATE`)

To create the collection, use the statement:

```
apex_collection.create_collection(
   p_collection_name => 'SHOPPING_CART');
```

You can use any column of appropriate data type, but by convention you use them in numerical order. So, you'll use column `c001` to store the Item code, `n001` to store the quantity, and `d001` to store the date. Therefore, to add two items to the cart, use code like this:

```
apex_collection.add_member(
   p_c001 => 'COMP-APPL-MBP-16',
   p_n001 => 2,
```


Copyright © 2026, Oracle and/or its affiliates.

```
   p_d001 => date'2025-05-27');
apex_collection.add_member(
   p_c001 => 'ACC-APPL-MAGICMOUSE',
   p_n001 => 1,
   p_d001 => date'2025-05-27');
```

To retrieve the members in the collection, write a query like the following. It is helpful to alias the columns to more meaningful names as follows:

```
select c001 as item_code,
       n001 as quantity,
       d001 as need_by_date
  from apex_collections
 where collection_name = 'SHOPPING_CART'
```

APEX automatically isolates the session state of one user session from another, so you can write the query knowing that you will only ever see the data the current user session has added to the collection named `SHOPPING_CART`. You can use a query like this in any APEX region to present collection data in any way you might need to.

You can also use a query over `APEX_COLLECTIONS` in a PL/SQL cursor `for` loop to iterate over the contents of the collection like this:

```
for i in (select c001 as item_code,
                 n001 as quantity,
                 d001 as need_by_date
            from apex_collections
           where collection_name = 'SHOPPING_CART')
loop
   -- Reference i.item_code, i.quantity,
   -- and i.need_by_date in the loop
end loop;
```

To proactively clear a collection before the session ends, use the `TRUNCATE_COLLECTION` procedure. Consult the APEX PL/SQL reference documentation for the `APEX_COLLECTION` package for other useful functions and procedures for working with collections.

> **Tip**
>
> If you combine an editable grid with a collection, you will need to make two modifications to the typical no-code setup. First, since the grid depends on a primary key value that does not change you will need to populate one of the collection columns with a unique key. Consider generating a `SYS_GUID` value as the primary key for each row as the package body in the following section does.

8.5.7.2 Tips for Maintainable Collection Use

For additional ease of maintenance, create a view for the collection query so your pages can work with the meaningful column names anywhere they need to.

Copyright © 2026, Oracle and/or its affiliates.

For example, you can create the following view. Notice that in addition to `item_code`, `quantity`, and `need_by_date`, it is also storing a unique `id` value that collection use with an interactive grid requires:

```
create view shopping_cart_v as
select c001 as item_code,
       c002 as id,
       n001 as quantity,
       d001 as need_by_date
  from apex_collections
 where collection_name = 'SHOPPING_CART'
```

You can use this view in a PL/SQL cursor `for` loop to iterate over the contents of the collection like this:

```
for i in (select item_code,
                 id,
                 quantity,
                 need_by_date
            from shopping_cart_v)
loop
   -- Reference i.item_code, i.id,
   -- i.quantity, and i.need_by_date in the loop
end loop;
```

For ultimate maintainability, consider writing a PL/SQL package API that isolates the interaction with the `APEX_COLLECTION` package behind a set of functions and procedures that describe the business functionality it provides. For example, consider the package:

```
create or replace package shopping_cart_api is
    ------------------------------------------------------------
    procedure clear_cart;
    ------------------------------------------------------------
    function add_item(
        p_item_code    in varchar2,
        p_quantity     in number,
        p_need_by_date in date)
    return varchar2;
    ------------------------------------------------------------
    procedure remove_item (
        p_item_id in varchar2);
    ------------------------------------------------------------
    procedure update_item (
        p_item_id      in varchar2,
        p_item_code    in varchar2,
        p_quantity     in number,
        p_need_by_date in date);
end shopping_cart_api;
```

Using a package specification like this, your application code is easier to understand for yourself and any teammates who can immediately infer that it's working with a shopping cart.

Copyright © 2026, Oracle and/or its affiliates.

The corresponding package body for the SHOPPING_CART_API package appears below. Notice how it's generating a unique SYS_GUID value to provide a primary key that does not change for each row in the collection:

```
create or replace package body shopping_cart_api is
    c_collection_name constant varchar2(40) := 'SHOPPING_CART';
    ----------------------------------------------------------
    procedure ensure_collection
    is
    begin
        if not apex_collection.collection_exists(c_collection_name) then
            apex_collection.create_collection(c_collection_name);
        end if;
    end ensure_collection;
    ----------------------------------------------------------
    procedure clear_cart
    is
    begin
        ensure_collection;
        apex_collection.truncate_collection(c_collection_name);
    end clear_cart;
    ----------------------------------------------------------
    function get_seq_id(
        p_item_id in varchar2)
    return number
    is
        l_ret number;
    begin
        for j in (select seq_id as seq
                    from apex_collections
                   where collection_name = c_collection_name
                     and c002 = p_item_id)
        loop
            l_ret := j.seq;
        end loop;
        return l_ret;
    end;
    ----------------------------------------------------------
    function add_item (
        p_item_code    in varchar2,
        p_quantity     in number,
        p_need_by_date in date)
    return varchar2
    is
        l_ret varchar2(255) := sys_guid();
    begin
        ensure_collection;
        apex_collection.add_member (
            p_collection_name    => c_collection_name,
            p_c001               => p_item_code,
            p_c002               => l_ret,
            p_n001               => p_quantity,
            p_d001               => p_need_by_date
        );
        return l_ret;
    end add_item;
```


Copyright © 2026, Oracle and/or its affiliates.

```
    ------------------------------------------------------------
    procedure remove_item (
        p_item_id in varchar2)
    is
        l_seq_id number;
    begin
        ensure_collection;
        l_seq_id := get_seq_id(p_item_id);
        if l_seq_id is not null then
            apex_collection.delete_member (
                p_collection_name => c_collection_name,
                p_seq             => l_seq_id);
        end if;
    end remove_item;
    ------------------------------------------------------------
    procedure update_item (
        p_item_id      in varchar2,
        p_item_code    in varchar2,
        p_quantity     in number,
        p_need_by_date in date)
    is
        l_seq_id number;
    begin
        ensure_collection;
        l_seq_id := get_seq_id(p_item_id);
        if l_seq_id is not null then
            apex_collection.update_member(
                p_collection_name => c_collection_name,
                p_seq             => l_seq_id,
                p_c001            => p_item_code,
                p_c002            => p_item_id,
                p_n001            => p_quantity,
                p_d001            => p_need_by_date);
        end if;
    end update_item;
end shopping_cart_api;
```

8.5.8 Temporary Files

Both the APEX *File Upload* and *Image Upload* page items provide a *Storage Type* setting of *Table APEX_APPLICATION_TEMP_FILES*. Both support uploading either single or multiple files.

When you configure their *Purge File at* setting to *End of Session*, then the uploaded files remain queryable in the `APEX_APPLICATION_TEMP_FILES` table for the duration of the session. Only files pertaining to the current session appear in the table.

The value of the `NAME` column uniquely identifies each temporary file. This is *different* from the `FILENAME` column in the table that reflects the actual name of the file. For a *File Upload* or *Image Upload* item that supports a *single* file, you can query the uploaded filename, MIME type, and binary contents with a query like:

```
select blob_content,
       mime_type,
       filename
```


Copyright © 2026, Oracle and/or its affiliates.

```
  from apex_application_temp_files
where name = :P1_SINGLE_FILE_UPLOAD
```

In contrast, if the page item allows multiple files, then its value will be a colon-separated list of unique uploaded file names. So the query needs to handle that list like this:

```
select blob_content,
       mime_type,
       filename
  from apex_application_temp_files
where name in (select column_value
                 from apex_string.split('P1_MULTI_FILE_UPLOAD',':'))
```

You can proactively clear the table using a regular SQL statement like:

```
DELETE FROM APEX_APPLICATION_TEMP_FILES
```

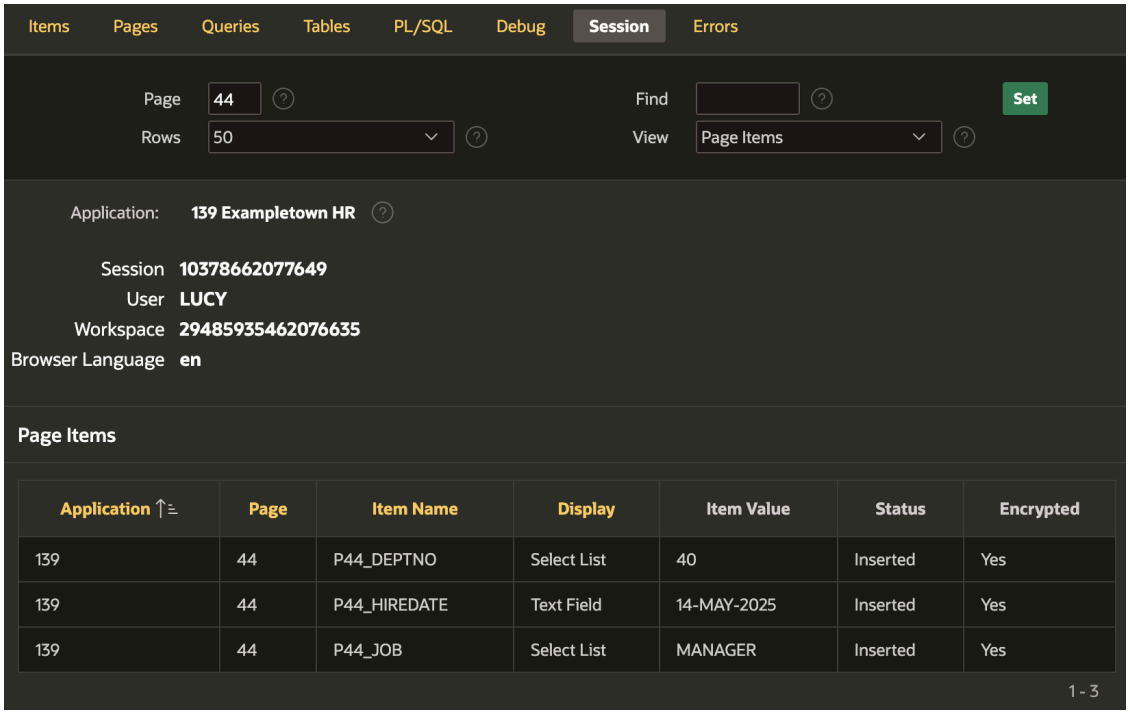
To only remove selected temporary files, just add an appropriate where clause.

8.5.9 Inspecting Session State for Debugging

When running your app from App Builder, the developer toolbar appears along the edge of the browser window and offers options useful for debugging.

Clicking the developer toolbar's *Session* > *View Session State* option opens a window with the session state browser. The figure below shows developer `LUCY` browsing her current session state for page 44 of the *Onboard New Employee* wizard. It shows the values of all page items stored persistently for the active user session.

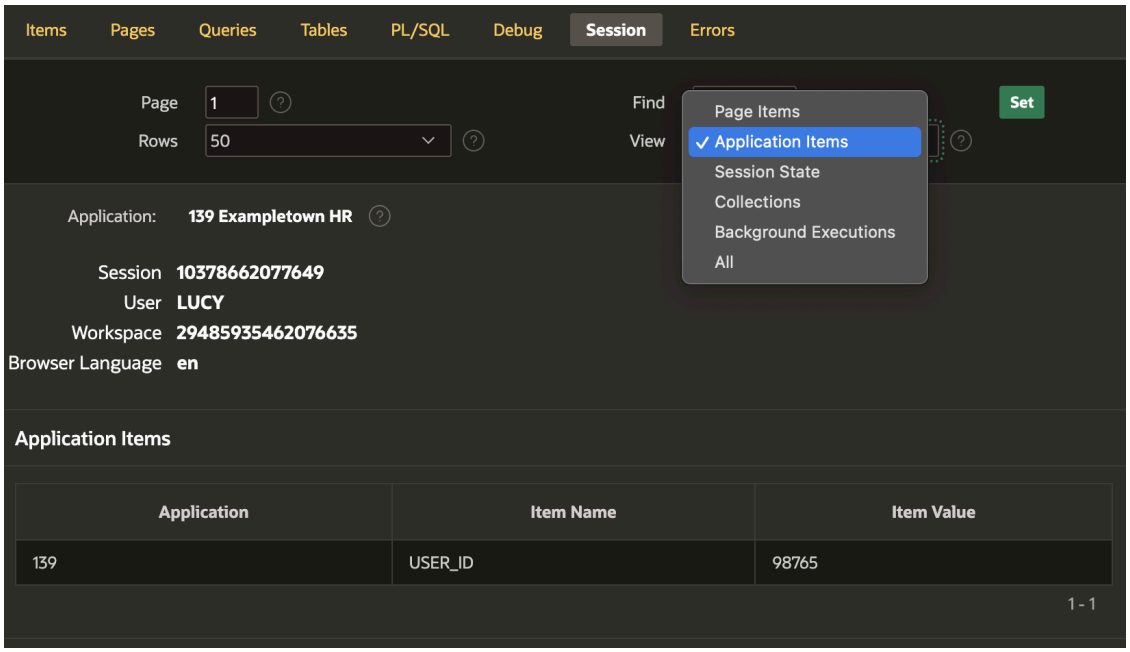
Figure 8-28 Browsing Page Item Session State for Page 44



Copyright © 2026, Oracle and/or its affiliates.

As shown below, you can also view application items, collections, and background executions. All of this information is useful when trying to understand why your APEX application is not working as you expected. The figure shows the *Session* tab of the session state viewing page, with the **View** select list open containing session state types you might need to inspect.

Figure 8-29 Browsing Application Items and Other Kinds of Session State



8.6 Studying a List and Edit Flow

The Create Page wizard builds list and edit pages for any data source in a few clicks. Understanding how those pages work together helps you use them well.

The wizard relies on declarative features you can reuse when customizing generated pages or building a Blank Page. The same ideas apply to any list page paired with an edit form.

- Creating List and Edit Pages
  The Create Page wizard's *Interactive Report* flow features an *Include Form Page* switch. Other report types offer a similar option.
- Running the List and Edit Pages
  Running your app, as shown below, the new navigation menu entry appears with the `fa-users-alt` icon you chose.
- Linking to the Edit Page
  The wizard configures the *Link Column* feature to target the form page it creates. This sets up the drill down behavior to let the user view details or edit a row in the report.
- Retrieving Existing Row by Primary Key
  The Form page has a page process in the *Before Header* phase of the *Pre-Rendering* section to retrieve the row to edit by primary key.
- Memory Only Storage for Form Page Items
  The wizard creates the form region's page items with a *Storage* setting of *Per Request (Memory Only)*.

Copyright © 2026, Oracle and/or its affiliates.

- Saving Changes with a Page Process
  The wizard configures a page process to automatically handle inserting, updating, or deleting the row in the form region.
- Conditional Database Action Buttons
  The wizard configures several important button properties affecting runtime behavior.
- Refreshing List On Dialog Close
  The wizard configures a native *Close Dialog* page process sequenced after the DML process.

8.6.1 Creating List and Edit Pages

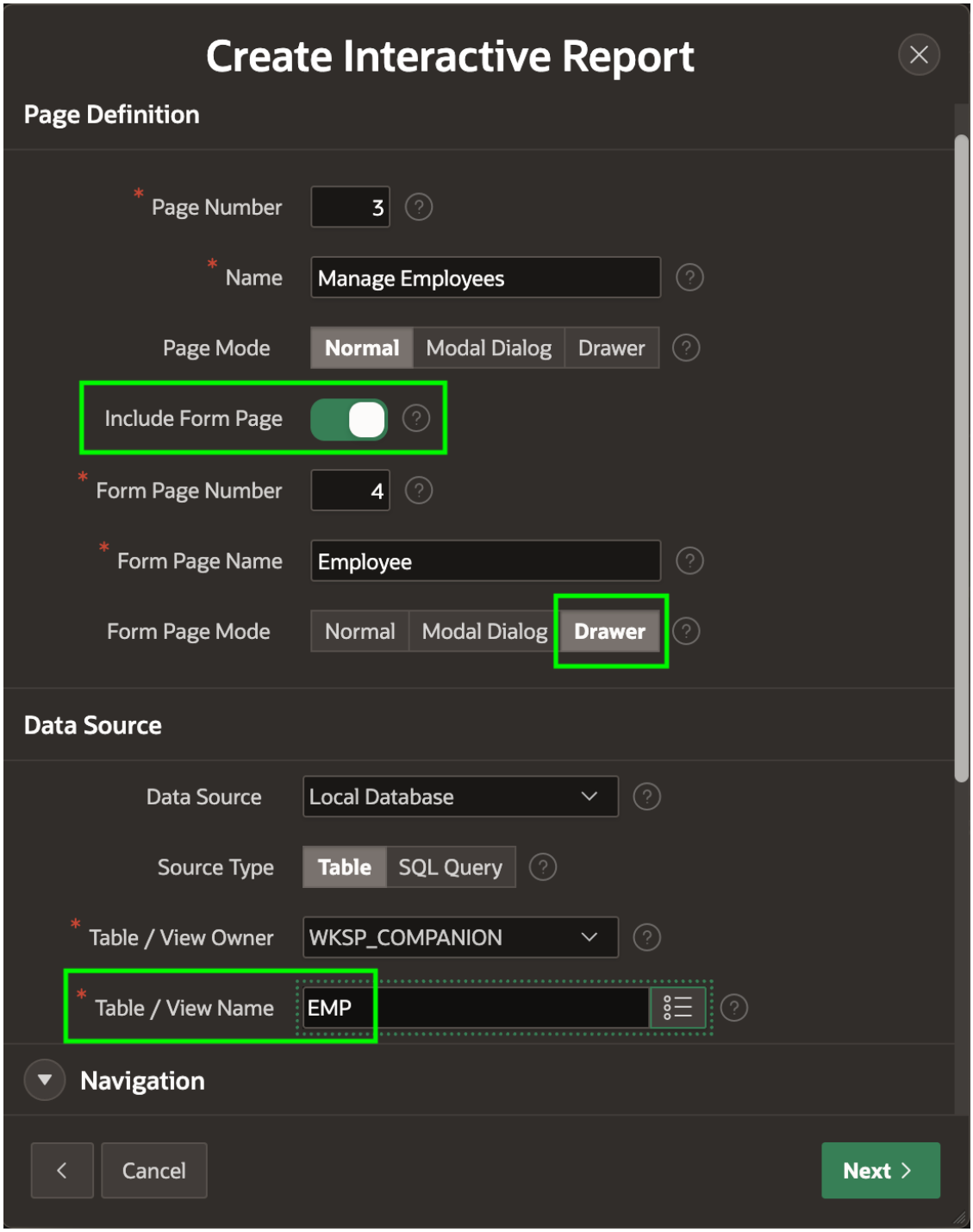
The Create Page wizard's *Interactive Report* flow features an *Include Form Page* switch. Other report types offer a similar option.

Enabling it lets you name a companion form page. In this example, you create a *Manage Employees* report page 3. By enabling *Include Form Page*, you also generate an *Employee* form page 4. By default, its *Form Page Mode* is a modal *Drawer* that appears on top of the list page by sliding in from the side.

This page *combo* lets your end users find existing rows to update or delete, as well as insert new ones. This example uses the `EMP` table in the local database, but the pages work the same way with any supported data source.

Copyright © 2026, Oracle and/or its affiliates.

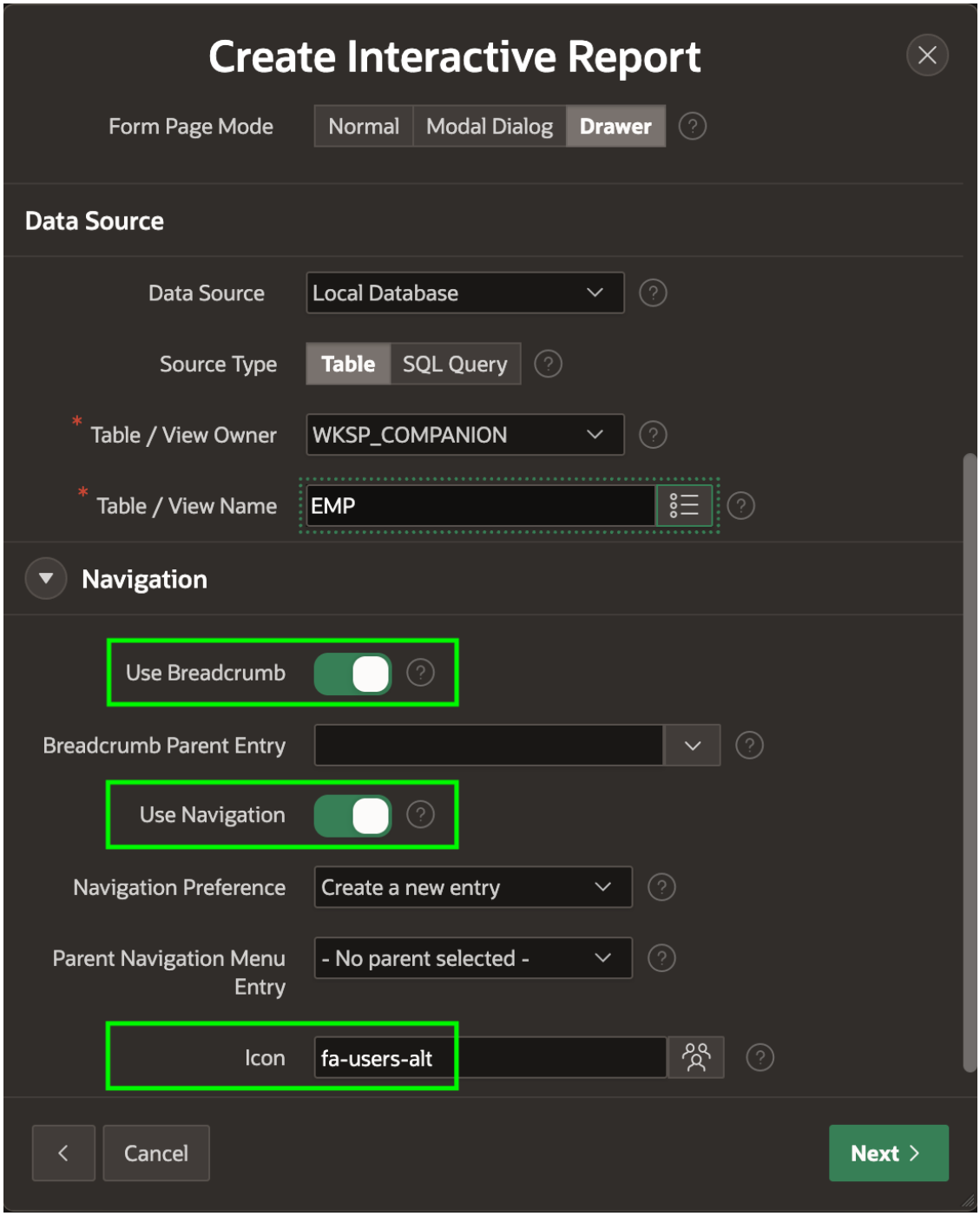
Figure 8-30 Create Page Wizard Choices for List and Edit Pages on the EMP Table



Scrolling the dialog contents to reveal the wizard's *Navigation* options, as shown below, you can opt to include a *breadcrumb* entry and region on the new page, to add a new *navigation menu entry*, and to customize the page icon by picking an icon name like `fa-users-alt` from the list of APEX icons. After you click *(Next >)*, confirm the `EMPNO` column as the primary key, and click *(Create Page)*. App Builder creates the two new pages.

Copyright © 2026, Oracle and/or its affiliates.

Figure 8-31 Configuring Breadcrumb, Navigation Menu Entry, and Icon



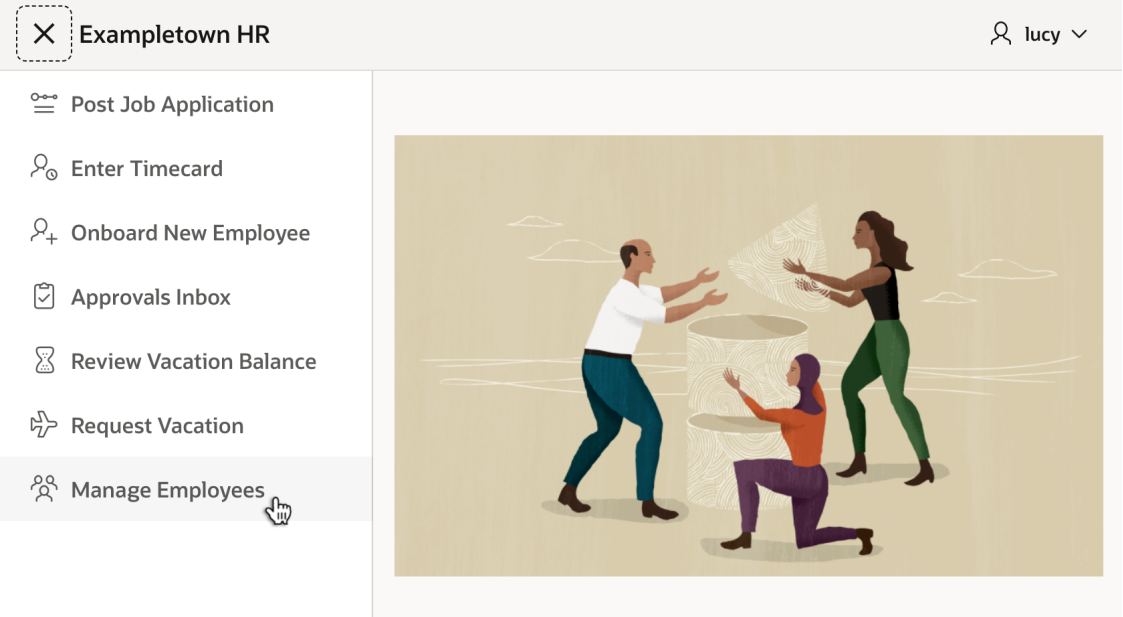
8.6.2 Running the List and Edit Pages

Running your app, as shown below, the new navigation menu entry appears with the `fa-users-alt` icon you chose.

The figure shows the new *Manage Employees* entry in the Exampletown HR app's menu.

Copyright © 2026, Oracle and/or its affiliates.

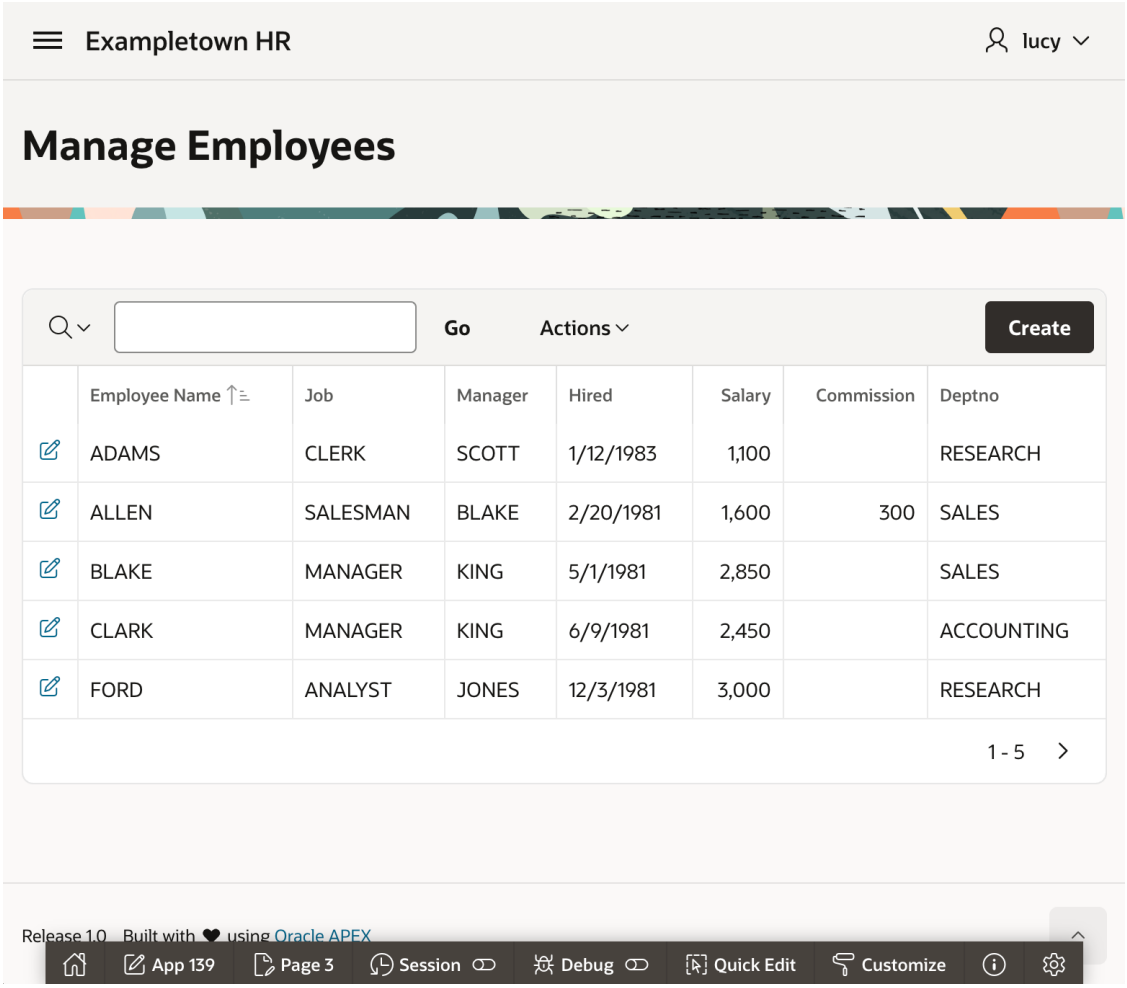
Figure 8-32 Navigation Menu Entry for Newly Created List Page



Clicking on *Manage Employees* in the navigation menu, you see the new list page you created shown below. It has a *(Create)* button to add a new employee. You can immediately see, search, and filter the employees. Since you are running the page from App Builder, notice the developer toolbar also appears at the bottom to provide quick access for iterative page development, inspecting session state, enabling debug tracing and viewing debug logs, as well as interactively customizing theme style settings.

Copyright © 2026, Oracle and/or its affiliates.

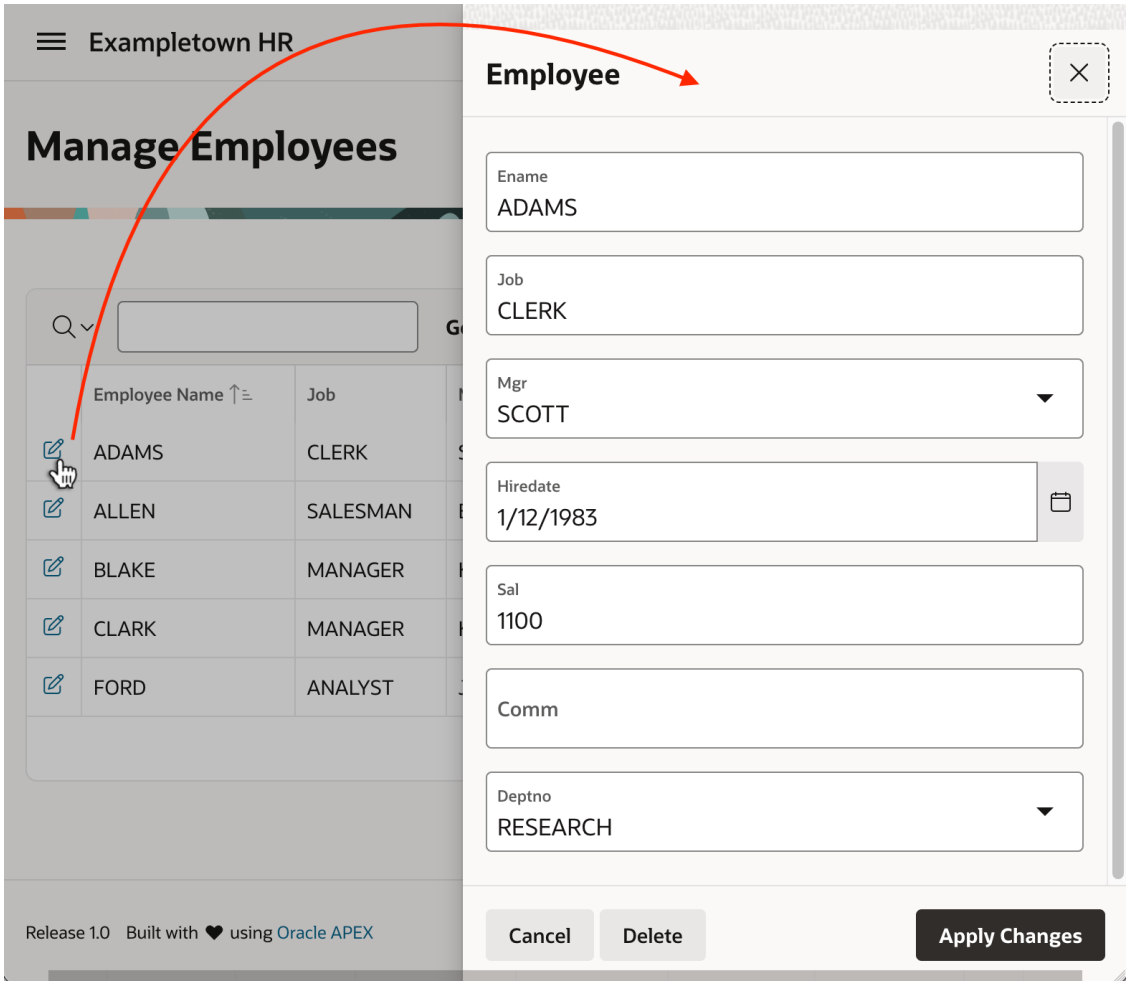
Figure 8-33 New Interactive Report Page Showing Employees



If you click the edit icon in the link column next to one of the employees, as shown below, the modal drawer page slides in from the right on top of the employees list to let you edit or delete the selected employee. If you prefer to have the developer toolbar stay mostly hidden until you need it, you can enable its *Auto Hide* setting from the "gear" settings icon in the toolbar. As shown below, in that mode it stays largely offscreen until you click on it to "pop" it up.

Copyright © 2026, Oracle and/or its affiliates.

Figure 8-34 Interactive Report and Form Modal Drawer Pages in Action



8.6.3 Linking to the Edit Page

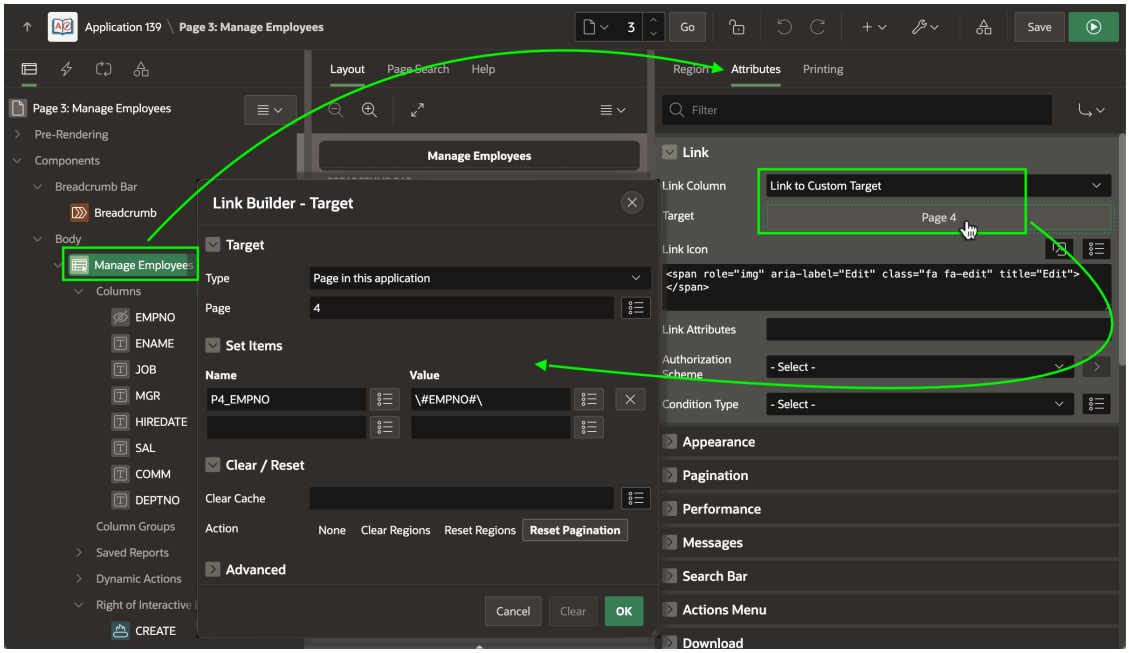
The wizard configures the *Link Column* feature to target the form page it creates. This sets up the drill down behavior to let the user view details or edit a row in the report.

After selecting the interactive report region in the *Rendering* tab and viewing its Property Editor *Attributes* tab, notice *Link Column* is set to *Link to Custom Target*. Clicking on the *Target* button labeled *Page 4*, the *Link Builder* dialog opens. You see the wizard configures a link targeting page 4: the number of the newly-created Form page. In the *Set Items* section, it adds the name of the target page's `P4_EMPNO` primary key page item, and the `#EMPNO#` substitution as the value to pass as a parameter. This expression references the value of the `EMPNO` column in the current row of the report.

Notice the *Value* `\#EMPNO#\` expression includes surrounding backslashes. This annotation tells the APEX engine to use value *verbatim*, even if it contains delimiter characters like a colon (`:`). Strictly speaking, the backslashes are not required to pass a simple `EMPNO` value that contains no delimiters. However, the wizard does this to ensure the link parameter value is passed correctly for any column.

Copyright © 2026, Oracle and/or its affiliates.

Figure 8-35 Examining Wizard-Configured Link Column Settings



Notice the value of the *Link Icon* property is the following HTML markup:

```
<span role="img" aria-label="Edit" class="fa fa-edit" title="Edit"></span>
```

The `class` attribute value indicates which icon to display. It uses a space-separated value containing `fa` to signal a Font APEX icon type and `fa-edit`, the native APEX icon name for a square with a pencil. The `title` attribute provides the text "*Edit*" an end user sees when hovering their mouse over the icon. The `aria-label` attribute ensures accessibility by specifying a label screen readers recognize.

The net result is that a native APEX `fa-edit` icon displays in the link column. When the user clicks it they are redirected to the modal drawer page 4, passing along the `EMPNO` primary key value of the clicked-on `EMP` row as the value of page 4's `P4_EMPNO` page item.

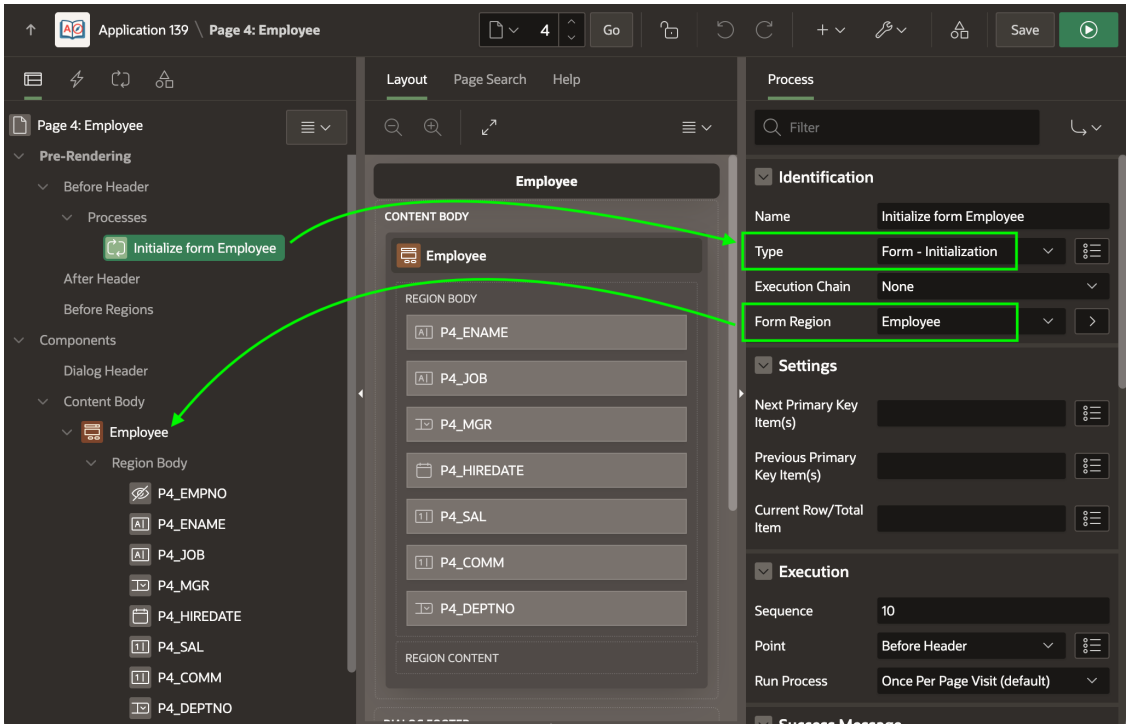
8.6.4 Retrieving Existing Row by Primary Key

The Form page has a page process in the *Before Header* phase of the *Pre-Rendering* section to retrieve the row to edit by primary key.

As shown below, in the *Rendering* tab, it uses the native *Form - Initialization* process type. This performs the lookup by primary key of a row to display, edit, or delete. Its *Form Region* property associates it with the *Employees* form region on the page. The page process uses this related form region to discover the data source to query and which columns comprise its primary key.

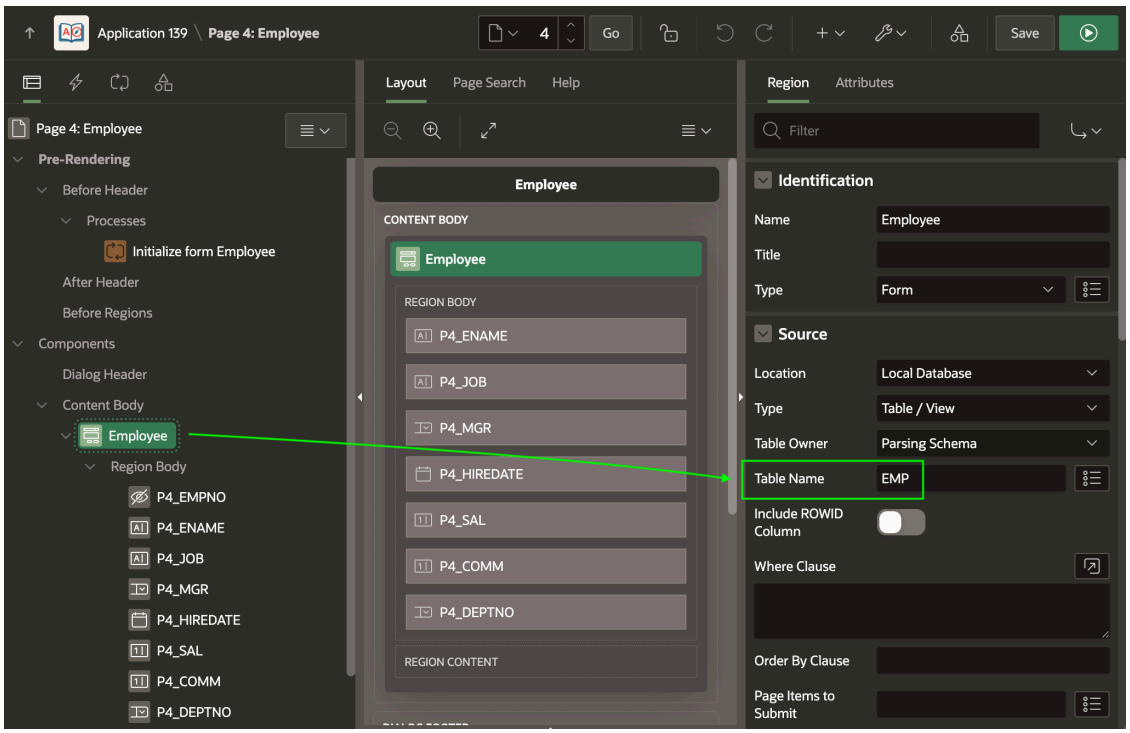
Copyright © 2026, Oracle and/or its affiliates.

Figure 8-36 Form Initialization Page Process Retrieves Row by Primary Key



The figure below shows how the wizard configures the form region's data source. With the *Employee* region selected in the component tree, the *Table Name* property in the *Source* group shows EMP.

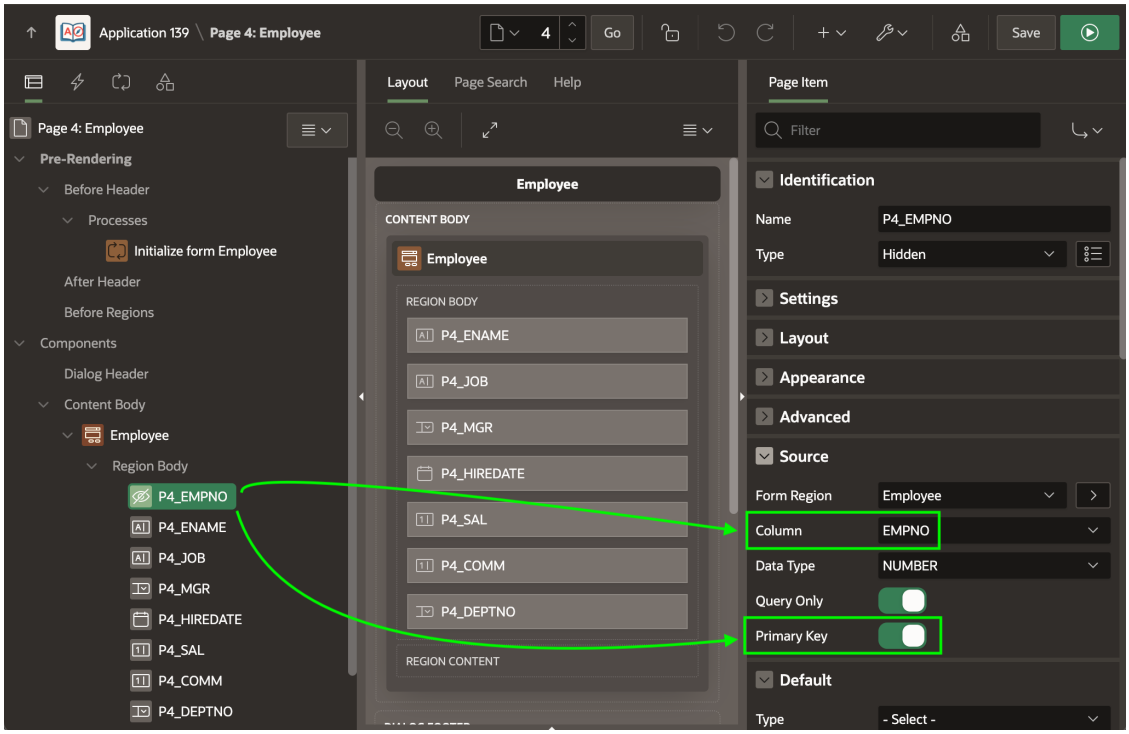
Figure 8-37 Form Region Using EMP Table as Data Source



Copyright © 2026, Oracle and/or its affiliates.

In a similar way, each page item in the form region has a corresponding *Source* column name configured. For example, the figure below shows the wizard configures the `P4_EMPNO` page item to use the `EMPNO` column as its source and indicates that it is the primary key.

Figure 8-38 Form Initialization Page Process Requires a Primary Key



When the *Form - Initialization* page process runs, it checks the values of the page items in its related form region that are marked as primary keys. If all primary key page item values have a non-null value at page render time, then this page process queries its related form region's data source to retrieve a row by primary key. In this `EMP` example, if `P4_EMPNO` has a value when the page renders, then the page process implicitly performs a SQL statement like the one below to retrieve all columns related to form page items by primary key and uses the values to populate the respective page items.

```
SELECT ENAME, MGR, HIREDATE, SAL, COMM, DEPTNO
INTO :P4_ENAME, :P4_MGR, :P4_HIREDATE, :P4_SAL, :P4_COMM, :P4_DEPTNO
FROM EMP
WHERE EMPNO = :P4_EMPNO
```

In contrast, if the `P4_EMPNO` primary key does *not* have a value when the page renders, the *Form - Initialization* page process does nothing.

Recall that the list page's *Link Column* target page link passes the value of the `EMPNO` column of the selected row as the value of the `P4_EMPNO` parameter. Before the *Pre-Rendering* phase occurs, the APEX engine assigns the value of the `P4_EMPNO` page item the value it finds in the `P4_EMPNO` parameter in the link URL. This ensures the primary key of the *Employees* form region has a value before the *Initialize form Employee* page process executes, which triggers its lookup of the existing employee's data to populate the form region's page items. This explains why clicking on one of the edit links for an employee row results in displaying the existing data for that employee.

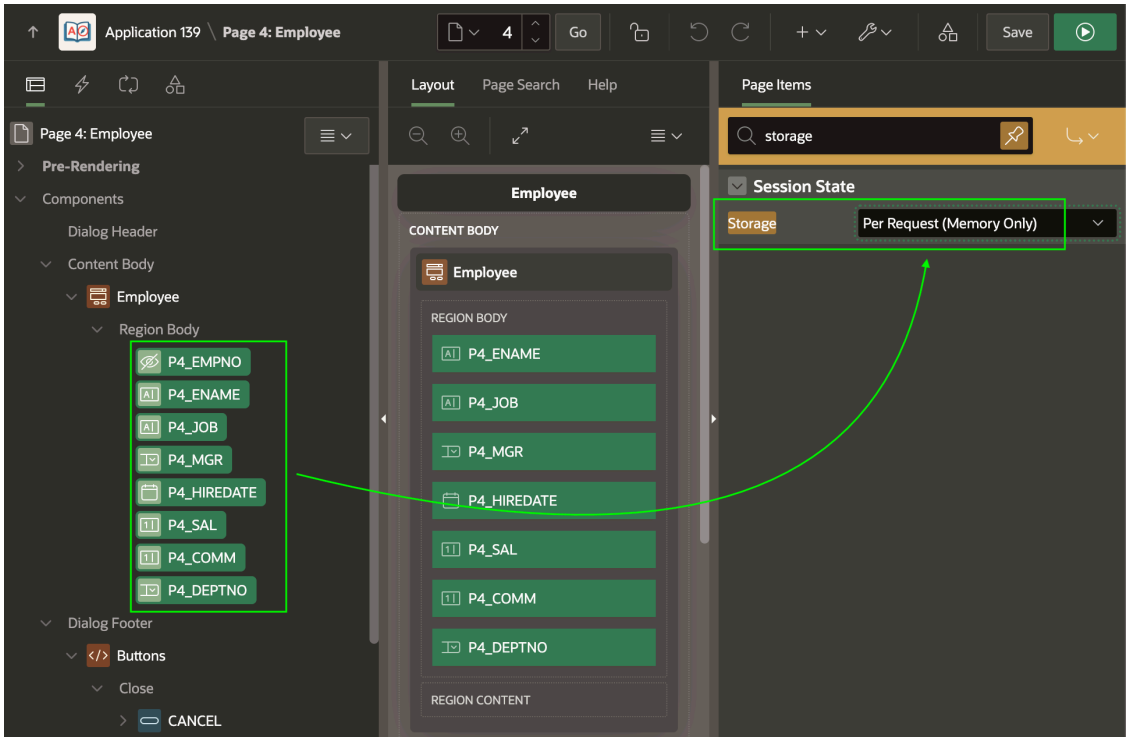
Copyright © 2026, Oracle and/or its affiliates.

8.6.5 Memory Only Storage for Form Page Items

The wizard creates the form region's page items with a *Storage* setting of *Per Request (Memory Only)*.

This means the APEX engine does *not* save their values in user session state after page processing is complete. This is not required because this form page saves any changes submitted by the user to the `EMP` table before the dialog closes. A side-effect is that the form region's page item values always start out `null` when the page renders before any *Pre-Rendering* computations or page processes execute.

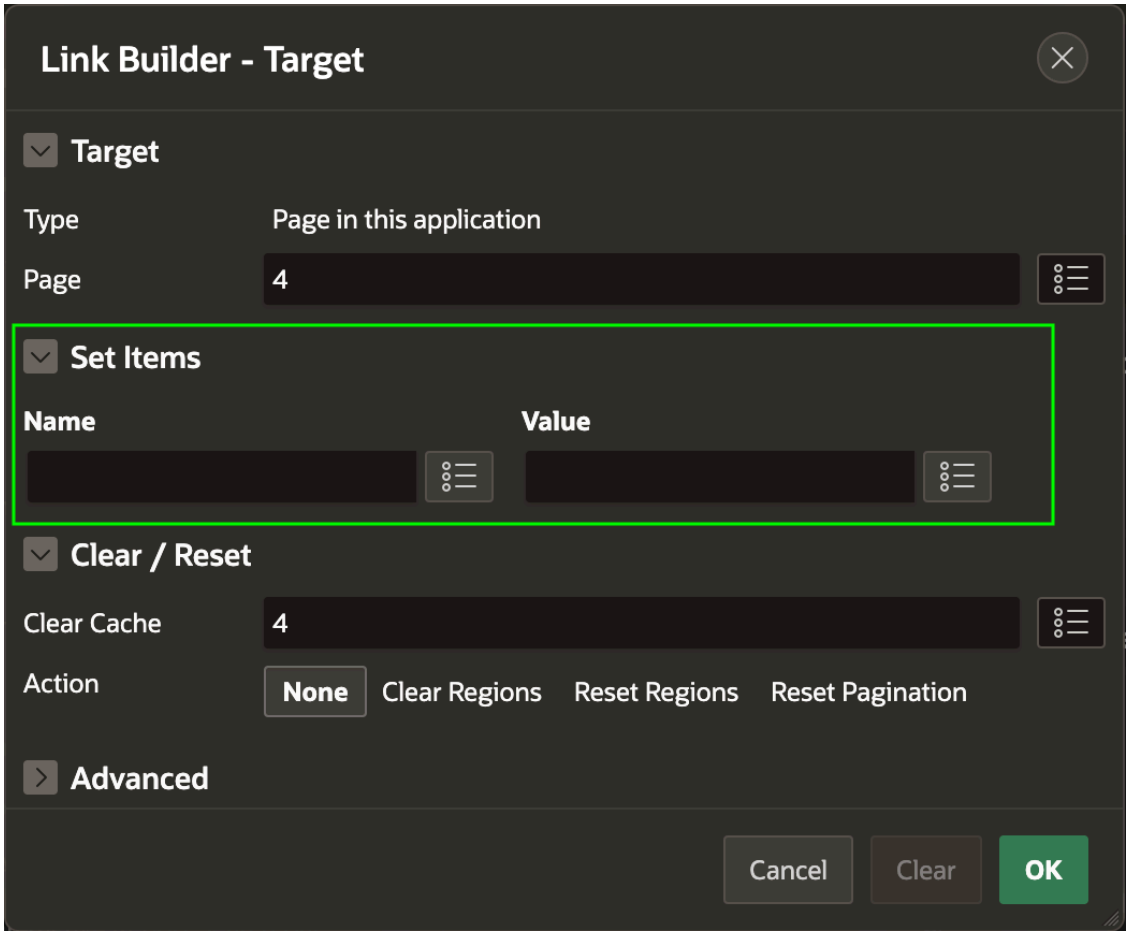
Figure 8-39 All Form Region Page Items Use "Memory Only" Storage



The figure below shows how the wizard configures the link target of the list page's *(Create)* button. Notice it passes no parameters to the target edit page.

Copyright © 2026, Oracle and/or its affiliates.

Figure 8-40 List Page's (Create) Button Passes No Parameters to Edit Page



This explains why the form page opens with empty fields when they click on the list page's *(Create)* button. The button's link target passes *no* value for the primary key page item `P4_EMPNO`, and it is one of the form region's page items with *Memory Only* storage. When the *Form - Initialization* process executes in the *Pre-Rendering* phase, it performs no lookup and the *Memory Only* page items remain empty. This leaves the page ready for the end user to enter data for a new employee.

8.6.6 Saving Changes with a Page Process

The wizard configures a page process to automatically handle inserting, updating, or deleting the row in the form region.

The *Process form Employee* page process shown below use the native page process type *Form - Automatic Row Processing (DML)*. If the button that submits the page has a *Database Action* setting of INSERT, UPDATE, or DELETE, this process performs the appropriate DML operation to save the end user's changes. If the submitting button has no *Database Action* configured, the page process performs no action.

Notice the page process specifies the associated *Form Region* and uses default values for the other settings to:

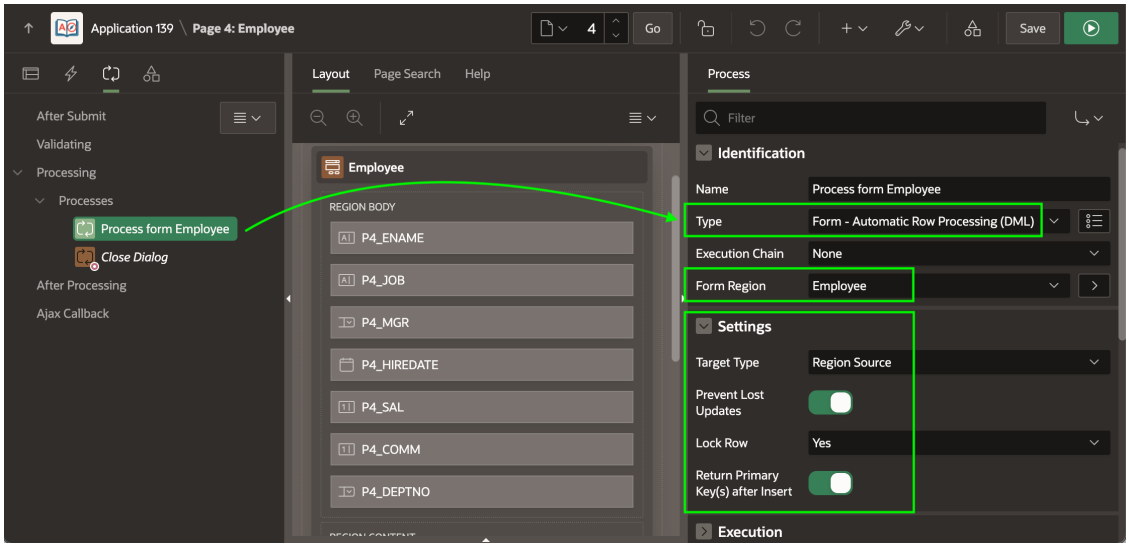
- Perform the DML against the form region's source,

Copyright © 2026, Oracle and/or its affiliates.

- Prevent lost updates by checking if another user changed the row before the current user did,
- Lock the row being updated or deleted, and
- Return the primary keys into corresponding page items after insert.

The *Target Type* settings support two other modes that let you target the automatic DML at a *different* table or view from the region's source, or alternatively to perform custom DML handling yourself by using PL/SQL. The figure shows the highlighted page process that handles the DML operation for the *Employee* form region and its declarative settings.

Figure 8-41 Declarative Form DML Handling with Native Page Process



8.6.7 Conditional Database Action Buttons

The wizard configures several important button properties affecting runtime behavior.

On a button whose *Action* is set to *Submit Page*, the *Database Action* setting indicates which SQL operation the *Form Automatic DML* page process executes. Failure to set this property results in *no* DML operation. The wizard configures four buttons on the page as follows:

Table 8-2 Wizard-Created Form Page Button Settings

Button	Behavior	Execute Validations	DB Action	Server-side Condition
CREATE	*Submit Page*	Yes	INSERT	P4_EMPNO is null
SAVE	*Submit Page*	Yes	UPDATE	P4_EMPNO is not null
DELETE	*Submit Page*	No	DELETE	P4_EMPNO is not null
CANCEL	*Dynamic Action*	No		

The server-side conditions on the four buttons result in the following behavior:

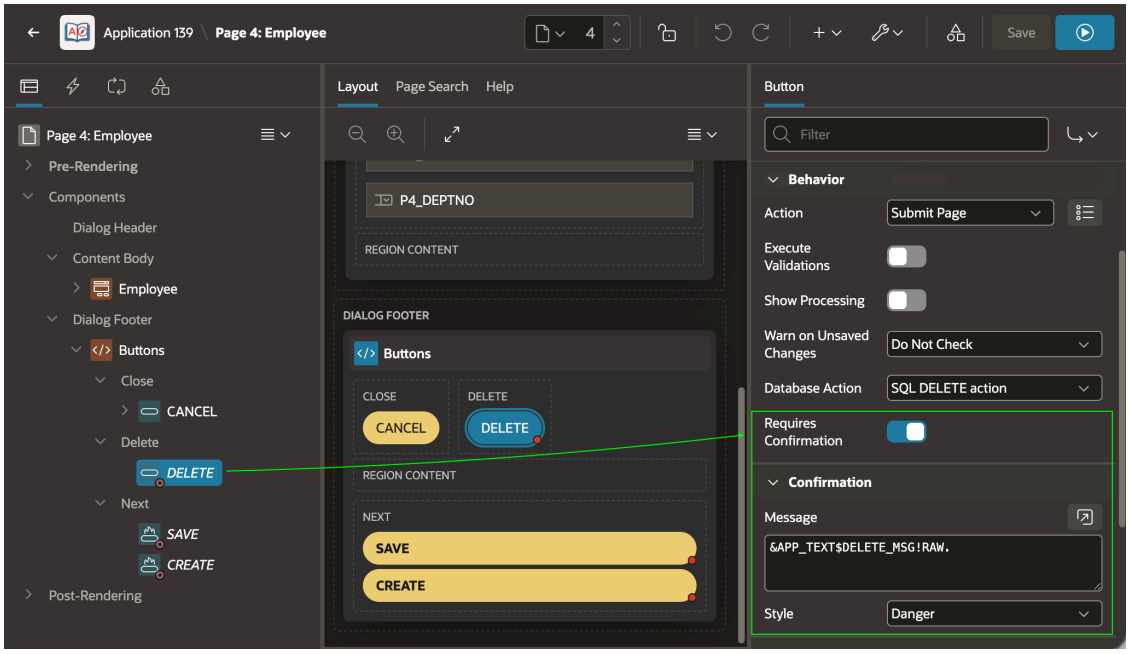
- *(Cancel)* always displays
- *(Create)* displays when the primary key page item is null

Copyright © 2026, Oracle and/or its affiliates.

- *(Save)* and *(Delete)* display when the primary key page item has a value

As shown below, the wizard also configures the *(Delete)* button to require a confirmation message to be shown in the *Danger* style. It uses the special substitution string `&APP_TEXT$DELETE_MSG!RAW.` to include the translatable text message with key `DELETE_MSG`. The `!RAW` modifier lets the text include HTML markup without being escaped. APEX defines a default runtime message for this message key, but you can customize it by creating a shared component text message with the same key having its *Used in JavaScript* property enabled.

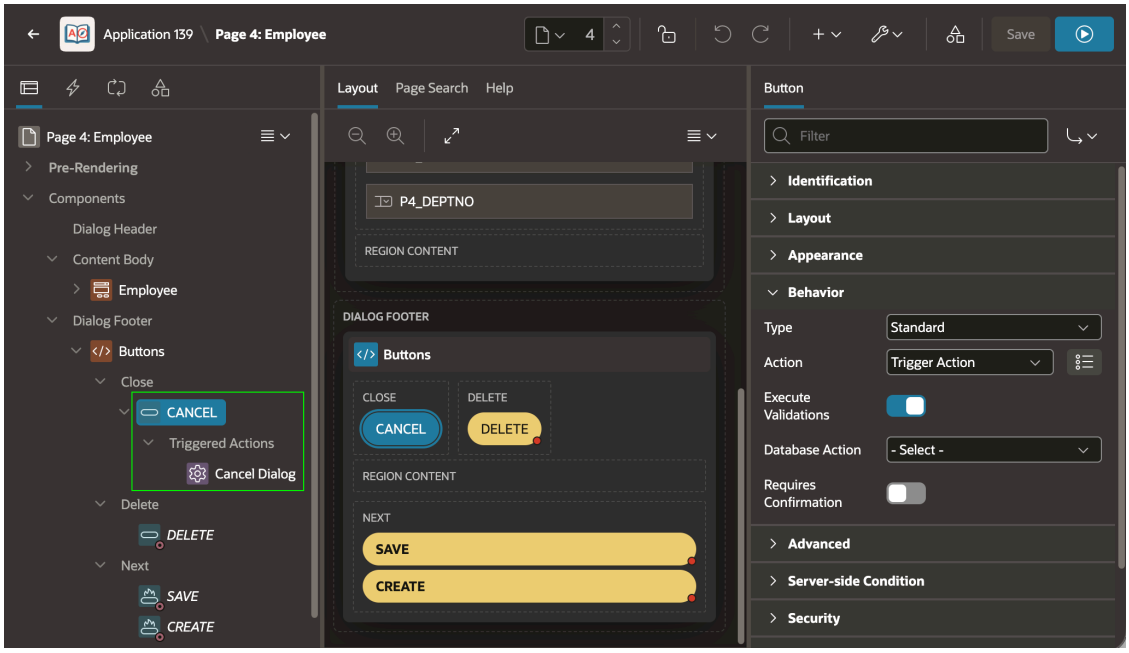
Figure 8-42 Wizard-Created Form Page Delete Button



In a *modal* form page the wizard configures a *Cancel Dialog* triggered action to fire as shown in the figure below. If you opt for a normal edit page instead of a modal page, then the wizard configures the *(Cancel)* button to submit the page, and a branch with its *When Button Pressed* condition set to the `CANCEL` button to return to the list page.

Copyright © 2026, Oracle and/or its affiliates.

Figure 8-43 Wizard-Created Modal Form Page Cancel Button



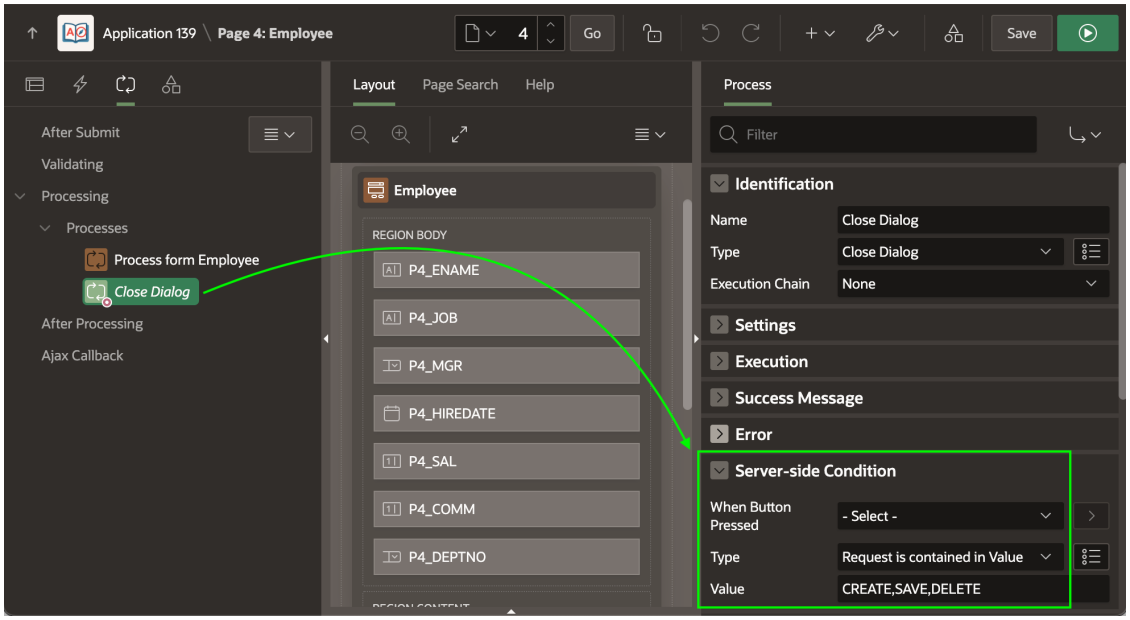
8.6.8 Refreshing List On Dialog Close

The wizard configures a native *Close Dialog* page process sequenced after the DML process.

Since the *(Create)*, *(Apply Changes)*, and *(Delete)* buttons should all close the dialog, it uses an appropriate server-side condition to achieve this. The APEX engine stores the name of the submitted button in the `REQUEST` substitution string. The process executes when `REQUEST` is one of the three button names in the comma-separated list containing `CREATE`, `SAVE`, or `DELETE`. The figure shows the *Server-side Condition* settings of the *Close Dialog* page process in Page Designer.

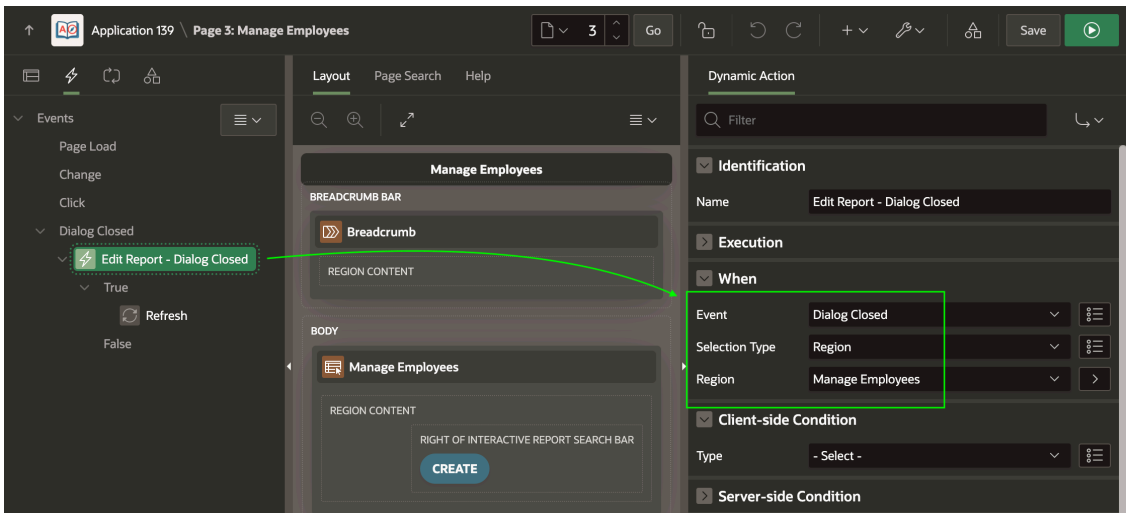
Copyright © 2026, Oracle and/or its affiliates.

Figure 8-44 Wizard-configured Close Dialog Page Process



When the modal dialog closes, the end user might have inserted, updated, or deleted an employee so they expect the list page to reflect that change. As shown below, the wizard configures a dynamic action in *Employees* list page to handle the *Dialog Close* event by refreshing the interactive report region. Using the *Selection Type* and *Region* properties, it sets the handler to listen for any *Dialog Closed* events that were from a dialog originally opened by the *Manage Employees* region.

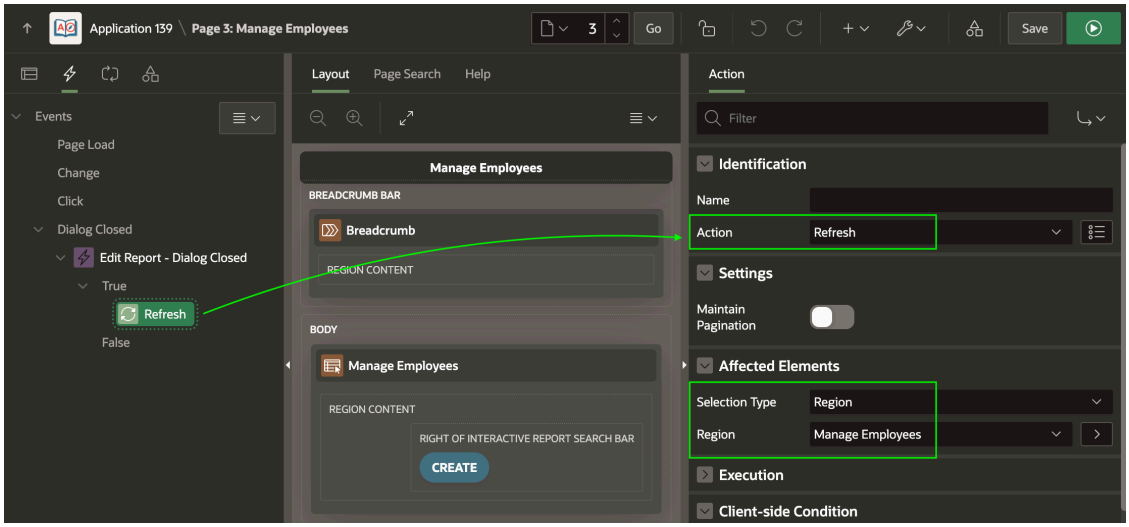
Figure 8-45 Dynamic Action Event Handler for the Close Dialog Event



When this event occurs, you refresh the interactive report region using a native *Refresh* dynamic action step as shown below and indicating the desired region as the affected element.

Copyright © 2026, Oracle and/or its affiliates.

Figure 8-46 Dynamic Action Step to Refresh the Interactive Report



Copyright © 2026, Oracle and/or its affiliates.

9

Editing Data in a Grid

Edit tabular data with Interactive Grid, overriding its save behavior for special cases, and customizing it with JavaScript APIs.

Interactive Grid lets users work with tabular data with a rich, in-place editing experience and automatic handling of multi-row inserts, updates, and deletes. Its unique editing, selection, clipboard, scrolling, and column display features complement its many *reporting* features inspired by Interactive Report. Study a special case of editing collection members in a grid to see how to override the default save handling when needed. Learn more about how the grid works at runtime, how to customize its actions and behavior with JavaScript APIs, and how to use its client-side data APIs to refresh a single grid row after a modal dialog closes.

- Getting to Know the Interactive Grid
  Explore Interactive Grid features for reporting, editing, selection, clipboard work, scrolling, and rich column display.
- Editing Collection Data with a Grid
  Edit session collection rows in an Interactive Grid using a stable key and custom save code.
- Understanding and Customizing Grid Behavior
  Learn how Interactive Grid works at runtime and how to use JavaScript APIs to customize its setup, actions, events, selection, and data refresh behavior.

9.1 Getting to Know the Interactive Grid

Explore Interactive Grid features for reporting, editing, selection, clipboard work, scrolling, and rich column display.

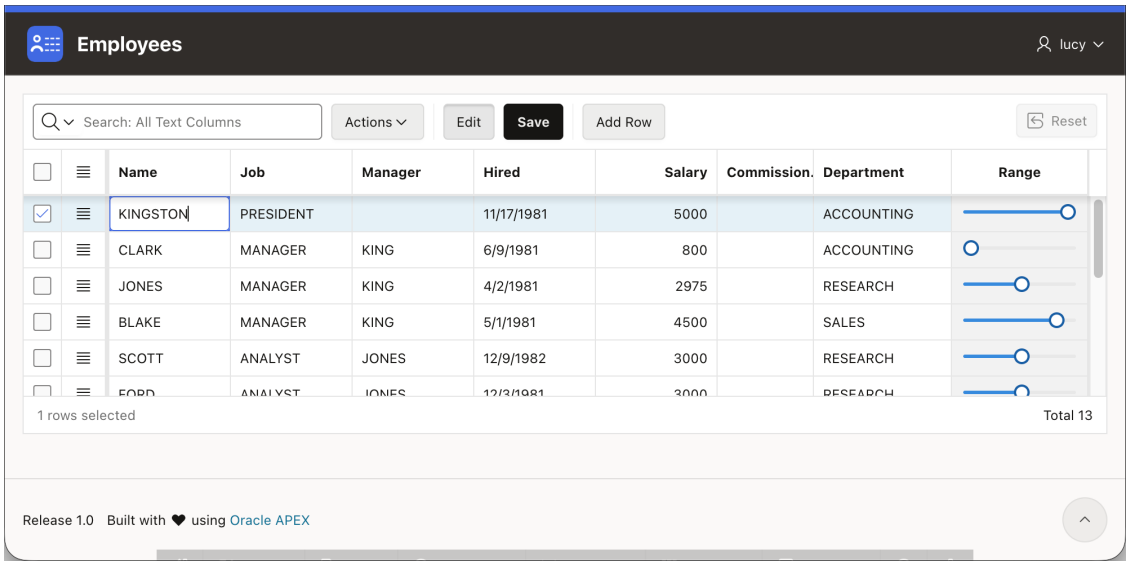
The Interactive Grid region presents query results as a rich data grid. Like Interactive Report, it lets users search, sort, filter, break rows into groups, add aggregates, highlight, chart, download, rearrange columns, and save personalized report layouts. Interactive Grid also adds grid-specific behaviors such as column resizing and freezing, scroll pagination, keyboard navigation and selection, clipboard operations, and optional in-place editing. When editing is enabled, users can add, update, duplicate, delete, refresh, and revert rows without leaving the page. Let users save one grid at a time without refreshing the page, or have them submit the page to process all changes together.

Exploring a Simple Employees Editable Grid

Consider the page shown in the figure below with an Interactive Grid based on the familiar `EMP` table. If editing is enabled, the user can click or navigate using the keyboard into the first row's **Name** column and edit an employee's name from `KING` to `KINGSTON`.

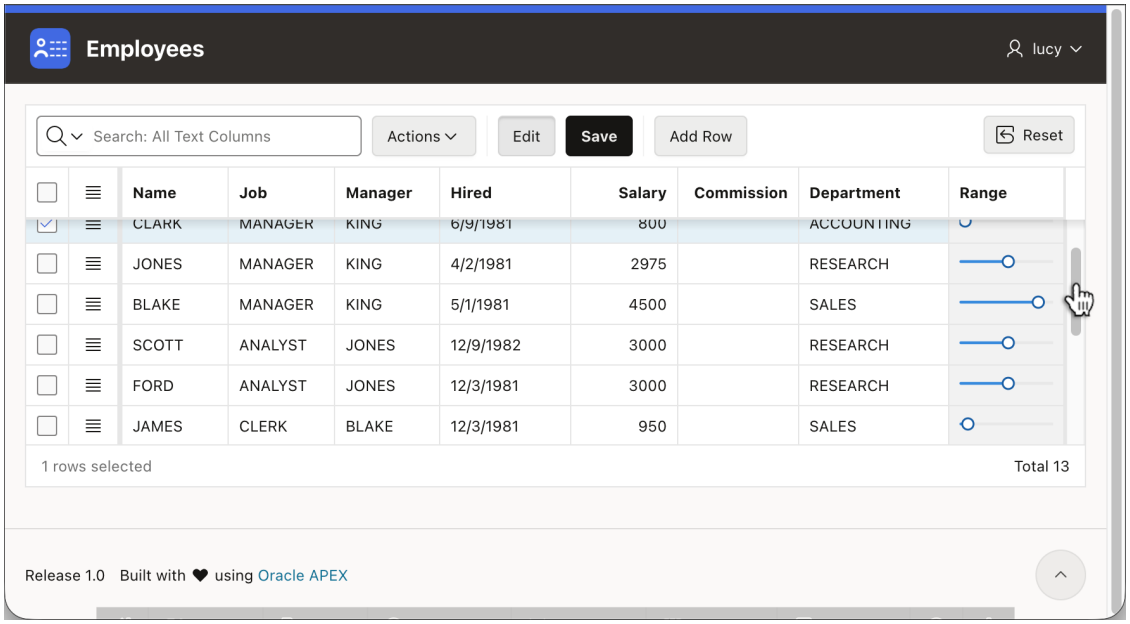
Copyright © 2026, Oracle and/or its affiliates.

Figure 9-1 User Can Insert, Update, and Delete Multiple Rows Then Save All the Changes



If you configure the grid to let users move through the data by scrolling, as shown below, the headings can stay visible as the user inspects even very large data sets. You configure which columns and rows are editable. Assuming they are authorized, the user can click into any editable row and any editable column to edit required values. When they save the grid or submit the page, all their changes are saved automatically.

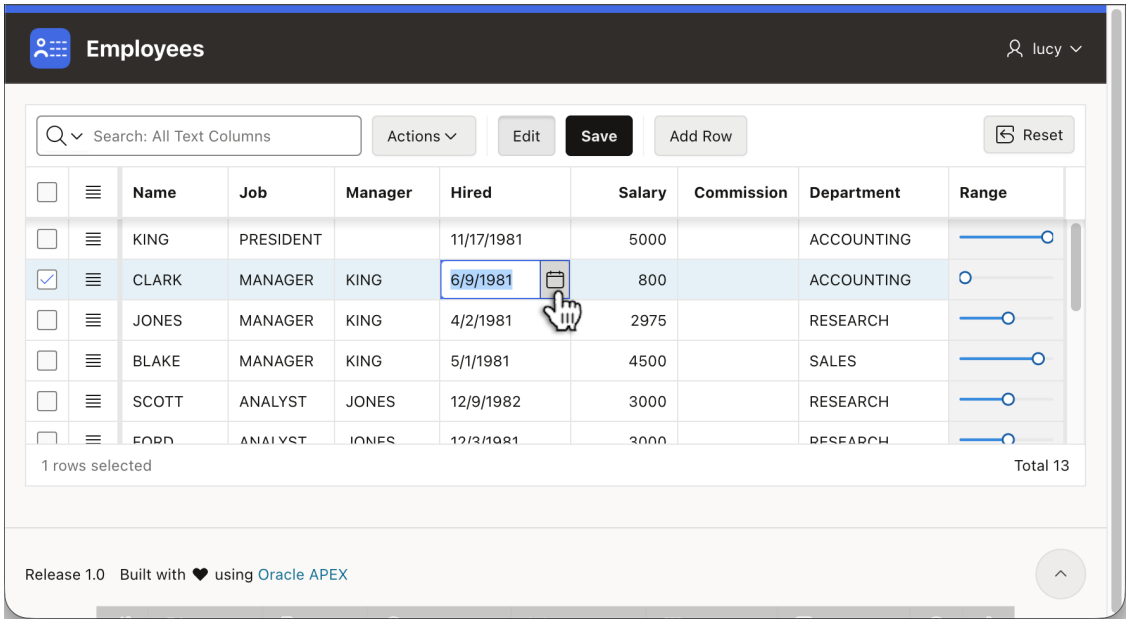
Figure 9-2 Grid Offers Scrolling Pagination with Fetch on Demand



You also pick the type of *input* control for each editable column that best serves end users. Available item types include many of the same kinds you can use in a form page. For example, in the figure below the user is about to pop open the *Date Picker* mini calendar to change employee *CLARK*'s date of hire.

Copyright © 2026, Oracle and/or its affiliates.

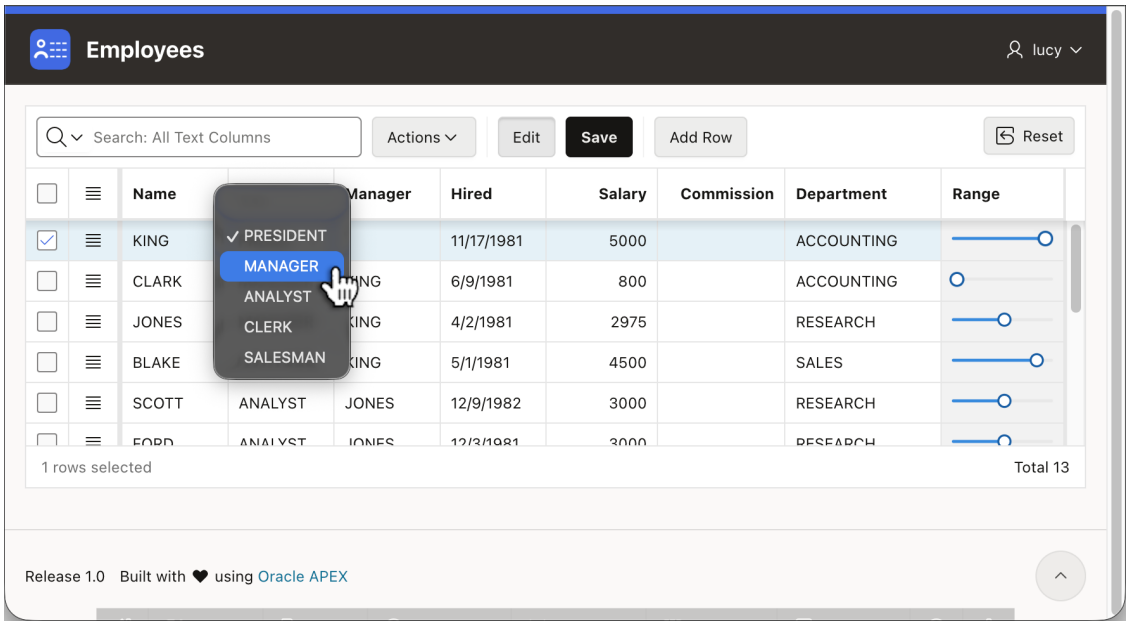
Figure 9-3 Date Picker is One of Many Item Types Available for Grid Column Editing



Your choice of column form controls includes list-driven ones like PopupLOV and Select List as well. The figure below shows the user editing employee *KING*'s job using a select list.

> **Tip**
>
> The **Range** column in the figure shows you can also use a *Template Component* to display column data. A later section explains how this one shows each employee's salary as a dot positioned along a salary range line.

Figure 9-4 Grid Columns Can Use List-Driven Items for Data Entry



Copyright © 2026, Oracle and/or its affiliates.

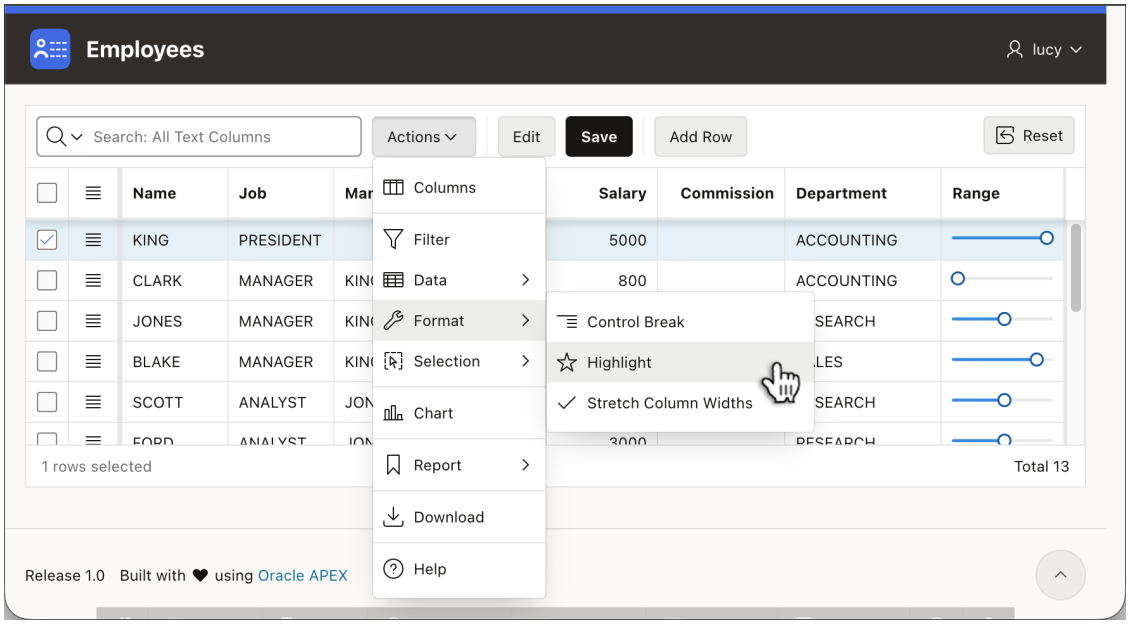
Understanding the Grid's Core End-User Capabilities

Interactive Grid shares Interactive Report's focus on letting end users control how reports look and behave, but gives them a more direct way to navigate and select rows and cells. When editing is enabled, they can also paste ranges of data directly into the grid. Depending on what features you enable, users access these capabilities through the default toolbar, Actions menu, column headers, keyboard shortcuts, and direct grid interactions. If you allow it, users can:

- Search across rows, optionally constraining search to a selected column.
- Choose, reorder, resize, freeze, and stretch columns.
- Sort by one or more columns and create filters using row-level or column-level conditions.
- Add aggregates, highlights, control breaks, groupings, and charts to summarize or reshape the displayed data.
- Use flashback, when enabled and supported, to view data as it existed at an earlier point in time.
- Save, rename, switch, reset, or delete saved reports, including private or public reports depending on authorization.
- Download or email the report in the formats made available by the developer and current APEX configuration.
- Use Help and keyboard shortcuts to navigate, select, edit, and invoke grid commands.

The figure below shows the Interactive Grid's **Actions** menu with the features you let users access.

Figure 9-5 Actions Menu Presents the Grid Features You Enable for Users



Examining Editable Grid Behavior

When you enable editing, the grid is a multi-row data entry component backed by a native page process that saves inserted, updated, and deleted rows automatically. When needed, you can customize processing to accommodate more advanced scenarios.

Copyright © 2026, Oracle and/or its affiliates.

Editable grids have navigation mode and edit mode. In navigation mode, the user moves through cells, selects rows or ranges, opens row actions, and invokes toolbar or menu commands. In edit mode, the active cell hosts the appropriate form control. Double click a cell, press [*Enter*], or click the (**Edit**) toolbar button to go into edit mode. Press [Esc] to return to navigating. Only one cell editor is active at a time; this avoids rendering an input control for every visible cell, improving performance, keyboard navigation, and accessibility.

You define whether users may add rows, update rows, delete rows, and whether the grid adds an empty row when the query returns no data. You can also use an **Allowed Row Operations** column to compute permitted actions dynamically, for example allowing updates only while a row is in a draft status. To make a column read-only in certain cases, just configure a condition. It can also depend on a user's authorization.

Like Form regions, editable grids can protect users from concurrency problems. The **Lost Update Type** setting compares row values by checksum or using a row version column. This ensures one user's edits don't silently overwrite a colleague's recent changes to the same record.

Using Row, Cell, and Selection Actions

The grid exposes both row actions and selection actions. Row actions work on the current row and typically include add, duplicate, delete, refresh, and revert. Selection actions work on selected rows or cell ranges and include duplicate, delete, refresh, revert, copy down, fill, clear, and clipboard operations where allowed.

Interactive Grid supports row selection and cell selection. In row selection mode, users can select one or more rows. In cell selection mode, users can select a rectangular range of cells, including keyboard support for selecting columns, rows, or all cells. Using simple JavaScript, you can opt to maintain selection across page-by-page pagination, and specify a selection state item to store the selected primary keys in a hidden page item for use in SQL, PL/SQL, branching, or other regions. See Understanding and Customizing Grid Behavior for examples.

These selection features are especially useful for business workflows such as approving several rows, recalculating values for selected rows, refreshing a dependent chart, passing selected keys to another page, or running a custom action on a batch of records.

Working with Clipboard and Spreadsheet Paste

Interactive Grid supports full copy, cut, paste, paste insert, and drag-and-drop of tabular data. This lets a user copy a range of cells from a spreadsheet and paste it into an editable grid.

Clipboard operations apply in navigation mode. Copy can place selected rows or cell ranges on the clipboard as plain text and HTML table data, and APEX also uses a custom JSON format when possible to preserve underlying model values for grid-to-gI rid copy and paste. Cut deletes selected rows or clears selected cells after copying, when the affected records and cells allow editing. Paste overwrites values starting at the current cell; Paste Insert uses the Shift-modified shortcut or command to insert new rows as needed.

Pasted values are set as if the user typed them, so dynamic actions and client-side validations can run. Read-only rows, read-only columns, and cells that cannot be edited are skipped. LOV-backed columns require care: the pasted value should match the return value that APEX stores, not merely the display value the user sees. Because APEX uses browser clipboard APIs for these operations, paste behavior can also depend on the browser and deployment context. For example, a browser might require HTTPS, a keyboard shortcut, or explicit user permission before exposing clipboard data to the page.

Copyright © 2026, Oracle and/or its affiliates.

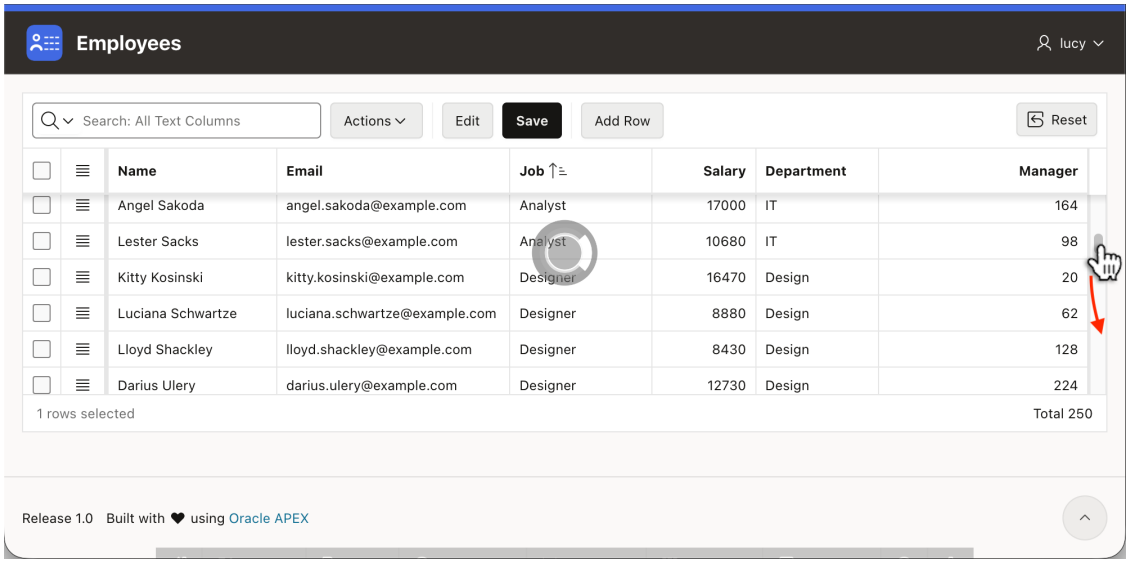
Navigating Larger Data Sets with Pagination and Scrolling

Users can move through Interactive Grid rows page-by-page like in an Interactive Report, showing a bounded set of rows with navigation controls. The grid *also* offers scroll pagination that presents a viewport into the data. Both approaches fetch only the necessary rows as the user moves through the data. The grid also supports fixed headers, frozen columns, column resizing, and drag-and-drop column reordering. These features help users work with larger result sets while staying oriented in the data.

The `EMP` table is small enough to keep examples easy to follow. But to illustrate scroll pagination, the figure below shows an Interactive Grid scrolling the built-in *Employees* sample data instead. Its 250 employees make scrolling behavior and on-demand data fetching easier to witness. The figure shows the progress indicator that appears when the user tugs the scrollbar down and the grid fetches additional data on demand.

When **Show Total Count** is off, Interactive Grid loads more rows as users scroll farther down, and the scrollbar can only indicate whether more rows remain. When **Show Total Count** is on, APEX knows the total row count and can use virtual scrolling, so the scrollbar reflects the size of the result set. This can improve the browsing experience, but requires an additional query to calculate the count.

Figure 9-6 Scrolling Through Large Dataset, Interactive Grid Fetches More Data On Demand



By default, when the Interactive Grid does scrolling pagination it uses the available vertical space on the page. Use the **Heading** > **Fixed To** setting on the **Attributes** tab to choose how the grid heading behaves:

- **Page**: heading stays pinned to the top of the page as rows scroll beneath it.
- **Region**: heading is above a scrolling grid sized by the **Fixed Report Height** setting.
- **None**: heading is not fixed, and scrolls normally with the page content.

The figure above shows a grid fixed to the **Region** with a report height of 300 pixels.

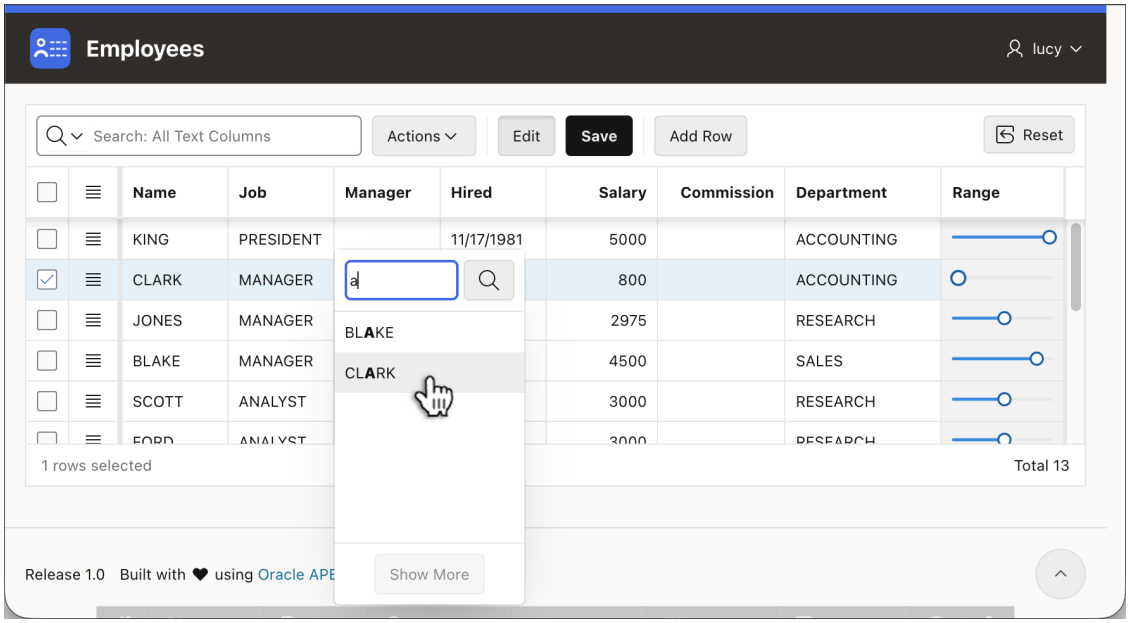
Copyright © 2026, Oracle and/or its affiliates.

Editing Rows with Form Controls

An editable Interactive Grid row is conceptually like a horizontal form. Each editable column specifies a column type that determines its cell editing experience for users. A column's form control behaves much like a page item, but the value belongs to the column's cell in the current row. This lets users edit many rows in place while still getting familiar APEX item behavior.

Grid columns can use many common item types, including text fields, number fields, date pickers, select lists, popup LOVs, switches, and checkboxes. This makes editable grids useful for more than simple text entry. For example, a row of an Interactive Grid on the familiar `EMP` table can include a select list for `DEPTNO`, a popup LOV for `MGR`, a date picker for `HIREDATE`, and number fields for amounts or percentages. The figure below shows the user choosing a new manager for employee CLARK by searching the popup LOV for names containing a letter `A`.

Figure 9-7 Updating a Grid Cell Using a Popup LOV



Column form controls can also participate in familiar form behaviors. Cascading lists can depend on other column values in the same row. Dynamic actions can react when a user changes a column value, validate input, calculate another column, enable or disable related fields, or refresh a dependent LOV. These behaviors make the editable row feel less like a plain table and more like a small data-entry form available on-demand for each row.

> ⚠ **Caution**
>
> Cascading lists and dynamic actions that execute code on the server can slow down grid bulk grid operations like paste, copy down, fill, and clear because they may make calls to the server for every row edited. When practical, consider doing calculations and validations in JavaScript on the client, then recompute them at save time using a page process with **Editable Region** set to the grid.

Copyright © 2026, Oracle and/or its affiliates.

Using a Template Component in a Grid Column

Template Components let any developer familiar with HTML markup create reusable components other team members can easily use in Page Designer. You'll see a simple example here, but for more information on creating them, see Reusable Components Using HTML Skills.

The **Percent Dot on Bar** template component displays a percent value as a dot on a bar using Scalable Vector Graphics. SVG is a W3C standard set of tags representing vector drawing instructions. All major modern browsers support it. This makes it a practical choice for lightweight, scalable graphics in web applications.

The template component uses the following fragment of SVG to draw the bar and the dot. Notice the markup references two Template Component custom attributes `LABEL` and `PERCENT`. When using this component on a page, developers provide appropriate values for these two attributes in the Property Editor that make sense for the context they use it in.

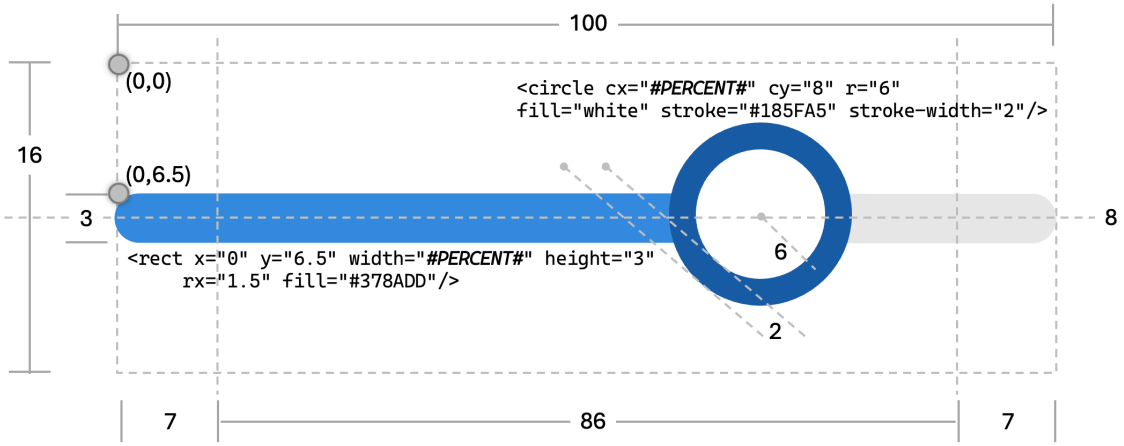
```
<svg xmlns="http://www.w3.org/2000/svg"
     viewBox="0 0 100 16" width="100%" height="16">
  <title>#LABEL#</title>
  <!-- Light gray rectangle with rounded ends -->
  <rect x="0" y="6.5"
        width="100" height="3"
        rx="1.5" fill="#e8e8e8"/>
  <!-- Blue rectangle with rounded ends -->
  <rect x="0" y="6.5"
        width="#PERCENT#" height="3"
        rx="1.5" fill="#378ADD"/>
  <!-- White circle with dark blue 2-wide border, radius 6 -->
  <circle cx="#PERCENT#" cy="8" r="6"
          fill="white" stroke="#185FA5" stroke-width="2"/>
</svg>
```

The `viewBox` attribute is for the *author*. It sets up a math-friendly coordinate space. The `viewbox="0 0 100 16"` specifies a rectangle 100 units wide and 16 units tall. In contrast, `width` and `height` are for the *browser*, controlling how that space maps to the page. The `width="100%"` asks it to use the full width of the container and `height="16"` makes the shape 16 pixels tall. The `width` and `height` of the rectangles (`<rect>`) and circle (`<circle>`) elements are expressed in logical units. The browser handles converting logical units into page units.

The figure below shows how the three SVG shapes get drawn into the `viewBox` area. The light grey rounded rectangle is 3 units tall and 100 units wide. The y-axis grows down from the origin (0,0) in the upper left corner. So drawing it at logical `y` coordinate 6.5 makes the 3-unit tall rectangle vertically centered. The blue rounded rectangle is similarly anchored, but only stretches `#PERCENT#` units across. The dark blue circle of radius 6 has a 2 unit border stroke, which SVG centers on the path of the circle, so the visual circle plus its border has an effective radius of 7 units. This means the circle's center needs to be shifted by 7 units if we don't want it to be clipped when showing percentage values near the left and right edges. So, the effective number range for drawing the circle's center needs to go from 7 to 93, which is a range of 86 units.

Copyright © 2026, Oracle and/or its affiliates.

Figure 9-8 SVG `viewBox` Area Used by Percent Dot on Bar Template Component Graphics



> **Tip**
>
> This example omits accessibility attributes and clipboard configuration for simplicity. Production use should consider both. Grid JavaScript APIs provide control over what a cell copies to the clipboard.

Adding SQL Expression Column to the Grid

To preserve the editability of the grid's data source, to support any requirement you can add additional computed columns that are read-only. This comes in handy to calculate the appropriate `PERCENT` value to pass to the template component to show where each employee's salary falls in the range of all salaries.

Choose **Create Column** on context menu of the Interactive Grid's nested **Columns** node in the component tree. Use the Property Editor to name it `PERCENT`. Set it's column type to **Hidden**. Its **Source** > **Type** defaults to **SQL Expression**. Lastly, set its **Data Type** to `NUMBER` and enter the following into the **SQL Expression** area:

```
ROUND(
  7 + (                      /* start 7 units from left edge  */
    (SAL - MIN(SAL) over ()) /* distance from the minimum     */
    /
    NULLIF(                  /* divided by total range        */
      MAX(SAL) over ()       /* guarding against              */
    - MIN(SAL) over (), 0)   /* divide-by-zero                */
    *
    86                       /* scale to 86-unit drawing area */
  )
)
```

This SQL formula converts a salary (`SAL`) into a position on a scale from 7 to 93. It works in three steps:

Copyright © 2026, Oracle and/or its affiliates.

1. **Find where this salary sits relative to the range**: subtract the lowest salary from the current salary, then divide by the difference between the highest and lowest. This gives a number between 0 and 1, where 0 means "at the bottom" and 1 means "at the top".
2. **Stretch that onto an 86-unit scale**: multiply by 86, the width of the drawable area inside the graphic.
3. **Shift it 7 units in from the left**:add 7 to keep the circle indicator from being cut off at the edges.

`MIN(SAL) over ()` and `MAX(SAL) over ()` are SQL window functions. The `over ()` part with empty parentheses means "look across all rows in the result set". So `MIN(SAL) over ()` gives the single lowest salary across all employees, repeated on every row. Without the `over ()` you would need a `GROUP BY`, which would collapse the rows and lose the per-employee detail.

The `NULLIF(..., 0)` function prevents a divide-by-zero error in the rare case where all values in the range are identical. The `ROUND()` removes any decimal places since SVG coordinates work best as whole numbers.

> **Tip**
>
> To use this technique for a different `NUMBER` column, replace every reference of `SAL` with your column name.

Displaying a Template Component in a Grid Column

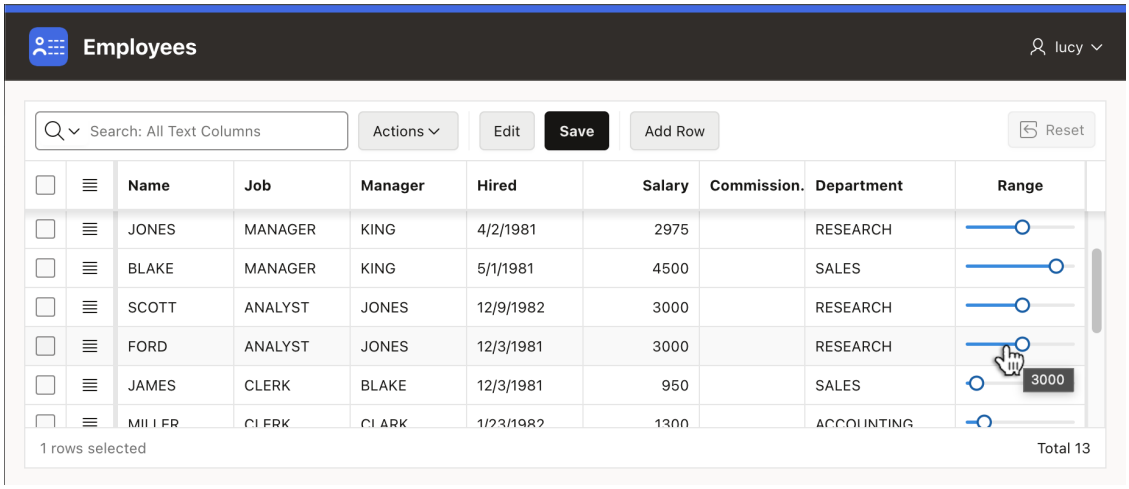
Now add a column to represent the salary range indicator. Repeat the **Create Column** step above. Since this column doesn't need a *SQL Expression* and doesn't come from a Database Column, set its **Source** > **Type** to **None**. Configure the column type to be an **HTML Expression**, and enter the four lines below into the **HTML Expression** area. This uses the `{with/}…{apply/}` template directive to include the `PERCENT_DOT_ON_BAR` template component, passing the value of the current row's `SAL` column for the `LABEL` attribute and the value of its `PERCENT` column for the component's `PERCENT` attribute.

```
{with/}
    LABEL   := &SAL.
    PERCENT := &PERCENT.
{apply PERCENT_DOT_ON_BAR/}
```

This combination produces the result shown in the figure below, where each employee's salary renders as a percent dot on a bar, proportional to the range of all employees salaries. Hovering the mouse over an individual bar like the one for *FORD*, the tooltip shows the salary amount you passed as the template component's `LABEL` attribute. In *FORD*'s case, that is `3000`.

Copyright © 2026, Oracle and/or its affiliates.

Figure 9-9 Visualizing Salaries as a Dot on a Bar Using a Template Component



Tip

In practice, you *could* have configured the `PERCENT` column directly as an **HTML Expression** instead of **Hidden**, and used the template directive above on it. However, sometimes having separate columns makes the solution easier to understand, and it is generally useful to know you can add grid columns with no source when needed.

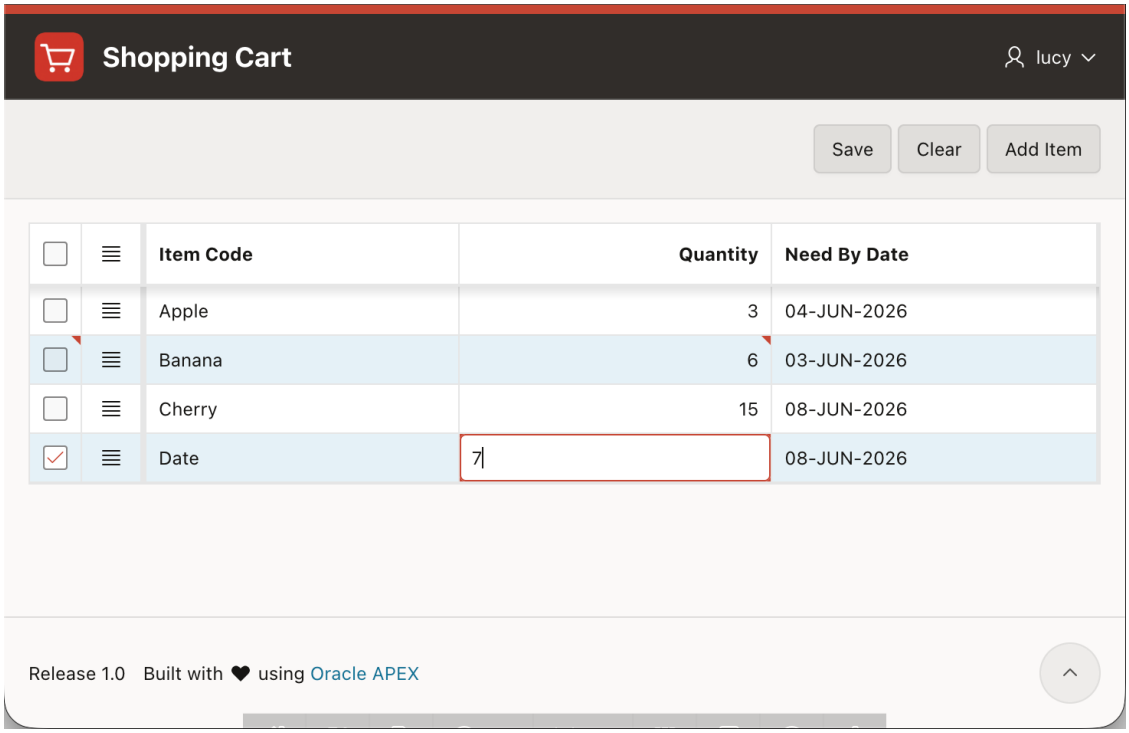
9.2 Editing Collection Data with a Grid

Edit session collection rows in an Interactive Grid using a stable key and custom save code.

Collections are useful for temporary storage of row values in a session. Normally your business logic works with them programmatically, but you can *also* let users directly edit collection members with an Interactive Grid. Consider the use case shown in the figure below: a simple shopping cart holding one or more rows with an item code, quantity, and need by date until the customer is ready to checkout. The item code column can be a Select List for a small number of items, or an inline Popup LOV if the list of products is large. The Need By Date can be a Date Picker.

Copyright © 2026, Oracle and/or its affiliates.

Figure 9-10 Editing Shopping Cart Items in a Collection with Interactive Grid



Interactive Grids use an *Interactive Grid - Automatic Row Processing (DML)* page process to edit a data source with no code. However, editing a collection requires a few lines of PL/SQL to call an `APEX_COLLECTION` API to save the rows instead. After first understanding a few details about how the grid works, and how collections assign their sequence id, configuring this use case is easy.

Changing a Grid Row's Primary Key is Not Allowed

An Interactive Grid manages an in-memory record set in the browser. This is called the *model*. An *editable* grid's model also tracks rows the end user has added, modified, and deleted. When the user submits the page or clicks the grid toolbar's *(Save)* button, the changes in the model are sent to the server where the *Interactive Grid – Automatic DML* page process saves any rows that were created, updated, or deleted.

The client-side model uniquely identifies rows using the values of the grid's *Primary Key* column and assumes those values *won't change*. For rows the user creates, the model generates a temporary unique key in the browser. Then, when the page process saves a new row, the database-assigned primary key replaces the client-assigned temporary value. That's the only key-change the grid allows.

Understanding Why Collection Sequence ID of a Member Row Can Change

Each collection member has a unique system-assigned sequence ID (`SEQ_ID`). It's an integer row index. When a member is added, if it's the first row its `SEQ_ID` is one (`1`), otherwise it's one more than the maximum sequence ID in the collection. As collection members are added, removed, moved, or sorted, using the `APEX_COLLECTION` API, a member's index can change. So, `SEQ_ID` is not the *unchanging* primary key the Interactive Grid requires.

Copyright © 2026, Oracle and/or its affiliates.

Instead, generate a unique key value with SYS_GUID() and store it in the collection in an additional generic VARCHAR2 column. The shopping cart uses the following collection columns, with the ID storing the unique GUID value for each member in C002:

- C001 as ITEM_CODE
- N001 as QUANTITY, and
- D001 as NEED_BY_DATE
- C002 as ID

Simplifying Collection Access with a View and Package

Create a view like following SHOPPING_CART_V to use as your Interactive Grid's data source. This way its column names are self-documenting:

```
create or replace view shopping_cart_v as
select c001 as item_code,
       c002 as id,
       n001 as quantity,
       d001 as need_by_date
  from apex_collections
 where collection_name = 'SHOPPING_CART'
```

Then create a PL/SQL package like the one below to encapsulate the use of the APEX_COLLECTION API. This lets your page's grid work with function and procedure names related to the task at hand. The add_item returns the unique GUID id the grid uses as its unchanging primary key. The update_item and remove_item accept this item id value as the unique identifier as well.

```
create or replace package shopping_cart_api is
    ------------------------------------------------------------
    procedure clear_cart;
    ------------------------------------------------------------
    function add_item(
        p_item_code    in varchar2,
        p_quantity     in number,
        p_need_by_date in date)
        return            varchar2;
    ------------------------------------------------------------
    procedure remove_item (
        p_item_id in varchar2);
    ------------------------------------------------------------
    procedure update_item (
        p_item_id      in varchar2,
        p_item_code    in varchar2,
        p_quantity     in number,
        p_need_by_date in date);
end shopping_cart_api;
```

The corresponding package body appears below:

```
create or replace package body shopping_cart_api is
    c_collection_name constant varchar2(40) := 'SHOPPING_CART';
    ------------------------------------------------------------
```


Copyright © 2026, Oracle and/or its affiliates.

```
procedure ensure_collection
is
begin
    if not apex_collection.collection_exists(c_collection_name) then
        apex_collection.create_collection(c_collection_name);
    end if;
end ensure_collection;
------------------------------------------------------------
procedure clear_cart
is
begin
    ensure_collection;
    apex_collection.truncate_collection(c_collection_name);
end clear_cart;
------------------------------------------------------------
function get_seq_id(
    p_item_id in varchar2)
return number
is
    l_ret number;
begin
    for j in (select seq_id as seq
                from apex_collections
               where collection_name = c_collection_name
                 and c002 = p_item_id)
    loop
        l_ret := j.seq;
    end loop;
    return l_ret;
end;
------------------------------------------------------------
function add_item (
    p_item_code    in varchar2,
    p_quantity     in number,
    p_need_by_date in date)
return varchar2
is
    l_ret varchar2(255) := sys_guid();
begin
    ensure_collection;
    apex_collection.add_member (
        p_collection_name    => c_collection_name,
        p_c001               => p_item_code,
        p_c002               => l_ret,
        p_n001               => p_quantity,
        p_d001               => p_need_by_date
    );
    return l_ret;
end add_item;
------------------------------------------------------------
procedure remove_item (
    p_item_id in varchar2)
is
    l_seq_id number;
begin
    ensure_collection;
```


Copyright © 2026, Oracle and/or its affiliates.

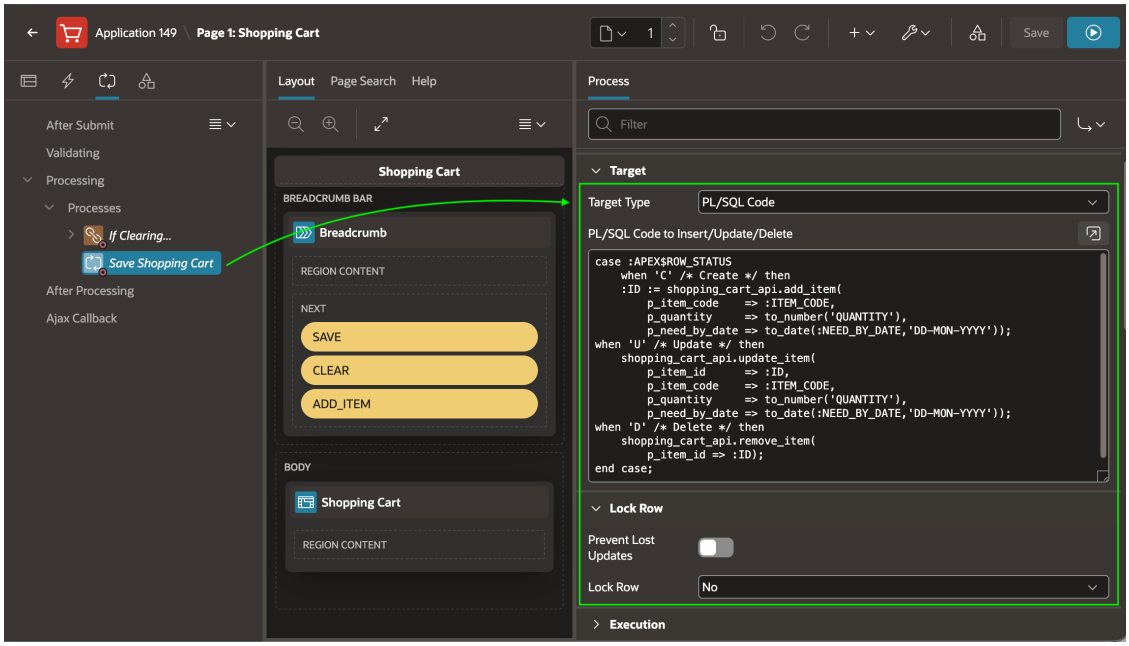
```
        l_seq_id := get_seq_id(p_item_id);
        if l_seq_id is not null then
            apex_collection.delete_member (
                p_collection_name => c_collection_name,
                p_seq             => l_seq_id);
        end if;
    end remove_item;
    ------------------------------------------------------------
    procedure update_item (
        p_item_id      in varchar2,
        p_item_code    in varchar2,
        p_quantity     in number,
        p_need_by_date in date)
    is
        l_seq_id number;
    begin
        ensure_collection;
        l_seq_id := get_seq_id(p_item_id);
        if l_seq_id is not null then
            apex_collection.update_member(
                p_collection_name => c_collection_name,
                p_seq             => l_seq_id,
                p_c001            => p_item_code,
                p_c002            => p_item_id,
                p_n001            => p_quantity,
                p_d001            => p_need_by_date);
        end if;
    end update_item;
end shopping_cart_api;
```

Configuring Interactive Grid to Save Using an API

With the view and package in place, set the Interactive Grid to use the view as its data source and configure its `ID` column as Primary Key. Page Designer adds the related page process to handle saving its rows when you enable editing on the **Attributes** tab in the Property Editor. You can rename the page process to **Save Shopping Cart**. The figure shows it selected in the processing tree. Its type is **Interactive Grid - Automatic Row Processing (DML)**, and its **Editable Region** property is set to the name of the Interactive Grid whose data it saves. As shown, set its **Target Type** to **PL/SQL Code**. The code appears below and uses the `SHOPPING_CART_API` package. You can also disable row locking and lost update protection as highlighted. Since the collection data is private to the current user session, there's no chance another user can lock or inadvertently change the same collection row.

Copyright © 2026, Oracle and/or its affiliates.

Figure 9-11 Configuring Custom Grid PL/SQL to Save Cart Data to Collection



The complete code for the *PL/SQL Code to Insert/Update/Delete* appears below. Since the page process' **Editable Region** property is set to an Interactive Grid, the code runs once for each row that the user created, updated, or deleted. Notice the `CASE` statement uses the built-in bind variable `:APEX$ROW_STATUS` to know if the current row being processed is for create (`'C'`), update (`'U'`), or delete (`'D'`). For the create, it assigns the unique item id GUID returned by the `add_item` function directly to the `ID` primary key grid column using the `:ID` bind variable.

```
case :APEX$ROW_STATUS
when 'C' /* Create */ then
    :ID := shopping_cart_api.add_item(
        p_item_code    => :ITEM_CODE,
        p_quantity     => to_number('QUANTITY'),
        p_need_by_date => to_date(:NEED_BY_DATE,'DD-MON-YYYY'));
when 'U' /* Update */ then
    shopping_cart_api.update_item(
        p_item_id      => :ID,
        p_item_code    => :ITEM_CODE,
        p_quantity     => to_number('QUANTITY'),
        p_need_by_date => to_date(:NEED_BY_DATE,'DD-MON-YYYY'));
when 'D' /* Delete */ then
    shopping_cart_api.remove_item(
        p_item_id => :ID);
end case;
```

Tip

When accessing grid columns of type date or timestamp, convert the `VARCHAR2` value using `to_date` or `to_timestamp` using an appropriate format mask. For columns of `NUMBER` type, use `to_number` instead.

Copyright © 2026, Oracle and/or its affiliates.

Wiring Button to Grid Action Declaratively

The Shopping Cart page's Interactive Grid has its toolbar and footer turned off, so the normal (**Add Item**) toolbar button does not appear. The page saves the grid data using a normal *(Save)* button that submits the page. It clears the shopping cart with another (**Clear**) button that submits the page to trigger an Invoke API page process with When Button Pressed = `CLEAR` that calls `shopping_cart_api.clear_cart`. It adds its own *(Add Item)* button that triggers the grid's built-in selection-add-row action in the browser.

To setup this declarative button behavior:

1. Assign the grid an HTML DOM ID like `shoppingcart`
2. Set the *Action* of the button to *Defined by Dynamic Action*, and
3. Set *Custom Attributes* to have the following text:

```
data-action="[shoppingcart]selection-add-row" data-no-update="true"
```

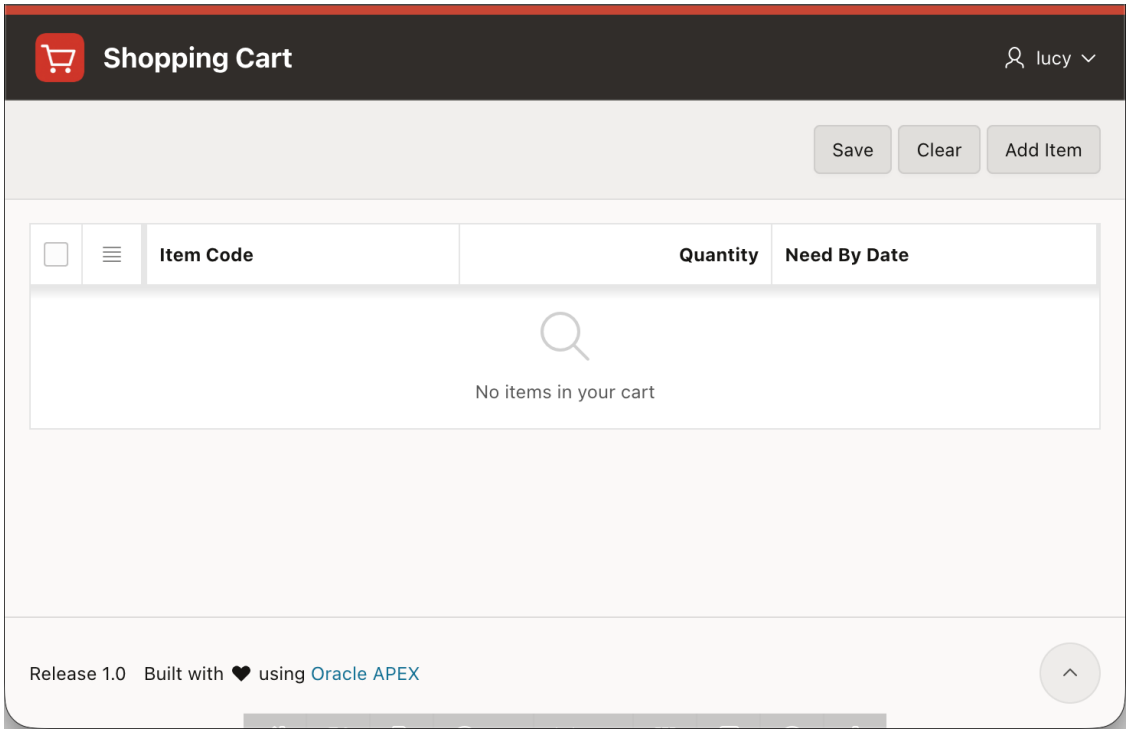
The grid region's HTML DOM ID `shoppingcart` appears in square brackets, and the `selection-add-row` action name follows. The same approach works for any of the built-in actions explained in the *Actions* section of the Interactive Grid's JavaScript API reference documentation. The `data-no-update` custom attribute asks the `selection-add-row` action to ***not*** automatically update your declaratively wired button at runtime to use the action's label and icon.

Customizing the No Rows Found Message

By default, a grid region with no rows will display the text *No data found*. To configure a more user-friendly message specific to what the grid is showing in your app, set its **When No Data Found** message on the **Attributes** tab of the Property Editor. For example, if you set it to "*No items in your cart*", then an empty shopping cart looks like the figure below.

Copyright © 2026, Oracle and/or its affiliates.

Figure 9-12 Custom When No Rows Found Message Improves Usability



Processing the Shopping Cart Contents

At checkout time, to process the shopping cart rows, your app can use code like the following using the view you create to give your collection members more meaningful names:

```
for j in (select item_code, quantity, need_by_date
            from shopping_cart_v)
loop
    -- Reference j.item_code, j.quantity, j.need_by_date
end loop;
```

9.3 Understanding and Customizing Grid Behavior

Learn how Interactive Grid works at runtime and how to use JavaScript APIs to customize its setup, actions, events, selection, and data refresh behavior.

Understanding Grid Behavior and Save Processing

The Interactive Grid region is a *model-view* component: a client-side model manages records, metadata, and change state; multiple views present and edit that model; and actions, events, and widget methods let developers invoke commands and respond to user interactions.

Toolbar buttons and menu items are wired to named Interactive Grid actions. When a user triggers one of these actions, the Interactive Grid widget coordinates the appropriate response, such as refreshing data, entering edit mode, adding a row, opening a dialog, or saving changes.

The Interactive Grid can be read-only, or enabled for editing. Each editable one's client-side model tracks inserted, updated, and deleted records. When a grid's *Save* action is triggered, it

Copyright © 2026, Oracle and/or its affiliates.

saves its *own* pending changes without refreshing the page. When the user submits the page, pending grid changes can be processed along with form region items, validations, and other page processes. Each grid's associated *Interactive Grid - Automatic Row Processing (DML)* page process handles saving its changes. This lets a master form and one or more detail grids be validated and saved in a single transaction. When using page submit to save grid data, you can optionally *hide* the grid's own (**Save**) toolbar button by turning off that feature in the Property Editor.

Customizing Grid Behavior with JavaScript

The *Oracle APEX JavaScript API Reference* covers the grid's rich set of client-side APIs. Developers commonly use them to:

- Configure high-level grid features and default view options.
- Customize the toolbar by adding, removing, or rearranging its controls.
- Configure built-in actions or add custom actions.
- Invoke grid commands such as save, refresh, edit, or open a built-in dialog.
- Respond to grid events such as selection, save, report, view, or mode changes.
- Work with selected records and model data, including refreshing a single row.

Use the Interactive Grid's *JavaScript Initialization Function* for settings that must be in place before the grid is created. Examples include:

- Setting initial grid options for features, toolbar layout, report settings, selection, editability, pagination, and default view or model behavior.
- Using `initActions` to add, hide, disable, relabel, or configure actions before the toolbar and menus are wired up.

Use `apex.util.getNestedObject` when setting initialization options inside objects like `defaultGridViewOptions` or `defaultModelOptions`. This lets your code change the values it cares about without accidentally discarding other settings that might already be in place. After the grid exists, you can use region, grid, actions, model, and event APIs. Examples include:

- Invoking named actions such as `save`, `edit`, or `show-filter-dialog`.
- Getting or setting selected records.
- Reacting to events such as selection change, mode change, save, report change, view change, or view model create.
- Using the current view and model APIs to read or update record values.
- Calling region, grid view, or model methods to refresh data, navigate to a cell, move focus, or refresh a specific row after a dialog closes.

Initialization code shapes the grid before it is created; runtime code works with the grid after it exists. Although `initActions` runs during initialization, the actions it configures can be triggered later by users or invoked later by JavaScript code.

Studying Some Simple Customization Examples

Suppose you want your grids to be able to:

- Hide the footer when not needed
- Let users select multiple rows in the grid
- Keep selected rows across page-by-page pagination

Copyright © 2026, Oracle and/or its affiliates.

- Automatically maintain primary keys of selected rows in a hidden page item.

Each requires setting just a single property in the grid's *JavaScript Initialization Function*:

- Hide the footer – **`footer = false`**
- Enable multiple row selection – **`multiple = true`**
- Track selected keys – **`selectionStateItem = "P2_SELECTED_EMPNOS"`**
- Keep selected rows across pagination – **`persistSelection = true`**

You can add one or more of these settings to a small initialization function: just add the property assignments you need. You define it in the **JavaScript Initialization Function** area on the ***Attributes*** tab of the Interactive Grid region. For example, to enable a selection state item in a grid with multi-row selection, the necessary code appears below. Notice the function receives an `options` object as a parameter from the grid. It uses the `getNestedObject()` utility function to access the default grid view settings object, then assigns two properties on it. Finally, it returns the same `options` object so APEX initializes the grid with the modified settings.

```
function( options ) {
    const gridOptions = apex.util.getNestedObject(
                            options,"defaultGridViewOptions");
    gridOptions.selectionStateItem = "P2_SELECTED_EMPNOS";
    gridOptions.multiple = true;
    // gridOptions.footer = false;
    // gridOptions.persistSelection = true;
    return options;
}
```

Surgically Refreshing an Edited Row

The Interactive Grid's model API has `getRecord()` and `fetchRecords()` functions. Use `getRecord()` to find a record by primary key in the client-side model. Call `fetchRecords()` to refresh one or more *specific* records with the latest data from the server. You can combine them to write a `refreshGridRowOrRegion()` helper function to add to a page's **Function and Global Variable Declaration**. The code appears below. It uses `apex.region()` to find the region by HTML DOM ID, then validates the region type to ensure it's working with an Interactive Grid. If the caller provides a *`primaryKey`*, it calls `getRecord()` and `fetchRecords()` to surgically refresh that record in the client-side model. Conversely, if the key is not provided it refreshes the whole grid instead.

```
function refreshGridRowOrRegion( regionHtmlDomId, primaryKey ) {
    const region = apex.region( regionHtmlDomId );
    // Ensure the static id identifies an Interactive Grid region
    if ( !region || region.type !== "InteractiveGrid" ) {
        throw new Error( "The region must be an Interactive Grid" );
    }
    // If we have a primary key that was updated, refresh just that row
    if ( primaryKey ) {
        const gridView = region.call( "getViews", "grid" );
        const model = gridView.model;
        const record = model.getRecord( primaryKey );
        // If we find the record in the model, refresh just that row
        if ( record ) {
            model.fetchRecords( [ record ] );
            return;
```


Copyright © 2026, Oracle and/or its affiliates.

```
        }
    }
    // Inserts and deletes require a full grid refresh
    // Although refresh() API supports "maintain pagination" flag,
    // in APEX 26.1, it only applies to Interactive Report regions
    region.refresh();
}
```

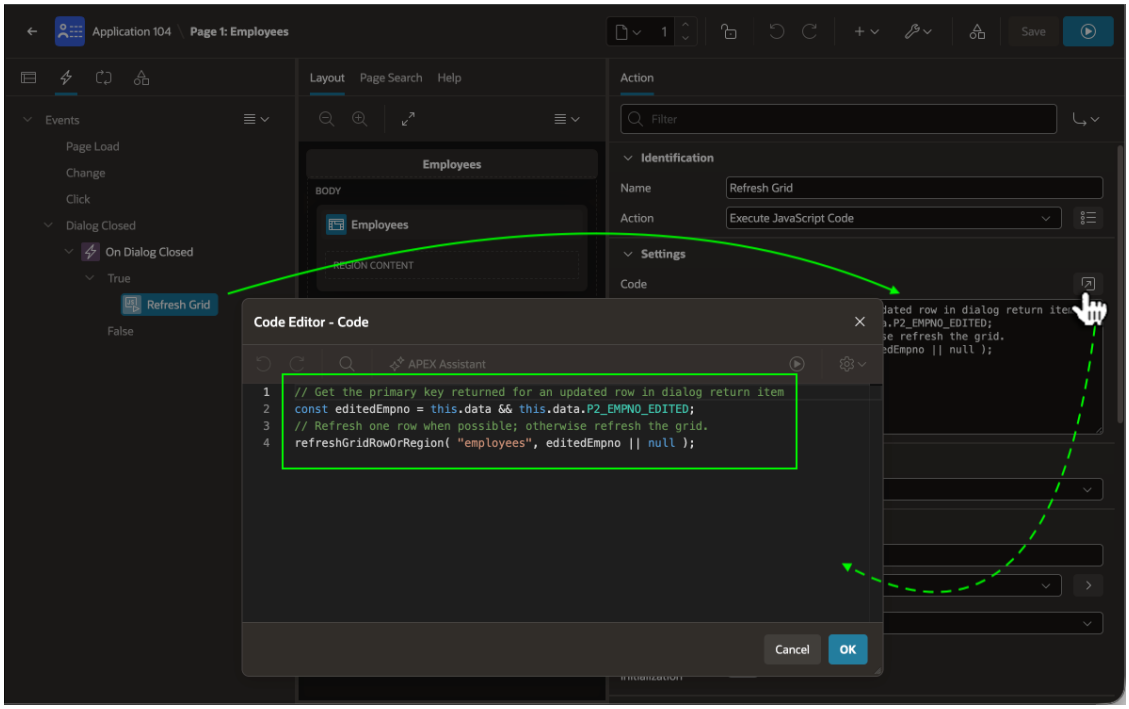
Using the Refresh Row Helper with a Modal Editing Dialog

In the calling page with an Interactive Grid of employees, to react to any dialog closure, add a dynamic action event handler on the *Dialog Closed* event targeting the `body` jQuery selector. This fires when the user closes the modal page that lets users create, edit, or delete an employee. In a dynamic action step of type *Execute JavaScript Code*, call the helper function to refresh the row or grid, passing in the primary key of the row that was edited. As explained in the next section, you can create your modal page to return `null` if a row was created or deleted. In those cases, the function refreshes the *full* grid instead of just the edited row. The two lines of code appear below. The first line gets the dialog return item `P2_EMPNO_EDITED` returned by modal dialog page. The second line calls `refreshGridRowOrRegion()`, passing `null` as a fallback if the dialog didn't return a primary key that was edited.

```
// Get the primary key returned for an updated row in dialog return item
const editedEmpno = this.data && this.data.P2_EMPNO_EDITED;
// Refresh one row when possible; otherwise refresh the grid.
refreshGridRowOrRegion( "employees", editedEmpno || null );
```

The figure shows these two lines of dynamic action JavaScript in the Code Editor. This handy window is one click away from the Property Editor whenever you want a more developer-friendly editing experience. It features code completion and syntax highlighting for JavaScript, PL/SQL, SQL, HTML, and CSS.

Figure 9-13 Calling Helper Function to Refresh Grid Row on Dialog Close

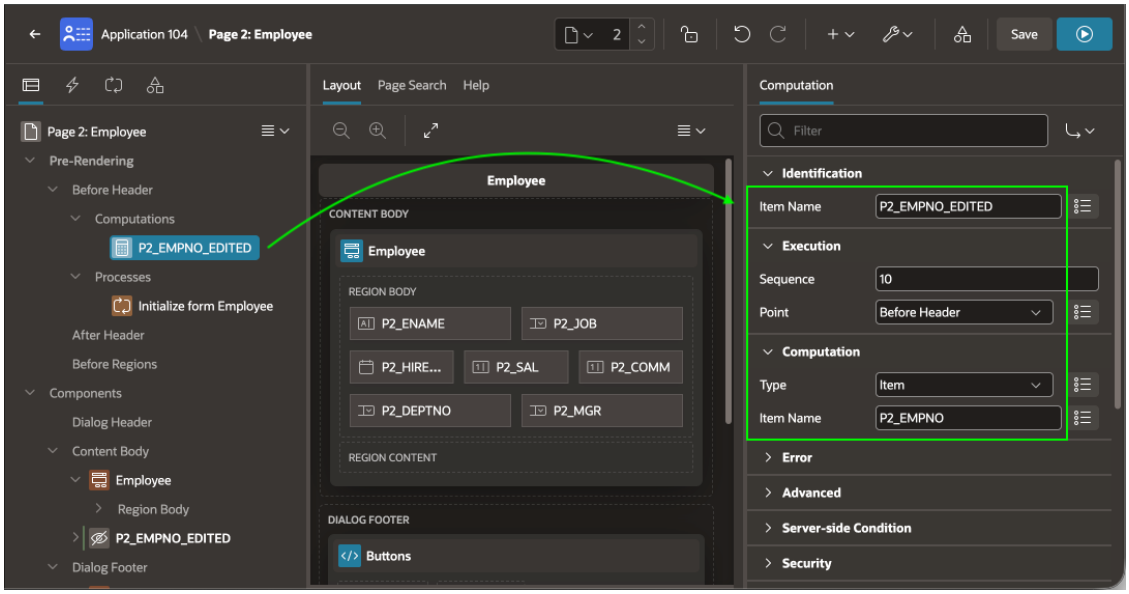


Copyright © 2026, Oracle and/or its affiliates.

Configuring Modal Dialog to Return Edited Primary Key or Null Otherwise

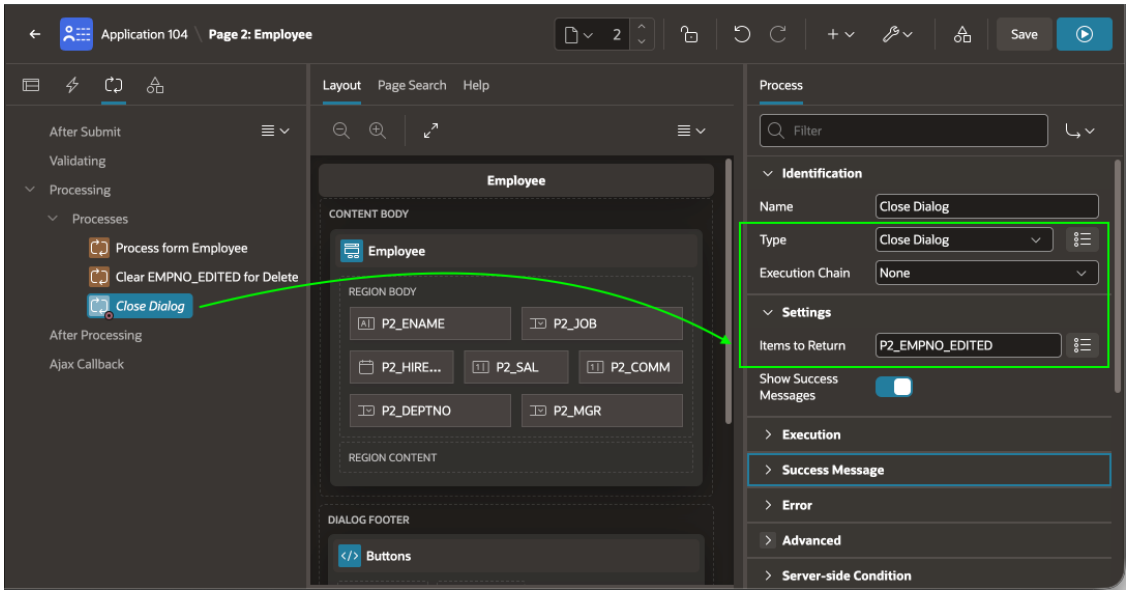
In the modal dialog page, as shown below, a Pre-Rendering computation sets the value of a hidden `P2_EMPNO_EDITED` page item to the `P2_EMPNO` item value passed in for the edit case.

Figure 9-14 Storing Primary Key Being Edited in a Hidden Item with a Computation



The following figure shows the **Close Dialog** page process returns `P2_EMPNO_EDITED` as a dialog return item by configuring its **Items to Return** setting.

Figure 9-15 Configuring Item Values to Return to Calling Page on Dialog Close

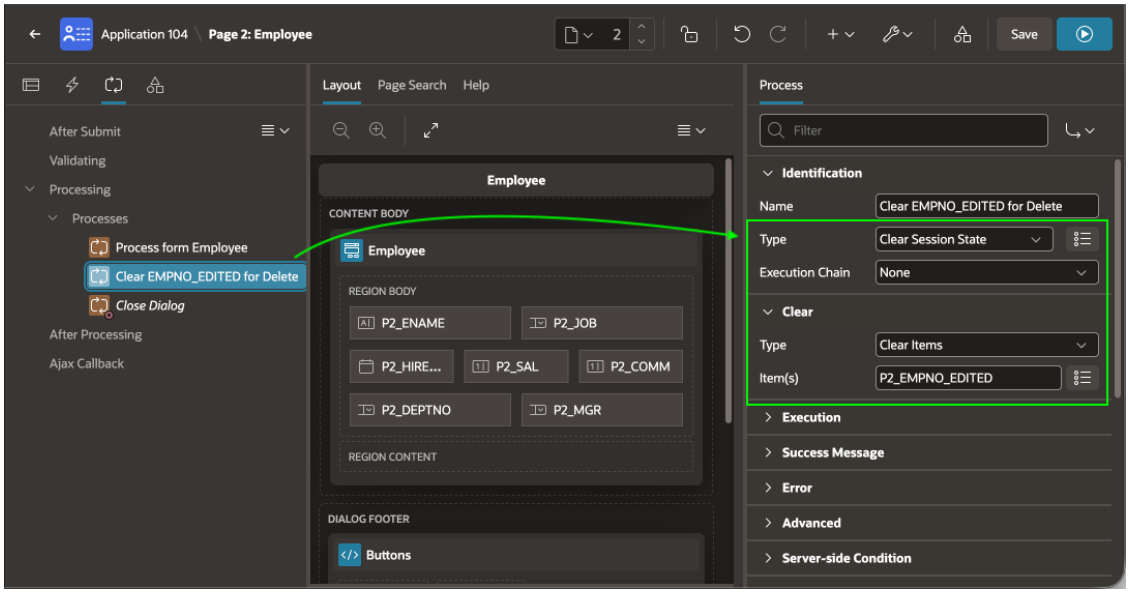


Just before closing the dialog, the figure shows the **Clear EMPNO_EDITED for Delete** page process of type *Clear Session State* clears `P2_EMPNO_EDITED` when the `DELETE` button is

Copyright © 2026, Oracle and/or its affiliates.

pressed. This ensures the Close Dialog handler on the calling page does a full grid refresh if a row is deleted.

Figure 9-16 Clearing the Edited Primary Key on Delete



Finally, the modal dialog can avoid the end user's ever seeing a database referential integrity violation error by showing the (**Delete**) button *only* when the employee being edited is not referenced as another employee's manager. Using the Server-side Condition type of *No Rows Returned*, it configures the following SQL query. If an employee is found with the current employee as manager, then a row is returned and the button does not display.

```
select empno
  from emp
 where mgr = :P2_EMPNO
```

This combination produces an Employees grid page that automatically updates just the edited row on return from the modal dialog.

Copyright © 2026, Oracle and/or its affiliates.

10

Working with Master/Detail Data

Work with related data using stacked grids, side-by-side selection, and drill-down editing pages.

Study three Master/Detail page styles generated by the Create Page Wizard to learn how to create and customize pages that show and edit related data.

- Reviewing Master/Detail Wizard Patterns
  Create master-detail pages using stacked grids, side-by-side selection, or drill-down editing.
- Configuring Master/Detail Grids
  Set up master-detail grids that stay coordinated and save together.
- Showing Master/Detail Data Side by Side
  Create a side-by-side page with a compact master list and selected details with related rows.
- Drilling Down to Edit Master and Details
  Open a selected master row on a page where users can edit both master and detail rows.

10.1 Reviewing Master/Detail Wizard Patterns

Create master-detail pages using stacked grids, side-by-side selection, or drill-down editing.

Enterprise apps often work with related data. An application data model usually contains many examples of tables in a one-to-many relationship. Even the simple EMP/DEPT sample dataset contains two such relationships:

- One department can be the organization unit for many employees
- One employee can manage many other employees

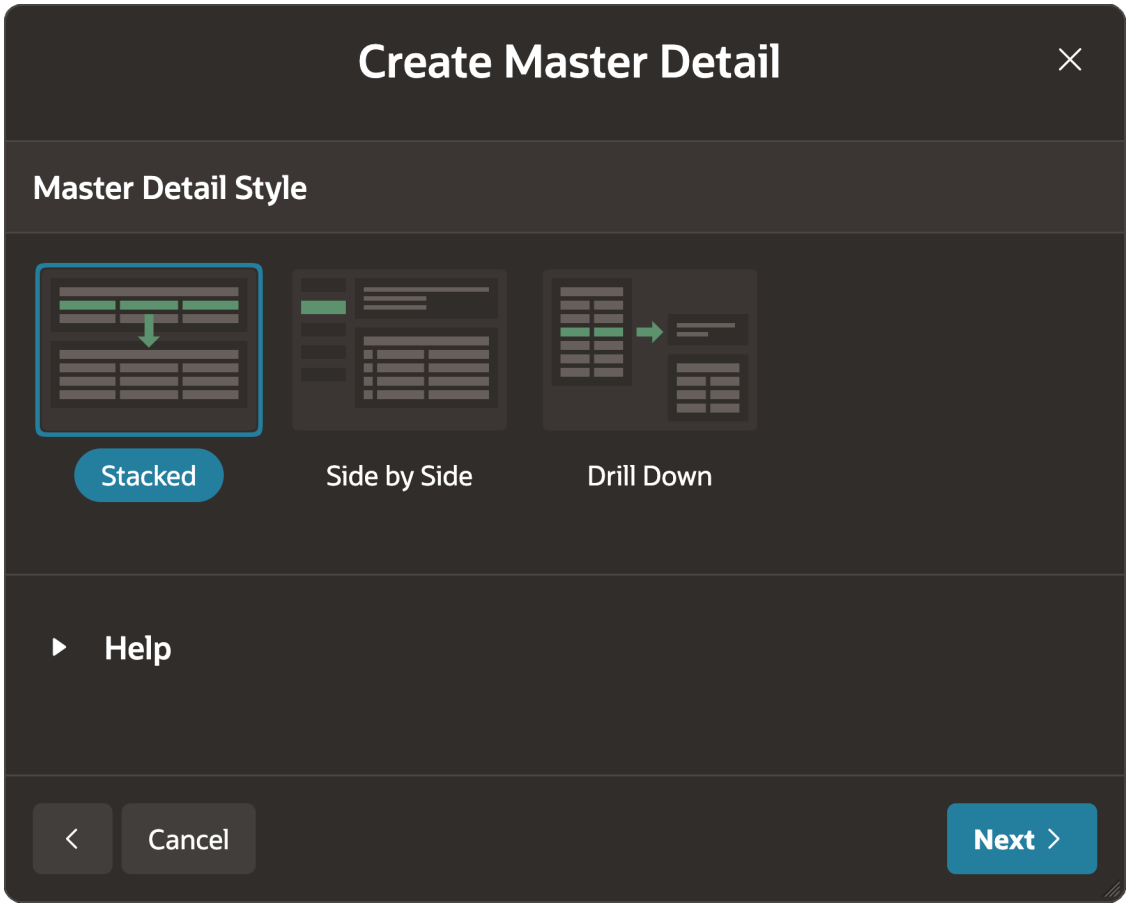
These kinds of one-to-many relationships go by many names, but three familiar ones are *Header/Lines*, *Parent/Child*, and *Master/Detail*.

End users commonly need both master and detail data to complete a task. In the App Builder Create Page Wizard, choose a master-detail interaction pattern, then specify the master table, detail table, and relationship. The figure shows three popular patterns as icons on the wizard's first step:

- **Stacked** - Edit master and detail rows in coordinated Interactive Grids
- **Side by Side** - View more info and detail rows for a master selected from a compact list
- **Drill Down** - Edit master and detail rows for the master row selected on the list page.

Copyright © 2026, Oracle and/or its affiliates.

Figure 10-1 Create Page Wizard Offers Three Master/Detail Patterns



10.2 Configuring Master/Detail Grids

Set up master-detail grids that stay coordinated and save together.

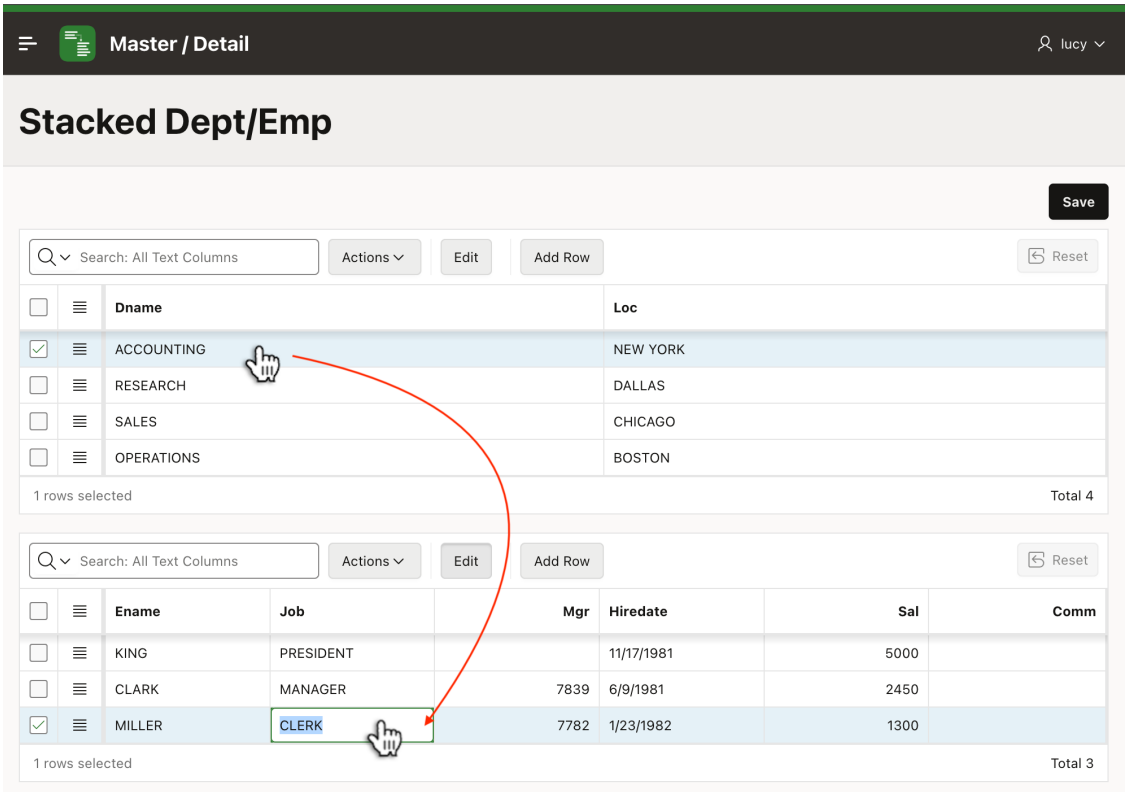
Exercising the Master/Detail Grid Behavior

If you choose the *Stacked* pattern in the Master/Detail wizard, the resulting page has two editable, coordinated Interactive Grids. As shown below, clicking on a row in the master grid refreshes the detail grid to show only rows that reference the selected master row's primary key.

Notice the two Interactive Grid toolbars. Each has an (**Add Row**) button, but no (Save) button. Users can work with departments and employees, but cannot save each grid separately. Instead, the page-level (**Save**) button submits the page so changes from both grids get processed in the same transaction. If validation fails, no edits are saved.

Copyright © 2026, Oracle and/or its affiliates.

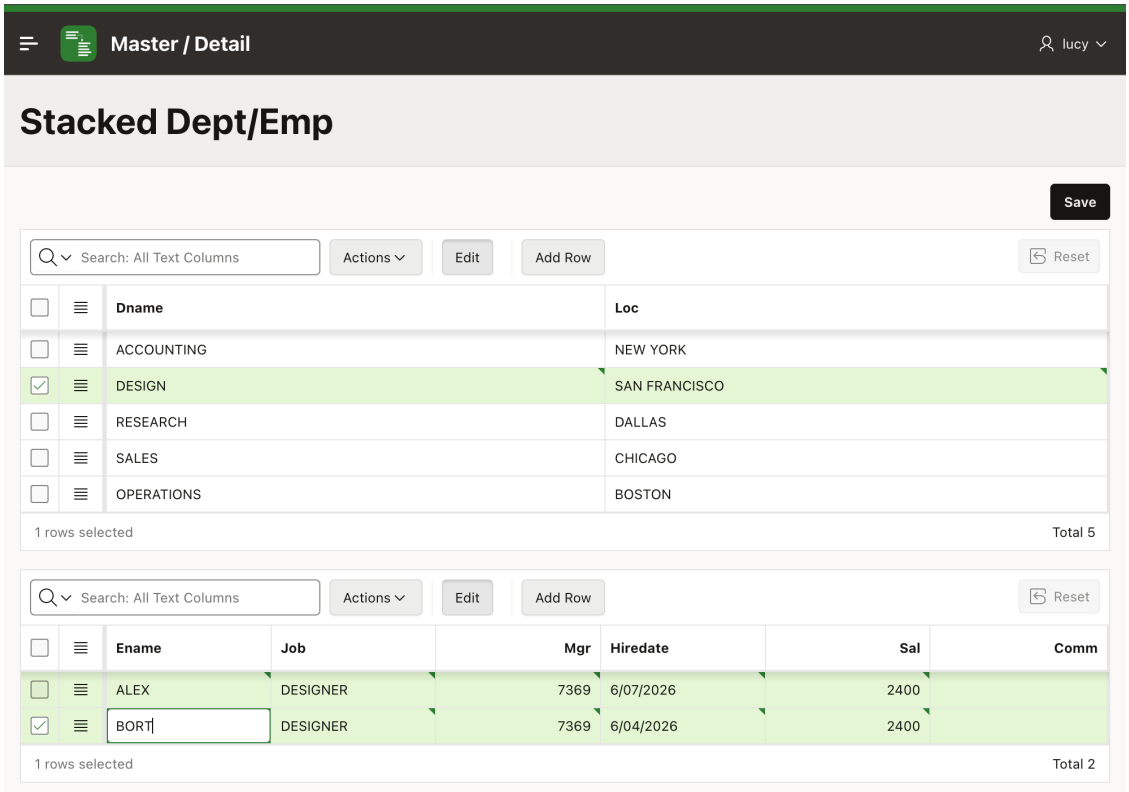
Figure 10-2 Edit Multiple Master Rows and Respective Detail Rows, Then Save



The Interactive Grid's client-side tracking of rows to display and edit automatically handles the master/detail relationship as well. Users can add, update, and delete departments and employees, accumulating changes to many departments and many employees, then save all the changes together. As shown below, the user can even add a new department *DESIGN* and multiple new employees in the new department as part of their changes. The user can continue to navigate through master and detail rows to complete necessary changes, then submit them all by clicking the page-level (**Save**) button.

Copyright © 2026, Oracle and/or its affiliates.

Figure 10-3 Enter New Master Row with New Detail Rows as Part of Other Changes

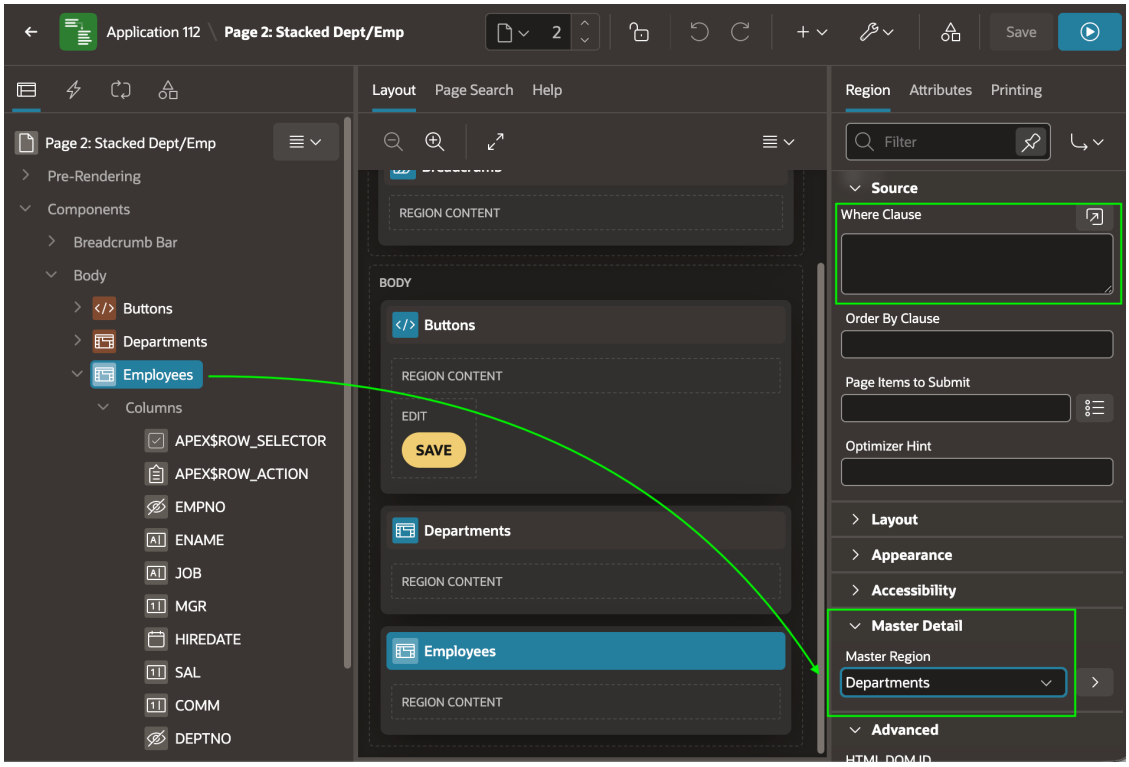


Understanding the Settings that Drive Master/Detail Grids

Two settings on the *detail* grid enable its Master/Detail coordination: **Master Region** and one or more columns with a **Master Column** assigned. First, as shown below, configure the **Master Region** property. Choose the name of the Interactive Grid whose row selection it should react to. Notice there is no need to include a **Where Clause** predicate related to master/detail coordination. The APEX runtime handles that automatically. If you require *additional* filtering, you can add those other criteria as needed.

Copyright © 2026, Oracle and/or its affiliates.

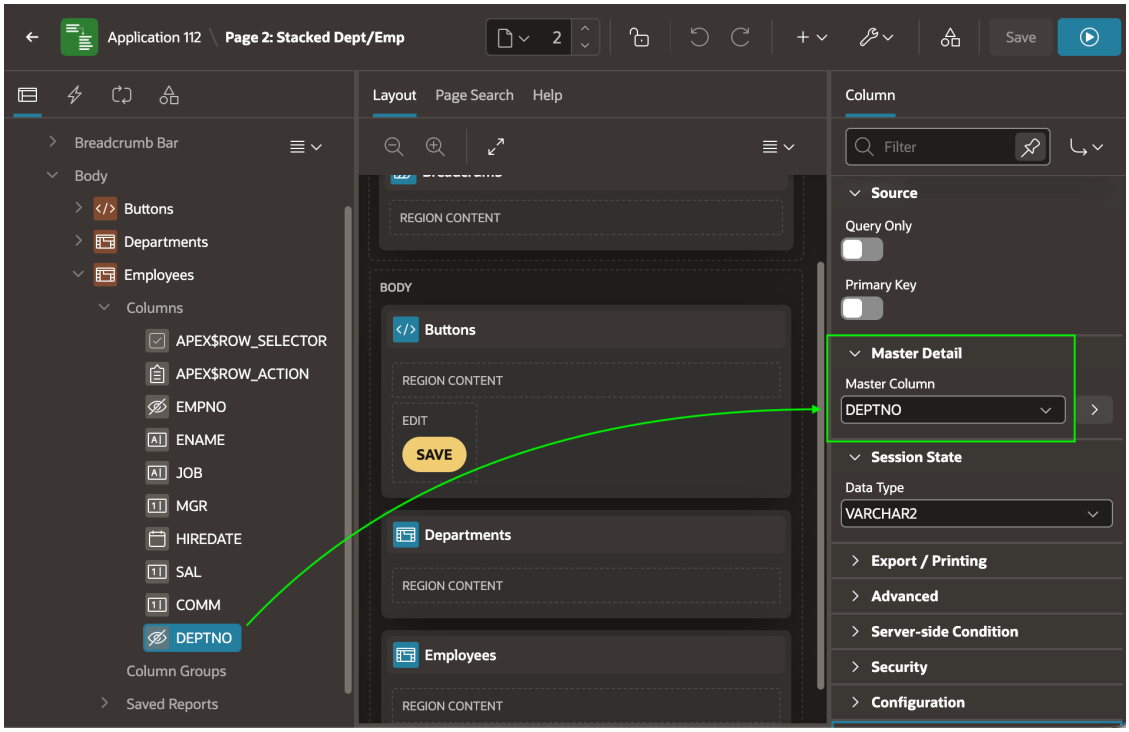
Figure 10-4 Configuring the Master Region of a Detail Grid



With the **Master Region** set on the detail grid, set the **Master Column** property on one or more detail grid columns. As shown below, after selecting the *Employees* grid's `DEPTNO` column, set its **Master Column** to the name of the master region column it relates to. In this case the master column is also named `DEPTNO`. The relationship between the `DEPT` and `EMP` tables only requires a single master column.

Copyright © 2026, Oracle and/or its affiliates.

Figure 10-5 Master Column



> **Tip**
>
> After setting up this parent/child relationship between two grids, you can continue to add a "grandchild" grid by repeating the same configuration. The grids stay coordinated and the user can edit records across multiple relationship levels.

10.3 Showing Master/Detail Data Side by Side

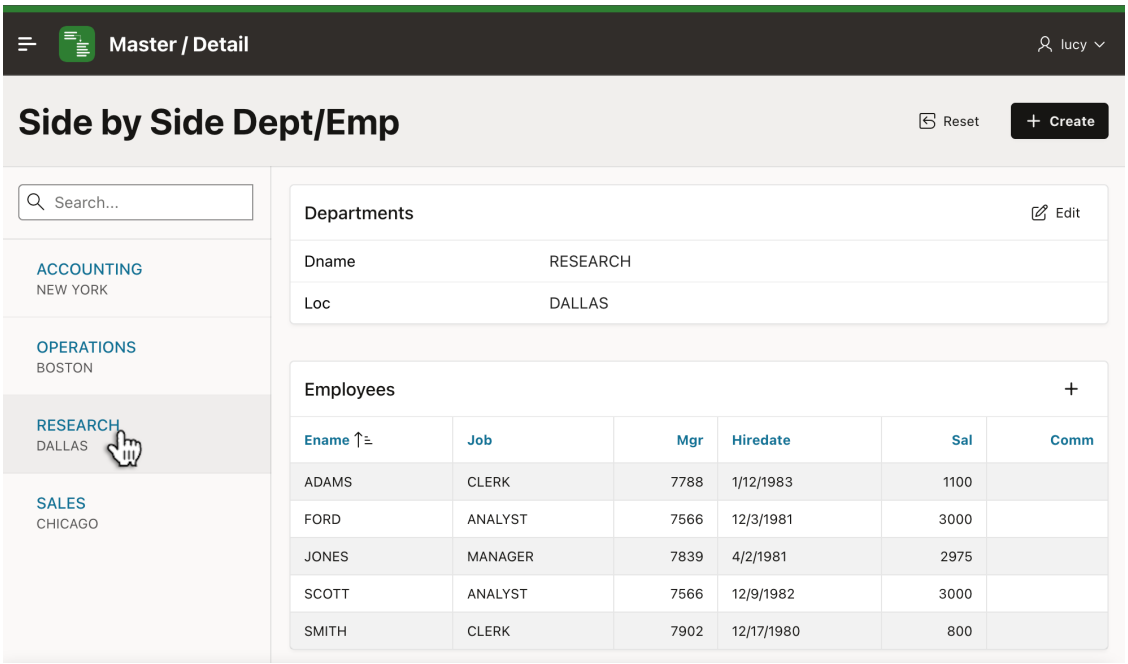
Create a side-by-side page with a compact master list and selected details with related rows.

Understanding the Side by Side Master/Detail Page

A Side by Side Master/Detail page features a compact master-row search experience in the left column. In the wizard, you specify a **Primary Display Column** and **Secondary Display Column** for the master table. Users see both master column values and can search on them. They click any master row in the left column to see more information about it in the body of the page, along with related detail table rows for the selected master.

Copyright © 2026, Oracle and/or its affiliates.

Figure 10-6 Side by Side Master/Detail Lets Users Search and See Details for Master Rows



As shown in the figure below, the page uses the *Left Side Column* template. Its *Left Column* slot has a *Search* Static Content region containing a `P4_SEARCH` field and a *Master Records* Classic Report below it. This report uses the *Media List* theme template to show the primary and secondary display column values of the master rows as a formatted set of links.

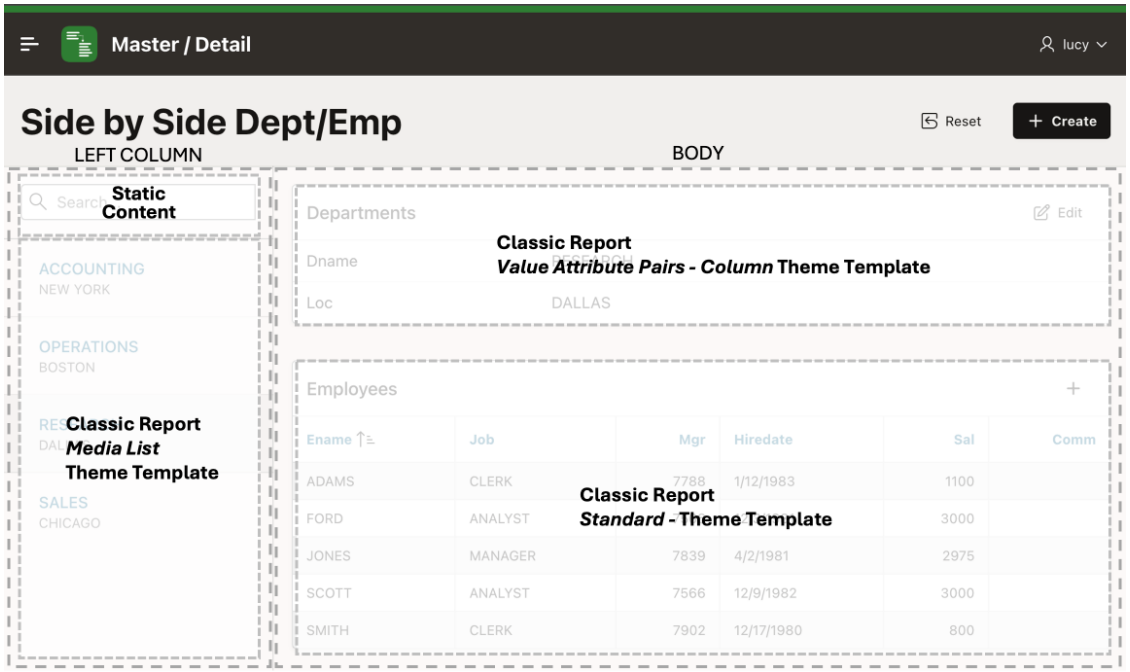
The *Body* slot has a master *Departments* Classic Report using the *Value Attribute Pairs - Column* theme template. This shows all of the columns of the master row in a label/value layout. Below it, an *Employees* Classic Report details region uses the *Standard* theme template for a tabular view of the selected department's employees.

> **Tip**
>
> The Create Page wizard's Master/Detail flow lets you choose *additional* detail tables if needed. They become additional detail repoditrt *regions* in the body of the page and work similarly to the *Employees* region explained here.

The (+ Create) button redirects to a *Departments* modal drawer Form page to enter a new department, and the (+) button redirects to an *Employees* modal drawer form page to enter a new employee in the selected department.

Copyright © 2026, Oracle and/or its affiliates.

Figure 10-7 Breakdown of Side by Side Page Template Slots and Regions in Them



Refreshing Master Records When User Enters a Search

The `P4_SEARCH` field lets users enter text to filter the list of master rows. It uses a *Perform Search* Dynamic Action handler on the *Key Press* event to react when the user presses the [*Enter*] key. The handler only fires for *this* key by using the following JavaScript expression for its **Client-side Condition**:

```
this.browserEvent.which === apex.jQuery.ui.keyCode.ENTER
```

The handler contains two Dynamic Action steps to:

- Refresh the *Master Records* region, then
- Cancel Event

> **Tip**
>
> The *Cancel Event* action prevents a browser behavior dating back to early HTML forms: submitting the page when the user presses [*Enter*] in its only text field.

The query in the *Master Records* region uses `:P4_SEARCH` in its `WHERE` clause below to match on the `DNAME` and `LOC` columns if a search is provided. The region mentions `P4_SEARCH` in its **Page Items to Submit** property as well. This ensures APEX sends the latest value of the search field to the server when refreshing the *Master Records* region.

```
select "DEPTNO",
    ▒
from "DEPT" x
where (:P4_SEARCH is null
```

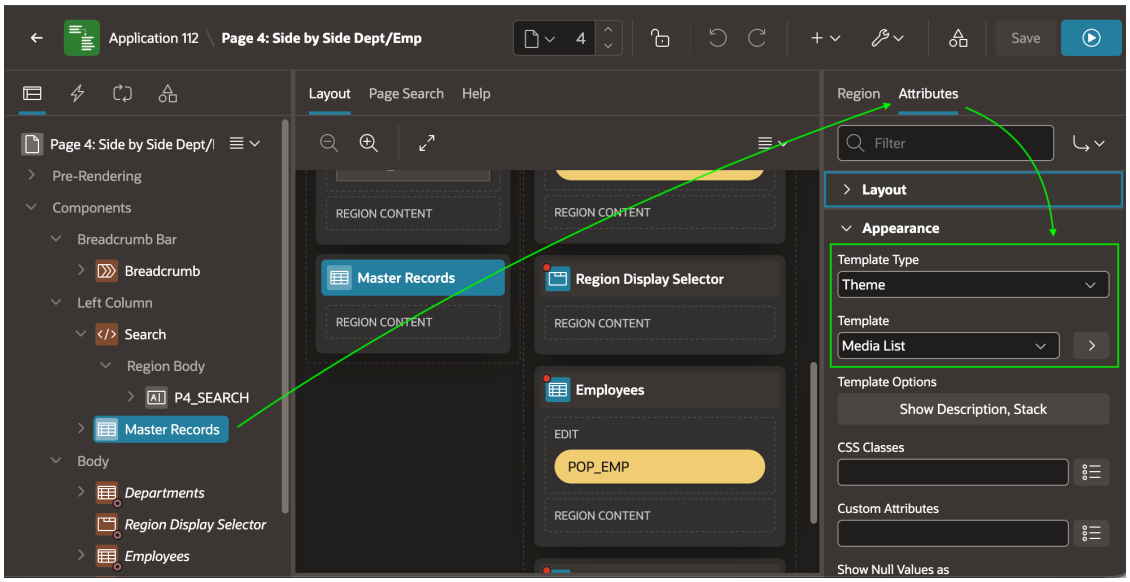

Copyright © 2026, Oracle and/or its affiliates.

```
        or upper(x."DNAME") like '%'||upper(:P4_SEARCH)||'%'
        or upper(x."LOC") like '%'||upper(:P4_SEARCH)||'%'
    )
order by "DNAME"
```

Using a Media List Classic Report

As shown below, the *Master Records* Classic Report region uses a *Media List* report template to format the master rows' primary and secondary display fields as links.

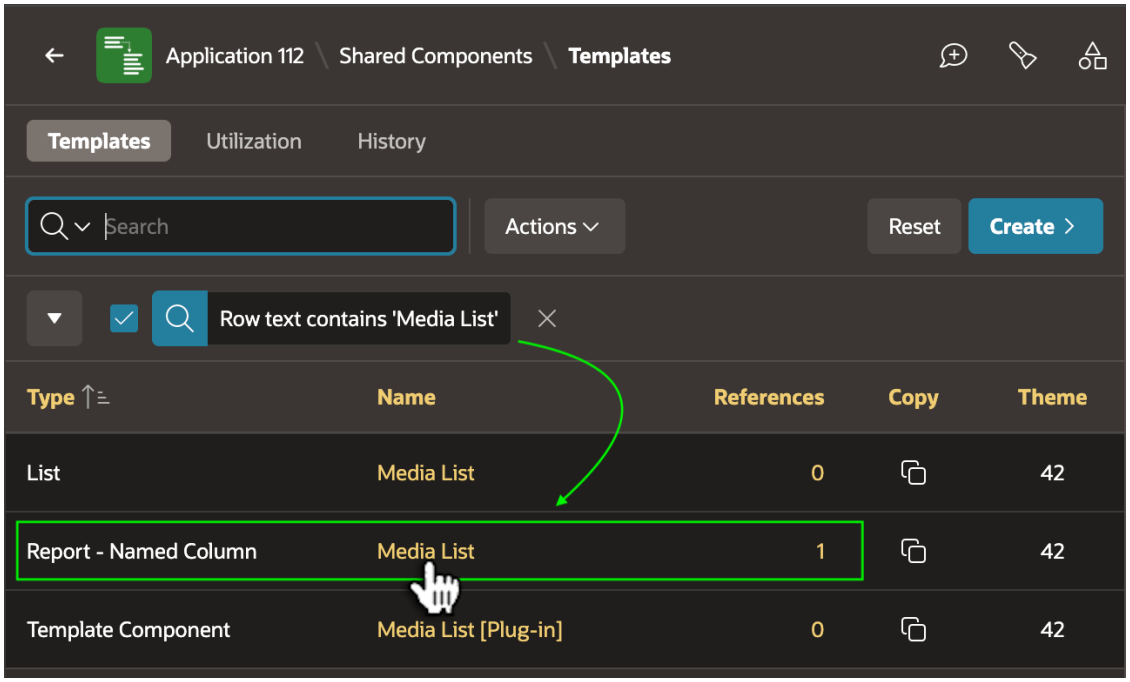
Figure 10-8 Configuring an Alternative Classic Report Template



Whenever you're curious to learn more about what a particular template does, seek it out in App Builder's *Shared Components* > *Templates* page as shown below. Searching for *Media List* reveals it's a single *Report* template of *Named Column* type.

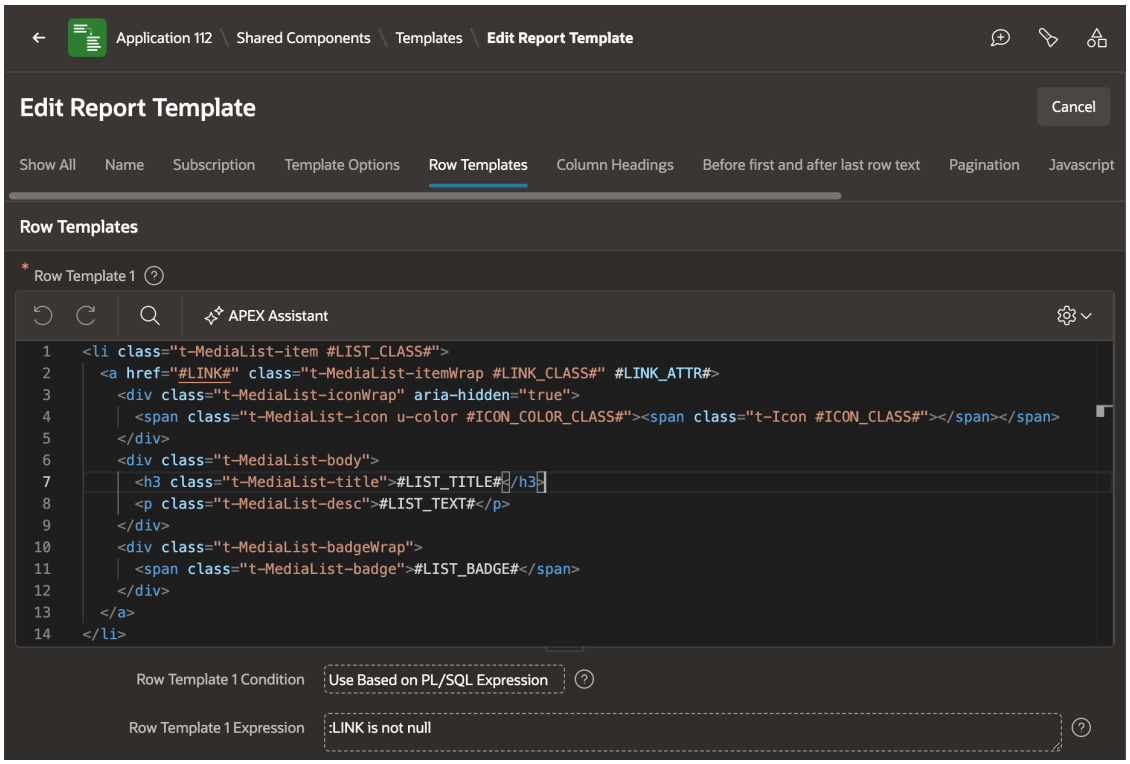
Copyright © 2026, Oracle and/or its affiliates.

Figure 10-9 Find Media List Report Template



A *Named Column* report template specifies the HTML markup that repeats for each row in the data source, referencing specific column names using *#NAME#* substitutions. On the *Row Templates* tab, you find the HTML Markup as shown below.

Figure 10-10 Study Row Template Substitutions



Copyright © 2026, Oracle and/or its affiliates.

Notice the LIST_CLASS, LINK, LIST_TITLE, and LIST_TEXT column substitutions in the row template HTML markup below. The LIST_CLASS is used to style each item in the list. The LINK provides the href URL of the anchor (<a>) tag, to define a link users can click on to select a particular master row. The LIST_TITLE and LIST_TEXT provide the visible text in the list of links.

```
<li class="t-MediaList-item #LIST_CLASS#">
  <a href="#LINK#" class="t-MediaList-itemWrap #LINK_CLASS#" #LINK_ATTR#>
    <div class="t-MediaList-iconWrap" aria-hidden="true">
      <span class="t-MediaList-icon u-color #ICON_COLOR_CLASS#"
      ><span class="t-Icon #ICON_CLASS#"></span></span>
    </div>
    <div class="t-MediaList-body">
      <h3 class="t-MediaList-title">#LIST_TITLE#</h3>
      <p class="t-MediaList-desc">#LIST_TEXT#</p>
    </div>
    <div class="t-MediaList-badgeWrap">
      <span class="t-MediaList-badge">#LIST_BADGE#</span>
    </div>
  </a>
</li>
```

When using a *Named Column* template in a Classic Report, the query needs to contain columns whose names match the template's substitutions you want to supply. The *Master Records* region's query appears below. Notice it selects data from the master DEPT table, but aliases columns to LINK, LIST_CLASS, LIST_TITLE, and LIST_TEXT.

```
select "DEPTNO",
    ▒
    apex_page.get_url(
        p_items  => 'P4_DEPTNO',
        p_values => "DEPTNO") LINK,
    ▒
    case
      when coalesce(:P4_DEPTNO,'0') = "DEPTNO"
      then 'is-active'
      else ' '
    end                         LIST_CLASS,
    (substr("DNAME", 1, 50)||
       (case when length("DNAME") > 50
        then '...'
        else '' end ))          LIST_TITLE,
    (substr("LOC", 1, 50)||
       (case when length("LOC") > 50
        then '...'
        else '' end ))          LIST_TEXT,
    ▒
from "DEPT" x
where (:P4_SEARCH is null
        or upper(x."DNAME") like '%'||upper(:P4_SEARCH)||'%'
        or upper(x."LOC")   like '%'||upper(:P4_SEARCH)||'%'
    )
order by "DNAME"
```


Copyright © 2026, Oracle and/or its affiliates.

Tracking the Selected Master Row

The page contains a hidden `P4_DEPTNO` page item that tracks the primary key of the selected master row. Its **Session State** > **Storage** property is set to **Per Session (Persistent)** so the selected master key value remains available throughout the user's session. Notice the *Master Records* classic report query above includes a `CASE` expression that returns `is-active` for the `LIST_CLASS` column value when the current `DEPTNO` equals the `:P4_DEPTNO` value of the selected master row. Otherwise, the `CASE` statement returns `NULL`. Including this `LIST_CLASS` value in the *Media List* row template markup renders the currently selected master row with a different style than the other rows.

Notice the `LINK` column in the query as well. It calls `apex_page.get_url` to generate the URL to link to the *current* page, by omitting a `p_page` parameter, passing the current master row's `DEPTNO` value as the value of the `P4_DEPTNO` page item. In this example, the Side by Side Master/Detail page is page 4. So, when the user clicks one of the links in the Master Records list, the link redirects to the current page passing the clicked row's primary key. The page assigns the passed value to the hidden `P4_DEPTNO` field and re-renders the current page to show the newly selected master record, along with *its* detailed information.

Using a Value Attribute Pairs - Column Classic Report for Master Row

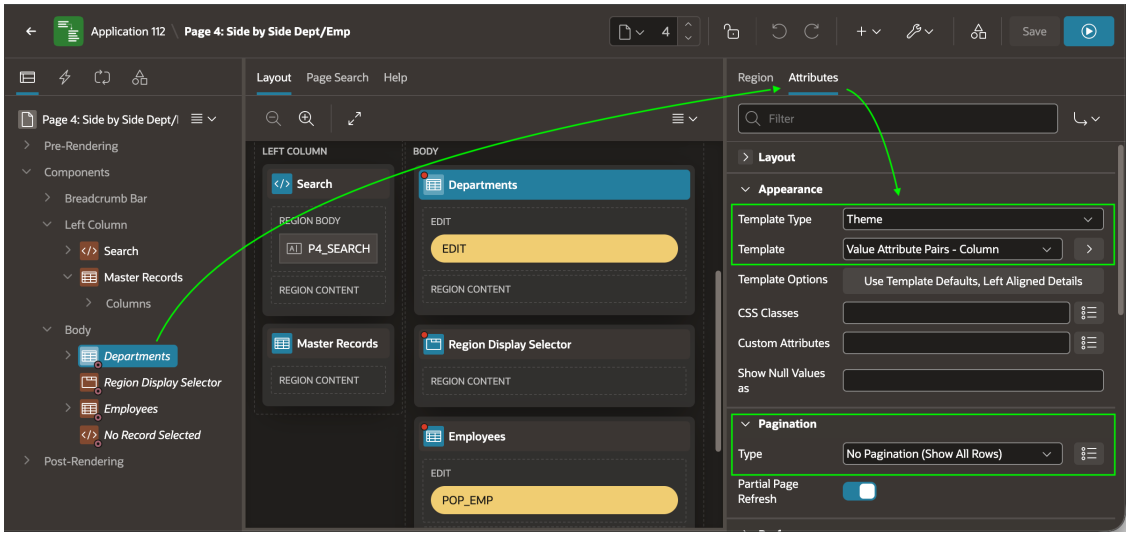
The *Departments* region in the Body slot displays the full information about the selected master row. Its `WHERE` clause references the hidden page item containing the primary key of the selected row:

```
DEPTNO = :P4_DEPTNO /* Filters on DEPT.DEPTNO */
```

It's also configured with a Server-side Condition that *hides* the region if `:P4_DEPTNO` is `null`.

To format the master row's column data in a Label/Value table, as shown in the figure below, the region uses the *View Attribute Pairs - Column* theme template and sets **Pagination** to *No Pagination* since this region will display a single row.

Figure 10-11 Configuring Value Attribute Pairs Classic Report Template



The *Value Attribute Pairs - Column* template is a *Generic Columns* type template. For each row in the report, it repeats the *Column Template* for each *column* in the row, with the ability to

Copyright © 2026, Oracle and/or its affiliates.

include appropriate HTML markup before and after each row. The column template uses generic substitutions `#COLUMN_HEADER#` and `#COLUMN_VALUE#` to include the header and value at an appropriate place in the markup. The result is a Label/Value row for each *column* in the report's data source.

> **Tip**
>
> This one-row master report region shows all columns other than the primary key. For the `DEPT` table, this is only `DNAME` and `LOC`.

Using a Standard Template Classic Report for Detail Rows

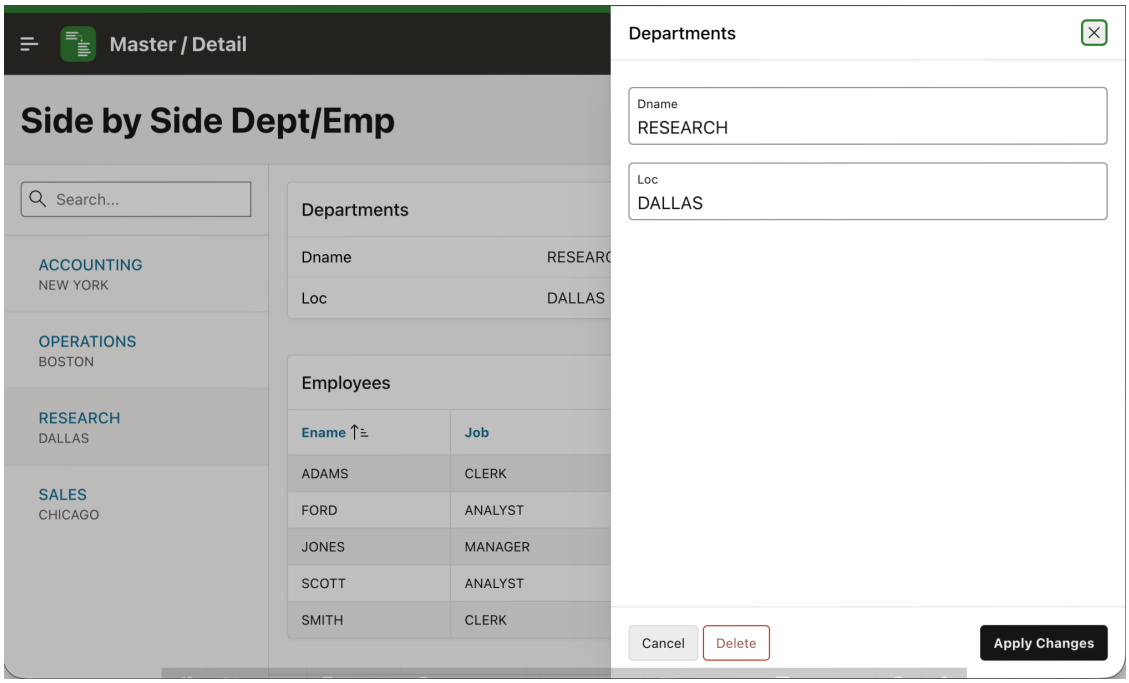
The *Employees* detail Classic Report uses the *Standard* template for a tabular result. It also includes the hidden `:P4_DEPTNO` in its `WHERE` clause. But in this case, it's filtering on the *foreign key* `DEPTNO` column of the detail region's `EMP` table.

```
DEPTNO = :P4_DEPTNO /* Filters on EMP.DEPTNO */
```

Creating and Editing a Master Row

The (**Create**) button in the breadcrumb bar redirects to a *Departments* form page, passing no parameters so it functions as a "Create New Department" page. The (**Edit**) button in the *Departments* region redirects to the same modal drawer page, but passes the value of the selected master record for its primary key page item. This results in presenting the edit page shown below.

Figure 10-12 Editing the Selected Master Row in a Modal Drawer



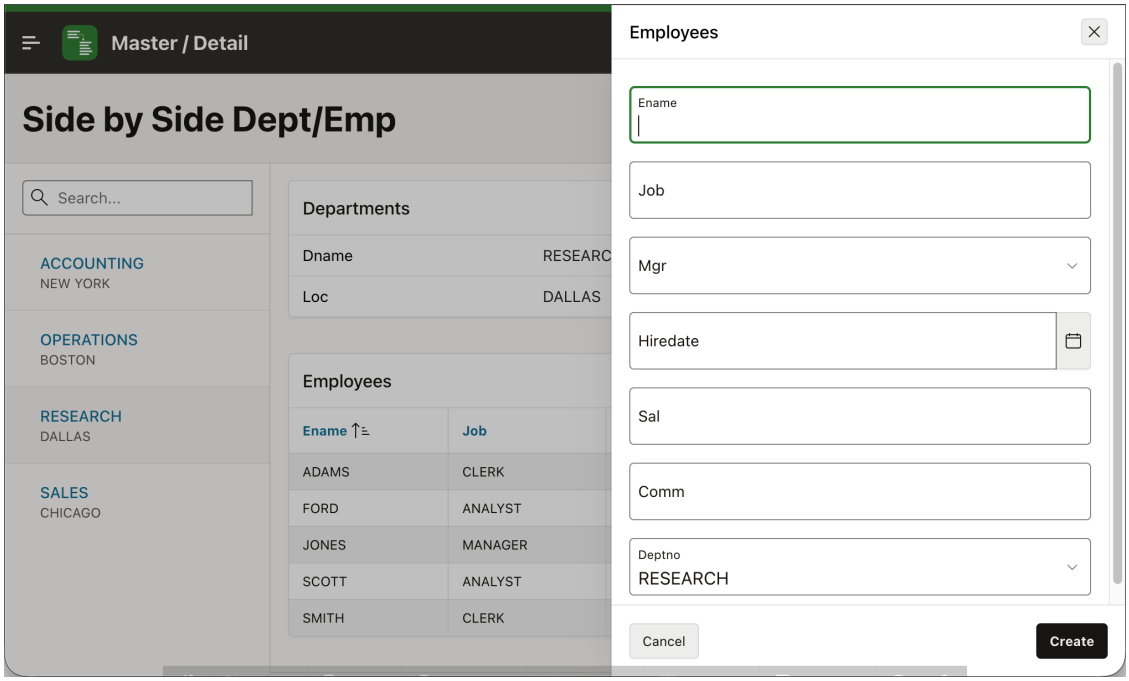
Creating a Detail Row for the Selected Master

The (**+**) button in the details *Employees* region redirects to an *Employees* modal drawer form page to create a new employee, as shown below. Notice the create form opens with the

Copyright © 2026, Oracle and/or its affiliates.

Employees form's **Deptno** field already set to the currently selected master row. This happens because the (**+**) button's link target passes the hidden `P4_DEPTNO` value to the foreign key page item `P7_DEPTNO` in the target form page.

Figure 10-13 Creating a New Detail Row for the Currently Selected Master



Enabling Edit of a Detail Row

By default, the wizard does not configure the page to edit *detail* rows, only to create them. However, doing so takes just a few clicks since you can repurpose the detail form page for editing. For example, you can enable clicking the *Employees* region's **Ename** column value to access the *Employees* edit page for the selected employee row. First, study the target of the existing create button for the detail region to identify the page number of the wizard-generated modal detail form page. Then, change the `ENAME` column type from *Plain Text* to *Link* instead, and configure its **Link** > **Target** to redirect to the same page. Pass the value of the detail row's primary key column as the value of the modal page's primary key page item. Now the page can also *edit* detail rows.

10.4 Drilling Down to Edit Master and Details

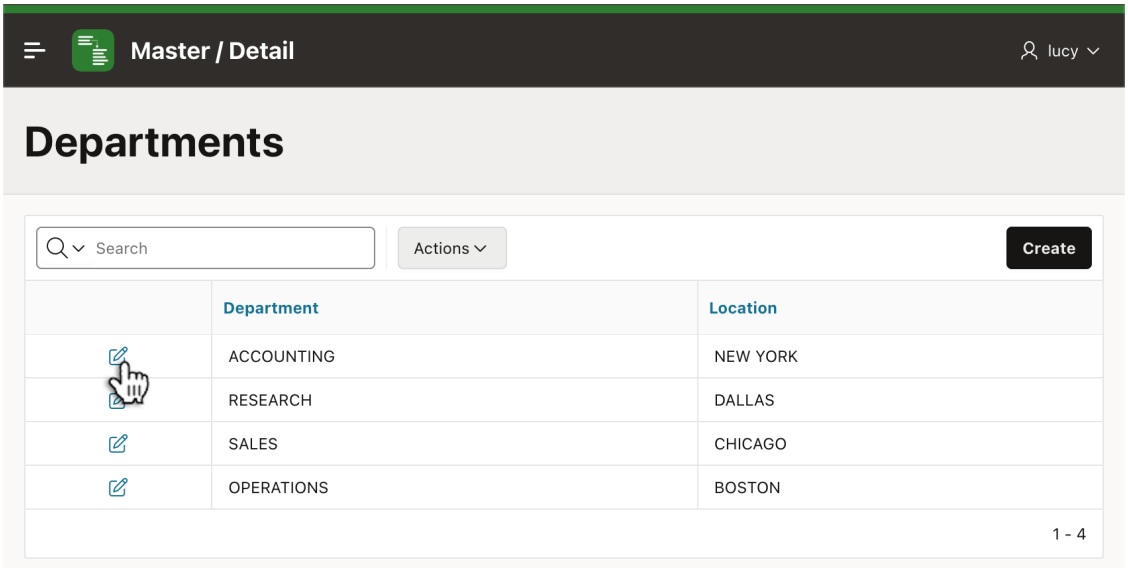
Open a selected master row on a page where users can edit both master and detail rows.

Reviewing the Drill Down Master/Detail Pattern Pages

When choosing the Drill Down Master/Detail pattern, as shown below the wizard creates an Interactive Report page showing master rows, each of which has an edit icon the user can click to navigate to an edit page.

Copyright © 2026, Oracle and/or its affiliates.

Figure 10-14 Master Detail in IR

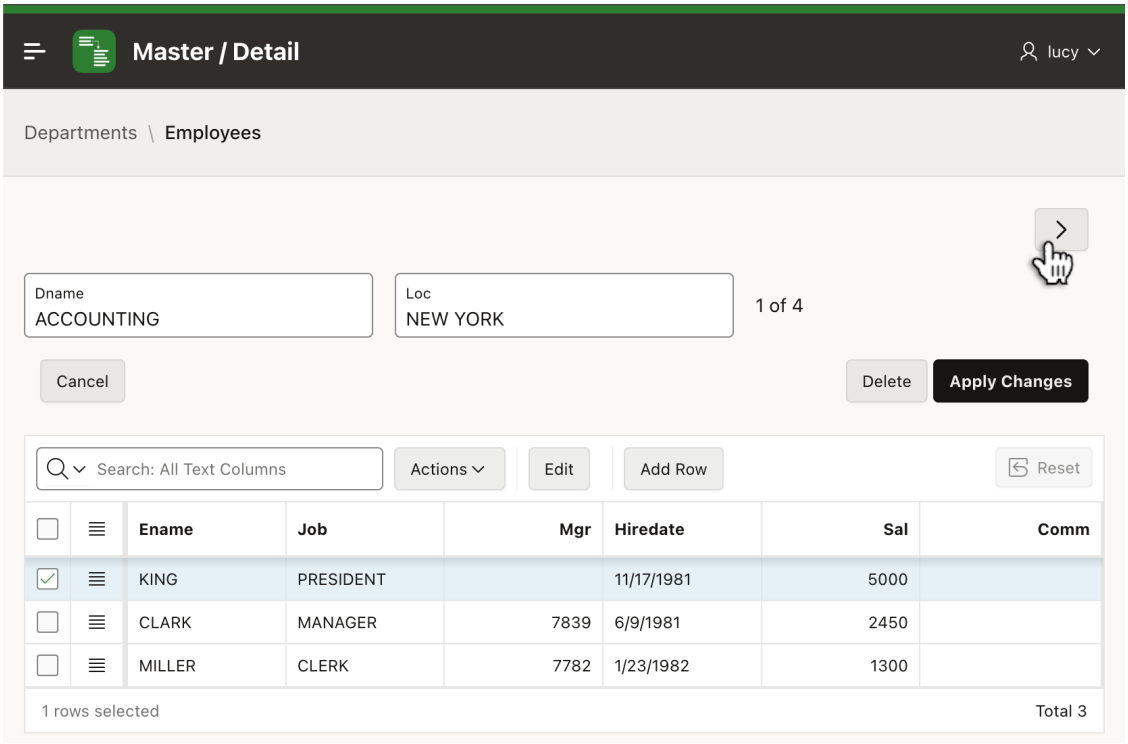


The edit page contains a Form region for the master row, and an editable Interactive Grid for the detail rows related to the current master row. From this page, the user can edit the current Department (*ACCOUNTING*) and all its employees in a single place. Notice the grid's (**Save**) toolbar button is turned off, so the (**Apply Changes**) button that submits that page saves master and details in a single transaction. As shown below, the page also contains next and previous navigation buttons to visit other master rows in department name order.

> **Tip**
>
> To make the master/detail edit page more compact, consider changing the breadcrumb region's template options to enable **Use Compact Style** and the master Form region's template options to set **Header** = *Hidden* and **Style** = *Remove UI Decoration*. The *Employees* edit page screenshots in this section show the effect of these space-saving changes.

Copyright © 2026, Oracle and/or its affiliates.

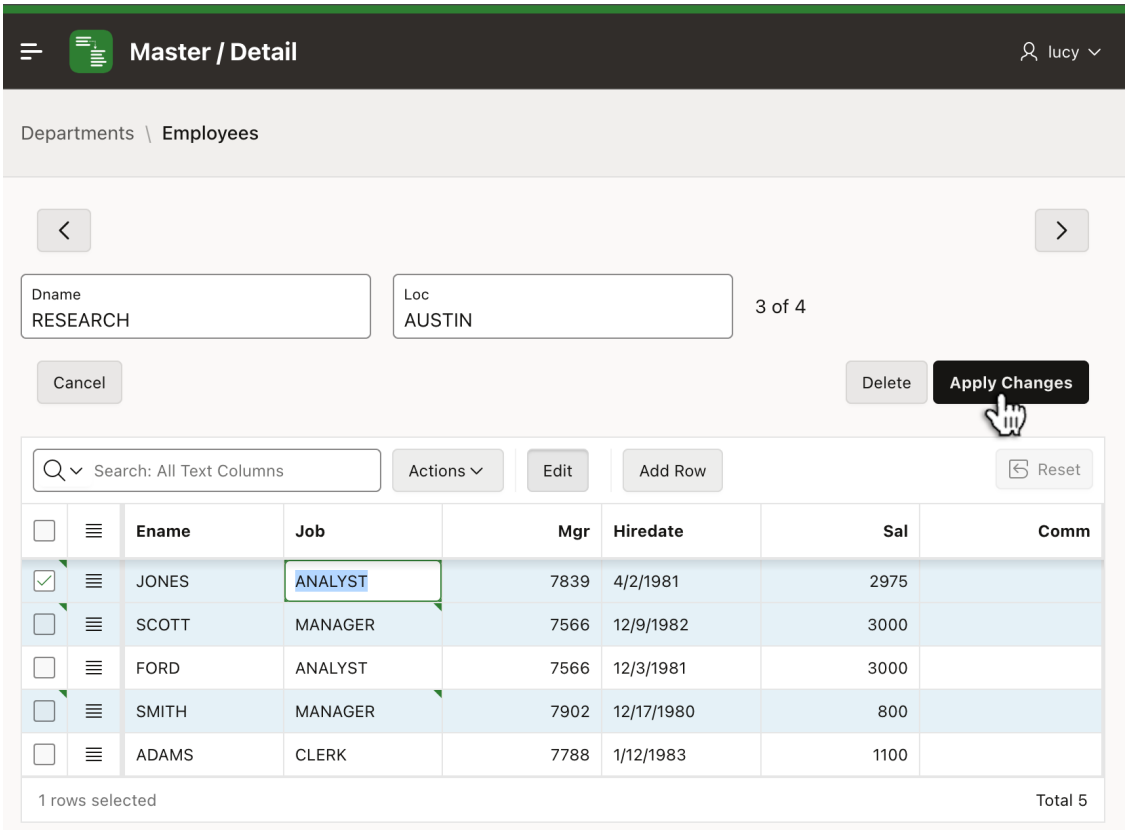
Figure 10-15 Master/Detail Edit Page Lets User Maintain Master and Details Together



For any master row, edit any combination of master and detail data and click (**Apply Changes**) or one of the navigation buttons to save the changes.

Copyright © 2026, Oracle and/or its affiliates.

Figure 10-16 Editing Master and Detail Before Saving or Navigating Previous or Next



Studying the Minimal Detail Grid Configuration Required

The grid uses the detail `EMP` table as its data source, and by default would display all `EMP` rows. So, the first detail to notice is its `WHERE` clause. It's configured to filter for only rows whose `DEPTNO` foreign key matches the primary key of the current master `DEPT` row.

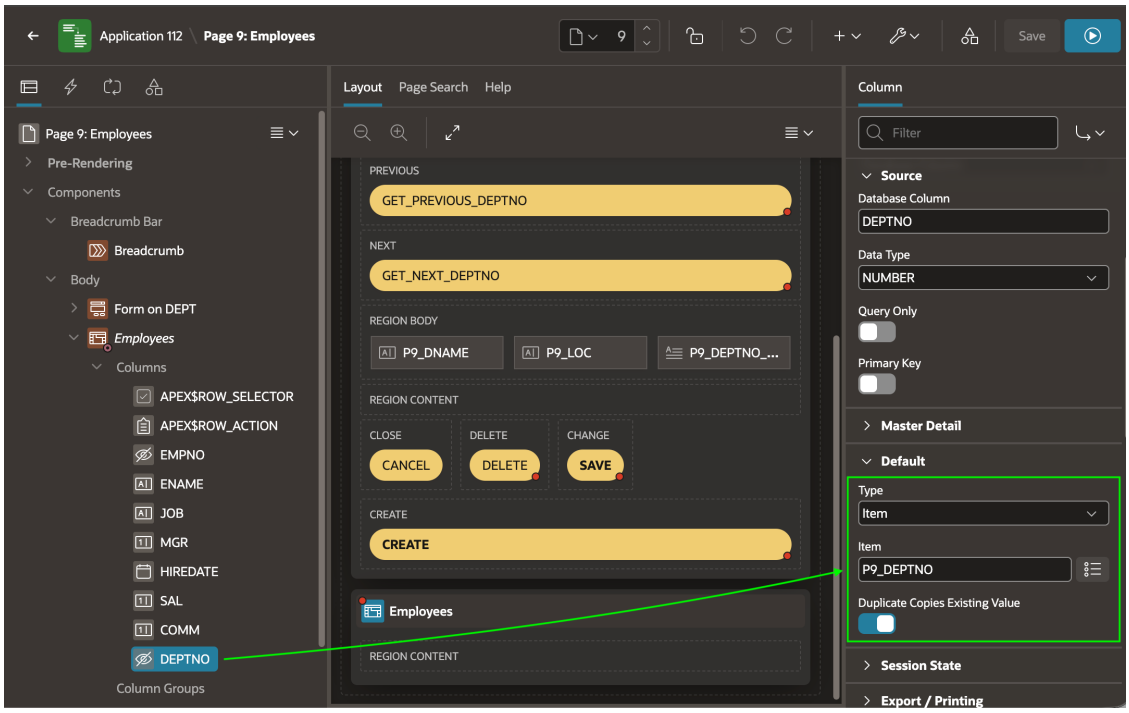
```
DEPTNO = :P9_DEPTNO
```

Since its `WHERE` clause references this master record page item as a bind variable, it also lists `P9_EMPNO` in its **Page Items to Submit** property.

The second detail involves the Employees grid's `DEPTNO` foreign key column. It is configured to use the current master row's primary key page item `P9_DEPTNO` as its default value as shown in the figure. This ensures that any new employees added to the grid while editing the *ACCOUNTING* department, are correctly associated with its `DEPTNO` value `10`.

Copyright © 2026, Oracle and/or its affiliates.

Figure 10-17 Defaulting Employee Department Foreign Key to Current Department Value



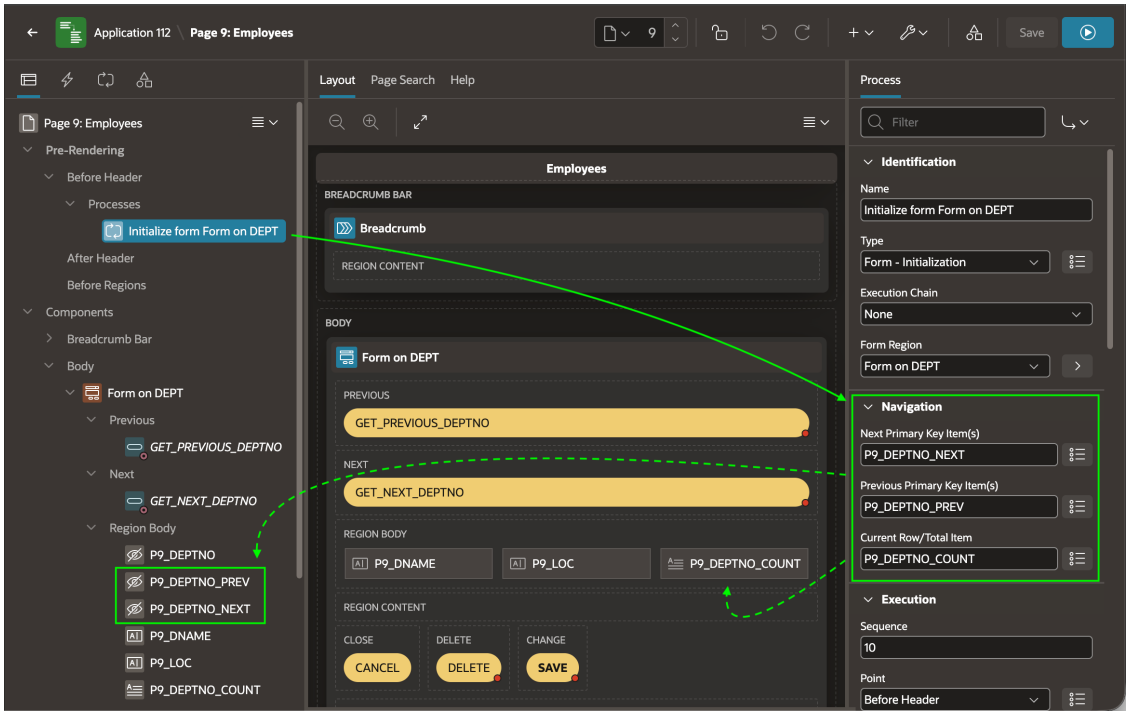
Understanding How to Configure Form Record Navigation

Like any Form region, the master region named *Form on DEPT* relies on a *Form - Initialization* page process in the Pre-Rendering section of the component tree. That process retrieves the master `DEPT` row by primary key if a value for `P9_DEPTNO` is present when the page renders. But, as shown in the figure, this Form region's initialization page process has three additional properties configured to name the page items to store **Navigation** information:

- **Next Primary Key Item(s)** → `P9_DEPTNO_NEXT`
- **Previous Primary Key Item(s)** → `P9_DEPTNO_PREV`
- **Current Row/Total Item** → `P9_DEPTNO_COUNT`

Copyright © 2026, Oracle and/or its affiliates.

Figure 10-18 Configuring Form Navigation Items on Initialization Page Process



In combination with configuring an **Order By Clause** on the Form region itself, the *Form - Initialization* page process computes these three navigation-related values at page render time and assigns them to the configured page items. The first two are the primary key of the next and previous row to edit. Either or both of these might evaluate to `null`, depending on the current row and how many rows are in the data source. The third property is a value like "3 of 4" representing the current row index and total row count. As shown in the figure above, the previous and next row keys go into hidden page items, while the *Count/Total* indicator goes into a *Display Only* page item `P9_DEPTNO_COUNT`.

The `GET_NEXT_DEPTNO` and `GET_PREVIOUS_DEPTNO` buttons use a Server-side Condition to show only when applicable, based on testing whether the values of the `P9_DEPTNO_PREV` and `P9_DEPTNO_NEXT` hidden page items are `not null`. Similarly, the branches associated to these buttons through **When Button Pressed** conditions decide whether to pass `P9_DEPTNO_NEXT` or `P9_DEPTNO_PREV` as the value of `P9_DEPTNO` when redirecting to the current page to rerender with the correct next or previous department number.

Creating New Master and Details at Once

The wizard configures the *Employees* edit page to *hide* the details *Employees* grid when `P9_DEPTNO is null`. So, by default, when the user clicks the (**Create**) button from the master Interactive Report page, the edit page shows *only* the Form region to create a new department. The user returns to the master page and has to edit the newly added department to add employees to the department in a second step.

To enable the generated *Employees* page to accept both master *and* detail data when creating a new department, only two small changes are required. First, remove the Server-side Condition on the *Employees* grid so that it always displays.

Then, add an *Assign Deptno Foreign Key* page process in the Processing tab as shown below, sequenced ***after*** the *Process form Form on DEPT* process and ***before*** the *Employees - Save Interactive Grid Data* process. This sequence is important so the form process can insert the

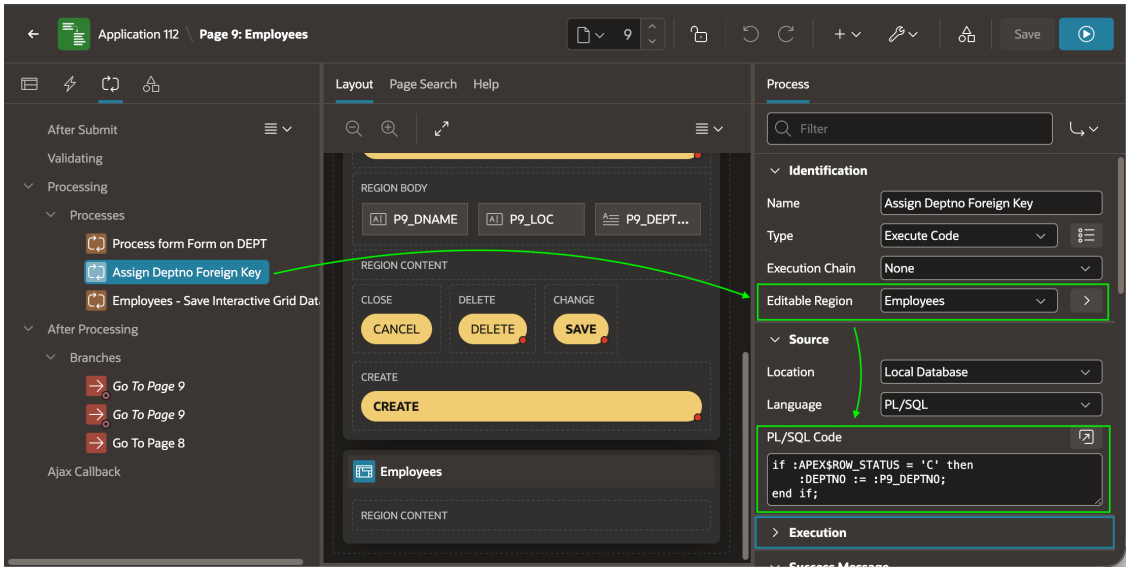
Copyright © 2026, Oracle and/or its affiliates.

new department and return its new system-assigned DEPTNO value into P9_DEPTNO. Then, your new *Execute Code* process can assign the DEPTNO foreign key column of any new employees being created. The code is very simple. It checks the value of APEX$ROW_STATUS to see if an *Employees* row is being created ('C'), then assigns the DEPTNO foreign key to the new master row's primary key in P9_DEPTNO. If the row being processed has row status of update ('U') or delete ('D'), then the assignment is skipped.

```
-- If Employee row being (C)reated, assign DEPTNO foreign key
if :APEX$ROW_STATUS = 'C' then
    :DEPTNO := :P9_DEPTNO;
end if;
```

Notice in the figure that the **Editable Region** property of the new process is set to the *Employees* grid. Since the region is a grid that can save multiple rows at once, APEX runs your process *once per row* that the *Employees* grid inserts, updates, or deletes. The APEX$ROW_STATUS check ensures the assignment is only performed for *Employees* grid rows being *created*.

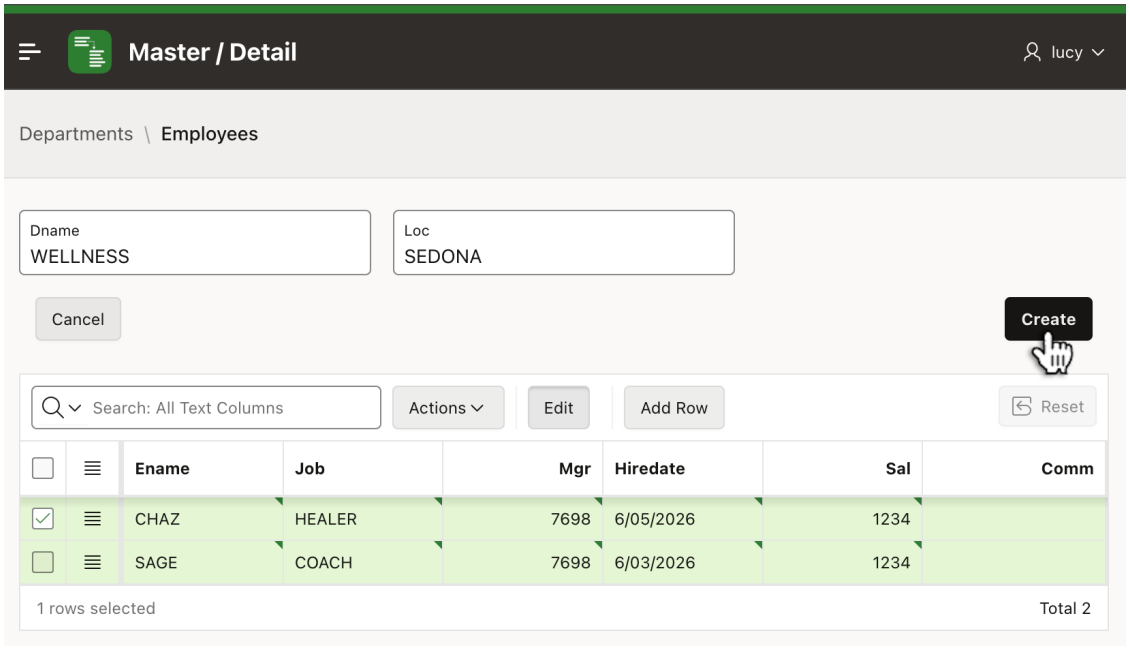
Figure 10-19 Execute Code Process with Editable Region Set Runs Once per Modified Grid Row



After making this change, clicking the (**Create**) button from the master Interactive Report page of *Departments* now shows the page below with both an empty department form *and* an empty Employees grid. You can enter a new department and multiple new employees and click (**Create**) to save them all in one transaction.

Copyright © 2026, Oracle and/or its affiliates.

Figure 10-20 Creating a New Master Row and Multiple New Details at Once



Copyright © 2026, Oracle and/or its affiliates.

11

Controlling Access with Users and Roles

Only authenticated users can access your app. They see only the pages and data they are authorized to use, and can perform only the actions their role allows. Their organizational duties shape their app experience.

Your app's *Authentication Scheme* defines how users sign in. *Roles* reflect relevant responsibilities, and *Role Assignments* link users to roles. APEX accounts are built-in, but role-based security works the same with external identity providers.

Each *Authorization Scheme* names a rule that grants or denies access based on flexible criteria. Role membership is most common, but custom rules can use SQL or PL/SQL, too. By configuring authorization rules on application elements, you tailor each user's experience based on their role.

When necessary, you can define row-level data security policies to control what data each user sees. A policy can reuse authorization schemes to keep access rules in one place.

The examples in this section use a simple *Woods HR* app based on `EBA_DEMO_EMP` and a copy of the familiar `EMP` table, with a few additional employees added. The figure below shows three colleagues, one uses a laptop, another holds a tablet, and a third uses his smartphone. Each one sees the application functionality and data that is automatically tailored to their functional role.

Figure 11-1 Deliver User Experience Tailored to Each User's Functional Role



Copyright © 2026, Oracle and/or its affiliates.

- Requiring Authenticated Access to a Page
  A page's *Authentication* property controls if users must login to view it.
- Authenticating Access with APEX Accounts
  Your app's *Authentication Scheme* controls how users login. The default option is to use native APEX user accounts.
- Assigning Users to Roles
  Assign users to one or more roles to reflect the functions they perform in the organization.
- Shaping Experience with Rules and Roles
  Authorization Schemes are rules you define and assign to virtually any kind of application element to declaratively shape the end user experience.
- Enabling Row-Level Data Security Policy
  Learn about where and when to use a row-level data security policy.
- Refining Salary Review with Data Security
  After setting up the view and row-level data security policy, you can enhance your app to use them.
- Integrating an External Identity Provider
  While APEX Accounts are easy and built-in, you might *already* manage users and roles with an external identity provider. If so, create a *Social Sign-in* type Authentication Scheme and *after authentication* dynamically enable the groups users belong to.
- Exploring Context for Authorization Rules
  Depending on your authorization scheme's evaluation point, additional context information is available to write more generic, data-driven rules.

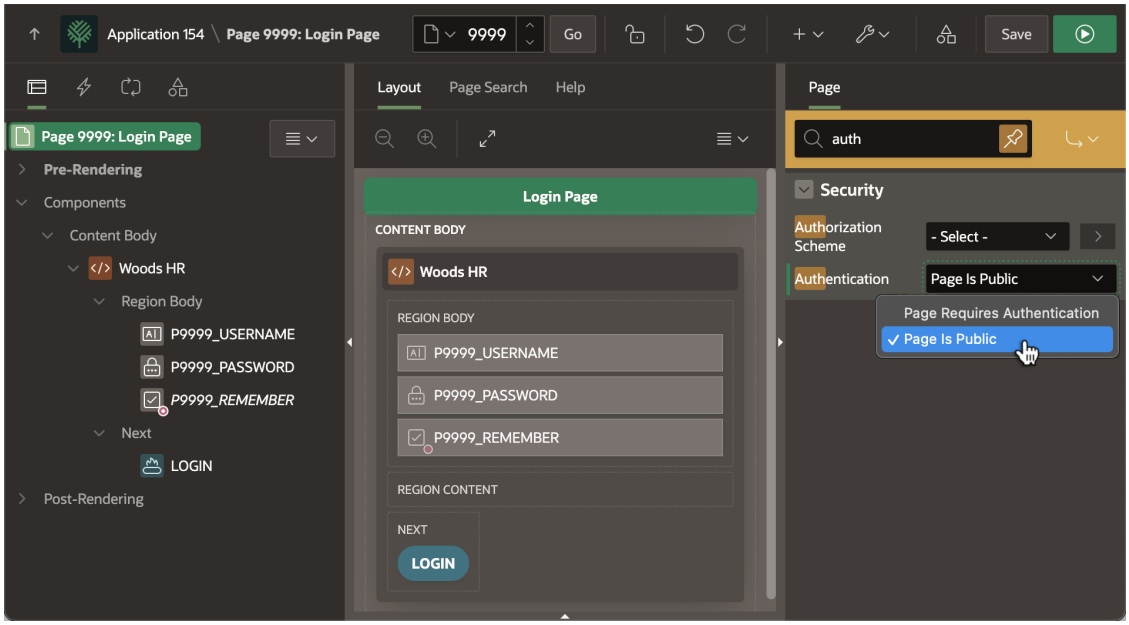
11.1 Requiring Authenticated Access to a Page

A page's *Authentication* property controls if users must login to view it.

Set the page's authentication to *Page Is Public* and anyone can visit. Choose *Page Requires Authentication* and users must log in first. As shown below, by default the *Login Page* that App Builder creates in a new app is public. Most *other* pages in your app should require authentication, but choose the setting that matches each page's needs. The figure shows the Woods HR application's *Login* page 9999 in Page Designer. The root page node is selected in the component tree, and the page's *Authentication* property is selected in the Property Editor.

Copyright © 2026, Oracle and/or its affiliates.

Figure 11-2 Deciding if a Page is Public or Requires Authentication

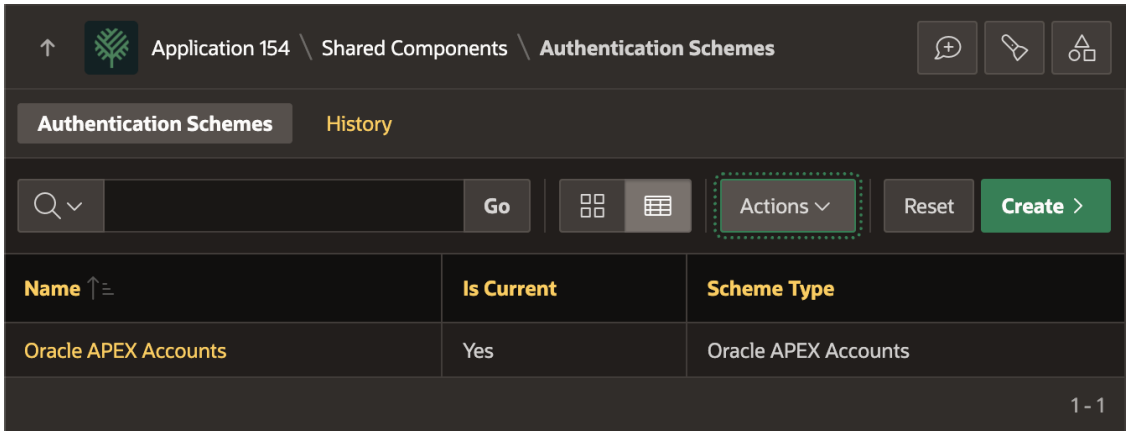


11.2 Authenticating Access with APEX Accounts

Your app's *Authentication Scheme* controls how users login. The default option is to use native APEX user accounts.

As shown below, in a new application the default scheme authenticates users based on *Oracle APEX Accounts*.

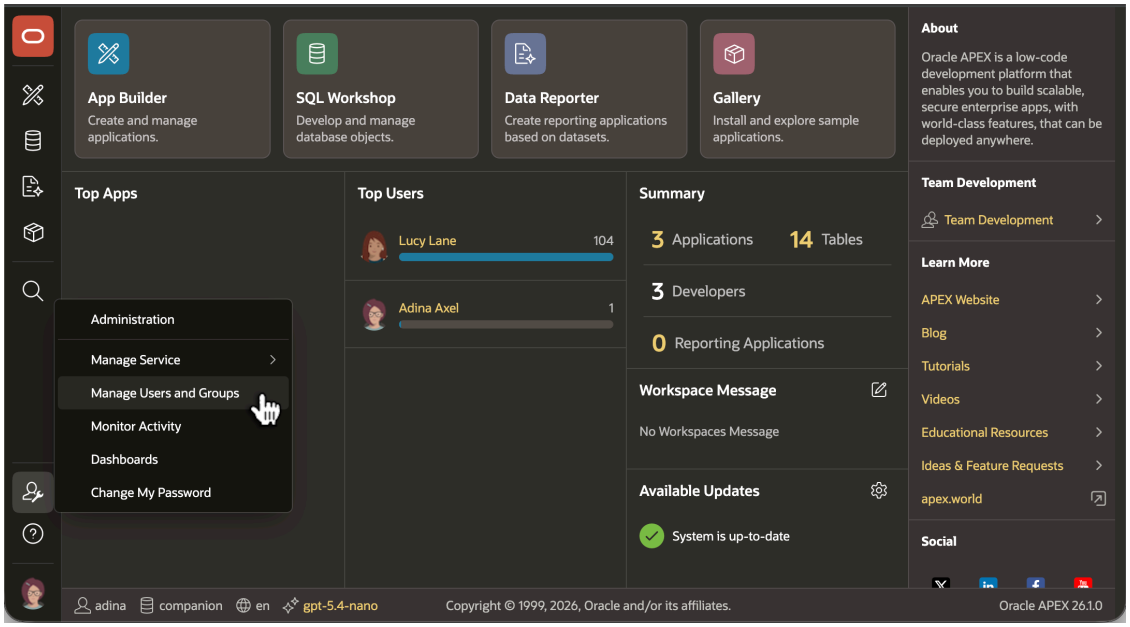
Figure 11-3 Oracle APEX Accounts Provide Default Authentication Scheme



As shown below, a workspace administrator like *Adina Axel* manages APEX user accounts by clicking on the button with a user and wrench, then choosing the *Manage Users and Groups* menu entry in the dropdown menu.

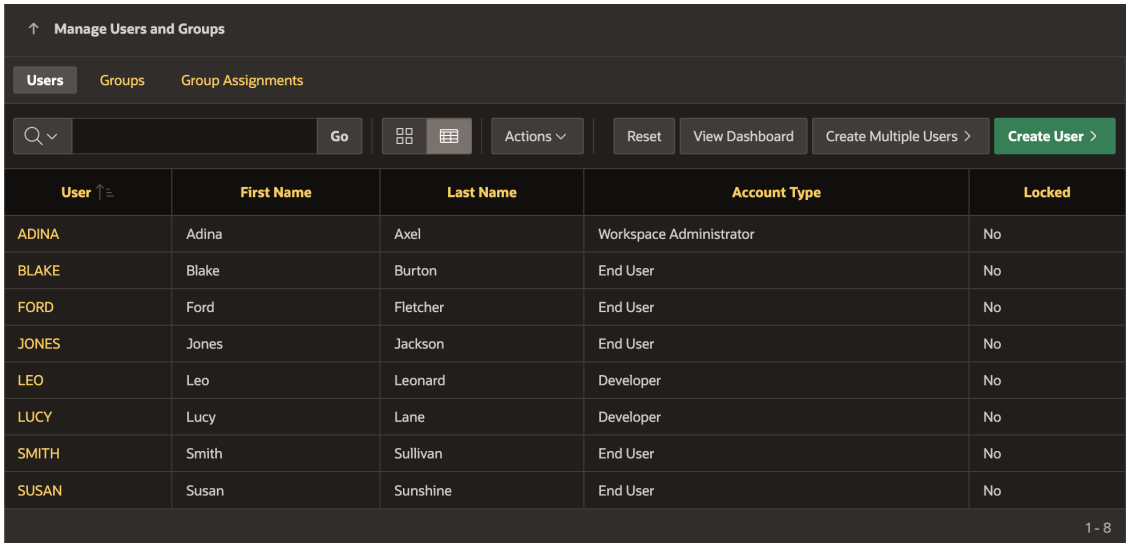
Copyright © 2026, Oracle and/or its affiliates.

Figure 11-4 Workspace Administrators Manage APEX User Accounts



When defining a new APEX user account, the workspace administrator sets the account type. As shown below, `ADINA` is the *Workspace Administrator*, `LEO` and `LUCY` have a *Developer* account, and the rest of the user accounts are end users. While `ADINA`, `LUCY`, and `LEO` can also log in to App Builder, the rest of the user accounts can only log in to APEX applications you develop.

Figure 11-5 Workspace Administrators Manage Users from Accounts List Page



Copyright © 2026, Oracle and/or its affiliates.

> **Tip**
>
> For more details on the many other built-in authentication schemes APEX supports, see Understanding Preconfigured Authentication Schemes in *Oracle APEX App Builder User's Guide*.

11.3 Assigning Users to Roles

Assign users to one or more roles to reflect the functions they perform in the organization.

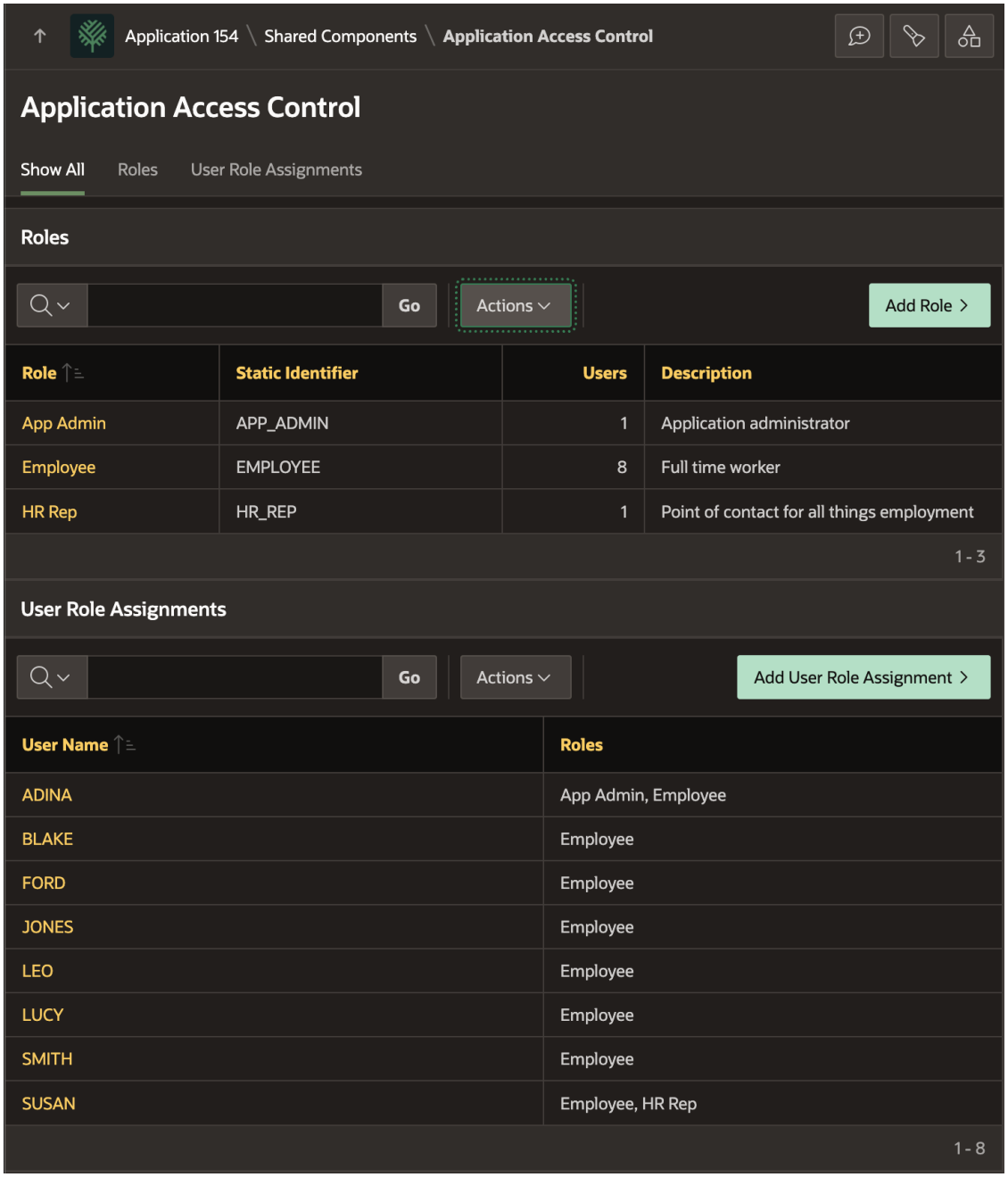
Combined with authorization schemes, these role assignments let your app tailor the user experience for each person automatically. As shown below, use the *Application Access Control* page under *Shared Components* to define roles and assign users to them. Notice *every* user has the *Employee* role, while:

- `ADINA` also has the *App Admin* role
- `SUSAN` also has the *HR Rep* role

> **Tip**
>
> Your roles are part of the app definition you export and import into other environments. In contrast, the role *assignments* are not. By design, your role assignments in a particular target environment remain intact when importing a newer version of your app.

Copyright © 2026, Oracle and/or its affiliates.

Figure 11-6 Configuring Roles and Role Assignments



11.4 Shaping Experience with Rules and Roles

Authorization Schemes are rules you define and assign to virtually any kind of application element to declaratively shape the end user experience.

While not *limited* to these elements, here you apply an authorization scheme to menu entries, pages, report columns, buttons, and edit links to control who can see and access what in the application.

Copyright © 2026, Oracle and/or its affiliates.

- Declaring Authorization Scheme Rules
  Name each *Authorization Scheme* after the rule it enforces. Each one can use role membership, SQL, or PL/SQL to authorize or deny the current user access.
- Authorizing Menu Access
  Your *Navigation Menu* list defines the entries users click to reach your app's business functions.
- Authorizing Page Access
  Configure page-level access by setting the *Authorization Scheme* for the page.
- Authorizing Page Item or Column Access
  You can set the *Authorization Scheme* for specific columns or page items.
- Authorizing Button Access
  Configure a button's Authorization Scheme to control which users can see it.
- Authorizing Edit Link Column Access
  Set an *Authorization Scheme* on the edit link of an Interactive Report to control who can see it
- Experiencing Authorization in Action
  Once you configure authorization for your application, every user's experience adapts automatically based on their role.

11.4.1 Declaring Authorization Scheme Rules

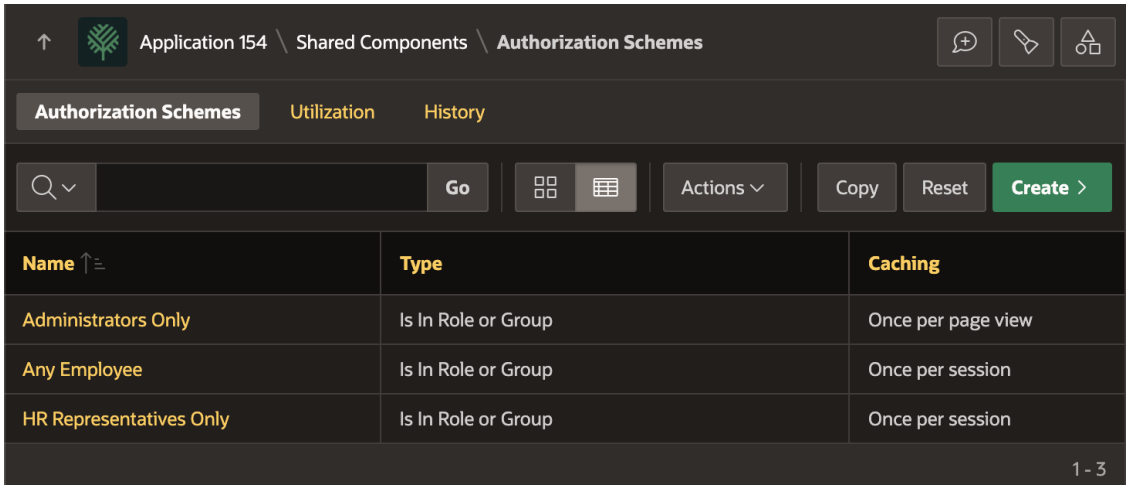
Name each *Authorization Scheme* after the rule it enforces. Each one can use role membership, SQL, or PL/SQL to authorize or deny the current user access.

As shown below, you define three rules:

1. *Administrators Only* – Authorizes a user who is an application administrator
2. *Any Employee* – Authorizes a user who is an employee
3. *HR Representatives Only* – Authorizes a user who is an HR representative.

All three of these authorization scheme rules use the built-in *Scheme Type* of *Is In Role or Group*.

Figure 11-7 Configuring Authorization Scheme Rules Using Role Membership

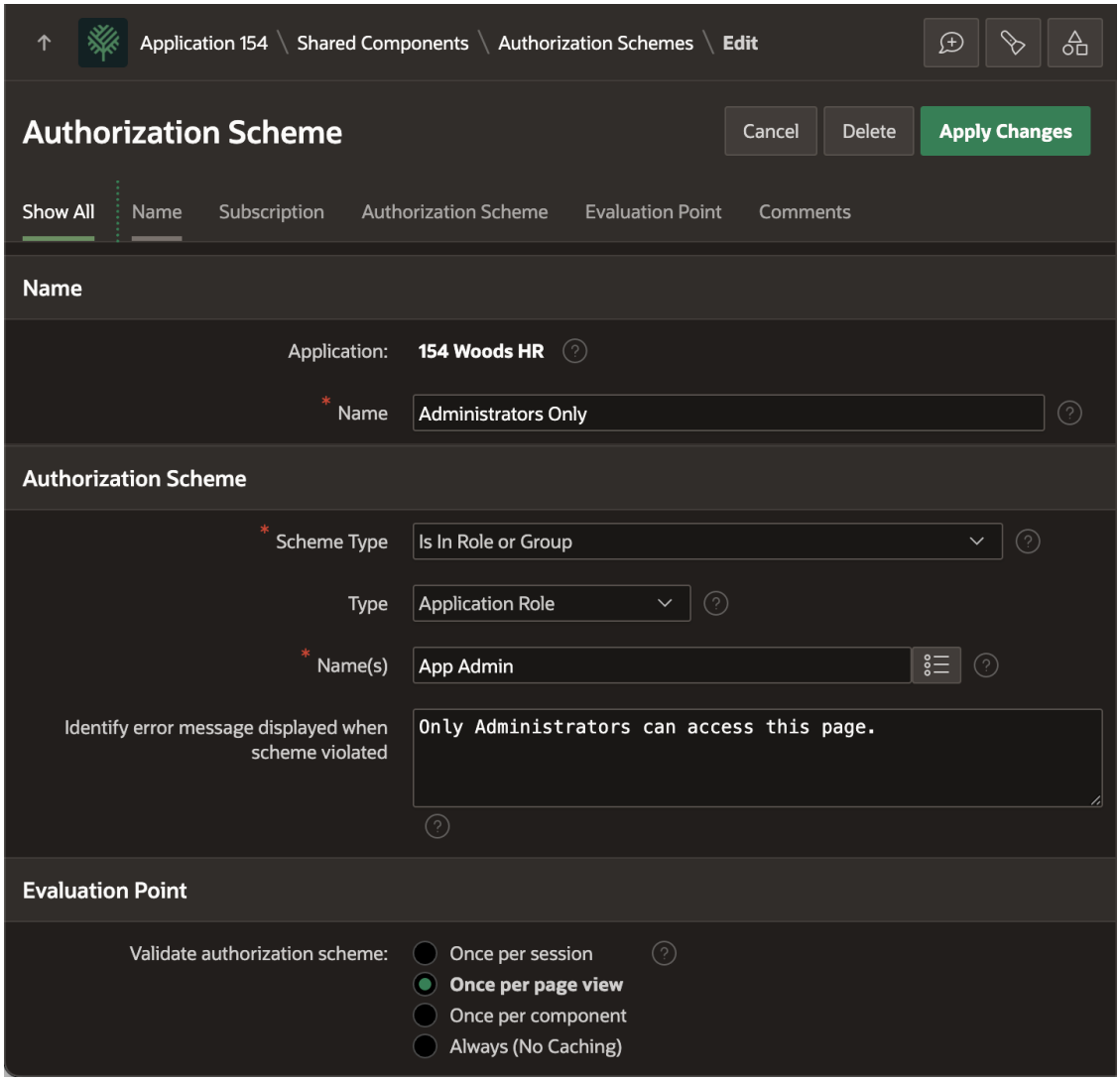


Copyright © 2026, Oracle and/or its affiliates.

As shown below, the *Administrators Only* rule identifies the *Type* of check as *Application Role*, and lists the role name *App Admin*. Given the sensitive nature of application administration functionality, this *particular* rule sets its evaluation point to be *Once per page view*. The *other* two rules are similar, using respective roles, but evaluate *Once per session* instead. As a concrete example, if during a work day a user becomes an *HR Representative* they need to sign out and login again to access new pages or elements only HR Reps get to see.

> **Tip**
>
> The *Name(s)* field can be a comma-separated list of multiple roles. In that case, the rule authorizes the user if they are a member of at least one of the listed roles.

Figure 11-8 Detail of Authorization Scheme Rule Based on Role Membership



Copyright © 2026, Oracle and/or its affiliates.

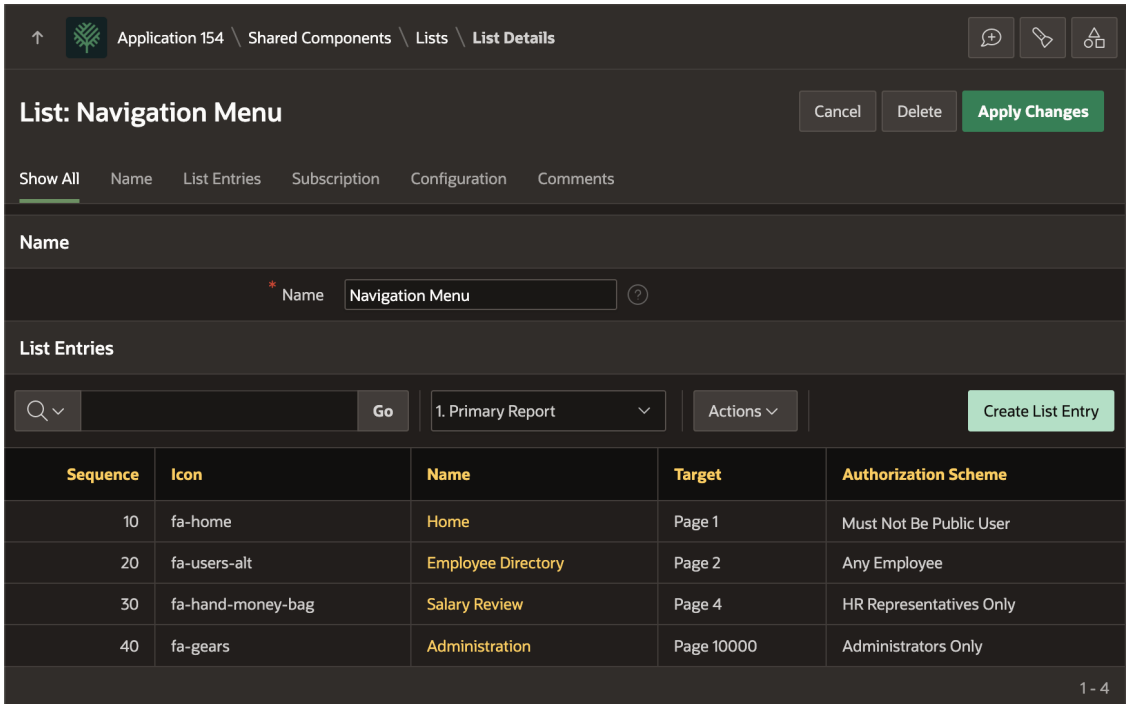
11.4.2 Authorizing Menu Access

Your *Navigation Menu* list defines the entries users click to reach your app's business functions.

By default, a menu entry is available to all users, including *unauthenticated* ones. As shown below, use the built-in *Must Not Be Public User* authorization scheme rule to ensure the user must have logged in successfully to see that entry. The *Woods HR* menu entries use the following authorization schemes:

- *Home* – **Must Not Be Public User**, so user must have logged in
- *Employee Directory* – **Any Employee**, so any employee can access the directory
- *Salary Review* – **HR Representative Only**, so only HR Reps can review salary details
- *Administration* – **Administrators Only**, so only App Admins can see and adjust app settings.

Figure 11-9 Applying Authorization Scheme to Navigation Menu Entries



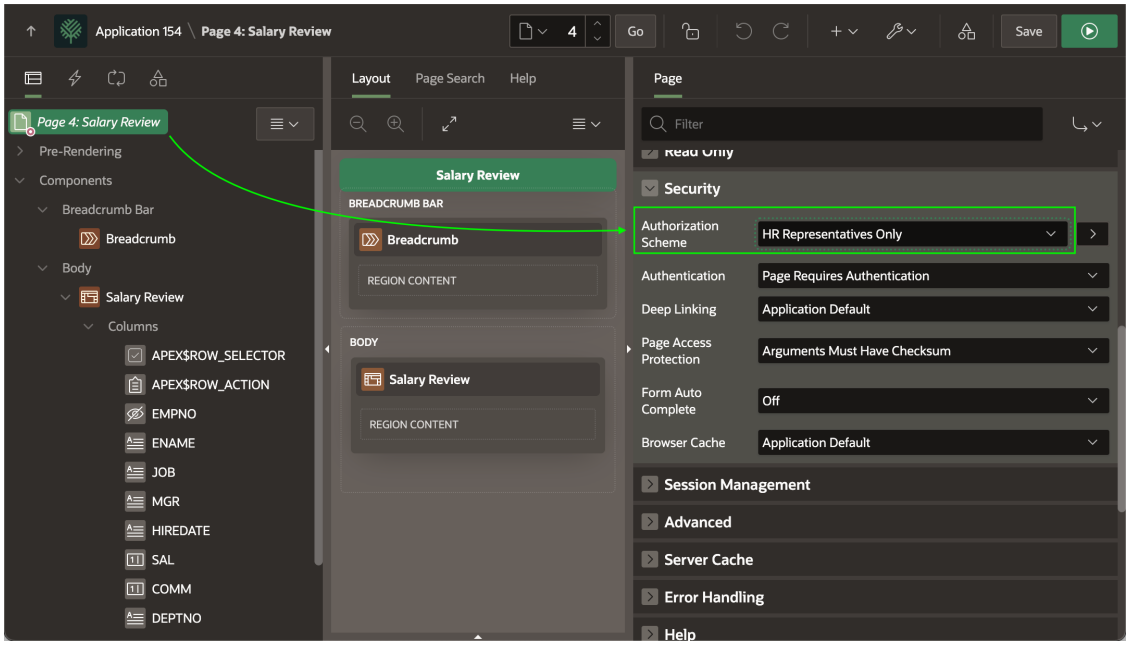
11.4.3 Authorizing Page Access

Configure page-level access by setting the *Authorization Scheme* for the page.

As shown below, in the *Salary Review* page, you set its page-level *Authorization Scheme* to **HR Representatives Only**. This authorizes HR Reps to access this page.

Copyright © 2026, Oracle and/or its affiliates.

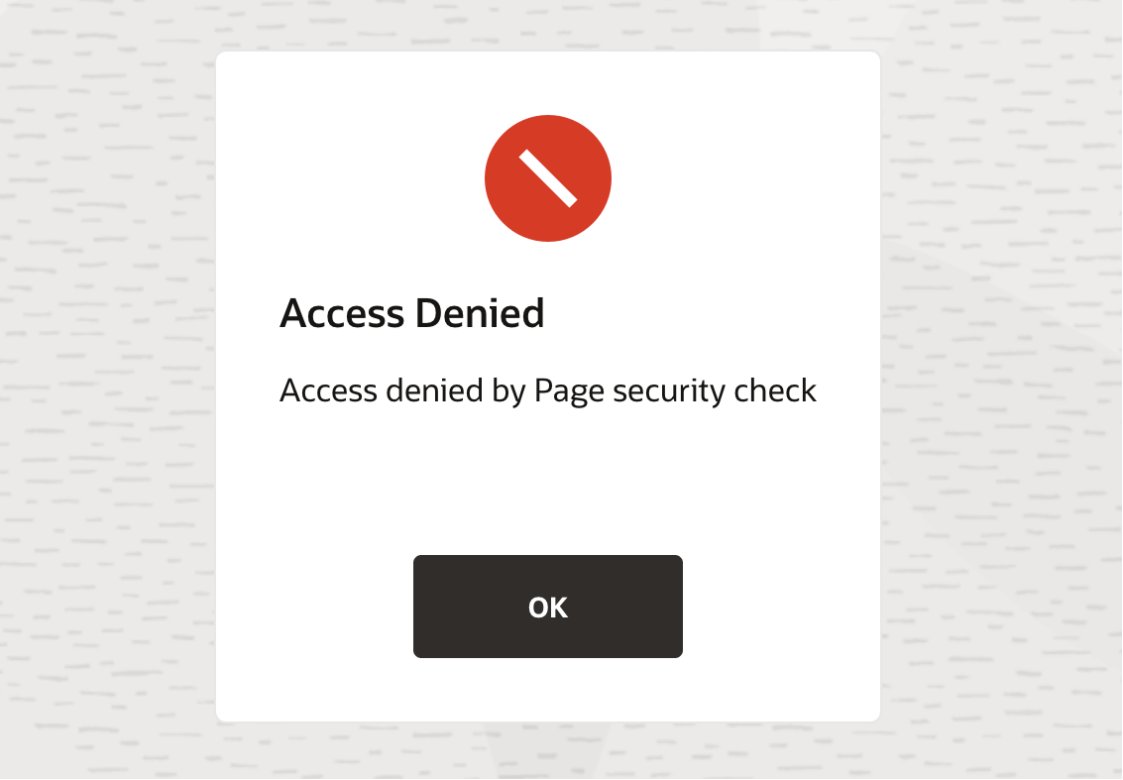
Figure 11-10 Applying Authorization Scheme to a Page



In contrast, any *other* users who might attempt to view the page by any means will receive the error shown below. You usually assign the same authorization scheme to the menu entry, so users can't click through to a page they can't access. However, if a clever user manually adjusts the URL in their browser address bar, they'll see this error if they are unauthorized. The error message in the figure reads, "Access Denied by Page security check."

Copyright © 2026, Oracle and/or its affiliates.

Figure 11-11 Non HR Representatives See Error Navigating to Unauthorized Page



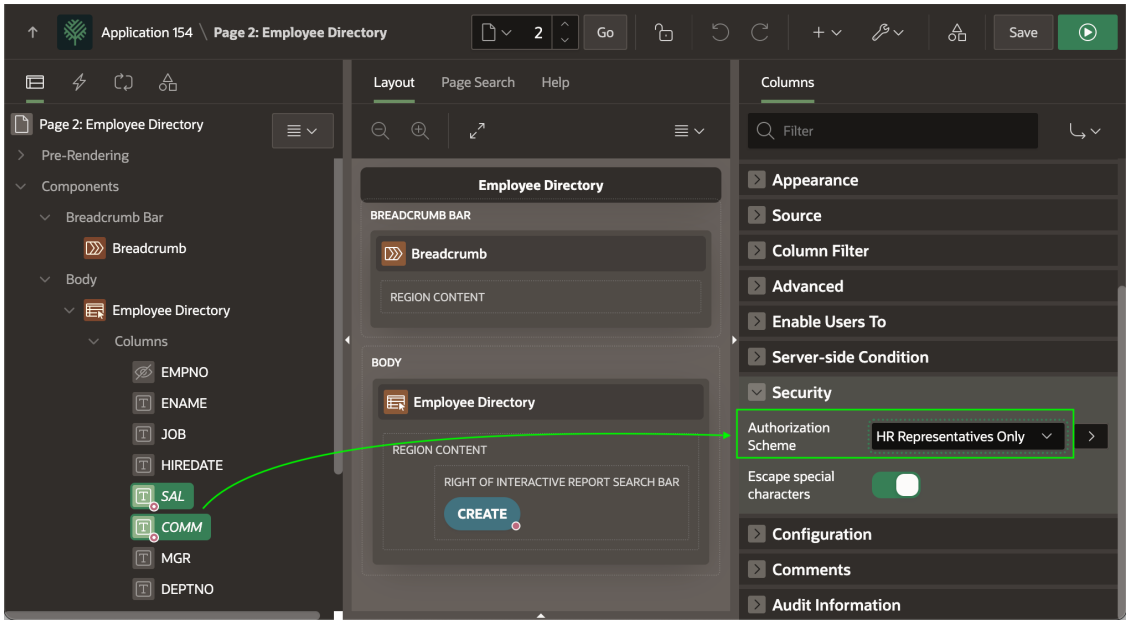
11.4.4 Authorizing Page Item or Column Access

You can set the *Authorization Scheme* for specific columns or page items.

As shown below, in the *Employee Directory* page, you set the *Authorization Scheme* of the `SAL` and `COMM` columns to **HR Representatives Only**. This authorizes HR Reps to see these columns when visiting the page. In contrast, any *other* users viewing the directory do not see the columns at all. You can apply an authorization scheme to page items as well to achieve the same effect.

Copyright © 2026, Oracle and/or its affiliates.

Figure 11-12 Applying Authorization Scheme to Columns



11.4.5 Authorizing Button Access

Configure a button's Authorization Scheme to control which users can see it.

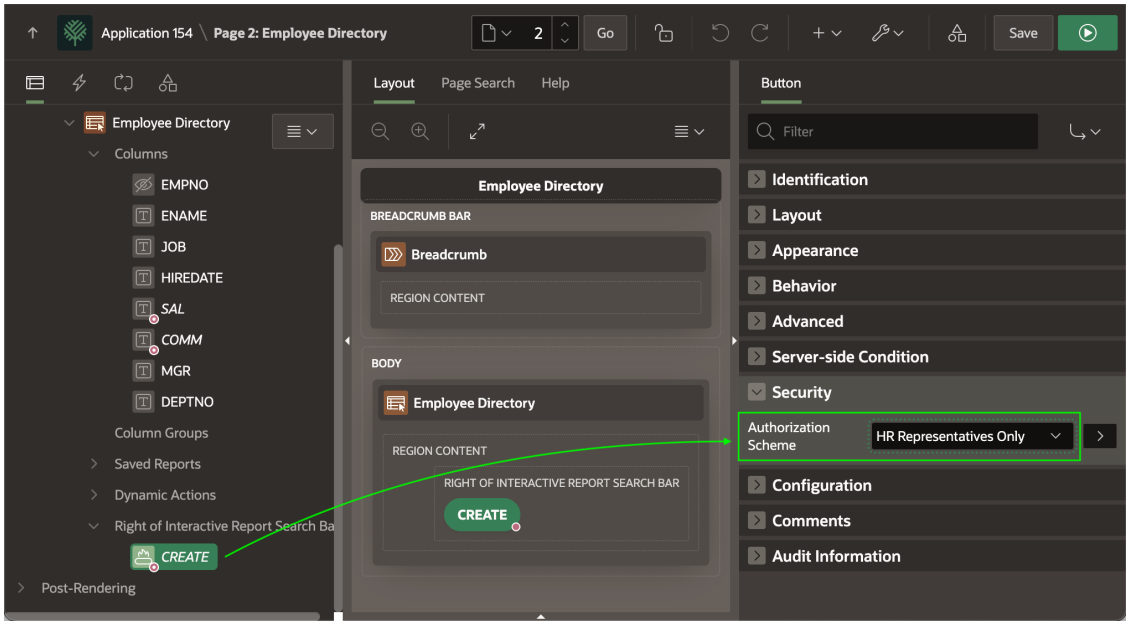
As shown below, in the *Employee Directory* page, you set the `CREATE` button's *Authorization Scheme* to **HR Representatives Only**. This lets HR Reps see this button to create a new employee. In contrast, any *other* users viewing the directory do not see the button at all, so they can't create employees.

If the button submits the page to trigger a page process, you can apply the same authorization scheme to the corresponding page process as an additional access check.

> **Tip**
>
> The *Employee* page the `CREATE` button targets *also* has a page-level authorization scheme of **HR Representatives Only**. So, any manual attempt by unauthorized users to visit that page results in an error.

Copyright © 2026, Oracle and/or its affiliates.

Figure 11-13 Applying Authorization Scheme to a Button



11.4.6 Authorizing Edit Link Column Access

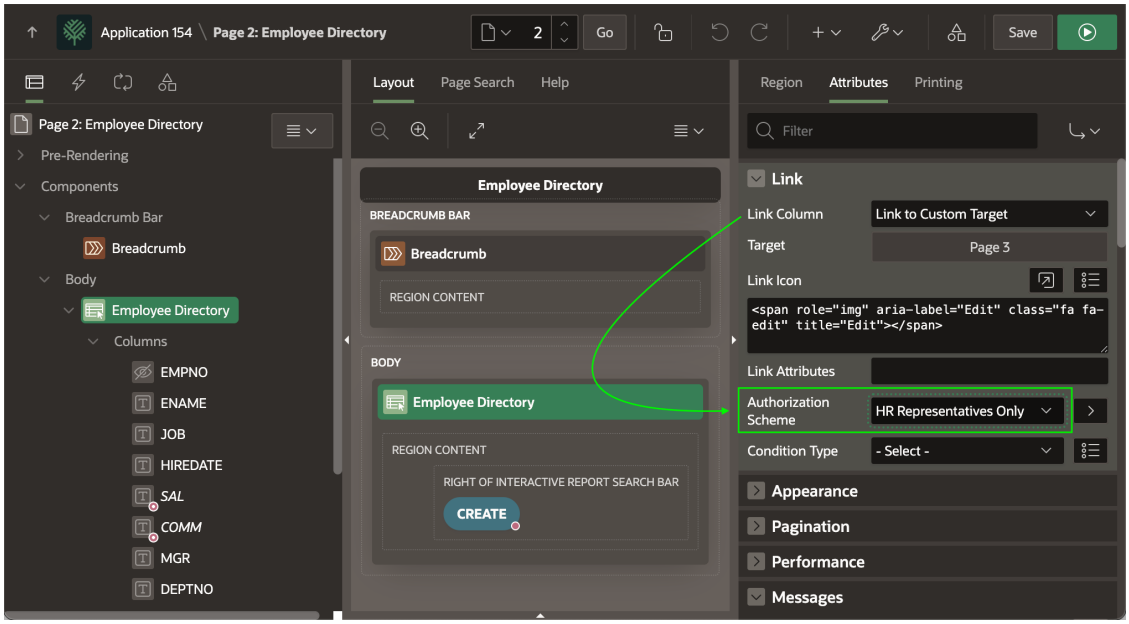
Set an *Authorization Scheme* on the edit link of an Interactive Report to control who can see it

As shown below, in the *Employee Directory* page, you set the *Authorization Scheme* of the *Link Column* to **HR Representatives Only**. This lets HR Reps see and click on this link to edit an existing employee. In contrast, any *other* users viewing the directory do not see the column at all, so they can't edit employees.

> **Tip**
>
> The *Employee* page the edit link targets also has a page-level authorization scheme of **HR Representatives Only**. So any manual attempt by unauthorized users to visit that page results in an error.

Copyright © 2026, Oracle and/or its affiliates.

Figure 11-14 Applying Authorization Scheme to an Edit Link Column



11.4.7 Experiencing Authorization in Action

Once you configure authorization for your application, every user's experience adapts automatically based on their role.

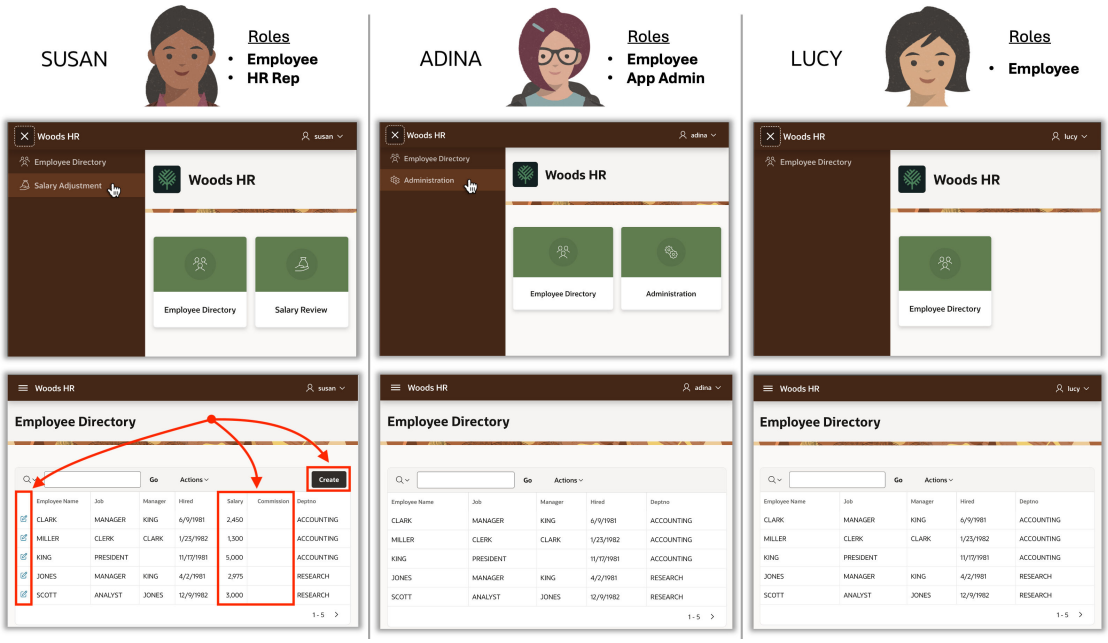
The figure below shows the outcome of applying authorization scheme rules to menu entries, pages, columns, buttons, and edit links. When `SUSAN` the HR Representative logs in, she sees the *Employee Directory* and *Salary Review* pages in the menu and on the home page that displays the navigation menu entries in a card list. When she accesses the *Employee Directory* page, the *Salary* and *Commission* columns appear in the interactive report, as do as the *(Create)* button and the edit link.

When App Admin `ADINA` logs in, she can access the *Employee Directory* and the *Administration* page. When she navigates to the *Employee Directory* page, she sees no pay information in the report, and she isn't even *aware* that creating or editing employees is possible.

When employee `LUCY` accesses the app, after logging in she see just the *Employee Directory* in the same simplified way that `ADINA` sees it.

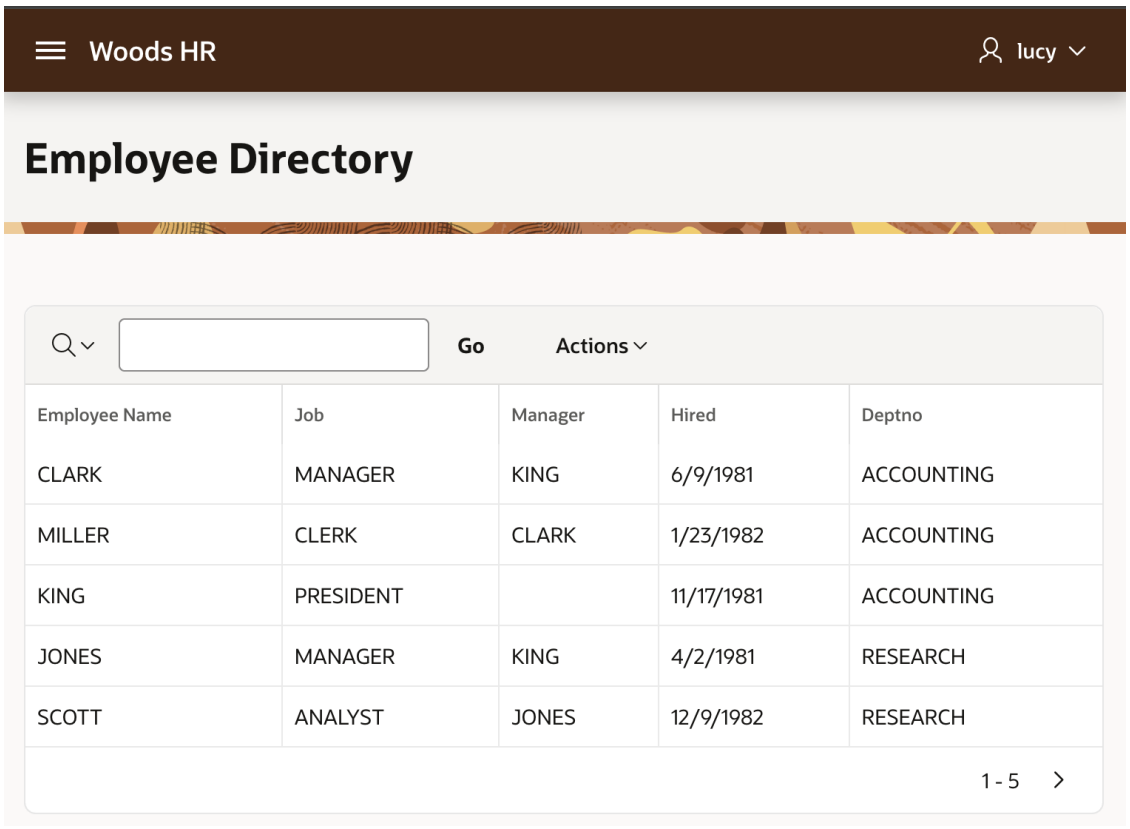
Copyright © 2026, Oracle and/or its affiliates.

Figure 11-15 End Users Experience and Features Adjust Automatically to their Role



As shown below, LUCY's *Employee Directory* page experience is a read-only report with no salary or commission information available.

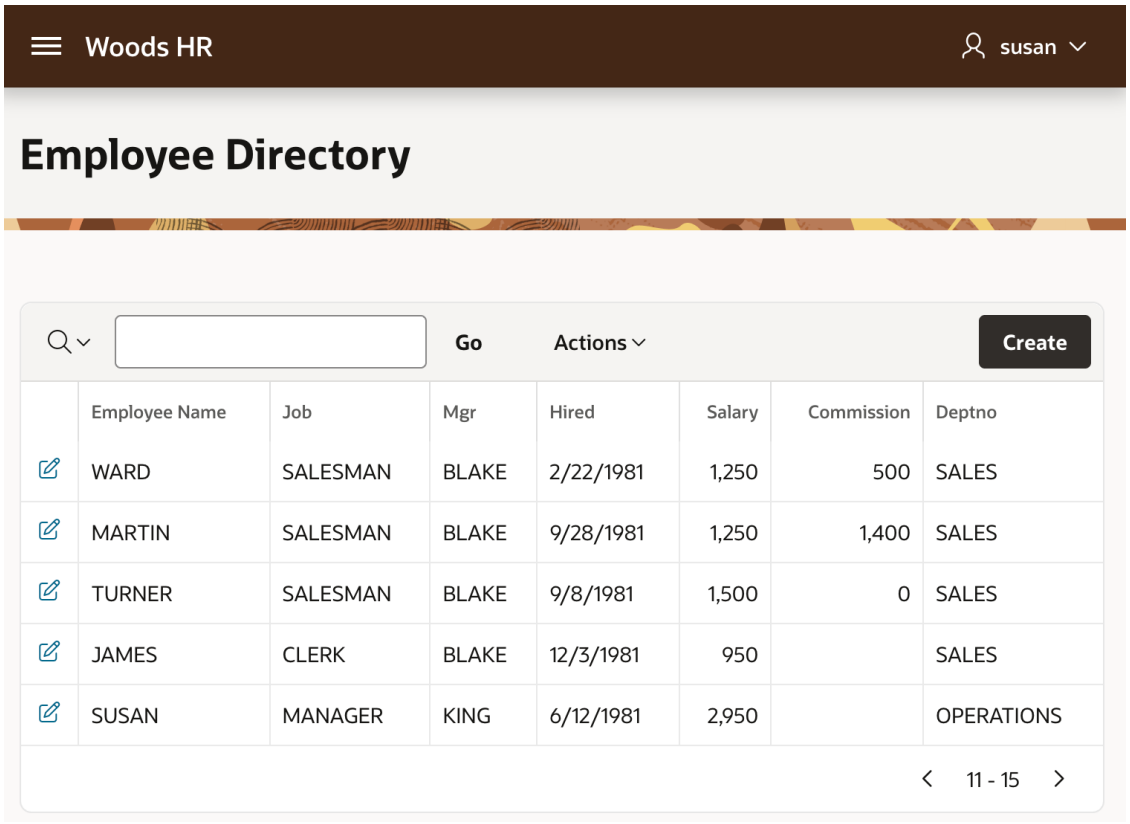
Figure 11-16 Regular Employees Don't See Certain Columns, Create Button, or Edit Link



Copyright © 2026, Oracle and/or its affiliates.

However, when SUSAN visits the same page, she can see pay information, and both create and edit employees since she's an HR Representative.

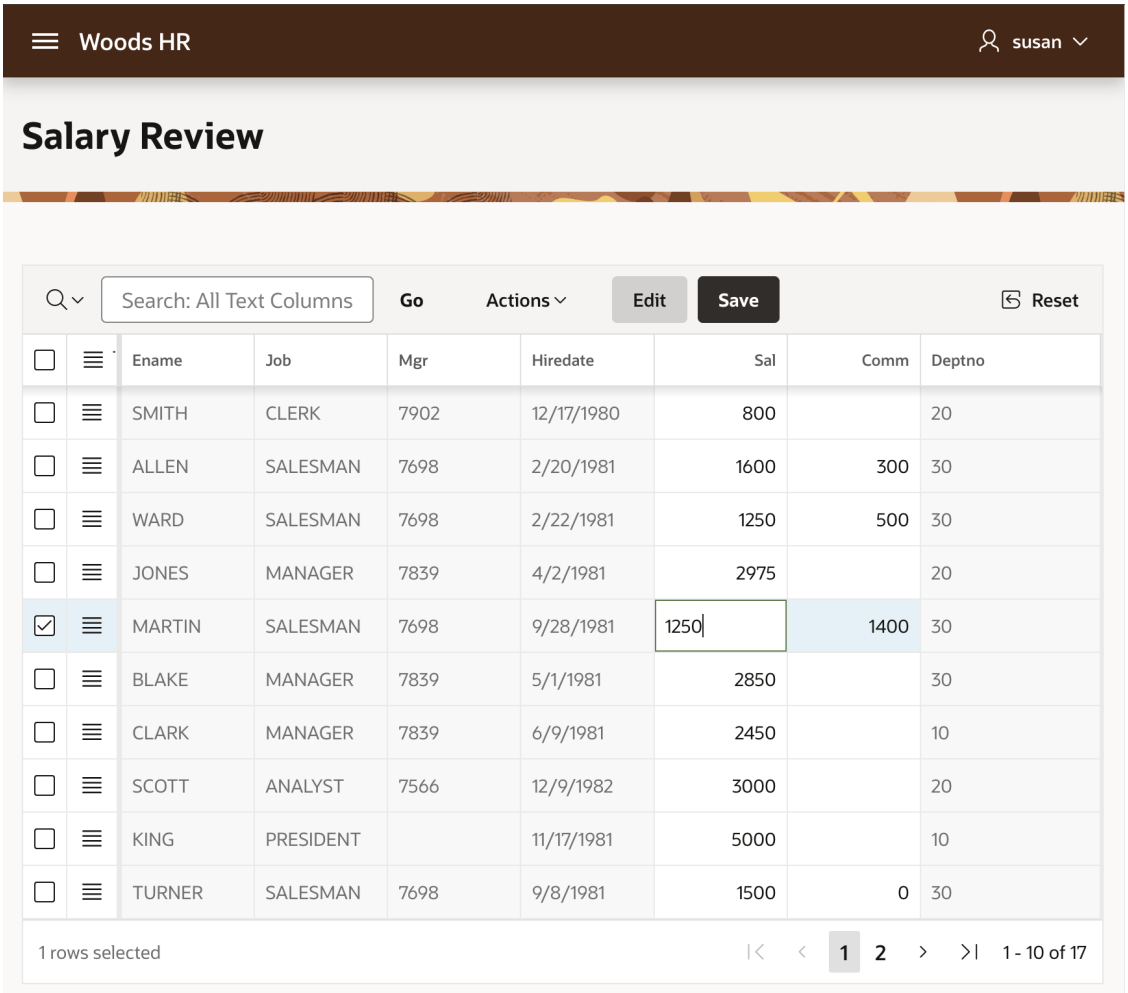
Figure 11-17 HR Representatives See Pay Columns, Create Button, and Edit Link



Only SUSAN can navigate to the *Salary Review* page shown below, featuring an Interactive Grid. You configured most of its columns to be *Display Only*, with only *Sal* and *Comm* columns as editable. So she can edit any employee's salary and commission from here.

Copyright © 2026, Oracle and/or its affiliates.

Figure 11-18 Only HR Representatives Can Use Salary Review Page



11.5 Enabling Row-Level Data Security Policy

Learn about where and when to use a row-level data security policy.

Imagine you are asked to implement a new feature with the following requirements:

- Managers should *also* be able to access the *Salary Review* page
- The page should display only their direct or indirect reports
- The employee data is read-only for managers.

You tackle this by creating two new authorization schemes, and combining them with a powerful Oracle database feature: policy-driven, row-level data security.

- Using SQL and PL/SQL Authorization Rules
  Use SQL or PL/SQL to define custom authorization schemes.
- Enabling Row-Level Data Security
  To define row-level security policies on tables or views, your workspace's parsing schema needs additional privileges your DBA can grant.

Copyright © 2026, Oracle and/or its affiliates.

11.5.1 Using SQL and PL/SQL Authorization Rules

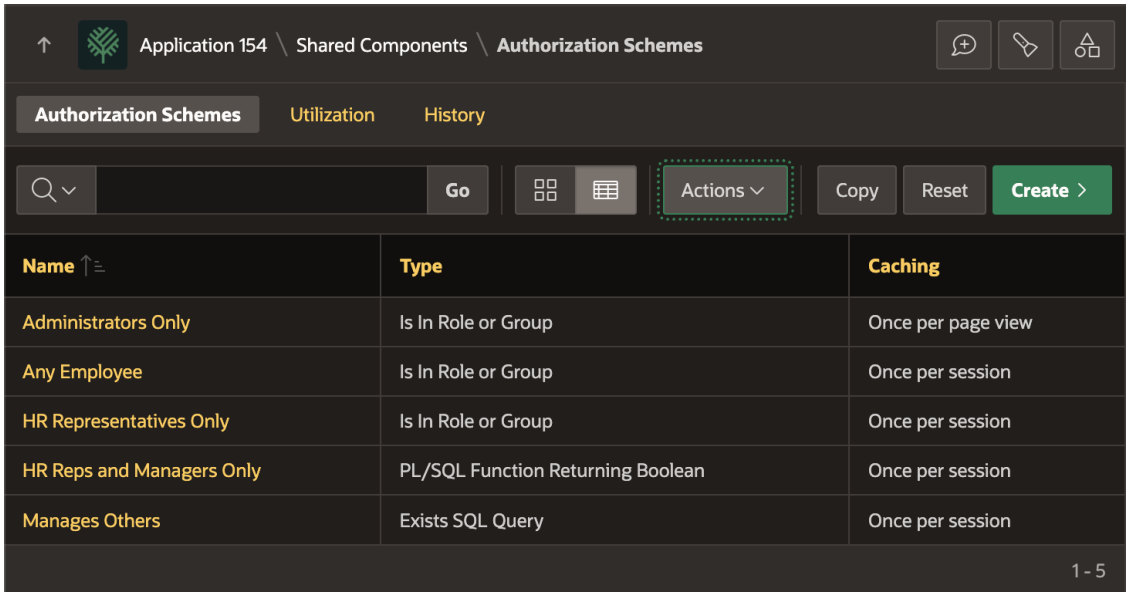
Use SQL or PL/SQL to define custom authorization schemes.

You define the two additional authorization schemes shown below:

- *Manages Others* – Authorizes users who manage at least one other employee
- *HR Reps and Managers Only* – Authorizes users who are either an HR Rep or manage others.

The figure shows the Authorization Schemes page in Shared Components, with the two new schemes added to the list.

Figure 11-19 Two Additional Authorization Schemes Using PL/SQL and SQL



- Defining Authorization Rule with SQL
  Use an *Exists SQL Query* authorization *Scheme Type* to define a rule based on a query.
- Using PL/SQL for an Authorization Rule
  Use a *PL/SQL Function Returning Boolean* scheme type to define a rule using custom logic.

11.5.1.1 Defining Authorization Rule with SQL

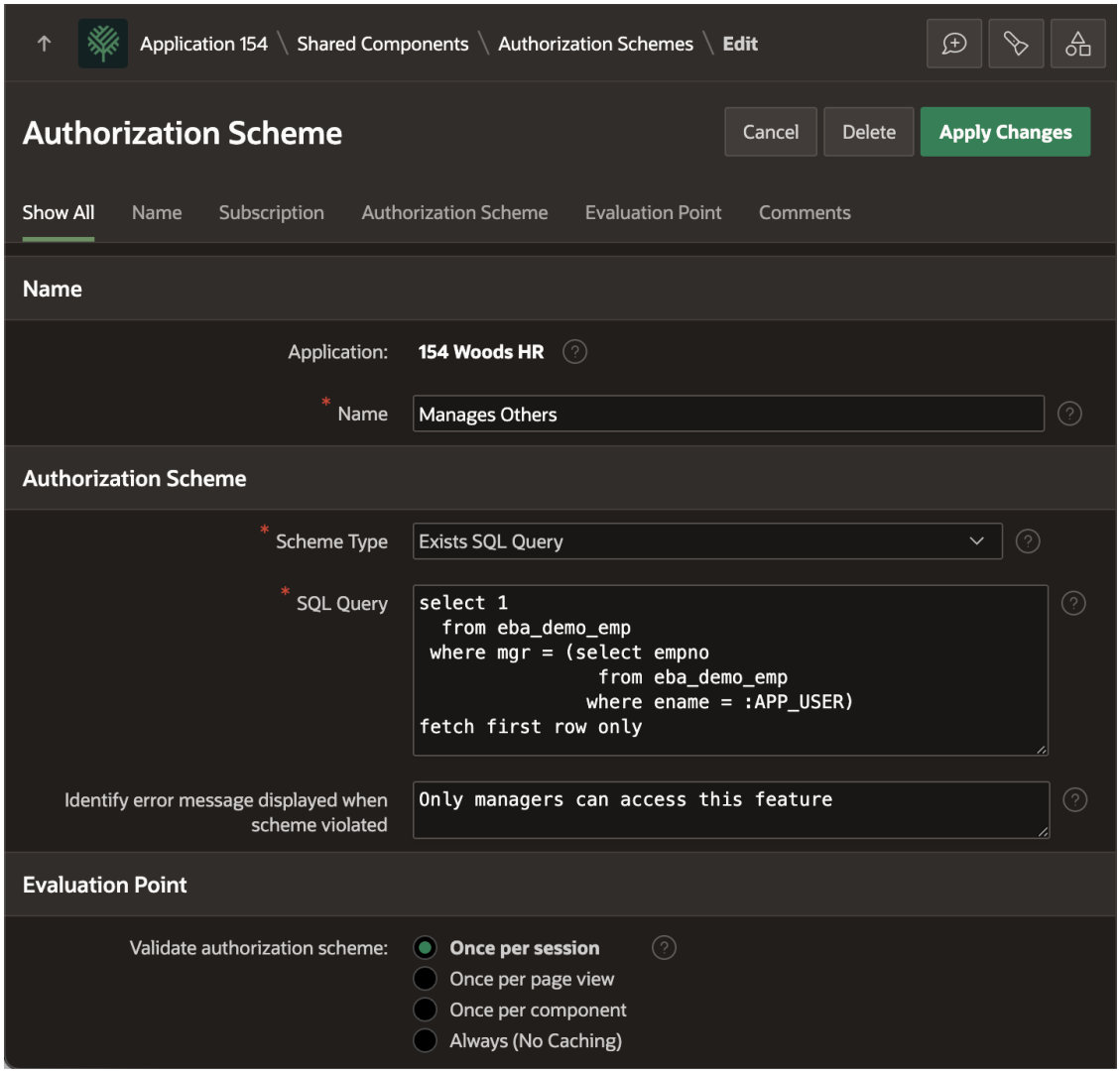
Use an *Exists SQL Query* authorization *Scheme Type* to define a rule based on a query.

As shown below, the *Manages Others* rule uses the following query that retrieves a row if at least one employee has the *current* user's `EMPNO` as its `MGR` value:

```
select 1
  from eba_demo_emp
 where mgr = (select empno
                from eba_demo_emp
               where ename = :APP_USER)
fetch first row only
```


Copyright © 2026, Oracle and/or its affiliates.

Figure 11-20 Using SQL to Define Authorization Scheme Rule for Managers



11.5.1.2 Using PL/SQL for an Authorization Rule

Use a *PL/SQL Function Returning Boolean* scheme type to define a rule using custom logic.

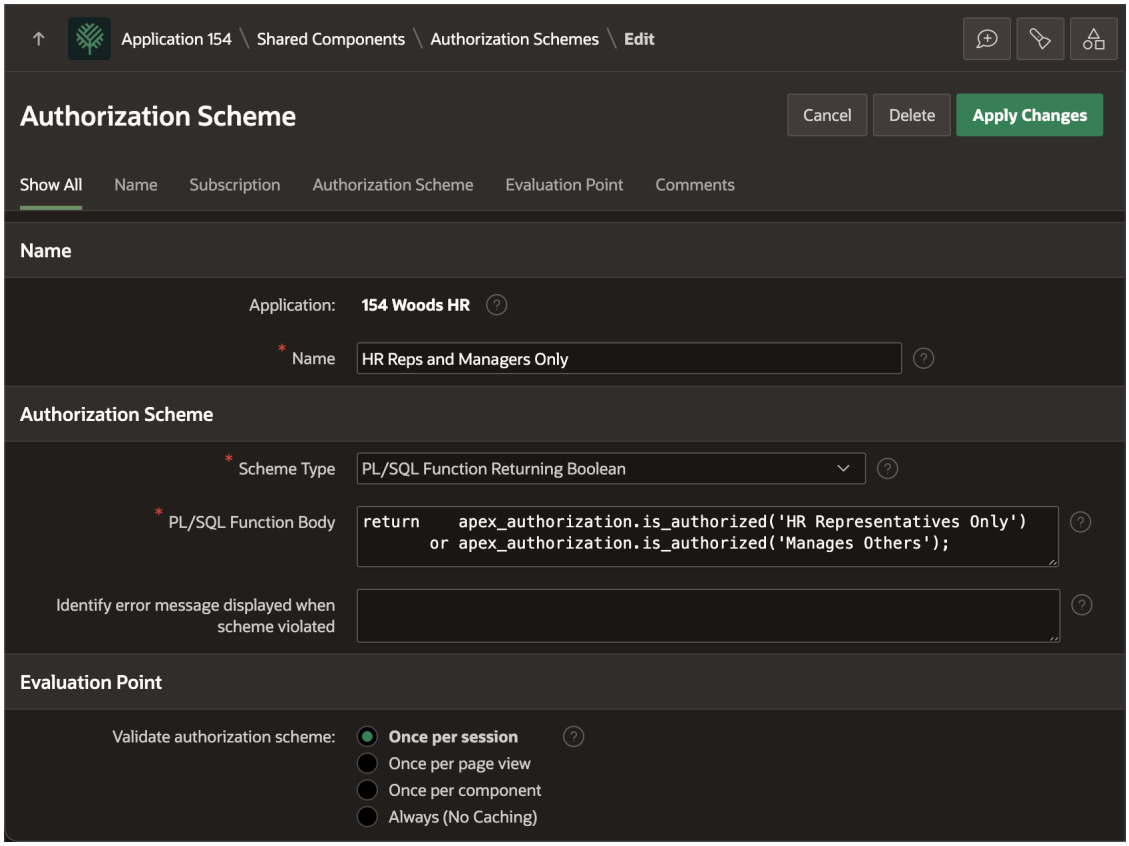
As shown below, the *HR Reps and Managers Only* rule combines two other authorization schemes. It uses a simple statement like the following that returns `true` if the current user manages others or they are an HR Representative:

```
return    apex_authorization.is_authorized('HR Representatives Only')
       or apex_authorization.is_authorized('Manages Others');
```

Notice it uses the `IS_AUTHORIZED` function in the `APEX_AUTHORIZATION` package to evaluate an authorization scheme by name on behalf of the current user. Since both the *HR Representatives Only* and *Manages Others* schemes test their condition *Once per session*, this programmatic use also benefits from that performance optimization.

Copyright © 2026, Oracle and/or its affiliates.

Figure 11-21 Using PL/SQL Expression to Combine Other Authorization Schemes



11.5.2 Enabling Row-Level Data Security

To define row-level security policies on tables or views, your workspace's parsing schema needs additional privileges your DBA can grant.

You use procedures in the `DBMS_RLS` package to manage data security policies. Once activated, the database enforces them transparently for all application users.

- Ensuring Execute Privilege on DBMS_RLS
  To use row-level data security, your workspace's parsing schema requires the `EXECUTE` privilege on the `DBMS_RLS` package.
- Defining the Security Policy Function
  A row-level security policy associates a *security policy function* with a table or view. This function returns the text of a `WHERE` clause predicate the database automatically applies to requested operations on the table or view.
- Adding and Enabling the Security Policy
  To add and enable the security policy, choose the target table or view and call the `ADD_POLICY` procedure in the `DBMS_RLS` package. Name the policy clearly, and indicate table or view to protect and the associated security policy function.
- Referencing Other Context Information
  In addition to calling `APEX_AUTHORIZATION.IS_AUTHORIZED`, your data security policy function can use other useful information to decide what `WHERE` clause to return.

Copyright © 2026, Oracle and/or its affiliates.

11.5.2.1 Ensuring Execute Privilege on DBMS_RLS

To use row-level data security, your workspace's parsing schema requires the `EXECUTE` privilege on the `DBMS_RLS` package.

It contains the procedures to manage data security policies. Run the following query to check if you already have the privilege to access this package:

```
SELECT 'DBMS_RLS accessible' as result
FROM dual
WHERE EXISTS (
    SELECT 1 FROM all_objects
    WHERE object_name = 'DBMS_RLS'
    AND object_type = 'PACKAGE'
    AND owner = 'SYS')
```

If the result is *No Rows Found*, then ask your DBA to run the following grant statement:

```
grant execute on dbms_rls to your_workspace_schema
```

11.5.2.2 Defining the Security Policy Function

A row-level security policy associates a *security policy function* with a table or view. This function returns the text of a `WHERE` clause predicate the database automatically applies to requested operations on the table or view.

To let you write more generic policies, two required parameters provide the name of the schema and table involved. The function returns a `VARCHAR2` containing the predicate.

```
function my_policy_function (
    schema_var in varchar2,
    table_var  in varchar2)
    return        varchar2
```

If no rows should be visible to the current user, the function returns the predicate `1=0` which is always false. If all rows should be visible, it returns `1=1`. Otherwise, it returns an appropriate predicate to enforce the rows the current user is able to see.

The `ONLY_OWN_REPORTS` policy function appears below. Notice it uses the `IS_AUTHORIZED` function in the `APEX_AUTHORIZATION` package to return:

- `1=1` – if the user is an HR Representative
- `empno in (` – if the user manages others, and
- `1=0` – otherwise.

Notice the function accesses the name of the current APEX user using the `sys_context` function.

```
function only_own_reports(
    schema_var in varchar2,
    table_var  in varchar2)
    return         varchar2
is
```


Copyright © 2026, Oracle and/or its affiliates.

```
    l_user_empno number;
    l_predicate varchar2(4000);
    l_apex_user varchar2(255);
begin
    -- Get the actual APEX application user
    l_apex_user := sys_context('APEX$SESSION','APP_USER');

    -- If no APEX session context, deny access
    if l_apex_user is null then
        return '1=0';
    end if;

    -- If user has HR Rep role, they can see all rows
    if apex_authorization.is_authorized('HR Representatives Only') then
        return '1=1';
    end if;

    -- If user does not manage others, they see nothing
    if not apex_authorization.is_authorized('Manages Others') then
        return '1=0';
    end if;

    -- If user manages others, they can see their own reports only
    -- Get current user's empno from the employee table
    begin
        select empno
          into l_user_empno
          from eba_demo_emp
         where ename = l_apex_user;
    exception
        when no_data_found then
            -- If user is not found in employee table, deny all access
            return '1=0';
    end;

    -- For managers, return predicate to show direct reports recursively
    l_predicate := 'empno in (
        select empno
        from eba_demo_emp
        start with mgr = ' || l_user_empno || '
        connect by prior empno = mgr)';

    RETURN l_predicate;
END only_own_reports;
```

11.5.2.3 Adding and Enabling the Security Policy

To add and enable the security policy, choose the target table or view and call the `ADD_POLICY` procedure in the `DBMS_RLS` package. Name the policy clearly, and indicate table or view to protect and the associated security policy function.

Copyright © 2026, Oracle and/or its affiliates.

Since the *Employee Directory* needs to show all employees, create the following `EBA_DEMO_EMP_V` view to use for the *Salary Review* page:

```
create view eba_demo_emp_v as
select empno,
       ename,
       job,
       mgr,
       hiredate,
       sal,
       comm,
       deptno
from eba_demo_emp;
```

To enforce data security on the `EBA_DEMO_EMP_V` view using the `ONLY_OWN_REPORTS` policy function, use the following PL/SQL block. It first drops, then *adds* and *enables* a policy named `ONLY_OWN_REPORTS_POLICY`:

```
-- Create the row-level data security policy on the view
begin
    -- First, try to drop the policy if it exists (ignore errors)
    begin
        dbms_rls.drop_policy(
            object_schema => sys_context('USERENV','CURRENT_USER'),
            object_name   => 'EBA_DEMO_EMP_V',
            policy_name   => 'ONLY_OWN_REPORTS_POLICY');
    exception
        when others then
            null; -- Ignore errors if policy doesn't exist
    end;

    -- Add the row-level data security policy to the view
    dbms_rls.add_policy(
        object_schema   => SYS_CONTEXT('USERENV','CURRENT_USER'),
        object_name     => 'EBA_DEMO_EMP_V',
        policy_name     => 'ONLY_OWN_REPORTS_POLICY',
        function_schema => SYS_CONTEXT('USERENV','CURRENT_USER'),
        policy_function => 'ONLY_OWN_REPORTS',
        statement_types => 'SELECT,INSERT,UPDATE,DELETE',
        update_check    => true,
        enable          => true);
end;
```


Copyright © 2026, Oracle and/or its affiliates.

Tip

To see all policies you've applied, run this query:

```
SELECT
    object_name,
    policy_name,
    function,
    policy_type,
    enable,
    sel, ins, upd, del,
    chk_option
FROM USER_POLICIES
ORDER BY object_name, policy_name
```

11.5.2.4 Referencing Other Context Information

In addition to calling `APEX_AUTHORIZATION.IS_AUTHORIZED`, your data security policy function can use other useful information to decide what `WHERE` clause to return.

To access the current APEX:

- *Username* – use `SYS_CONTEXT('APEX$SESSION','APP_USER')`
- *Application ID* – use `SYS_CONTEXT('APEX$SESSION','APP_ID')`
- *Session ID* – use `SYS_CONTEXT('APEX$SESSION','APP_SESSION')`
- *Tenant ID* – use `SYS_CONTEXT('APEX$SESSION','APP_TENANT_ID')`

It can also access the list of APEX ACL roles assigned to the user with the query:

```
select role_name
  from apex_appl_acl_user_roles
 where user_name = sys_context('APEX$SESSION','APP_USER')
   and application_id = sys_context('APEX$SESSION','APP_ID')
```

When using dynamic groups with an external identity provider, it can access the list of dynamic groups the user belongs to with the query:

```
select group_name
  from apex_workspace_session_groups
 where apex_session_id = sys_context('APEX$SESSION','APP_SESSION')
   and user_name = sys_context('APEX$SESSION','APP_USER')
```


Copyright © 2026, Oracle and/or its affiliates.

> **Tip**
>
> The *Tenant ID* is a context value you can assign in an *After Authentication* application process or a post-authentication procedure using the `SET_TENANT_ID` procedure in the `APEX_SESSION` package. In a multi-*tenant* app – where users from different customer *companies* access it – the value represents the unique id of the current user's company. Your data security policies can use it to ensure users only see data pertaining to their own company. However, you can set and use the value in any way that proves useful to you.

11.6 Refining Salary Review with Data Security

After setting up the view and row-level data security policy, you can enhance your app to use them.

You now have the following new elements in place:

- `EBA_DEMO_EMP_V` view to anchor the data security policy
- `ONLY_OWN_REPORTS_POLICY` data security policy added and enabled on the view
- *Manages Others* and *HR Reps and Managers Only* authorization schemes.

You are ready to adjust the *Salary Review* page to implement the requested new functionality.

- Switching to Data Security-Enabled View
  When a region uses a table or view with a row-level data security policy, it inherits the enforcement automatically.
- Driving Read-Only with Authorization Rule
  Use the `IS_AUTHORIZED` function in a PL/SQL Read Only expression to decide when data can be edited.
- Adjusting Salary Review Page Access
  You can adjust the salary review page access by changing the associated authorization scheme.
- Enabling Salary Review Menu for Managers
  You can modify the authorization scheme on the navigation menu entry to change who has access to it.
- Testing Enhanced Salary Review Page
  At runtime, authorization schemes and row-level data-security work together to grant data and feature access based on a user's role.

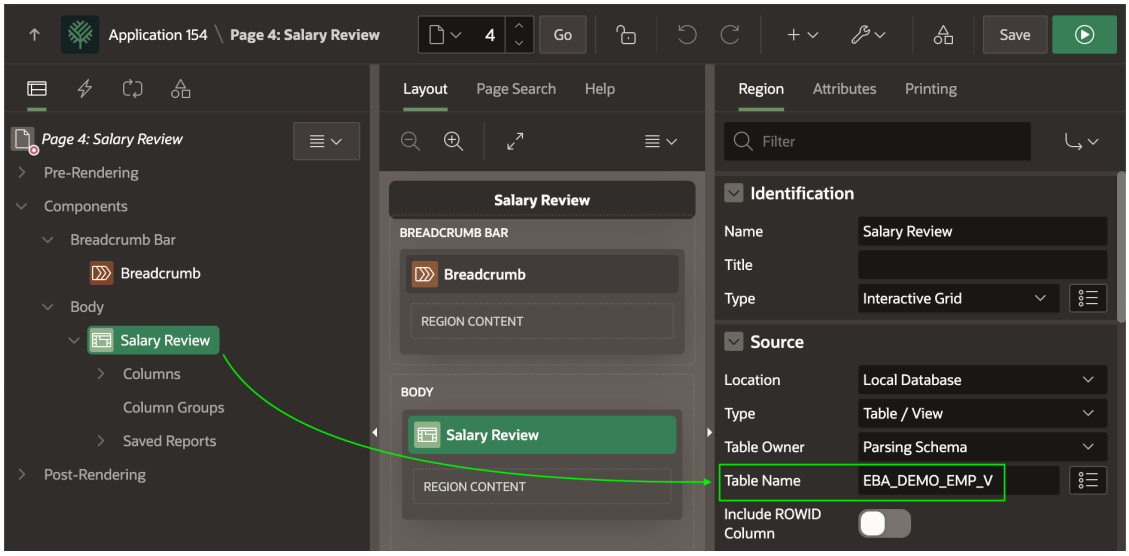
11.6.1 Switching to Data Security-Enabled View

When a region uses a table or view with a row-level data security policy, it inherits the enforcement automatically.

As shown below, start by changing the *Table Name* the *Salary Review* Interactive Grid region is using to instead use the `EBA_DEMO_EMP_V` view. The *Salary Review* page transparently inherits the data security policy enforcement simply by accessing the view.

Copyright © 2026, Oracle and/or its affiliates.

Figure 11-22 Change Salary Review to Use View with Data Security Policy



11.6.2 Driving Read-Only with Authorization Rule

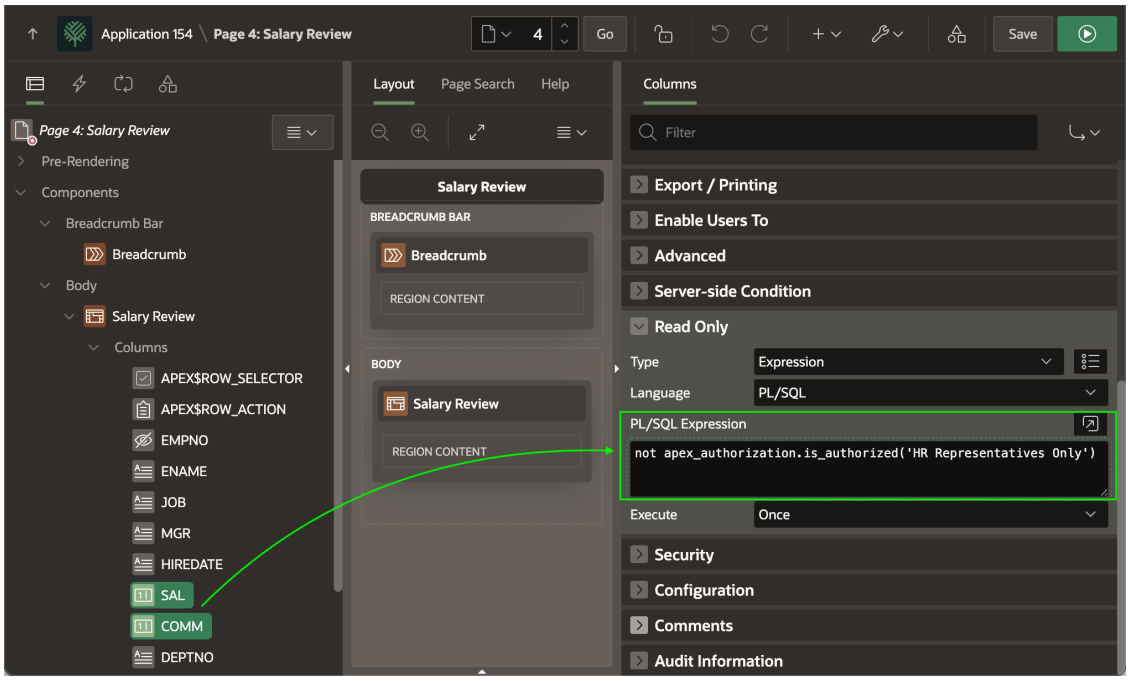
Use the `IS_AUTHORIZED` function in a PL/SQL Read Only expression to decide when data can be edited.

The *Salary Review* page needs to let HR Representatives edit *Salary* and *Commission*, but managers should see a read-only view of their own reports only. To make these two columns *read-only* for users who are *not* HR Representatives, as shown below, use a PL/SQL expression to configure their *Read Only* behavior. These two columns are read-only when:

```
not apex_authorization.is_authorized('HR Representatives Only')
```


Copyright © 2026, Oracle and/or its affiliates.

Figure 11-23 Make Sal and Comm Columns Read Only Unless User is HR Representative

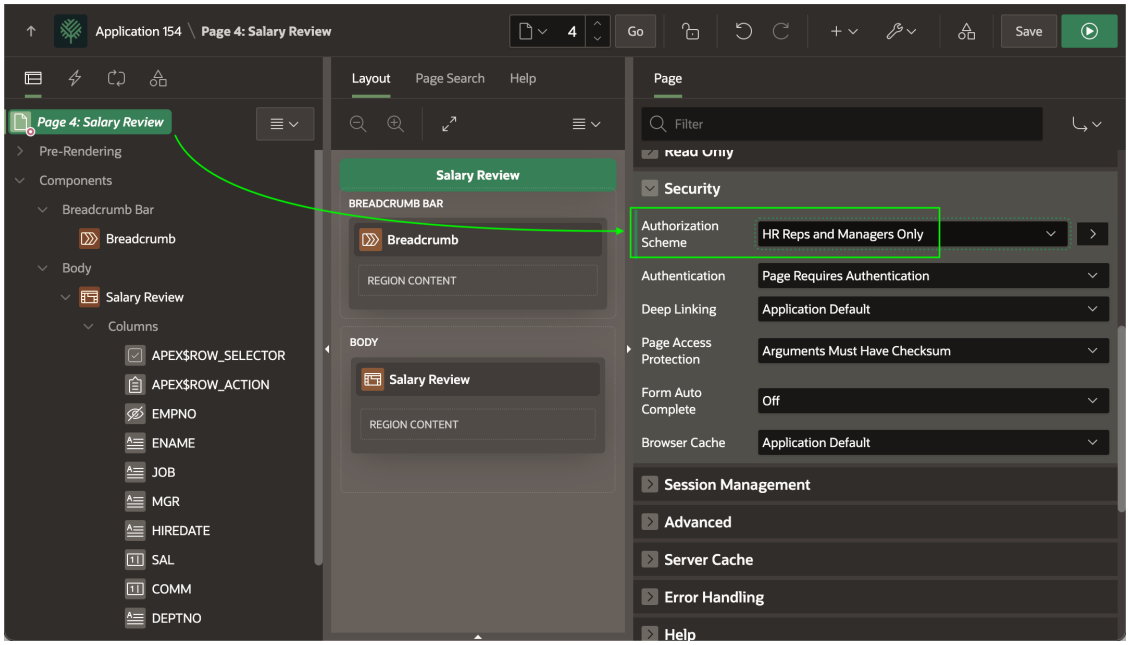


11.6.3 Adjusting Salary Review Page Access

You can adjust the salary review page access by changing the associated authorization scheme.

To let *both* HR Representatives and managers access the *Salary Review* page, as shown below, you adjust its *Authorization Scheme* to use the new *HR Reps and Managers Only* rule.

Figure 11-24 Adjust Authorization Scheme to Let Managers and HR Reps Access



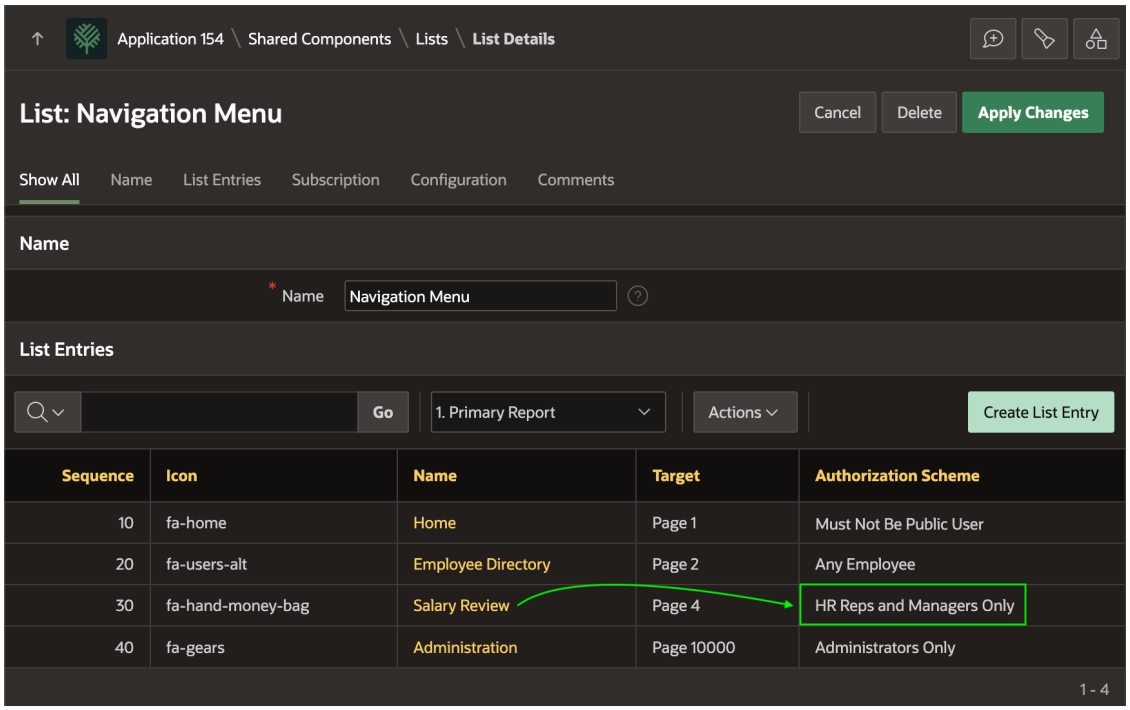
Copyright © 2026, Oracle and/or its affiliates.

11.6.4 Enabling Salary Review Menu for Managers

You can modify the authorization scheme on the navigation menu entry to change who has access to it.

To enable the *Salary Review* navigation menu entry for managers, in addition to HR Reps, as shown below, you change the authorization scheme to use *HR Reps and Managers Only*.

Figure 11-25 Adjust Menu Entry Authorization Scheme for Managers and HR Reps



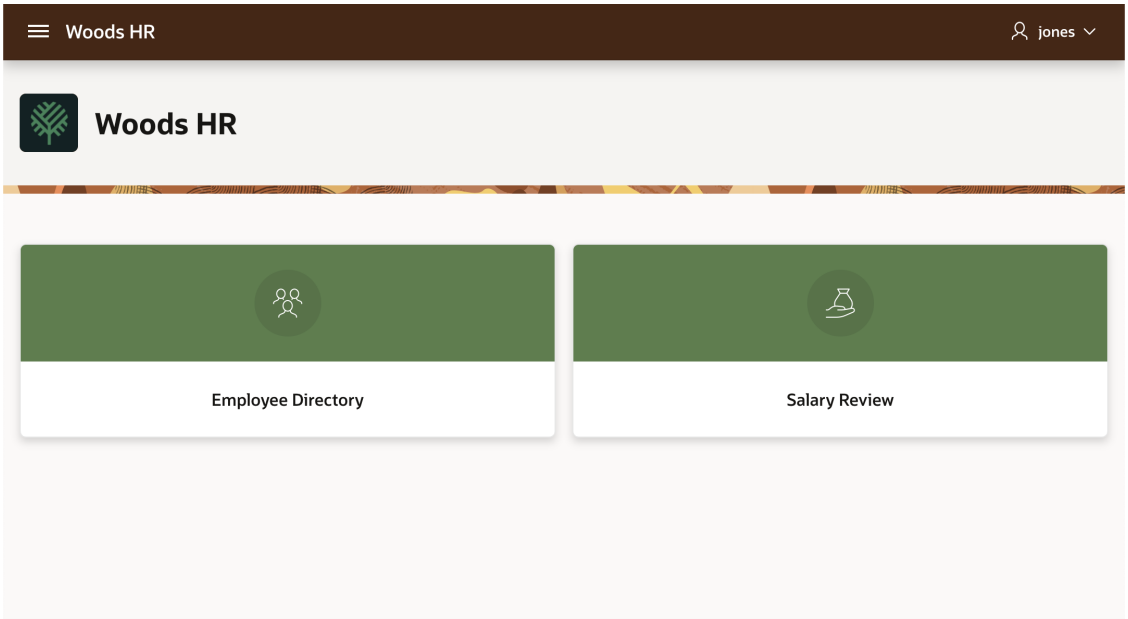
11.6.5 Testing Enhanced Salary Review Page

At runtime, authorization schemes and row-level data-security work together to grant data and feature access based on a user's role.

With all your changes in place, when a manager like `JONES` logs in, he now sees the *Salary Review* page in his navigation menu as shown below.

Copyright © 2026, Oracle and/or its affiliates.

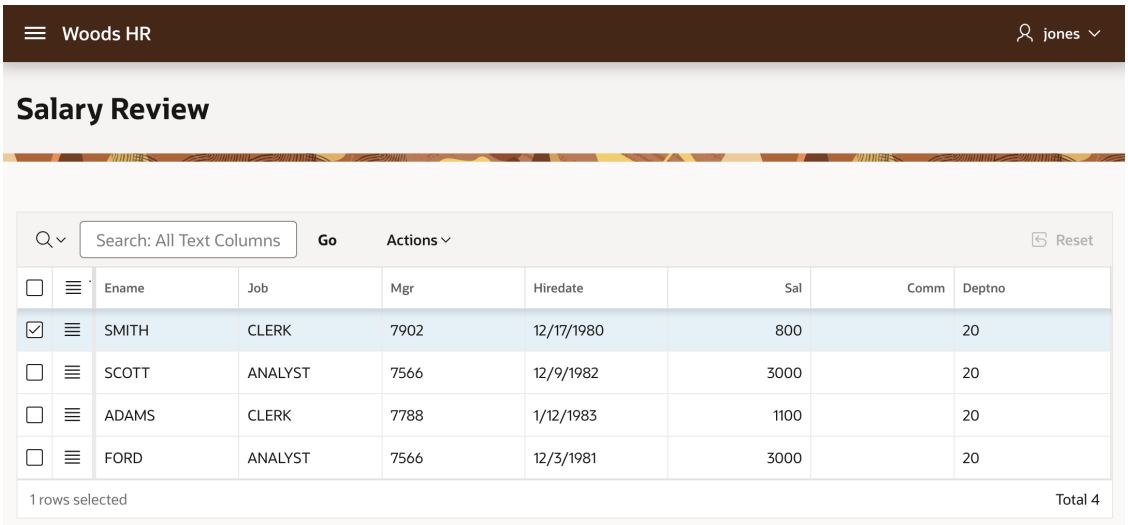
Figure 11-26 Now Managers Like Jones See the Salary Review Page



Opening the page, as shown below, `JONES` sees the four reports that either report directly to him, or to one of the managers who work for him. The data security policy automatically restricts the data `JONES` can see. Also, the *Sal* and *Comm* columns are read-only as requested.

When an HR Representative like `SUSAN` logs in, the page continues to let her see all employees and edit their salary and commission as before.

Figure 11-27 Data Security Ensures Jones Sees Only Own Staff Members



Copyright © 2026, Oracle and/or its affiliates.

11.7 Integrating an External Identity Provider

While APEX Accounts are easy and built-in, you might *already* manage users and roles with an external identity provider. If so, create a *Social Sign-in* type Authentication Scheme and *after authentication* dynamically enable the groups users belong to.

This scheme type authenticates users with Google, Facebook, or any provider supporting OpenID Connect or OAuth 2.0. You can use the *Identity and Access Management* (IAM) service on Oracle Cloud Infrastructure (OCI) to experiment with setting it up. The basic OCI IAM features shown here work in an *Always Free* OCI tenancy as well. Using another external identity provider requires similar steps.

Assume you create an IAM domain called *WoodsHR* for use by the Woods HR app in this section. Among other identity-related artifacts, an IAM domain contains users, groups, user/group assignments, and integrated application clients. To easily compare APEX Accounts with an external identity provider, configure the same users and roles.

> **Tip**
>
> Other IAM features beyond these basics may require a paid tenancy.

- Defining Users in an IAM Domain
  Define your IAM domain users on the *User management* tab.
- Creating Groups in an IAM Domain
  OCI *Identity and Access Management* (IAM) uses the term "group" instead of role. Create groups on your domain's **User management** tab.
- Assigning IAM Domain Users to Groups
  Assign users to a group on the *Users* tab of the group edit page.
- Configuring Confidential App for APEX
  A domain's *Integrated applications* tab lists applications you allowed to authenticate through it.
- Defining Credential for Identity Provider
  Before defining the *Social Sign-in* Authentication Scheme to use your external identity provider, start by defining the Web Credential it requires in Workspace Utilities.
- Authenticating with an External Provider
  With the web credential defined, create a new *Social Sign-in* Authentication Scheme that references the Web Credential protecting the OAuth client ID and secret for the confidential app used for authentication.
- Enabling Groups in Post-Auth Procedure
  To use the groups defined in your external identity provider, write a post authentication procedure and configure your *Social Sign-in* Authentication Scheme to invoke it.
- Sourcing App Groups from Custom Code
  When using `ENABLE_DYNAMIC_GROUPS` in a post-authentication procedure to set the groups the current user belongs to, as shown below, set the *Source for Role or Group Schemes* application security setting to *Custom Code*.
- Switching Authorization Rules to Custom
  When using `ENABLE_DYNAMIC_GROUPS`, edit your authorization schemes to change their *Type* to *Custom*.

Copyright © 2026, Oracle and/or its affiliates.

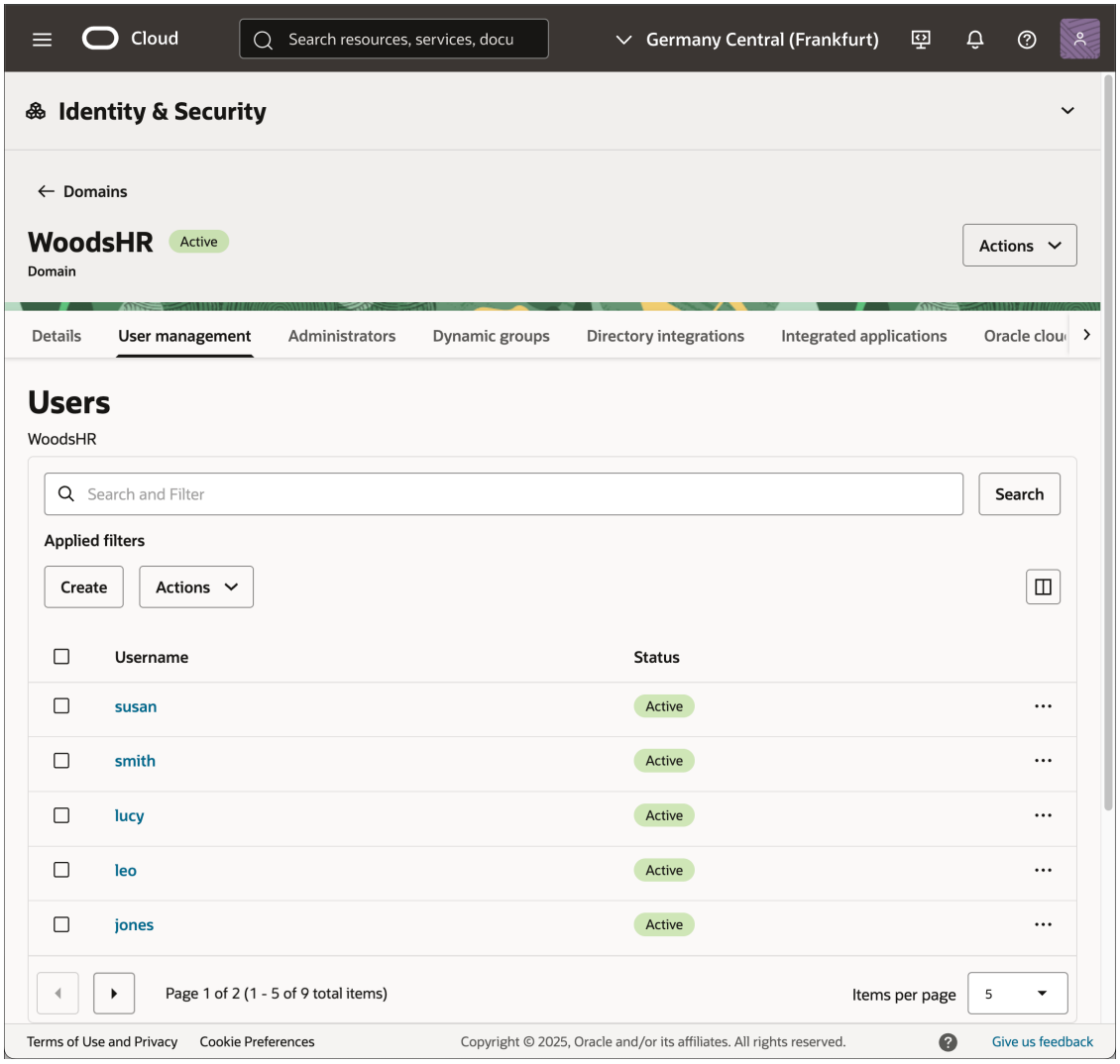
- Making Social Sign-in the Current Scheme
  To try out a new authentication scheme at runtime in your application, make it the current scheme.
- Experiencing External Identity Provider
  With all the changes ready, run the app to test out authentication using the external identity provider.

11.7.1 Defining Users in an IAM Domain

Define your IAM domain users on the *User management* tab.

In the context of your *WoodsHR* domain, as shown below, define your users on the domain's *User management* tab. Notice all the same usernames from your APEX Accounts list are there.

Figure 11-28 Domain Users in Identity Cloud Service



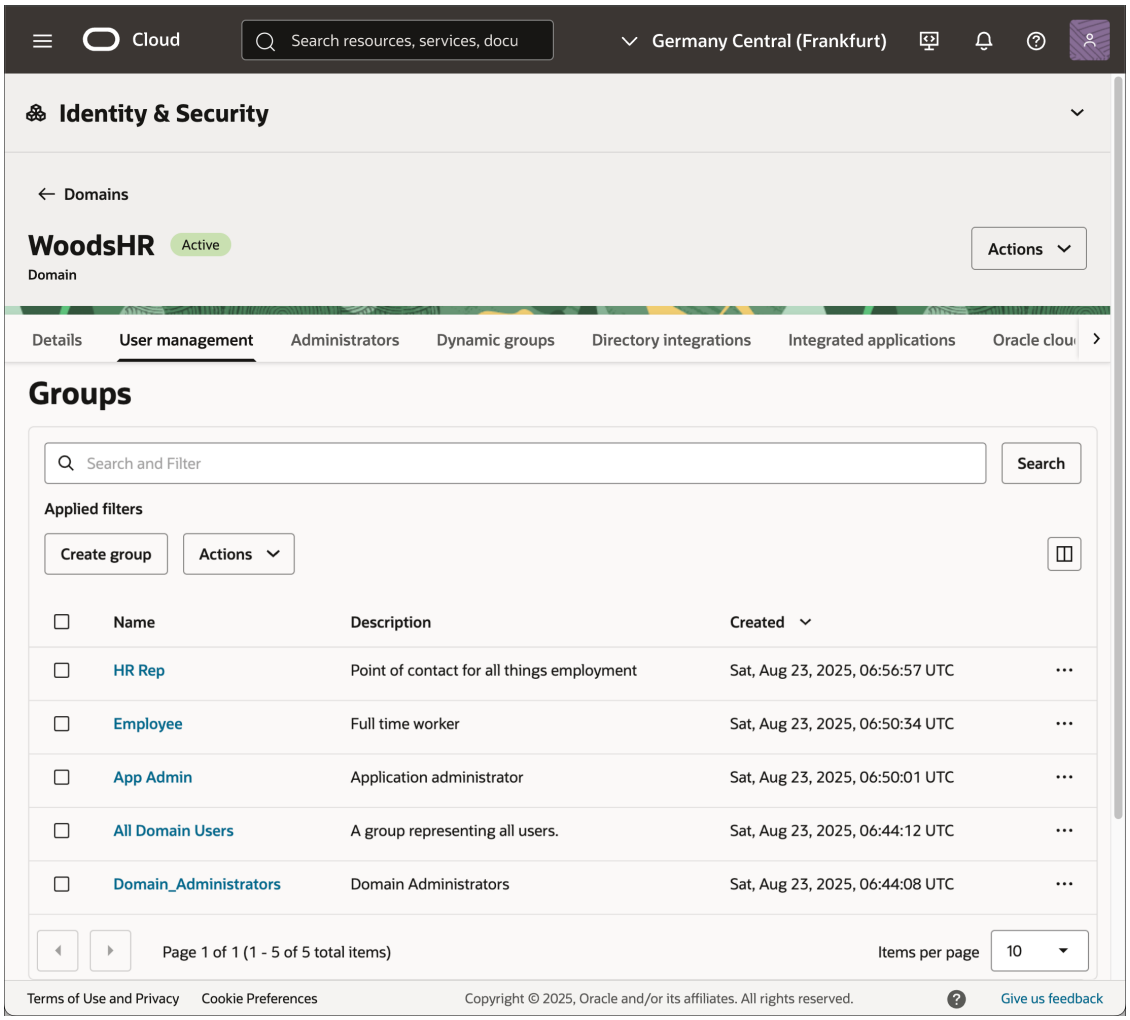
Copyright © 2026, Oracle and/or its affiliates.

11.7.2 Creating Groups in an IAM Domain

OCI *Identity and Access Management* (IAM) uses the term "group" instead of role. Create groups on your domain's **User management** tab.

In addition to the two default groups named *All Domain Users* and *Domain_Administrators*, as shown below, you create the three groups named after the APEX Roles: *Employee*, *HR Rep*, and *App Admin*.

Figure 11-29 Domain Groups in Identity Cloud Service



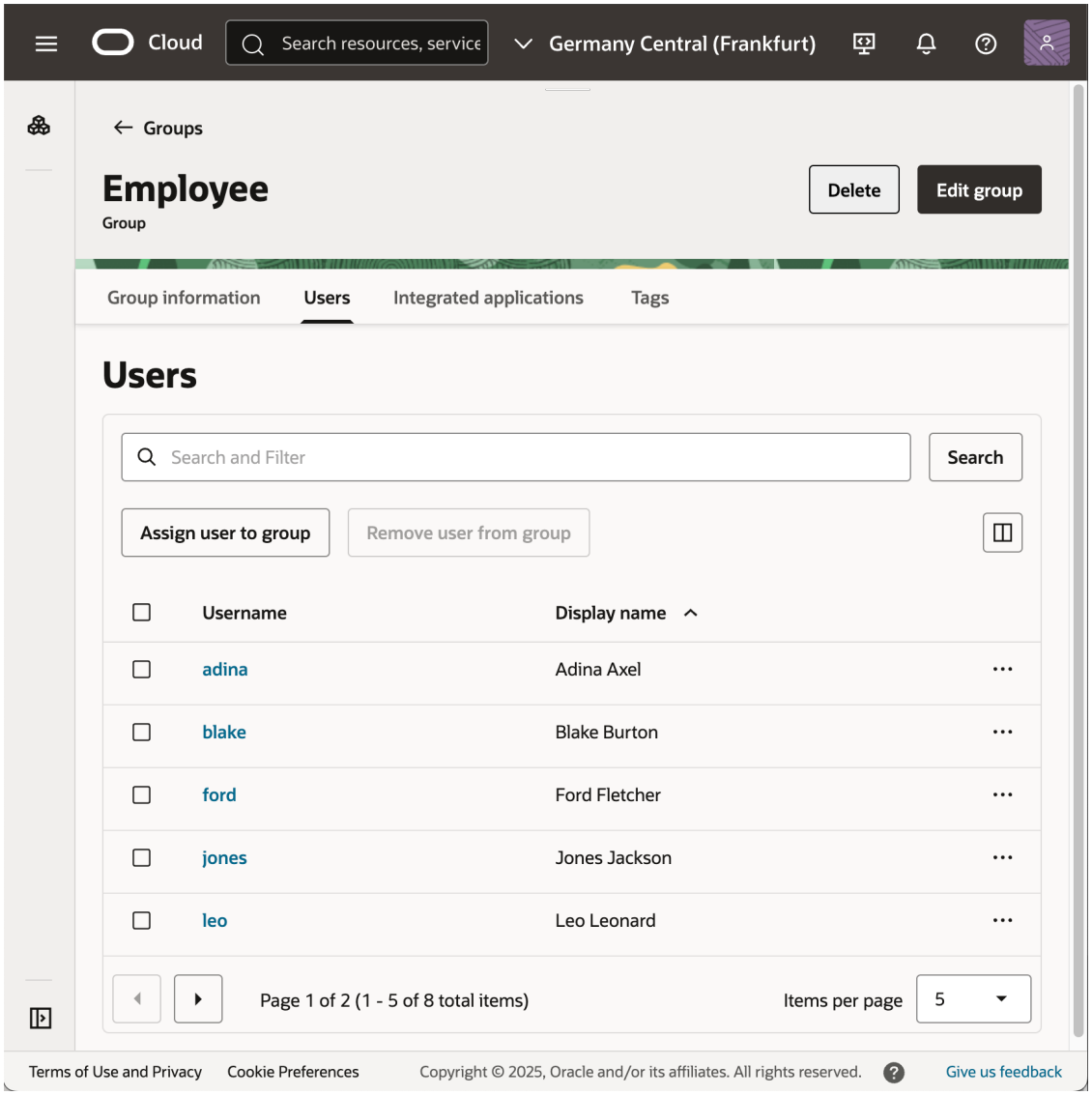
11.7.3 Assigning IAM Domain Users to Groups

Assign users to a group on the *Users* tab of the group edit page.

As shown below, while editing a group like *Employee*, use the *Users* tab to assign users to the group. All eight (8) users are members of the *Employee* group. As in the APEX Accounts example, you use the same approach to assign user `susan` to the *HR Rep* group and user `adina` to the *App Admin* group.

Copyright © 2026, Oracle and/or its affiliates.

Figure 11-30 Domain Group Assignments in Identity Cloud Service



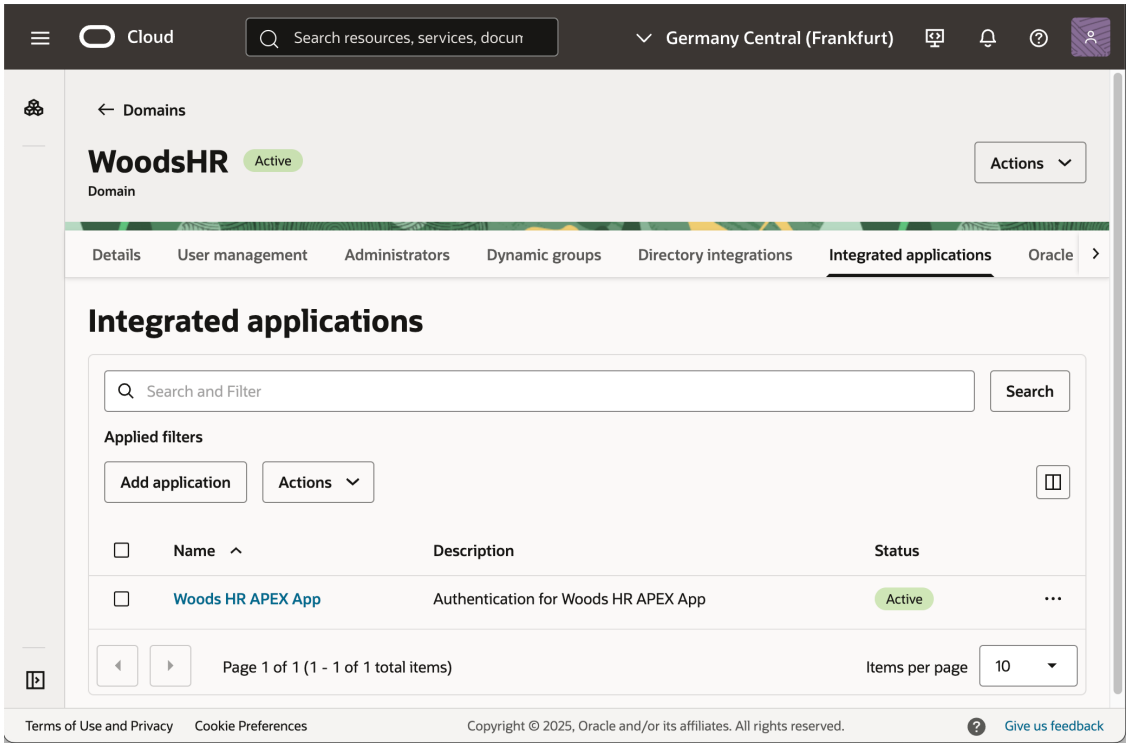
11.7.4 Configuring Confidential App for APEX

A domain's *Integrated applications* tab lists applications you allowed to authenticate through it.

Notice the *Woods HR APEX App* in the list shown below. You created it from this page using the *(Add application)* button and supplying the application URL to the Woods HR APEX application.

Copyright © 2026, Oracle and/or its affiliates.

Figure 11-31 Domain Integrated App for Use by Woods HR APEX Application



After initial creation, as shown below, you edit the *OAuth configuration* to enable:

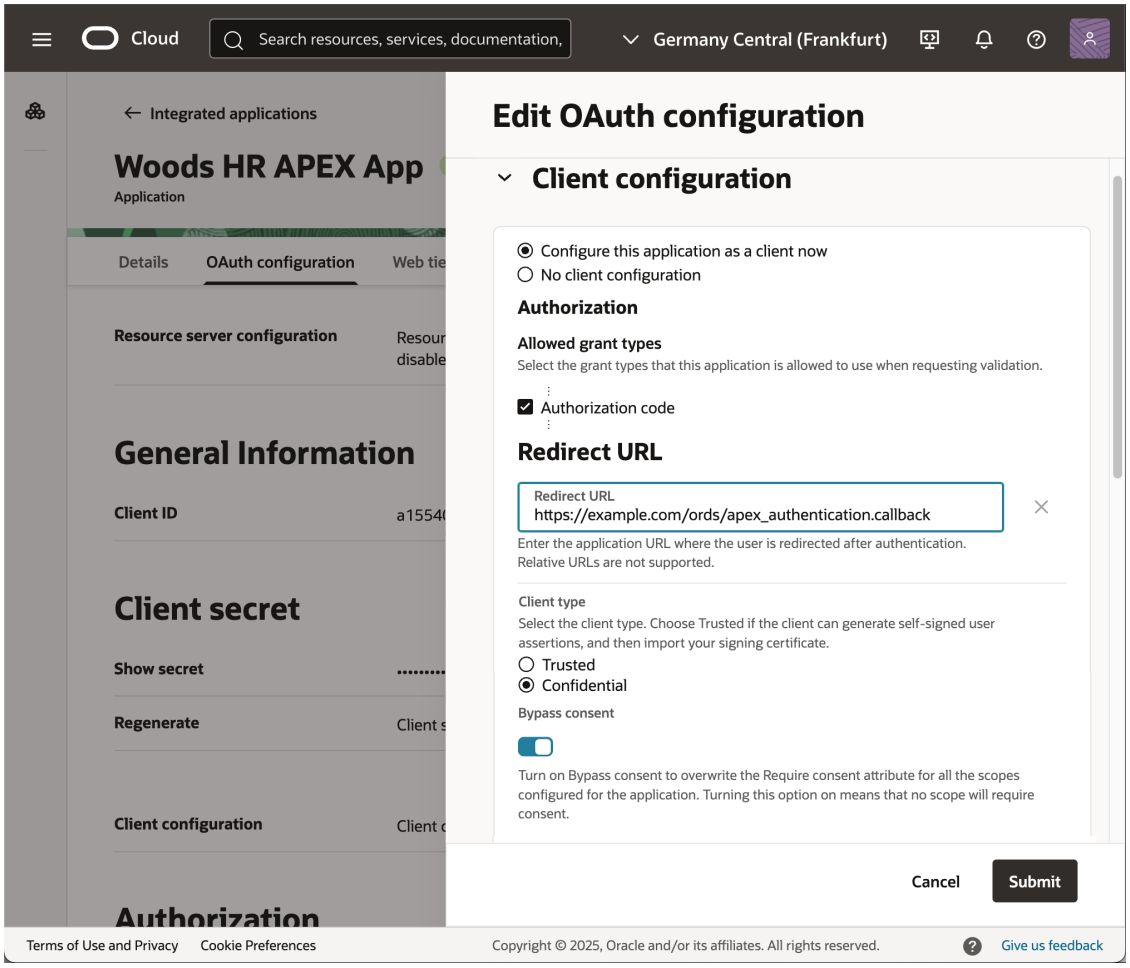
- *Configure this application as a client now* – selected
- *Allowed grant types* – **Authorization code**
- *Redirect URL* – Absolute URL to your APEX instance's `/ords/apex_authentication.callback`
- *Client type* – **Confidential**
- *Bypass consent* – optionally enable to skip user consent to return profile & group info.

This process yields a *Client ID* and *Client Secret* to communicate securely to the app developer who needs to authenticate users using this IAM domain as its identity provider. In this exercise, that is you. The key pieces of information you need are:

- Confidential App Client ID and Secret
- Domain URL
  - `https://idcs-xxxx.identity.oraclecloud.com`
- Discovery URL
  - `https://idcs-xxxx.identity.oraclecloud.com/.well-known/openid-configuration`

Copyright © 2026, Oracle and/or its affiliates.

Figure 11-32 Configuring Domain Integrated App OAuth Client



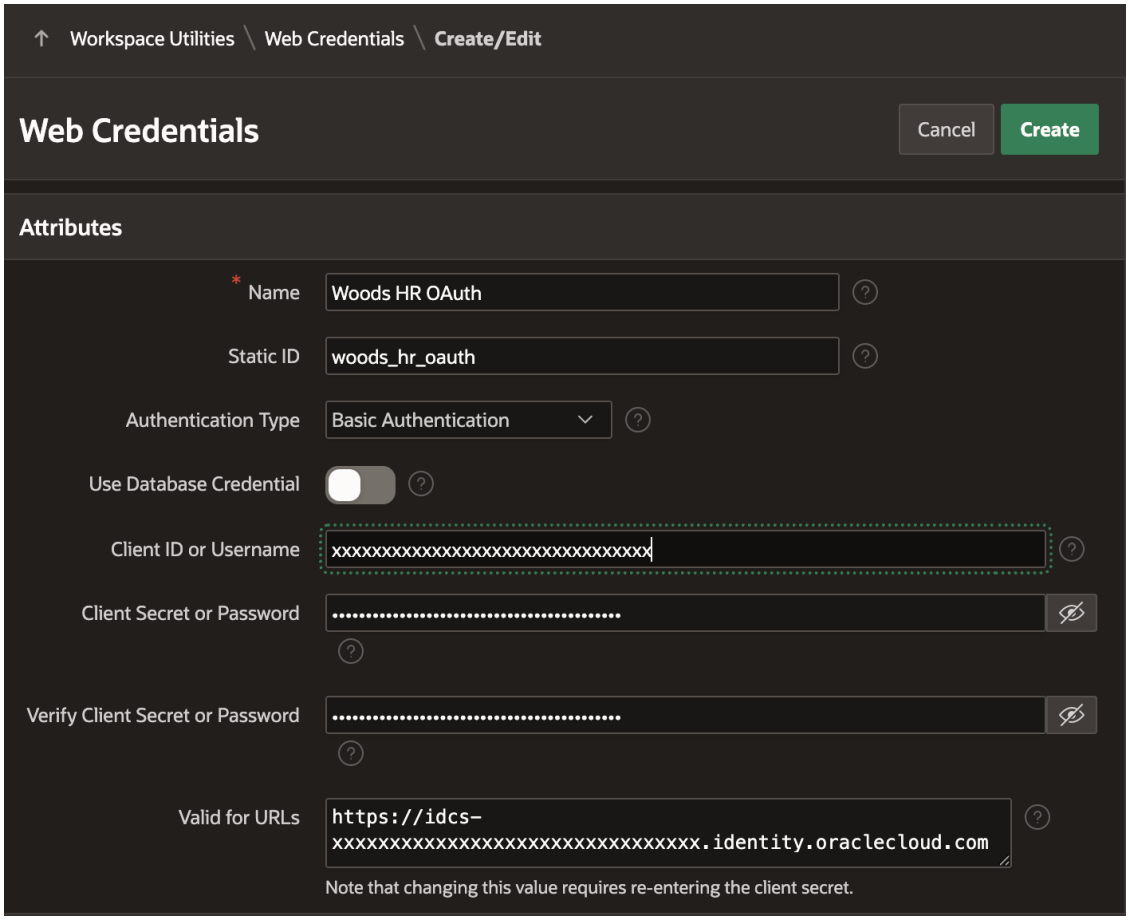
11.7.5 Defining Credential for Identity Provider

Before defining the *Social Sign-in* Authentication Scheme to use your external identity provider, start by defining the Web Credential it requires in Workspace Utilities.

As shown below, the *Woods HR OAuth* credential is of *Basic Authentication* type, securely storing the OCI IAM's confidential application *Client ID* and *Client Secret*. As a best practice, also provide your IAM *Domain URL* in the *Valid for URLs* field to ensure this credential cannot be inadvertently used for an unintended purpose.

Copyright © 2026, Oracle and/or its affiliates.

Figure 11-33 Web Credential for Use with IDCS Social Sign-in Authentication



11.7.6 Authenticating with an External Provider

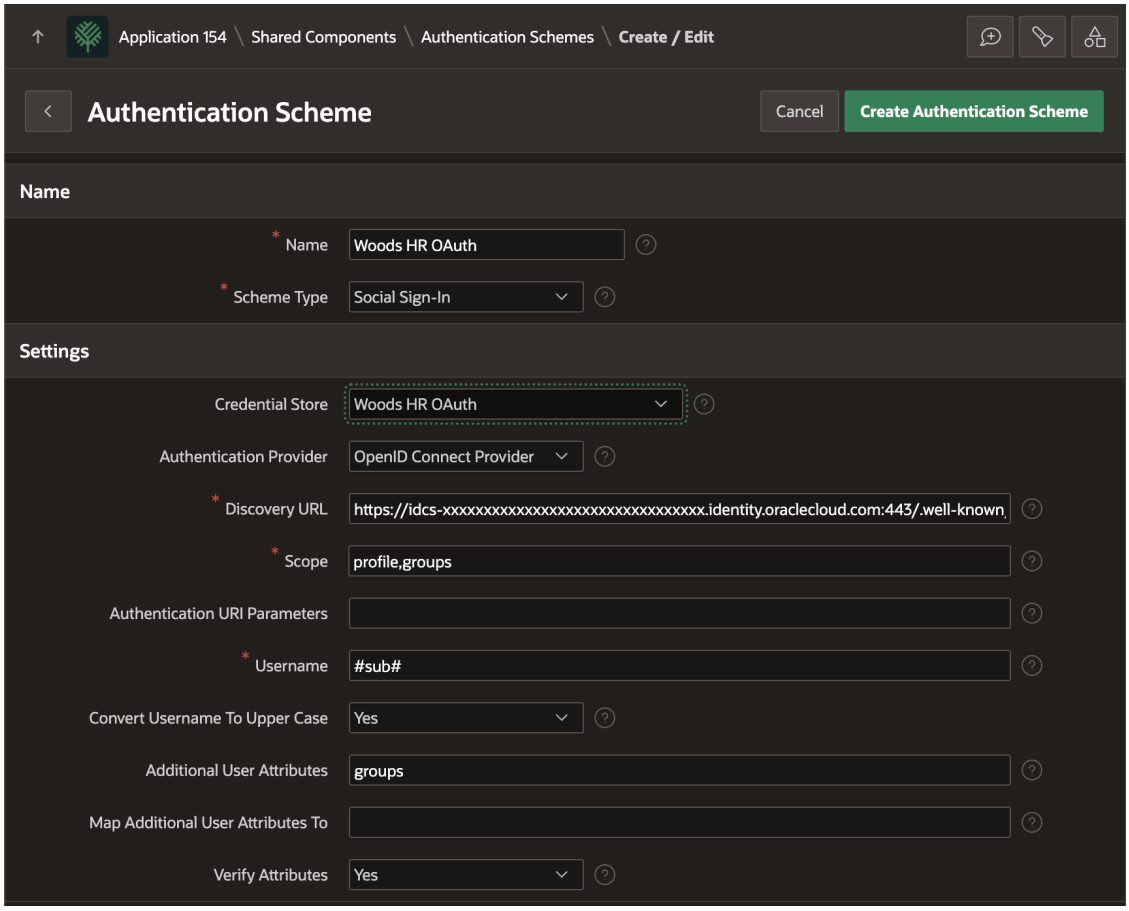
With the web credential defined, create a new *Social Sign-in* Authentication Scheme that references the Web Credential protecting the OAuth client ID and secret for the confidential app used for authentication.

Create a new *Social Sign-in* Authentication Scheme named *Woods HR OAuth*. Configure the following options as shown below:

- *Credential Store* – Choose the web credential you just created
- *Authentication Provider* – Ensure it's **OpenID Connect Provider**
- *Discovery URL* – Enter the discovery URL of your IAM domain: **`https://idcs-xxxx.identity.oraclecloud.com/.well-known/openid-configuration`**
- *Scope* – Enter the comma-separated list: **`profile,groups`**
- *Username* – Ensure it's **`#sub#`** to use the IAM subject at the APEX username
- *Convert Username To Upper Case* – **Yes**
- *Additional User Attributes* – Enter the attribute name: **`groups`**

Copyright © 2026, Oracle and/or its affiliates.

Figure 11-34 Configuring Social Sign-in Authentication Scheme for IDCS



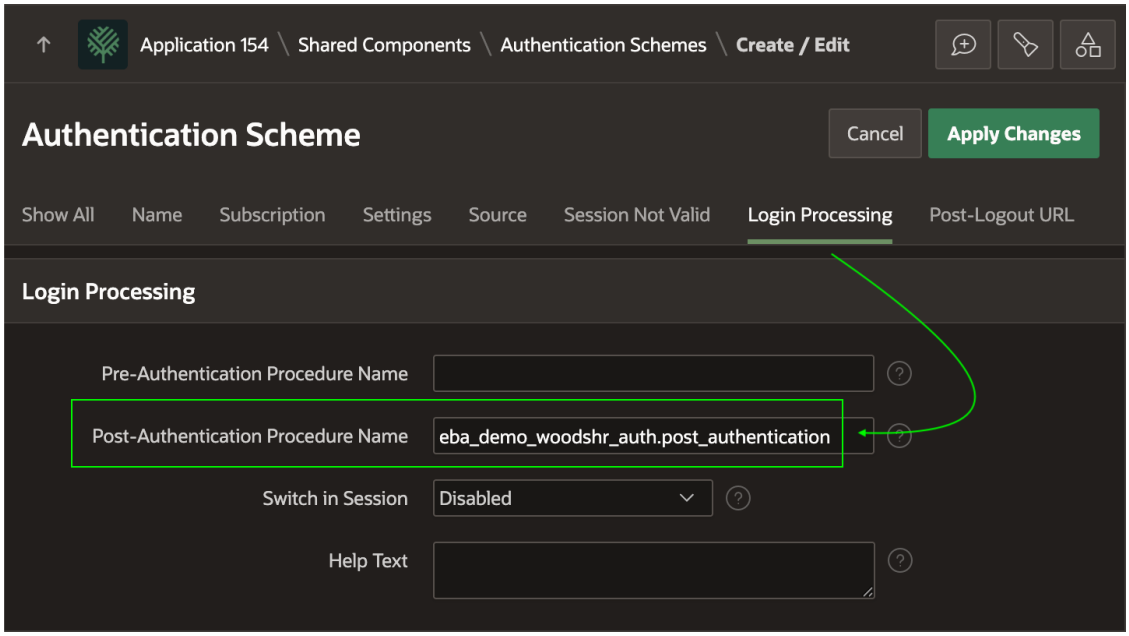
11.7.7 Enabling Groups in Post-Auth Procedure

To use the groups defined in your external identity provider, write a post authentication procedure and configure your *Social Sign-in* Authentication Scheme to invoke it.

As shown below, the *Post-Authentication Procedure Name* field on the *Login Processing* tab can reference the name of a procedure in a package.

Copyright © 2026, Oracle and/or its affiliates.

Figure 11-35 Configuring Post-Authentication Procedure to Enable Groups



Your post-authentication procedure can reference the JSON response from the REST API call made to the external identity provider. Retrieve the group names the user belongs to from this document. They will be present in the JSON based on the value of the *Scope* and *Additional User Attributes* you configured previously.

For example, when user `susan` logs in using the OCI IAM Domain identity provider login page, the response looks like the following JSON document. Notice it contains a `groups` property whose value is an array of JSON objects. The `name` property of each object in this array provides the group name to which `susan` belongs (e.g. `Employee` and `HR Rep`).

```
{
  "family_name": "Sunshine",
  "given_name": "Susan",
  "groups": [
    {
      "id": "xxxxxxxxxxxx",
      "name": "Employee",
      "$ref": "http://dp-admin:9246/admin/v1/Groups/xxxxxxxxxxxx"
    },
    {
      "id": "yyyyyyyyyyyyy",
      "name": "HR Rep",
      "$ref": "http://dp-admin:9246/admin/v1/Groups/yyyyyyyyyyyyy"
    }
  ],
  "name": "Susan Sunshine",
  "preferred_username": "susan",
  "sub": "susan",
  "updated_at": 1755958517
}
```


Copyright © 2026, Oracle and/or its affiliates.

The code for the post-authentication procedure follows. Using two private helper functions, the `post_authentication` procedure:

- gets the most recently parsed JSON document the `APEX_JSON` package processed
- accesses the JSON array property named `groups`
- processes the JSON objects in the `groups` array
  - pushing the value of each one's `name` property into an `APEX_T_VARCHAR` string list
- calls `APEX_AUTHENTICATION.ENABLE_DYNAMIC_GROUPS` to register the user's group names.

> **Tip**
>
> If your external provider requires calling an *additional* service to get group information, do that from the post-authentication procedure using `MAKE_REST_REQUEST` in the `APEX_WEB_SERVICE` package or using a REST Data Source. In this alternate way, you can still produce a list of the group names to enable dynamically.

```
package eba_demo_woodshr_auth is
    procedure post_authentication;
end eba_demo_woodshr_auth;
--
package body eba_demo_woodshr_auth is
    ------------------------------------------------------
    -- Private helper function to return most recently
    -- parsed APEX_JSON document as JSON_OBJECT_T
    ------------------------------------------------------
    function post_auth_json
    return json_object_t
    is
        l_ret json_object_t;
    begin
        if apex_json.g_values.count > 0 then
            apex_json.initialize_clob_output;
            apex_json.write( p_values => apex_json.g_values );
            l_ret := json_object_t(apex_json.get_clob_output);
            apex_json.free_output;
        else
            l_ret := json_object_t();
        end if;
        return l_ret;
    end post_auth_json;
    ------------------------------------------------------
    -- Private helper function to return JSON object in array
    ------------------------------------------------------
    function get_object(
        p_array in json_array_t,
        p_index in pls_integer)
        return     json_object_t
    is
    begin
        return treat(p_array.get(p_index) as json_object_t);
    end get_object;
    ------------------------------------------------------
```


Copyright © 2026, Oracle and/or its affiliates.

```
    -- Public Post Authentication Procedure enables dynamic
    -- groups returned from external identity provider
    ----------------------------------------------------
    procedure post_authentication
    is
        l_auth_data   json_object_t := post_auth_json;
        l_groups_arr  json_array_t;
        l_groups      apex_t_varchar2 := apex_t_varchar2();
    begin
        if     l_auth_data.has('groups')
           and l_auth_data.get('groups').is_array
        then
            l_groups_arr := l_auth_data.get_array('groups');
            for i in 0 .. l_groups_arr.get_size - 1 loop
                apex_string.push(l_groups,
                                 get_object(l_groups_arr,i)
                                 .get_string('name'));
            end loop;
            if l_groups.count > 0 then
                apex_authorization.enable_dynamic_groups(l_groups);
            end if;
        end if;
    end post_authentication;
end eba_demo_woodshr_auth;
```

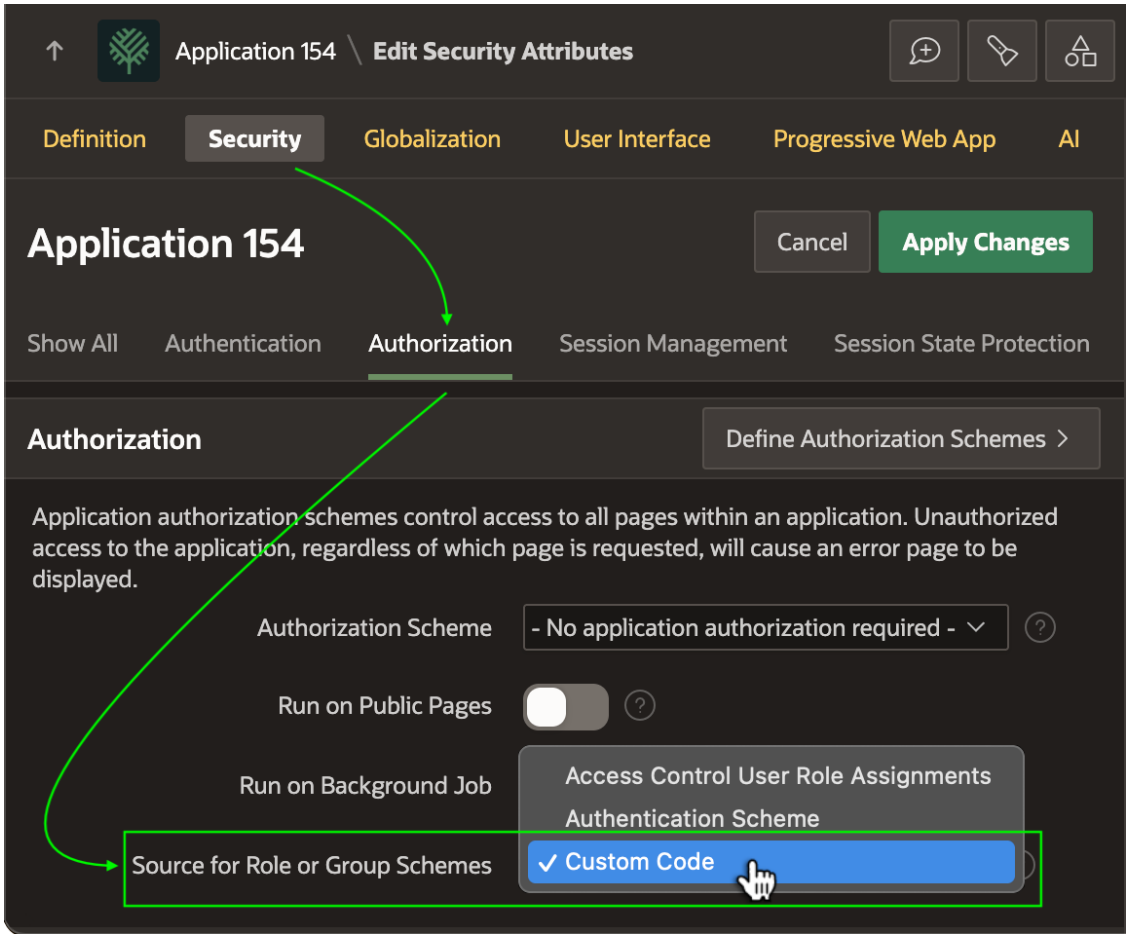
11.7.8 Sourcing App Groups from Custom Code

When using `ENABLE_DYNAMIC_GROUPS` in a post-authentication procedure to set the groups the current user belongs to, as shown below, set the *Source for Role or Group Schemes* application security setting to *Custom Code*.

The figure below shows changing the *Source for Role or Group Schemes* application security setting to *Custom Code* on the Authorization tab of the application's security settings. Access this page under *Shared Components* > *Security Attributes*.

Copyright © 2026, Oracle and/or its affiliates.

Figure 11-36 Changing App to Source Groups from Custom Code



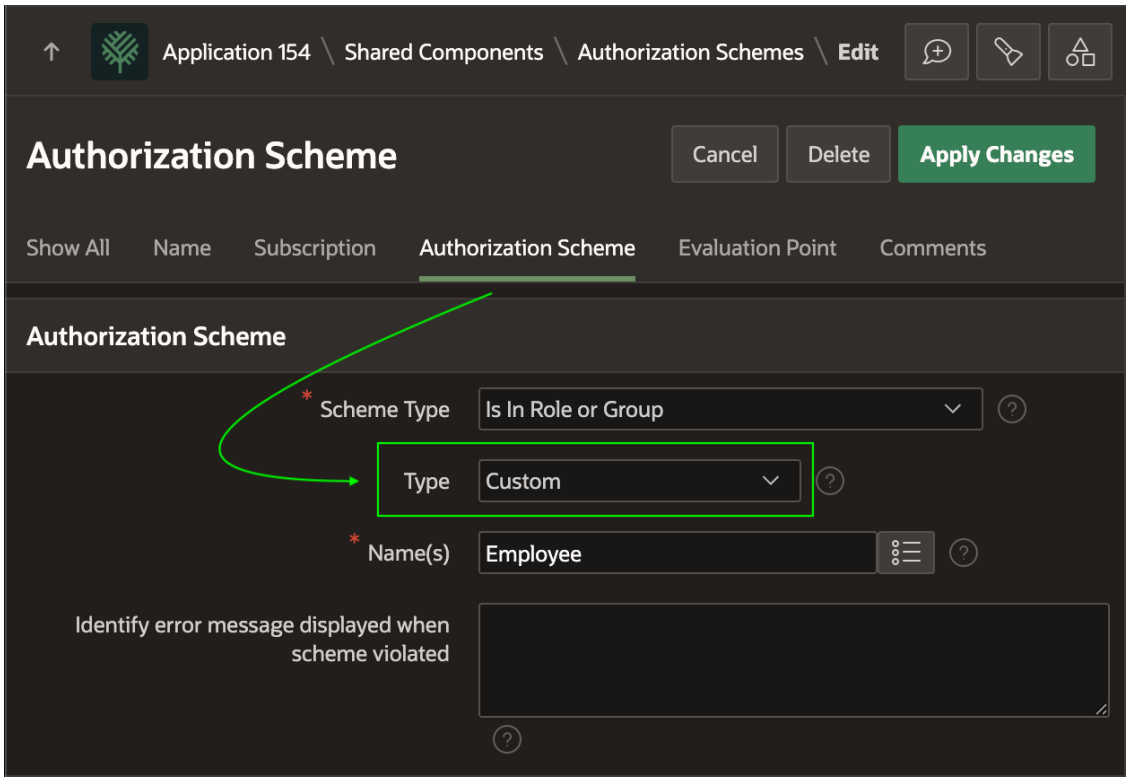
11.7.9 Switching Authorization Rules to Custom

When using `ENABLE_DYNAMIC_GROUPS`, edit your authorization schemes to change their *Type* to *Custom*.

The figure shows setting the *Type* of the *Employee* authorization scheme to *Custom*. This ensures the scheme evaluates group membership against the dynamic groups you enable with custom code.

Copyright © 2026, Oracle and/or its affiliates.

Figure 11-37 Changing Authorization Schemes to Use Custom Type for Dynamic Groups



11.7.10 Making Social Sign-in the Current Scheme

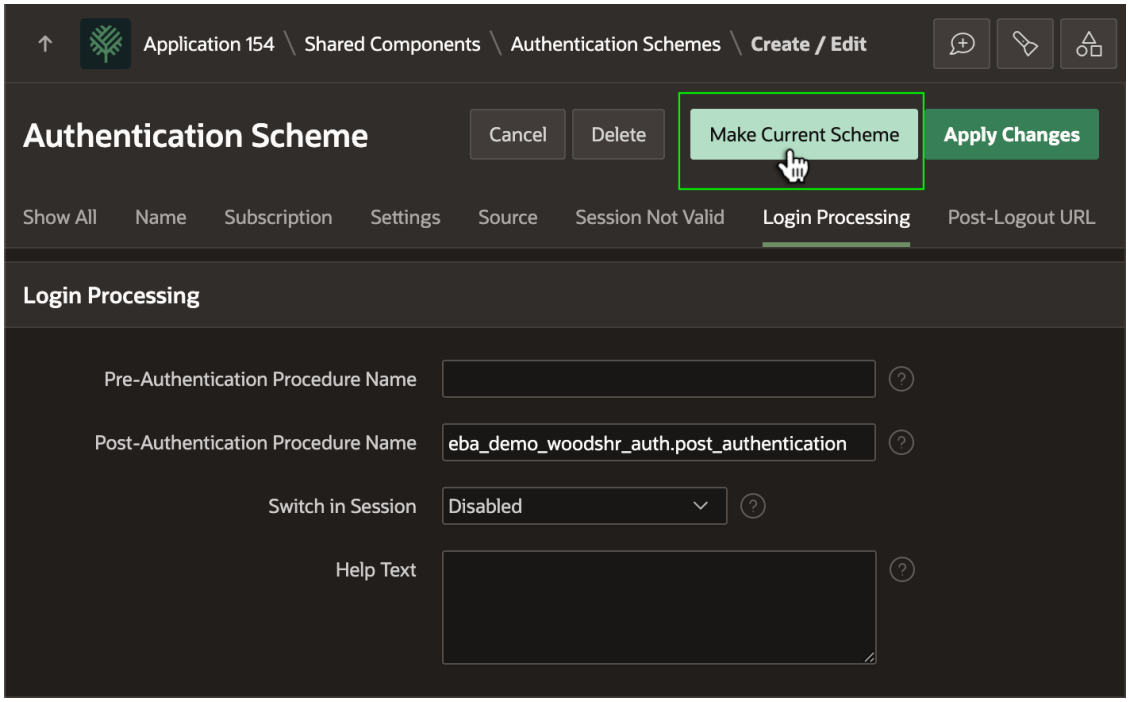
To try out a new authentication scheme at runtime in your application, make it the current scheme.

As the final step, as shown below, edit the *Woods HR OAuth* social sign-in Authentication Scheme and click the *(Make Current Scheme)* to make it the current one to use by default.

> **Tip**
>
> For more advanced scenarios, you can enable *Switch in Session* and dynamically determine by user which of multiple authentication schemes to use. For more information, see Using a Procedure to Configure Authentication at Runtime in *Oracle APEX App Builder User's Guide*.

Copyright © 2026, Oracle and/or its affiliates.

Figure 11-38 Making Social Sign-in Authentication Scheme the Current One



11.7.11 Experiencing External Identity Provider

With all the changes ready, run the app to test out authentication using the external identity provider.

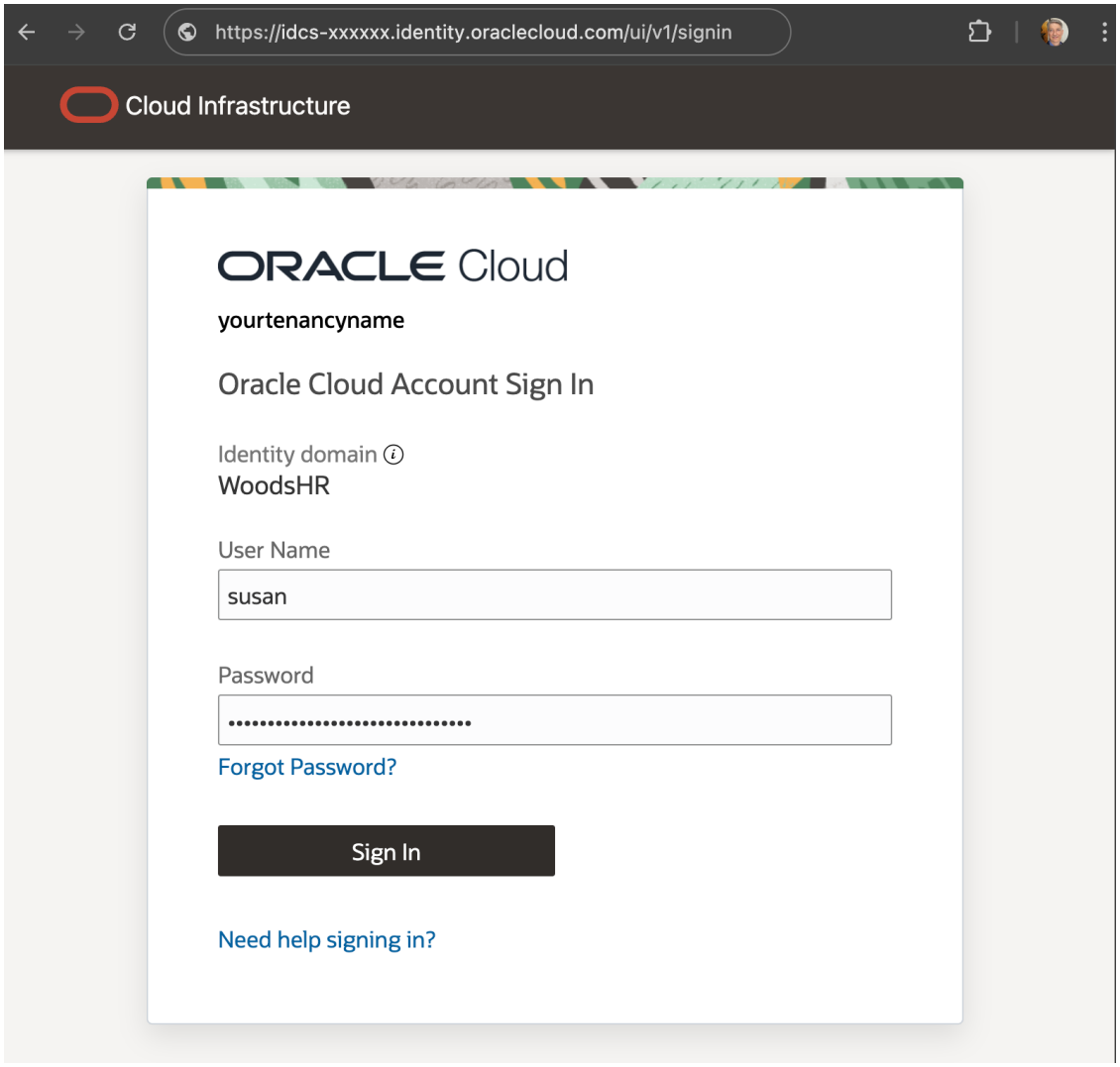
You put all the ingredients in place:

- OCI IAM Domain with users, groups, group assignments, and a confidential app
- Web credential to securely store confidential app's client id and secret
- Social sign-in authentication scheme with credential and Open ID Connect discovery URL
- Post-authentication procedure to enable user's groups with custom code
- Application uses custom code to resolve authorization schemes
- Authorization schemes use custom type to resolve *Is In Role/Group* membership.

Now, when end users run the app, as shown below, they see the external identity provider's login page. Once users login, the rest of the application behaves identically to before, including all the role-based access control.

Copyright © 2026, Oracle and/or its affiliates.

Figure 11-39 Users Now Login to Woods HR Using OCI IAM Sign In



11.8 Exploring Context for Authorization Rules

Depending on your authorization scheme's evaluation point, additional context information is available to write more generic, data-driven rules.

So far, you used authorization schemes with an evaluation point of *Once per session* and *Once per page view*. Rules evaluated only once per session are appropriate when the decision to grant or deny access relies only on the logged-in username and the roles or groups they belong to.

If the rule depends on information that might change during the user's session, then *Once per page view* rules using SQL or PL/SQL can reference standard bind variables to reference additional context information like:

- `APP_USER` – Current logged-in username (or `nobody` if not logged in)
- `APP_ID` – Current application ID
- `APP_ALIAS` – Current application alias

Copyright © 2026, Oracle and/or its affiliates.

- `APP_PAGE_ID` – Current page ID
- `APP_PAGE_ALIAS` – Current page alias
- `WORKSPACE_ID` – Current workspace ID

If your authorization scheme uses an evaluation point of *Once per component* or *Always (No Caching)*, then your SQL or PL/SQL can reference three additional bind variables:

- `APP_COMPONENT_TYPE` – Dictionary view name containing current component
- `APP_COMPONENT_NAME` – Name of the current component
- `APP_COMPONENT_ID` – Unique ID of the current component in its dictionary view.

This additional context information lets you design more generic, data-driven rules your app administrator might configure from an application settings page. If you do, for best performance consider caching the result for each unique combination of factors that affect the authorization decision.

Copyright © 2026, Oracle and/or its affiliates.

12

Applying Artificial Intelligence

Add AI features that help users explore data, ask questions, generate text, complete requests, and find results by meaning.

Start by configuring the Generative AI Service your app uses. From there, you can let users shape Interactive Reports with natural language, define AI Agents to achieve a goal with tools that retrieve permitted app data or perform allowed actions, add an agent-driven chatbot, and generate draft text users can review and refine. You also learn how vector search helps users find relevant rows by meaning, even when their words do not match the data.

- Configuring a Generative AI Service
  Specify a Generative AI service by choosing the provider, endpoint, model, and credentials your apps use.
- Exploring Data with Natural Language
  Let users shape Interactive Reports with plain-language requests that apply filters, sorts, highlights, charts, breaks, and other report features.
- Defining an Agent to Achieve a Goal
  Create an AI Agent with a clear goal and tools to retrieve app data and run business logic.
- Creating an Agent-Driven Chatbot
  Create a chatbot using an agent to answer questions and fulfill user requests.
- Generating Text to Save Users Time
  Generate text drafts from application data so users can review, refine, and finish faster.
- Finding by Meaning with Vector Search
  Use Oracle AI Database 26ai Vector Search with a Search Configuration to find results by meaning, even when words differ.

12.1 Configuring a Generative AI Service

Specify a Generative AI service by choosing the provider, endpoint, model, and credentials your apps use.

A generative AI service accepts a natural language prompt and returns a response based on its training. Most services offer multiple models, each of which has been fine-tuned in a particular way to prioritize certain use cases. To integrate AI into your app, start by configuring a Generative AI service definition. It specifies which service you want to work with, its REST API endpoint URL, and the model name you want to use. You can do this in the APEX Builder under *App Builder* > *Workspace Utilities* > *All Workspace Utilities* > *Generative AI*.

APEX supports working with Generative AI services from Oracle Cloud Infrastructure (OCI), Open AI, Cohere, Google Gemini, Anthropic Claude, Mistral AI, Ollama, as well as services that implement an Open AI compatible API. After selecting an *AI Provider*, give the service definition a name and a unique identifier called the *Static ID* that you'll use when working with the service programmatically with the `APEX_AI` package. In the *Settings* section, enable the *Used by App Builder* switch if you want the *APEX Assistant* in the builder to use this AI service. Leave this switch disabled if you only want your own applications to use this AI service definition.

The *(Test Connection)* button lets you verify that everything is working before clicking *(Create)*.

Copyright © 2026, Oracle and/or its affiliates.

12.2 Exploring Data with Natural Language

Let users shape Interactive Reports with plain-language requests that apply filters, sorts, highlights, charts, breaks, and other report features.

The Interactive Report region offers filtering, sorting, conditional highlighting, grouping, charting, pivoting, computed fields and aggregates, and more. Your end users can gradually learn its features as they need them, but there's a simpler way.

By enabling Natural Language support on any Interactive Report in your app, users can explain what data they want to see in their own words and carry on a conversation with the report. APEX uses your app's configured AI Service to understand the request and apply Interactive Report features that produce the result. The features applied are always visible, inspectable chips identical to the functionality an Interactive Report power user could have done manually.

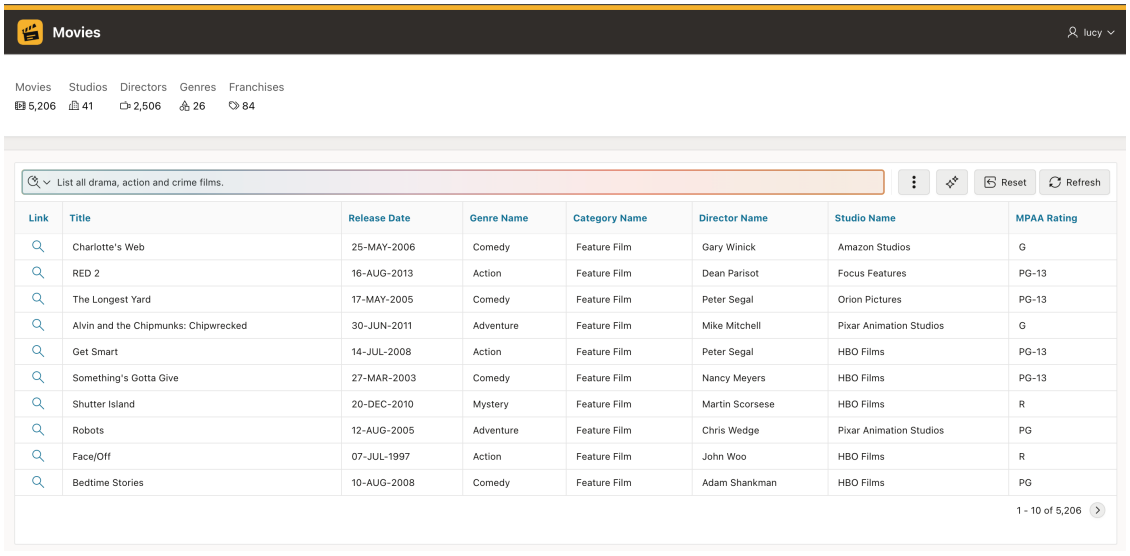
Asking Questions of Your Data and Seeing Results Immediately

The figure below shows an Interactive Report in a *Movies* app with **Natural Language Support** enabled. The developer configured **Search with AI** as the default search experience in the search field. The usual Row Search is also available from the dropdown at the start of the search field, but the Search with AI is smart enough to handle both row search and natural language prompts.

With this Interactive Report feature enabled, a user Lucy can directly type into the search field:

List all drama, action, and crime films. [Lucy]

Figure 12-1 Typing a Request in Natural Language into the Interactive Report Search Field



The report updates to show films in those requested genres, with an appropriate report filter applied. As shown below, you can see the filter "chip" for the **Genre Name** field above the results area.

This visibility is fundamental to confidence in the results. The Assistant does not return a standalone answer with no evidence of how it was produced. It maps Lucy's request to

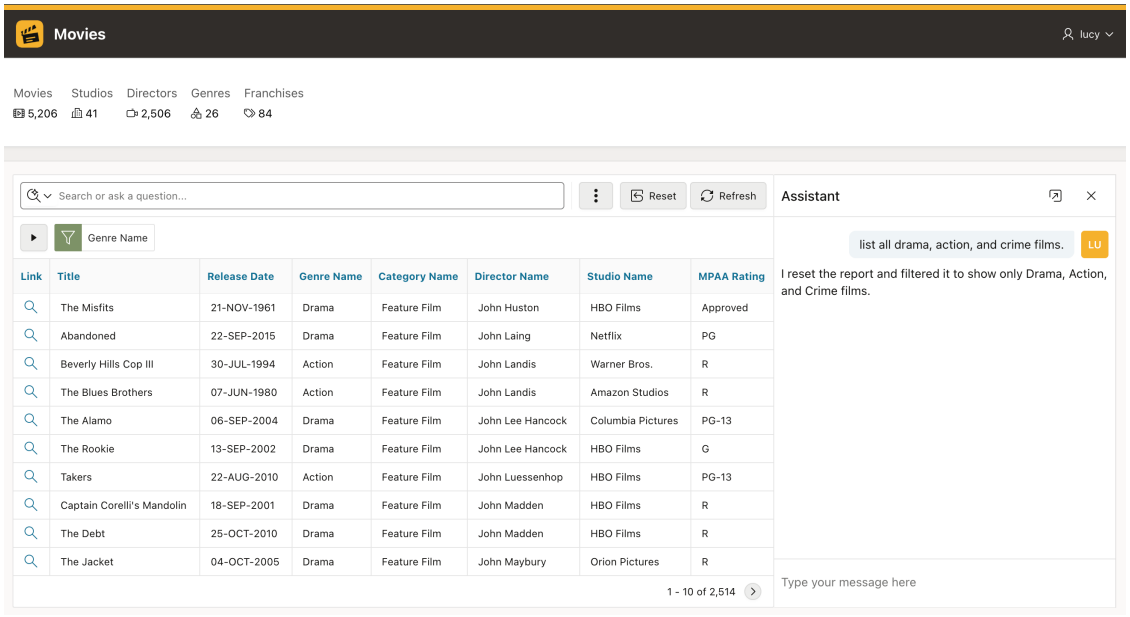
Copyright © 2026, Oracle and/or its affiliates.

ordinary Interactive Report features she can see, inspect, change, or remove. The chips above the report show the working: which filters, sorts, highlights, charts, breaks, and other settings produced the current result.

Clicking on the **Assistant** button in the toolbar, a chat window opens at the end of the report. It shows Lucy's first request, and the Assistant's reply:

> I reset the report and filtered it to show only Drama, Action, and Crime films. [Assistant]

Figure 12-2 Applied Report Features Appear as Chips, and Assistant Button Opens Chat Area



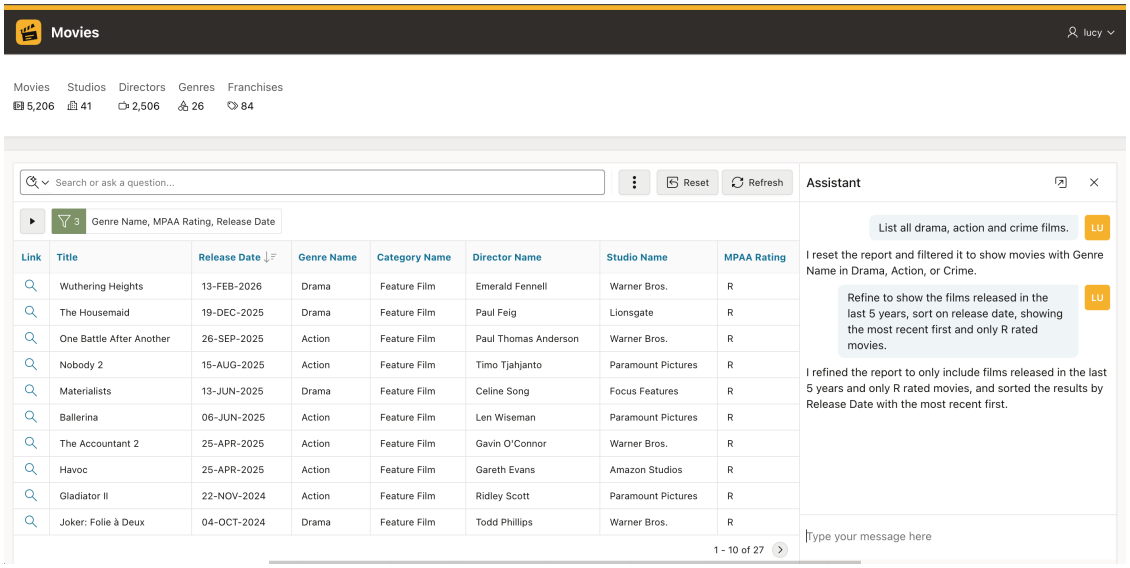
In the **Assistant** chat area, Lucy continues the conversation to refine her search results:

> *Refine to show the films released in the last 5 years, sort on release date, showing the most recent first and only R rated movies.* [Lucy]

The Assistant updates the report automatically to apply additional filters with the requested sort applied.

Copyright © 2026, Oracle and/or its affiliates.

Figure 12-3 User Can Refine Results By Continuing the Conversation in the Chat Area

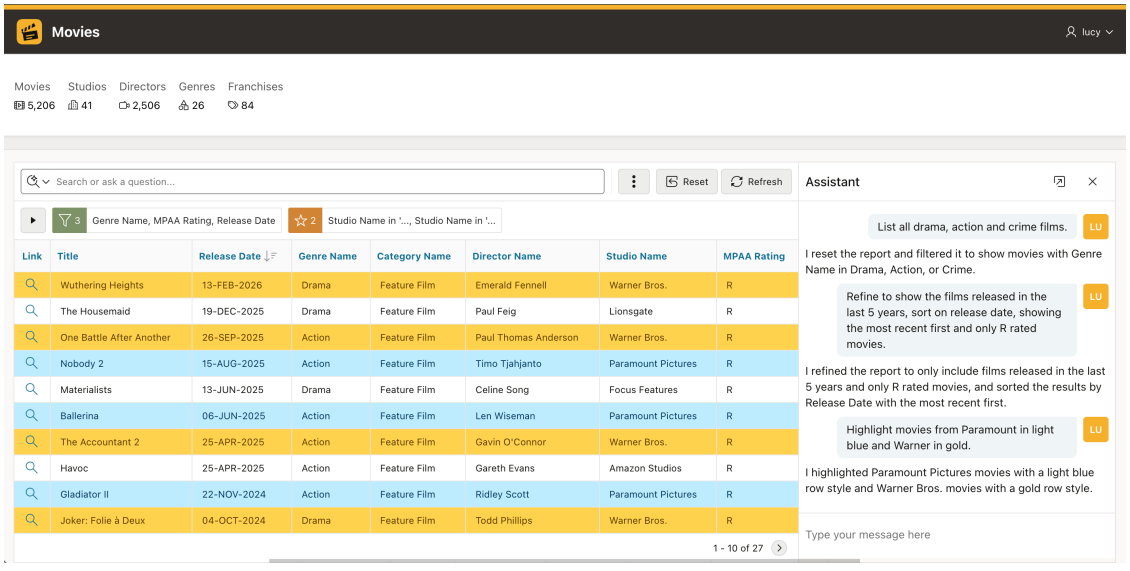


Next Lucy asks:

> *Highlight movies from Paramount in light blue and Warner in gold.* [Lucy]

The Assistant adds multiple conditional highlight features to the report and the results appear.

Figure 12-4 Interactive Report Features Like Highlighting Get Configured Using Natural Language



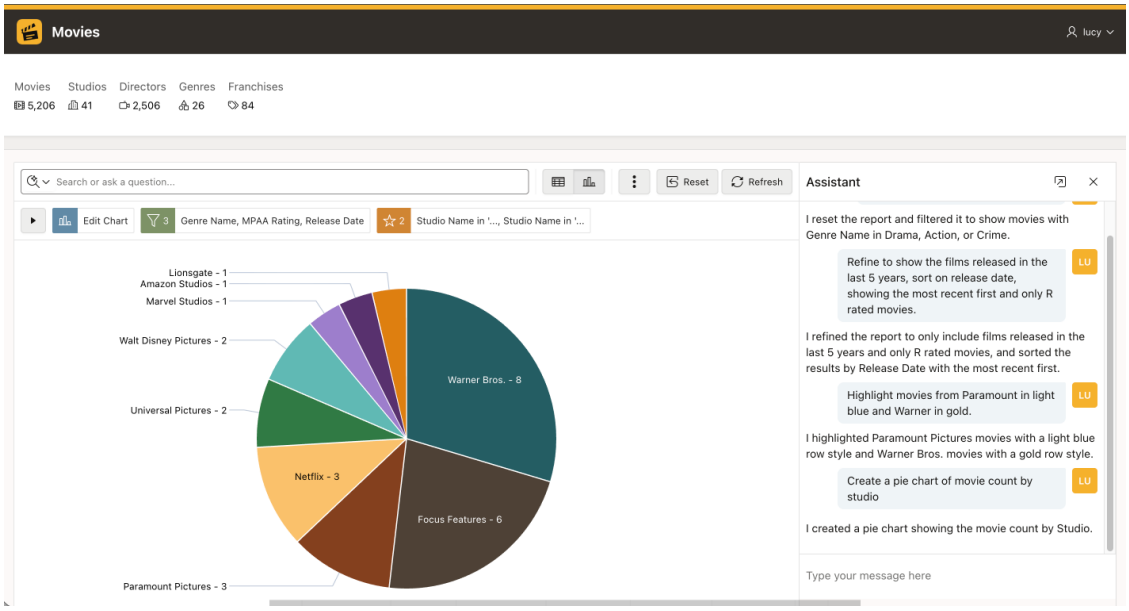
To better understand the filtered results, Lucy asks to:

> *Create a pie chart of movie count by studio* [Lucy]

Copyright © 2026, Oracle and/or its affiliates.

A chart view appears, and the usual table and chart toolbar buttons appear to let Lucy toggle between the two views of her data.

Figure 12-5 Charting and Pivoting Data Are Also Available Just by Asking



Since this is an Interactive Report, Lucy could save the current report with a meaningful name and then revisit this view of the data at any time in the future by selecting it from her saved reports list.

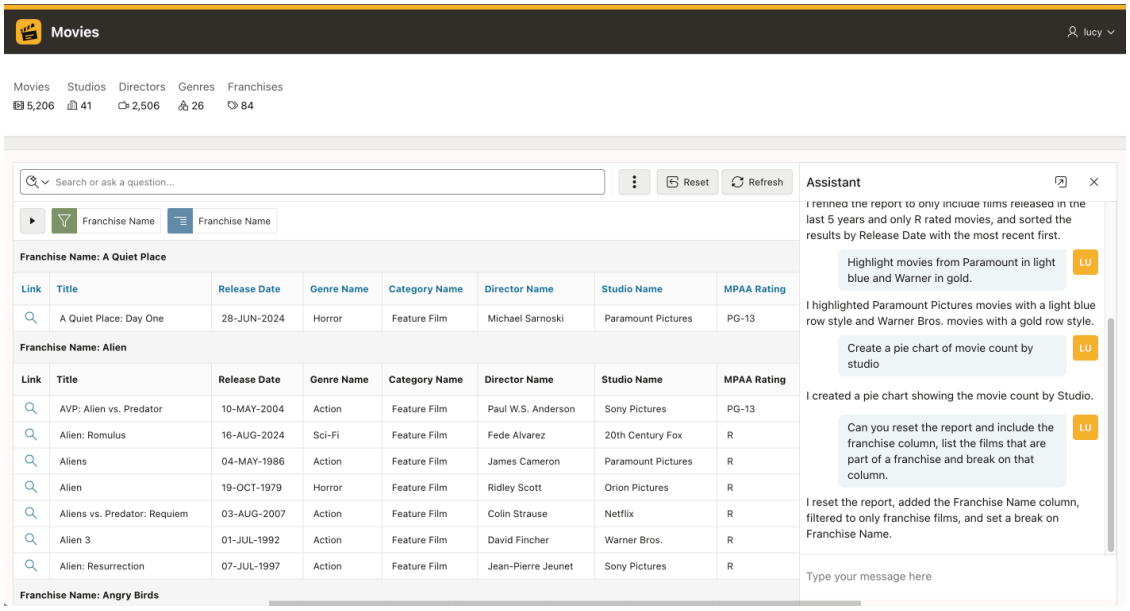
Lucy changes her mind, and now asks:

> *Can you reset the report and include the franchise column, list the films that are part of a franchise and break on that column.* [Lucy]

The Assistant removes the existing report features, shows the requested column, uses it to separate films that share the same franchise into groups.

Copyright © 2026, Oracle and/or its affiliates.

Figure 12-6 Hide or Show Columns and Add Break Groups with a Prompt

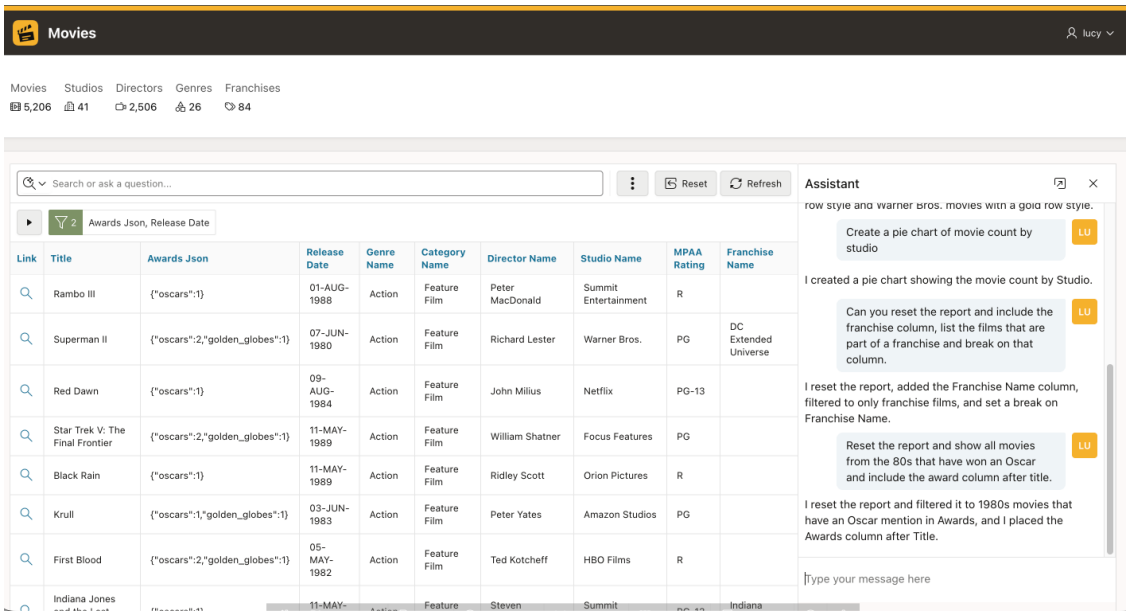


Lucy proceeds:

> *Reset the report and show all movies from the 80s that have won an Oscar and include the award column after title.* [Lucy]

The Assistant replaces previous report features with appropriate new filters and the requested results appear.

Figure 12-7 Context You Add Helps Assistant Turn "the 80s" and "won an Oscar" into Filters



For a final refinement, Lucy asks:

Copyright © 2026, Oracle and/or its affiliates.

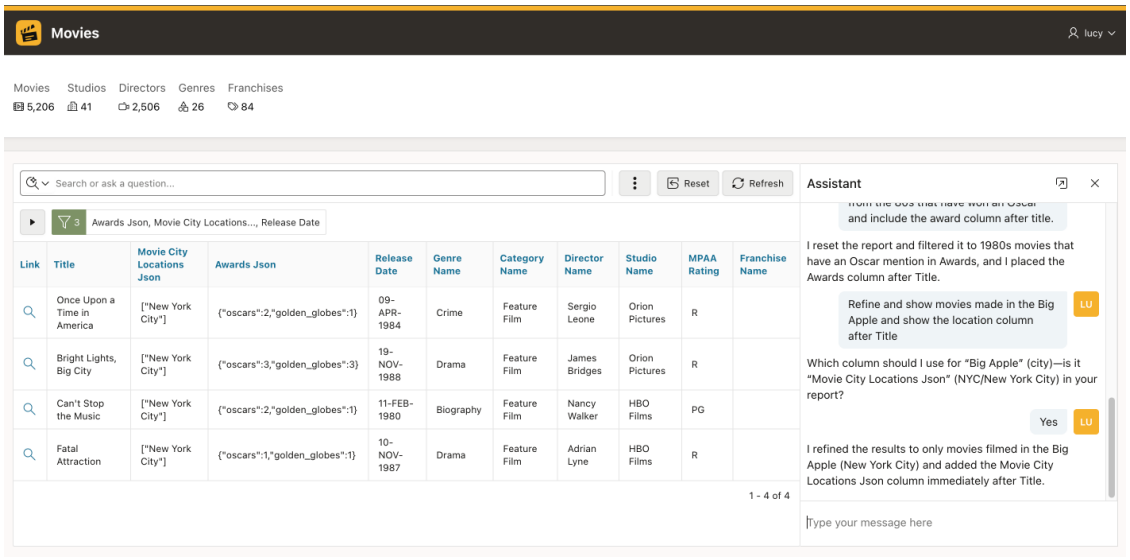
Refine and show movies made in the Big Apple and show the location column after Title [Lucy]

If *Assistant* needs Lucy to clarify something, it prompts her for a clarification:

Which column should I use for "Big Apple" (city) – is it "Movie City Locations Json" (NYC/New York City) in your report? [Assistant]

Lucy responds "Yes" and the report updates to show the subset of existing films that were filmed in New York City.

Figure 12-8 The Assistant Prompts the User for Clarification When Necessary



At any time while chatting with the Assistant, Lucy can expand or collapse the feature "chips" area of the report to inspect what Interactive Report features her prompts have produced. This makes the result transparent rather than opaque. She can continue to use the report in the normal way, adding or removing any features with the Actions menu. The Assistant is always aware of the report's current settings. If the user adds, removes, or changes features, the Assistant stays in sync when interpreting the next request.

Understanding What Is Sent to the AI Service

When the Assistant uses the AI service to understand the user's prompt, it also sends the current report settings, column names and labels, and additional context you can add in Page Designer. The extra report and column context descriptions can help the AI Service better understand the kinds of questions users may ask about the report.

The Assistant ***never*** sends application data to the AI Service. Its only job is to understand the prompt and available columns names, and additional context, then translate the request into one or more existing Interactive Report features to apply. It produces report settings, not untraceable answers.

As a result, users can only see data they are already allowed to see, using features they could have applied manually. Natural language just gives them an easier way to do it.

Copyright © 2026, Oracle and/or its affiliates.

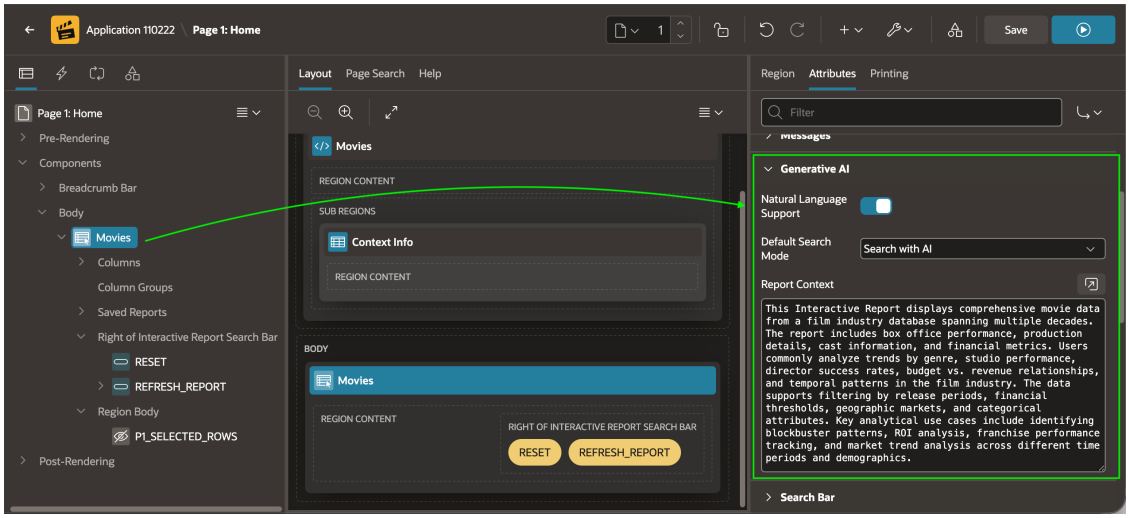
Enabling Natural Language Support and Configuring Additional Context

To enable *Search with AI* functionality on an Interactive Report, as shown below, set its **Natural Language Support** switch on. The **Default Search Mode** property controls the report search field's initial behavior. If set to *Row Search*, then users access *Search with AI* in the search field dropdown. Conversely, when *Search with AI* is the default users see AI-powered search by default. Users can always toggle to the other mode using the dropdown, as well as search on a specific field from there, too.

Add **Report Context** information to help the Search with AI Assistant handle the user's prompt. As shown below, the *Movies* region configures the following report context description:

> This Interactive Report displays comprehensive movie data from a film industry database spanning multiple decades. The report includes box office performance, production details, cast information, and financial metrics. Users commonly analyze trends by genre, studio performance, director success rates, budget vs. revenue relationships, and temporal patterns in the film industry. The data supports filtering by release periods, financial thresholds, geographic markets, and categorical attributes. Key analytical use cases include identifying blockbuster patterns, ROI analysis, franchise performance tracking, and market trend analysis across different time periods and demographics.

Figure 12-9 Report Context Helps Search with AI Assistant Better Understand User Requests



Sometimes a column's name and label suffices for the AI service to understand a column's purpose. If your testing indicates additional context is needed on a particular column, as shown below, you can add that **Column Context** as needed. For example, the `DIRECTOR_NAME` column has the following information added:

> Context: Primary director of the film. Common Use: Filmmaker analysis, director performance tracking, auteur studios. AI Hints: Users may refer to famous directors by last name only ("Spielberg", "Scorsese"). Support partial name matching, Handle "directed by" phrases naturally.

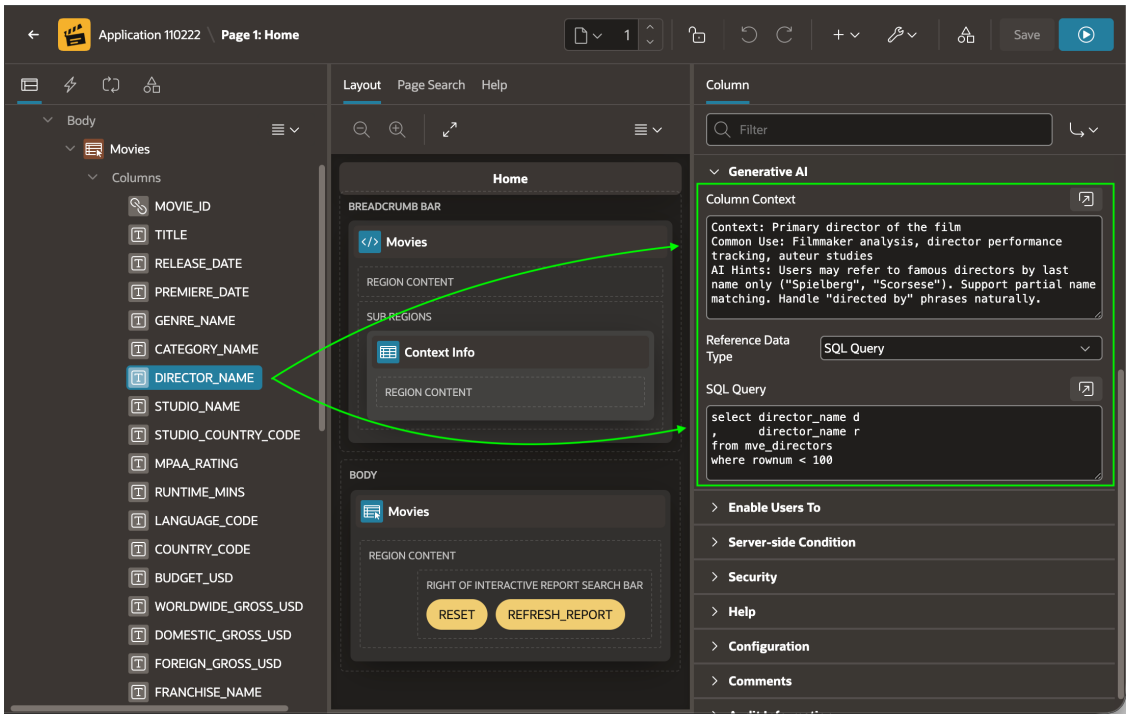
Copyright © 2026, Oracle and/or its affiliates.

You can also optionally define additional **Reference Data** to send to the AI service. For example, the *Movies* Interactive Report below adds a SQL Query to identify up to 100 director names.

```
select director_name d, director_name r
  from mve_directors
 where rownum < 100
```

> **⚠ Caution**
>
> Consider carefully which reference data you send to the AI service. Like the report and context information, reference data uses extra tokens. Evaluate whether better prompt handling is worth the added token cost.

Figure 12-10 Column Context



12.3 Defining an Agent to Achieve a Goal

Create an AI Agent with a clear goal and tools to retrieve app data and run business logic.

Defining an Agent's Goal

After defining an *AI Service*, create an *AI Agent* to handle an app-specific goal using artificial intelligence. It collaborates automatically with the AI service you configure to process a request and produce a response. An end user or your business logic supplies the input.

Copyright © 2026, Oracle and/or its affiliates.

You define your agent's mission with a system prompt in natural language. For example, if the goal is providing an *HR Assistant* that can answer questions for HR representatives, a simple prompt might be:

```
You are a friendly Woods HR assistant.
You only answer questions about Woods HR employees and no others.
```

You configure the unique aspects of meeting the goal, and the APEX engine handles interacting with the AI Service to answer requests.

Adding Tools for App Data and Allowed Actions

You can give your agent *AI Tools* to make it even smarter. They either augment the initial system prompt with dynamic data, or function as callbacks the AI Service invokes on demand to produce a final response.
AI Tools let your AI Agent expose additional capabilities that help the AI Service to achieve a goal. When responding to a user prompt, the AI Service can call your agent's tools to retrieve additional data or perform tasks.

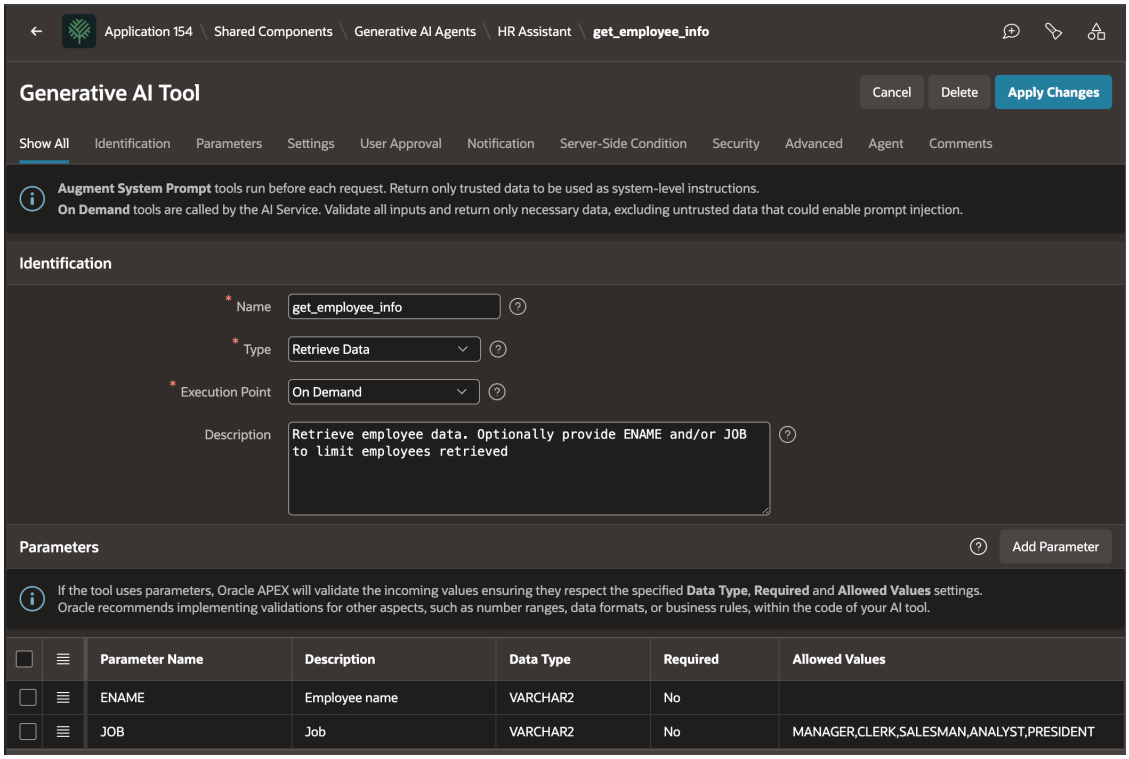
Each tool has a name, and may have a description and parameters. Your agent runs any *Augment System Prompt* tools on the initial exchange with the AI Service, and gives the service a list of *On Demand* tool descriptions. You implement tools using SQL, PL/SQL, or JavaScript. Any tool you configure can have a Server-side condition to be included only when relevant.

Exploring the HR Assistant Agent's Two AI Tools

For example, consider the on-demand tool `get_employee_info` shown below. It lets the AI service retrieve employee information. As shown below, you include a description of the tool in natural language and define the parameters the AI Service can supply. Notice this tool includes two *optional* parameters `ENAME` and `JOB`, both of type `VARCHAR2`. The tool **Type** is set to **Retrieve Data**, so you can implement the tool using a SQL query.

Copyright © 2026, Oracle and/or its affiliates.

Figure 12-11 Defining a Retrieve Data On-Demand Tool with Optional Parameters



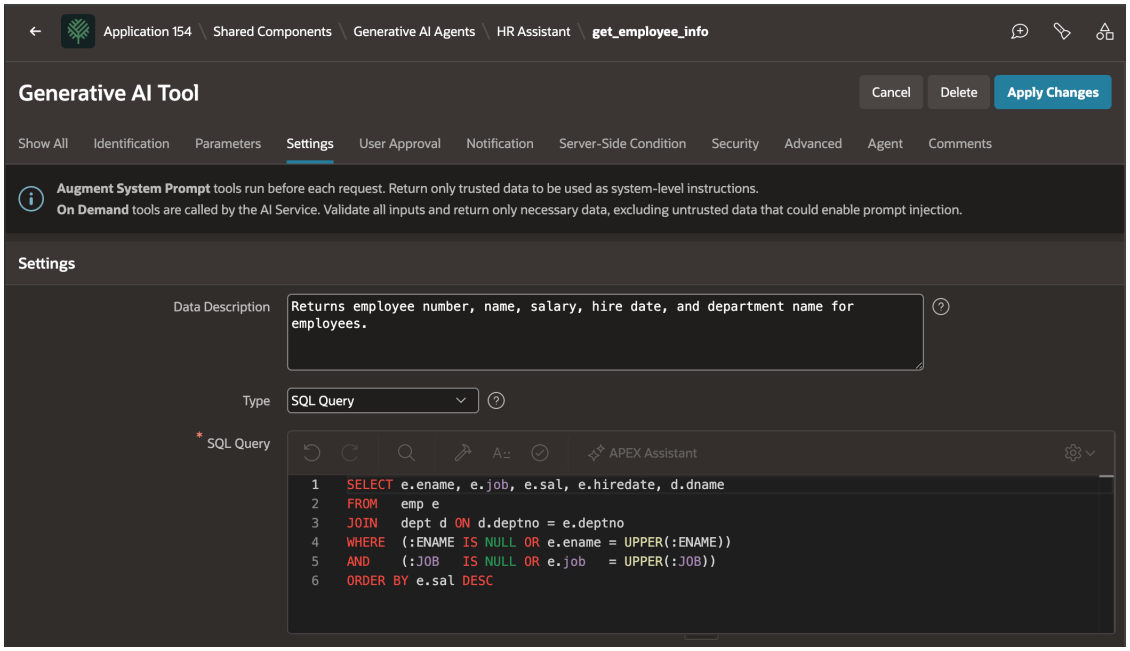
On the **Settings** tab, as shown below, you describe the data the tool returns in natural language and provide the SQL query that can reference the parameters as bind variables. Notice that since the parameters are optional, the query uses them if they have a value to filter on employee name and job. The query statement is:

```
SELECT e.ename, e.job, e.sal, e.hiredate, d.dname
FROM   emp e
JOIN   dept d ON d.deptno = e.deptno
WHERE  (:ENAME IS NULL OR e.ename = UPPER(:ENAME))
AND    (:JOB   IS NULL OR e.job   = UPPER(:JOB))
ORDER BY e.sal DESC
```

> **Tip**
>
> The query can also reference APEX session state like application items as bind variables.

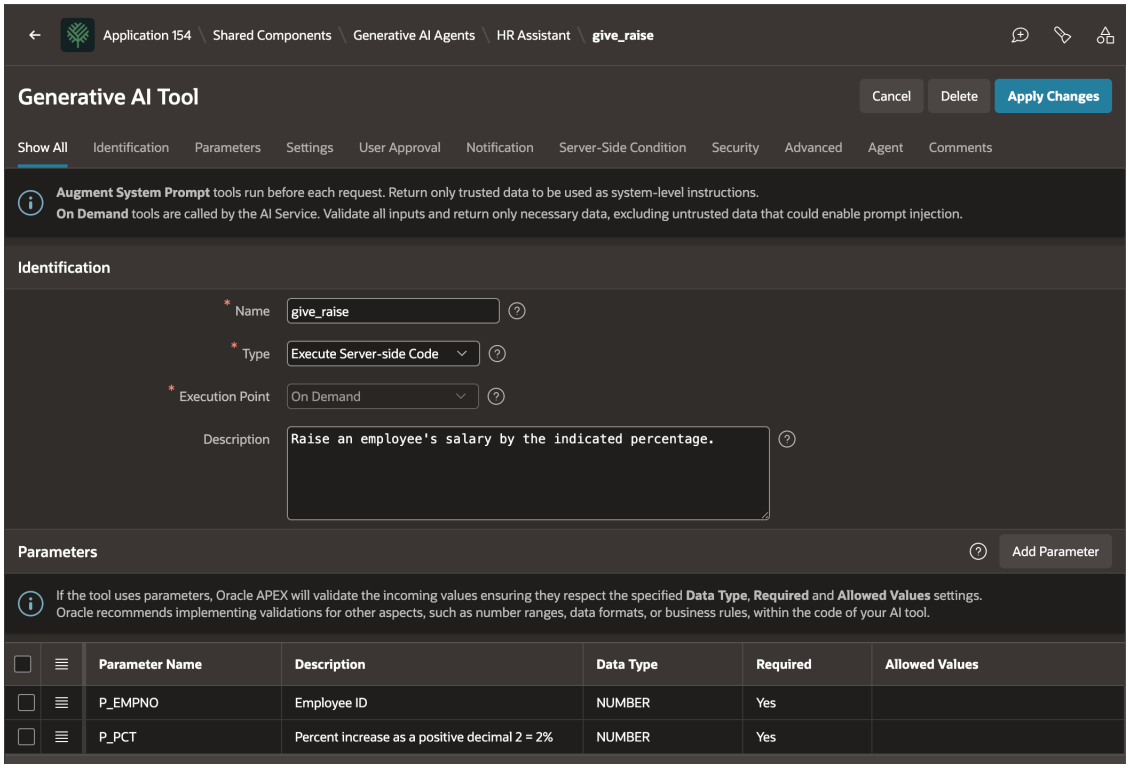
Copyright © 2026, Oracle and/or its affiliates.

Figure 12-12 Writing the SQL Query to Implement a Retrieve Data Tool



A second `give_raise` tool lets the AI service increase an employee's salary by a certain percent. This one sets **Type** to **Execute Server-side Code**, specifies a natural language description, and defines `P_EMPNO` and `P_PCT` *required* `NUMBER` parameters.

Figure 12-13 Defining an Execute Server-side Code On-Demand Tool with Required Parameters



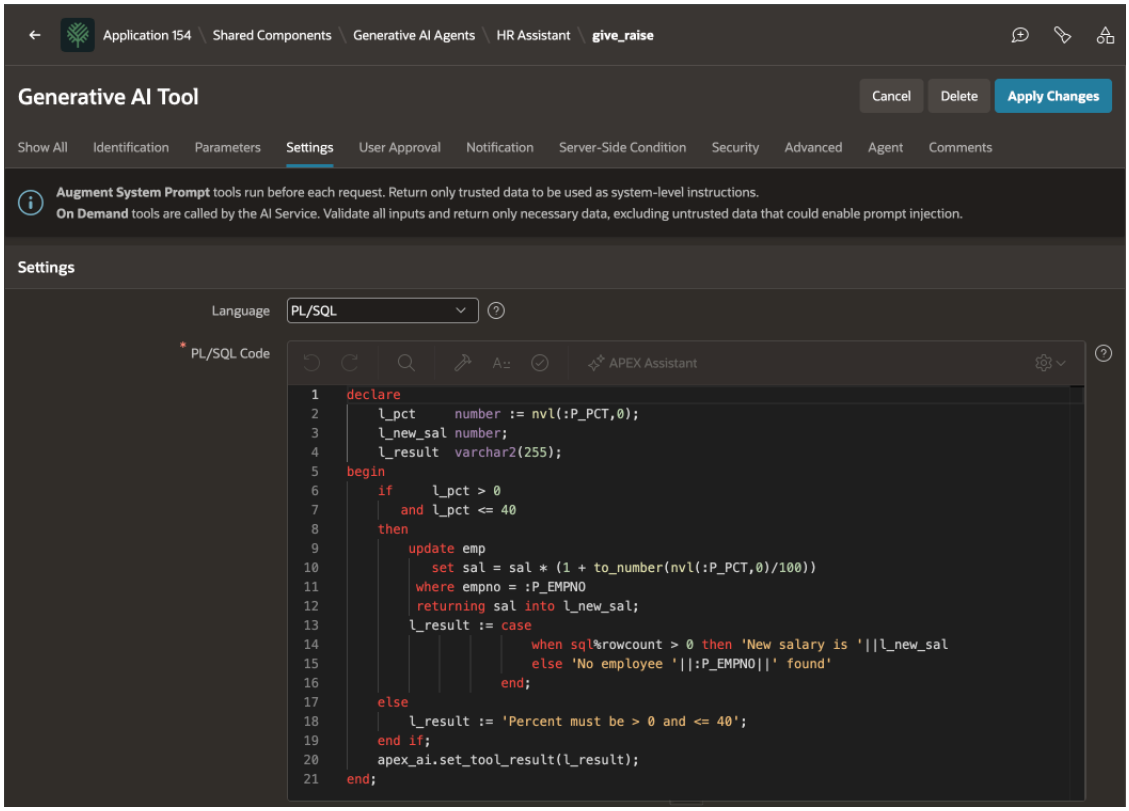
Copyright © 2026, Oracle and/or its affiliates.

You implement an *Execute Server-side Code* tool using PL/SQL or server-side JavaScript. As shown below, the code appears on the **Settings** tab. It ensures the percent parameter supplied is within the range of 0 to 40, and if so attempts to update the salary of the employee identified by the `P_EMPNO` parameter value. If the update is successful, it sets the result message to inform the AI service of the new salary. If the percent value is out of range, or the employee is not found, it sets a helpful error message instead. It calls `set_tool_result` in the `APEX_AI` package to provide the message APEX returns to the AI service as the tool invocation result.

```
declare
    l_pct     number := nvl(:P_PCT,0);
    l_new_sal number;
    l_result  varchar2(255);
begin
    if     l_pct > 0
       and l_pct <= 40
    then
        update emp
           set sal = sal * (1 + to_number(nvl(:P_PCT,0)/100))
         where empno = :P_EMPNO
         returning sal into l_new_sal;
        l_result := case
                        when sql%rowcount > 0
                        then 'New salary is '||l_new_sal
                        else 'No employee '||:P_EMPNO||' found'
                    end;
    else
        l_result := 'Percent must be > 0 and <= 40';
    end if;
    apex_ai.set_tool_result(l_result);
end;
```


Copyright © 2026, Oracle and/or its affiliates.

Figure 12-14 Writing the PL/SQL Code to Implement an Execute Server-side Code Tool



Witnessing Tools in Action in an Agent-Driven Chatbot

The Woods HR application uses role-based authorization to show HR representatives like Susan an **HR Assistant** button in the Navigation Bar. Clicking it, an **Assistant** chatbot dialog opens where Susan gets help using natural language about the employees who work at Woods. As shown below, she asks:

> *Which clerk earns the most?* [Susan]

The assistant replies:

> *The highest-paid clerk at Woods HR is **MILLER**, earning **$1,300**.* [Assistant]

Susan continues:

> *Which employee earns the least?* [Susan]

The assistant says:

> *The employee who earns the least is **SMITH**, with a salary of **$800**.* [Assistant]

Susan asks the assistant to increase Smith's pay:

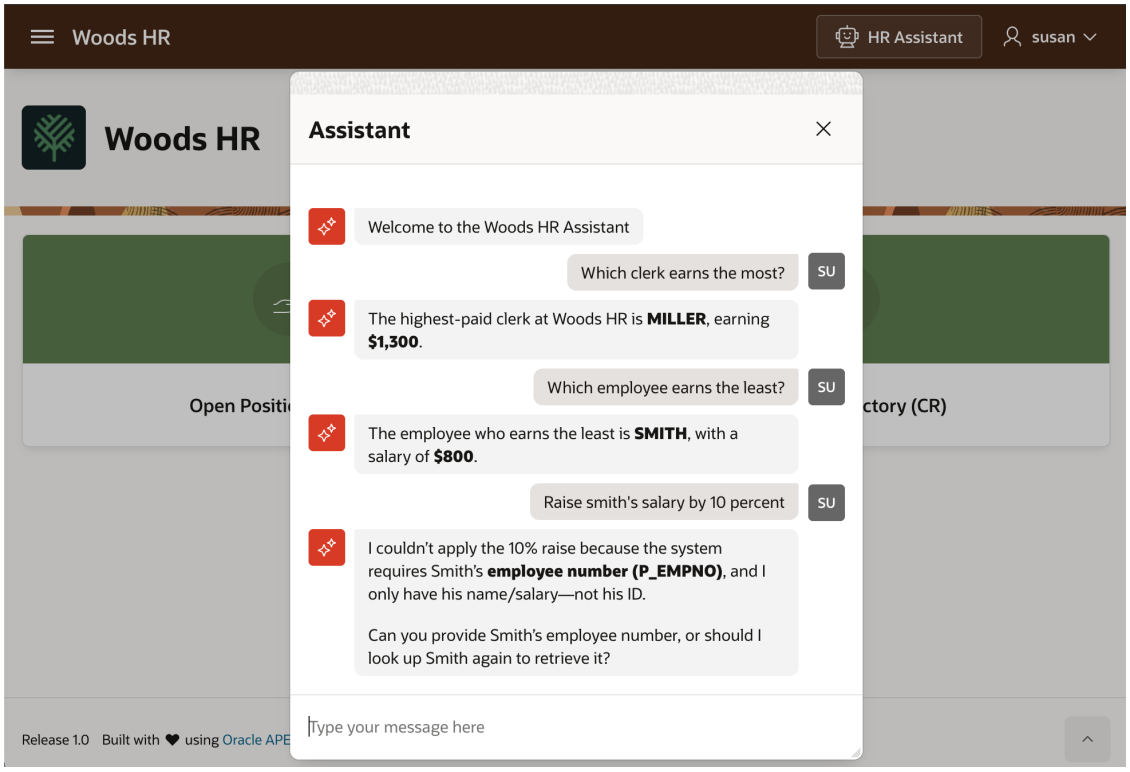
> *Raise smith's salary by 10 percent* [Susan]

The assistant answers:

Copyright © 2026, Oracle and/or its affiliates.

I couldn't apply the 10% raise because the system requires Smith's employee number, and I only have his name/salary – not his ID. Can you provide Smith's employee number, or should I look up Smith again to retrieve it? [Assistant]

Figure 12-15 Using the HR Assistant AI Agent in an AI Assistant Chatbot



The conversation illustrates the chatbot is using the *HR Assistant* AI Agent and calling the `get_employee_info` and *considering* calling the `give_raise` tools as needed. When HR reps complain about manually typing in the employee number to perform salary changes, you identify the problem.

The `get_employee_info` tool's SQL query does not include the `EMPNO` column in the returned data. You change the tool's query to select `EMPNO` as well, and update its **Data Description** to mention employee id. Then, the chatbot can complete the task with no further input. In the example conversation below, if Susan asks the HR assistant to try again, this time the chatbot replies:

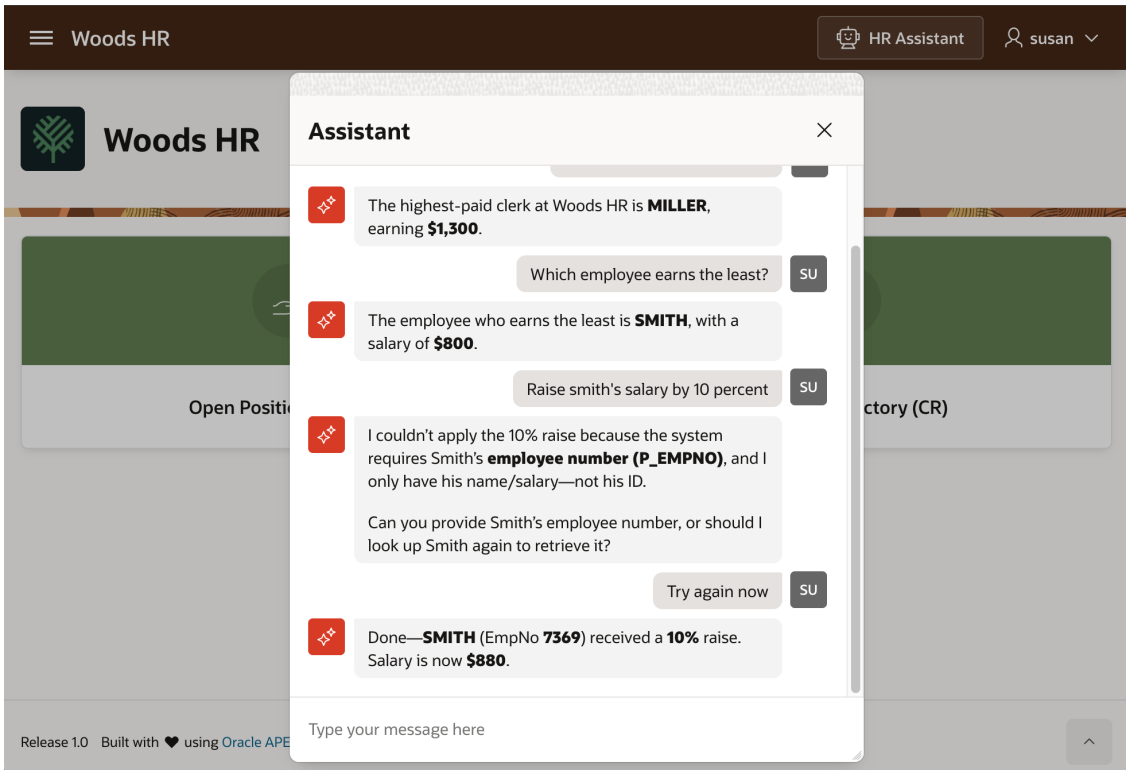
*Done – **SMITH** (EmpNo **7369**) received a **10%** raise. Salary is now **$880**.* [Assistant]

Tip

See Creating an Agent-Driven Chatbot for more info.

Copyright © 2026, Oracle and/or its affiliates.

Figure 12-16 Once `get_employee_info` Returns `EMPNO`, Assistant Completes Task Directly



Understanding the Lifecycle of an Agent-Driven Chatbot Request

The diagram below illustrates the flow of control from the time an end user asks a question in the browser like "*Which clerk earns the most?*" to when they see the answer. The APEX engine runs any *Augment System Prompt* tools before it sends the request to the AI Service. Their results accompany the AI Agent's system prompt in the first message sent to the service's Large Language Model (LLM).

If the AI Service has everything it needs, it may respond directly. If not, it may call one or more of your agent's *On Demand* tools. For example, the LLM may request to invoke the `get_employee_info` tool, passing in `"clerk"` for the value of the tool's `JOB` parameter.

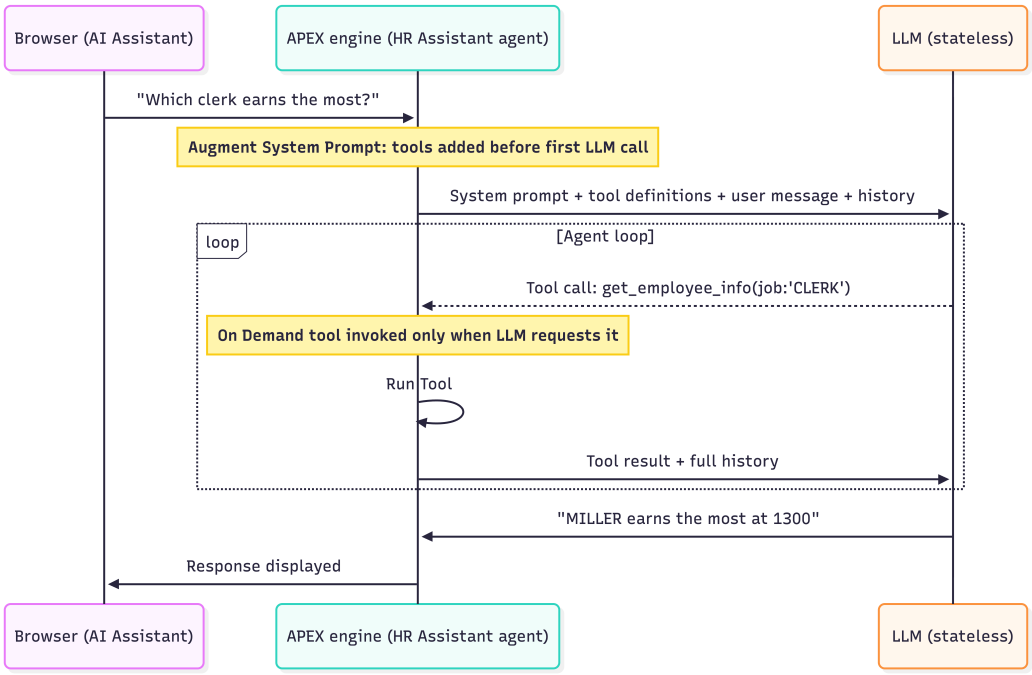
> **Tip**
>
> The AI service ***never*** sees how your tools are implemented and only sees the data your tools return in their response. APEX sends your tool name, description, and parameter definitions only. These establish the "contract" of available "functions" the LLM can invoke, and each is in full control of what it returns in response.

The APEX engine runs any tools the LLM requests to invoke, passing in parameters the service provides. APEX responds with your tools' results. This "agent loop" conversation continues until the AI Service decides to respond with the final answer. In this example, the service returns the response "*MILLER earns the most at 1300*", and APEX returns it to the browser where the user sees it appear as the answer to their question.

Copyright © 2026, Oracle and/or its affiliates.

As Susan carries on a conversation of multiple messages and responses, APEX accumulates the conversation history. Since the LLM is stateless, APEX sends the full history with every request to the LLM so it has the full context to reason over. If later in the conversation Susan asks to raise Smith's salary by 10%, the same request lifecycle occurs. The only difference in that case is that the LLM requests to invoke the `give_raise` tool instead.

Figure 12-17 Lifecycle of a Request Involving an AI Agent



Deciding Between Execution Points

Your AI Agent tool's *Execution Point* determines when it runs: *On Demand* or *Augment System Prompt*.

The APEX engine runs your agent's *On Demand* tools only when the AI Service requests them. In contrast, *Augment System Prompt* tools run before your agent sends the initial request to the AI Service. Any results become additional system prompts that precede user input.

You can configure a Server-side condition on both kinds of tools. When an On-Demand tool's condition is `false`, it is omitted from the list of tool descriptions sent to the AI service. When an Augment System Prompt tool's condition is `false`, that tool is skipped and adds no data to the initial request.

Any tool can perform Retrieval Augmented Generation (RAG). If your agent's goal benefits from up front information that will *always* be useful to prepare an answer, then use an *Augment System Prompt* tool. Otherwise, create a well-described *On Demand* tool instead. Define appropriate parameters and return just the information the LLM needs, *when* it decides it is useful to have it.

Understanding Native Tool Types

APEX offers the three native tool types:

Copyright © 2026, Oracle and/or its affiliates.

- *Retrieve Data*
  - Returns data based on a SQL Query, Function Body returning a CLOB, or static text.
- *Execute Server-side Code*
  - Performs server-side logic using PL/SQL or JavaScript (MLE) code. Override the implicit "success" result with `set_tool_result` in the `APEX_AI` package.
- *Execute Client-side Code*
  - Request end user-input or call browser APIs in an async context. Override the implicit "success" return by returning another string.

You can use all tool types for an on-demand tool. To augment the system prompt, choose either *Retrieve Data* or *Execute Client-side Code*.

Specifying Tool Parameters

On demand tools can declare named parameters with optional description, and specify which if any is required. Supported data types include `VARCHAR2`, `CLOB`, `NUMBER` and `BOOLEAN`. Tools using SQL or PL/SQL reference parameter values as bind variables (`:PERSON_ID`). In JavaScript, they are available via `this.data` (`this.data.PERSON_ID`).

Ensuring a Human is "In the Loop" Before Invoking a Tool

Each AI Tool can insist that the end user is aware of and approves its invocation. If you set the **Requires Confirmation** switch on in the **User Approval** section of your tool definition, then the end user is alerted if the LLM requests the tool, and the tool is only executed if the end-user approves.

Using an Agent Declaratively and Programmatically

Your *Show AI Assistant* and *Generate Text With AI* dynamic actions can reference an AI Agent, as can your *Generate Text with AI* page process or workflow activity. For programmatic use, pass the AI Agent's static ID to `APEX_AI` package APIs. If the AI Service calls your agent's tools, APEX automatically runs them. Client-side tools are allowed when dynamic actions like *Show AI Assistant*, or *Generate Text With AI* initiate the request from the browser.

Extending Native AI Tool Types with Plug-ins

The "Generative AI Tool" plug-in type lets you create your own reusable tools to extend the built-in set.

12.4 Creating an Agent-Driven Chatbot

Create a chatbot using an agent to answer questions and fulfill user requests.

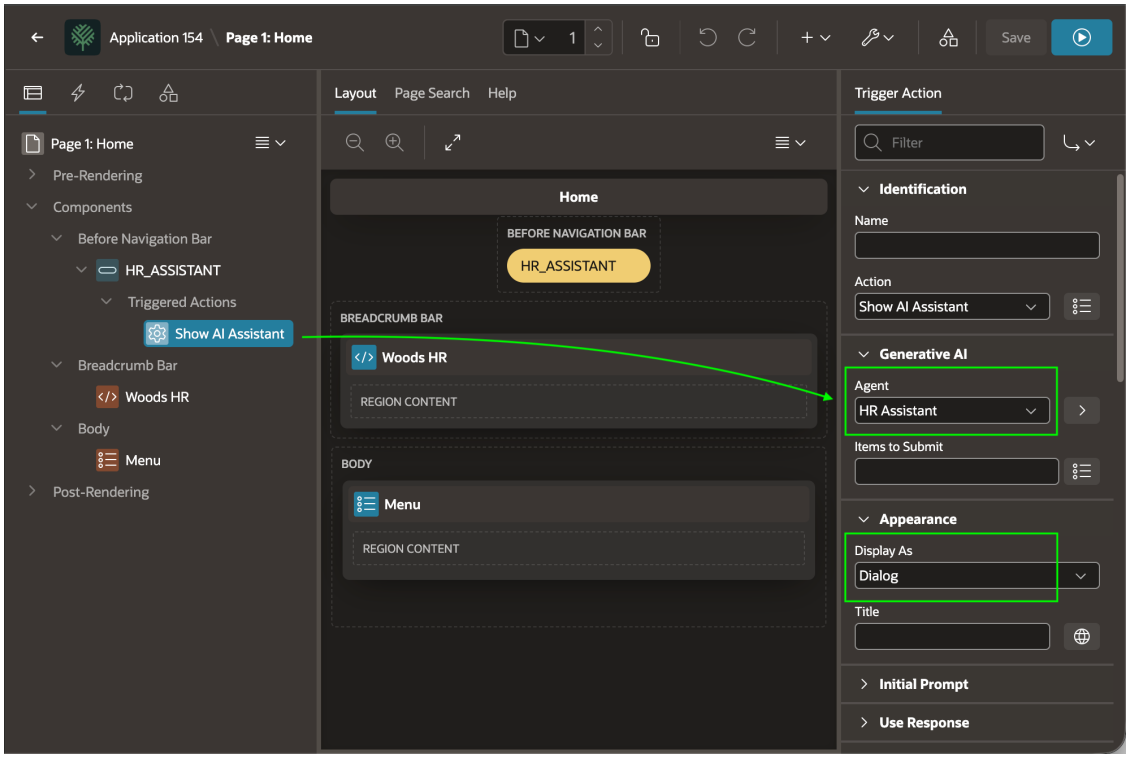
To create a chatbot, use the *Show AI Assistant* dynamic action, typically on page load or in response to a button click. It can open its own dialog window, or you can point it at a *Static Content* region in your page as an inline display area. In either case, at runtime a welcome message greets the end user and they can begin asking questions. The display is familiar to anyone who has used a messaging app. The end user and the bot chat in an alternating timeline of question and answer speech bubbles, each marked with an avatar icon.

As shown below, choose an **Agent** in the **Generative AI** section to handle the chatbot's conversation and decide whether it will **Display As** a dialog or inline. The agent's prompt and tools offer users a more complete and effective experience. You can also configure the dynamic action to use a Generative AI Service directly when the chatbot only needs a prompt.

Copyright © 2026, Oracle and/or its affiliates.

> **Tip**
>
> To configure the avatar icons or images that represent the AI service and the end user, visit the *Shared Components* > *Component Settings* > *Show AI Assistant* in your application and adjust the respective settings.

Figure 12-18 Showing an AI Assistant Dialog Using a Trigger Action on a Button



To create a *custom* chatbot experience, you can:

- engage the AI service with the `CHAT()` function in the `APEX_AI` package,
- store questions and responses in a collection using the `APEX_COLLECTION` package, and
- display the conversation by combining a *Comments* region with the `APEX_COLLECTIONS` view.

12.5 Generating Text to Save Users Time

Generate text drafts from application data so users can review, refine, and finish faster.

Use a *Generate Text with AI* dynamic action, page process, or workflow activity when your app needs to draft, rewrite, summarize, classify, or explain text for a user. For example, a clinic app might draft discharge instructions for a patient from selected recommendations, or a sales app might summarize recent customer activity before a call.

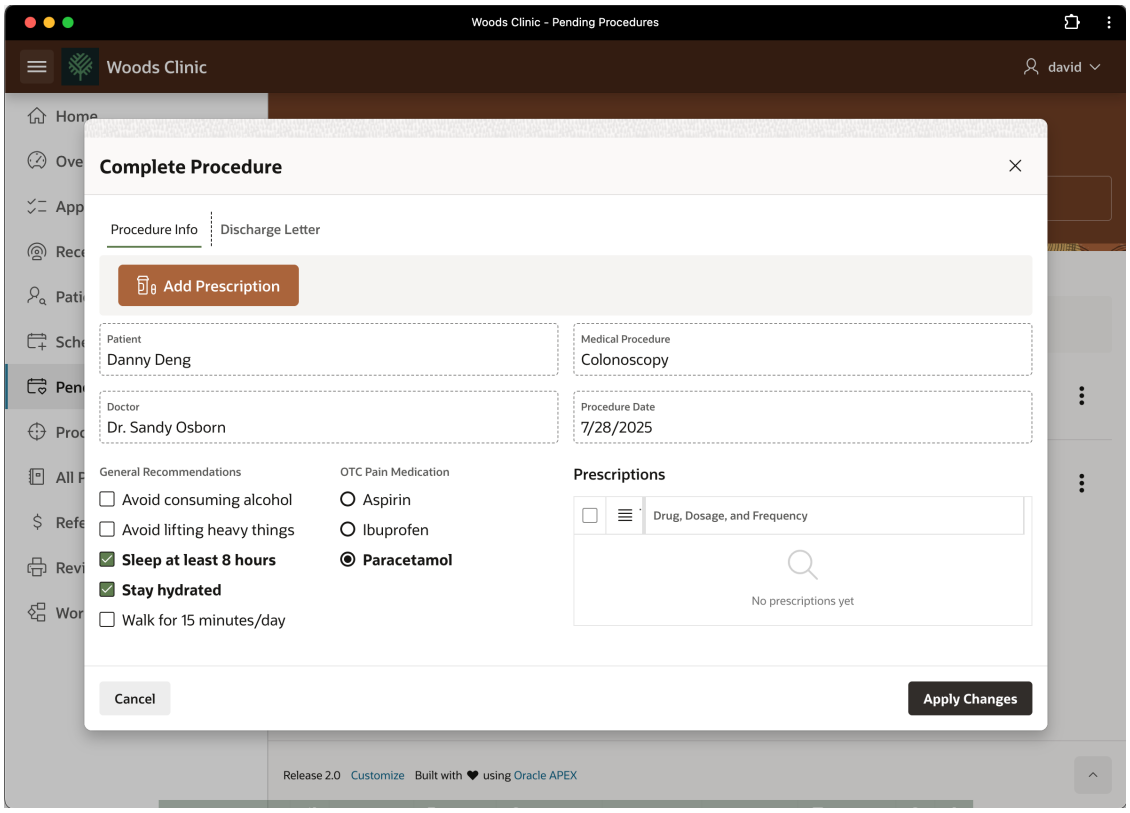
Choose an *AI Agent* when the request benefits from permitted application data or actions exposed through Tools. You can also use a Generative AI Service directly when a fixed prompt is enough.

Copyright © 2026, Oracle and/or its affiliates.

To perform AI text generation in custom code, use the `GENERATE()` function in the `APEX_AI` package.

For example, when discharging a patient, *Woods Clinic* staff members fill out the **Complete Procedure** form shown below. It lists common recommendations that doctors suggest to help patients recover more quickly after a medical procedure. It also lets staff indicate over the counter pain medication and enter any medications the physician has prescribed.

Figure 12-19 Entering Patient Discharge Instructions



On the **Discharge Letter** tab of the same dialog, as shown below, staff can click (**Generate Initial Letter**) to get help writing the discharge letter with doctor's instructions and prescriptions. They give this letter to each patient before they leave the clinic to return home.

The first trigger action on the (**Generate Initial Letter**) button is an *Execute Server-side Code* that assigns the staff member's input into a series of application items (`PATIENT`, `PROCEDURE`, `GENERAL_RECS`, `PAIN_REC`, and `PRESCRIPTIONS`). The second one is a *Generate Text with AI* associated with a *Generate Discharge Letter Draft* AI Agent. It engages the agent to populate a draft discharge letter into a Rich Text Editor page item on the page. The agent defines the following System Prompt with template directives to conditionally populate some of the application item values.

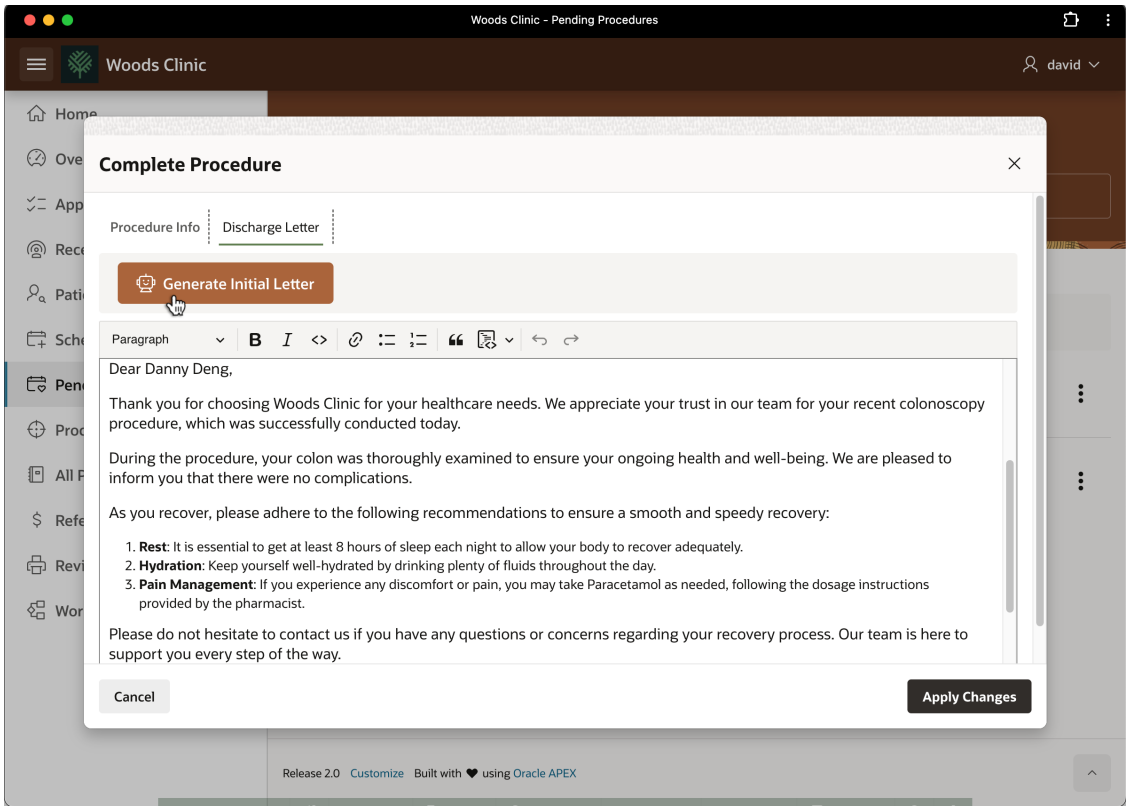
```
You are a friendly nurse at a medical clinic named "Woods Clinic".
Your task is to write a succinct letter in a professional style as
discharge instructions for a patient #PATIENT# who
underwent a #PROCEDURE# procedure at the clinic today.
Using a gender-neutral greeting, start by
thanking the patient for trusting Woods Clinic for their healthcare
needs and summarize the procedure.
```


Copyright © 2026, Oracle and/or its affiliates.

```
{if GENERAL_RECS/} Include general recommendations like: #GENERAL_RECS#.
{endif/}
{if PAIN_REC/} For any pain experienced, recommend: #PAIN_REC#.{endif/}
{if PRESCRIPTIONS/} Summarize prescribed medicines: #PRESCRIPTIONS#{endif/}
```

The result appears below. Woods Clinic staff members love the time it saves them from having to write the letter by hand. They can review the draft letter to make any changes, then print it to send the patient home with the instructions they need.

Figure 12-20 Preparing Discharge Letter Using AI Agent



12.6 Finding by Meaning with Vector Search

Use Oracle AI Database 26ai Vector Search with a Search Configuration to find results by meaning, even when words differ.

Understanding the Potential for User Frustration When Searching

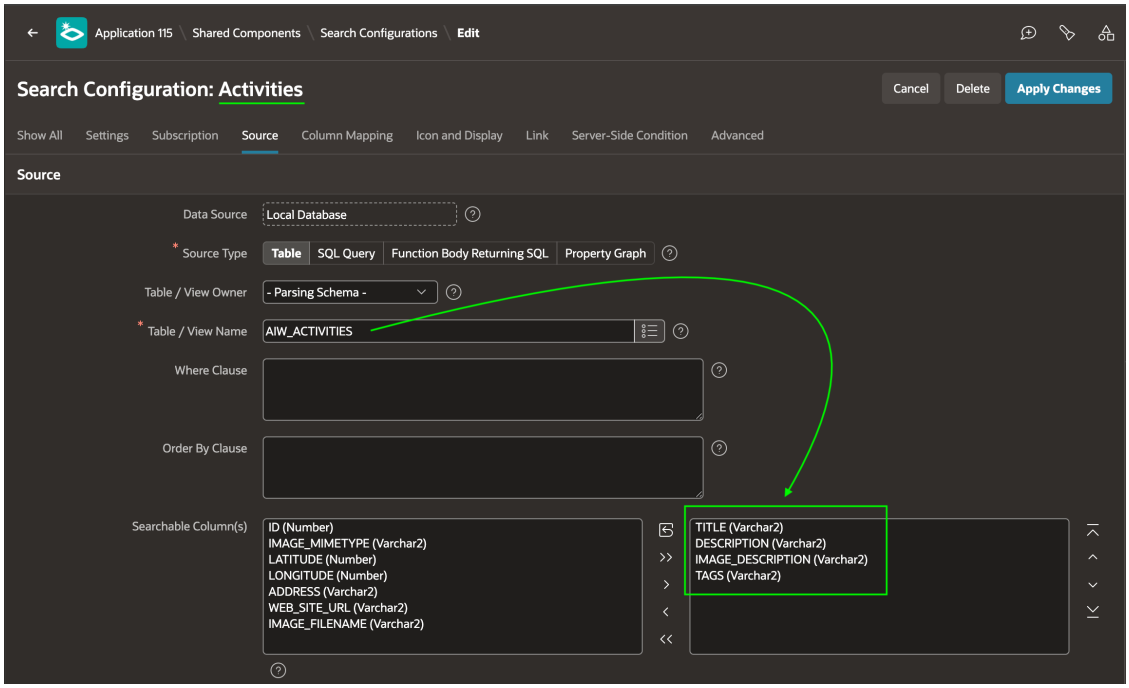
Users know what they are looking for, but might search for it using words that are different than the ones in your tables. When their search finds no results, that can be frustrating. For example, suppose Lucy brings her family to Las Vegas while she attends a conference during the day. She wants to find family-friendly activities to do with the kids in the evenings, so she uses your *Vegas Family Nights* app to search for ideas.

After reading Unified Search Results Across Sources, you created a page with a Search region to give user's like Lucy an internet-search experience to find fun things to do together. As shown below, your *Activities* Search Configuration relates to your `AIW_ACTIVITIES` table with all

Copyright © 2026, Oracle and/or its affiliates.

activities Las Vegas has to offer that are appropriate for children. You identify `TITLE`, `DESCRIPTION`, `IMAGE_DESCRIPTION`, and `TAGS` as good columns to search over.

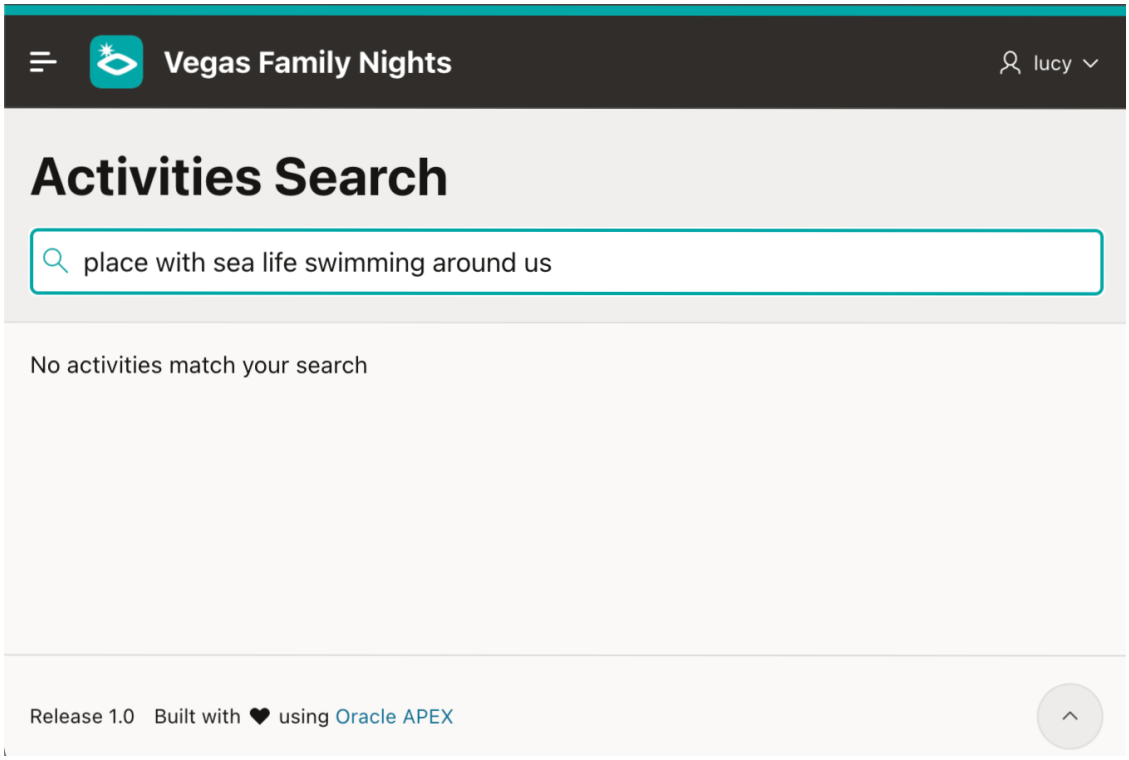
Figure 12-21 Activities Standard Search Configuration on Activities Table Columns



However, when Lucy types in her search for "*place with sea life swimming around us*", as shown below she sees that no activities match her search.

Copyright © 2026, Oracle and/or its affiliates.

Figure 12-22 Lucy's Choice of Words Produces No Matching Activities



Lucy tries another search for "*location where we can see illuminated old hotel signs*", but again her search produces no activities. She leaves your application feeling frustrated.

Capturing the Meaning of Words Using Vectors

An AI service provider trains a large language model on an enormous set of text or image inputs. Then they fine-tune the model to excel at recognizing and classifying new inputs based on its training data. For some new input, the model applies its training to "understand" it. In the process, it evaluates potentially hundreds or thousands of different traits and assigns the input a "score" for each trait it has been trained to recognize.

For example, imagine training an AI model on millions of family friendly activities from around the world. Fine-tuning can incorporate professional activity planner expertise, helping the model suggest activities from natural language phrased the way an average parent might ask. For a family activity, traits might include what the experience is like and who it suits, such as activity type, age range, indoor or outdoor setting, energy level, educational value, time commitment, cost, distance from the Strip, accessibility, noise level, food availability, physical effort required, and weather sensitivity.

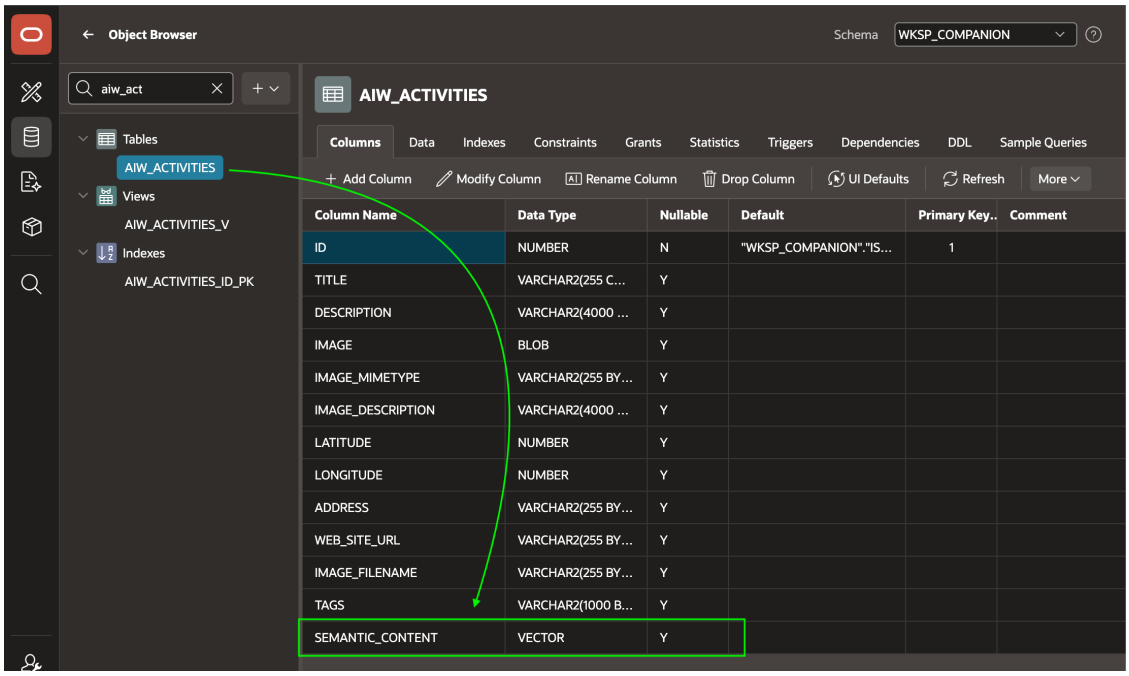
The model assigns a number between 0 and 1 for each of these traits, capturing how strongly a particular activity expresses that quality. For instance, a food availability score of 0 means no food available, while a score of 1 would indicate the activity is absolutely food-centric. You can write down the sequence of an activity's trait scores as an ordered list of numbers called a *vector*. The Oracle AI Database 26ai database supports a `VECTOR` data type to store these "scorecards" in a column. It also indexes vectors efficiently to quickly identify ones that are closest in *meaning* to a vector representing the intent of a user's search.

Copyright © 2026, Oracle and/or its affiliates.

Incorporating a Vector into Your Existing Table

Your database contains an `AIW_ACTIVITIES` table. You can alter the table to add a `SEMANTIC_CONTENT` column of type `VECTOR` to store the list of trait scores the AI model infers for each activity. That vector encodes the model's understanding of each activity's complete profile: a compact numeric summary of all the trait scores the AI recognized about it.

Figure 12-23 Adding a VECTOR Column to an Existing Table in Object Browser



Installing a Model in Your Database

The AI model you use to encode the semantics of your data can be an external AI service, an external program, or an industry-standard Open Neural Network Exchange (ONNX) model that you load directly into Oracle 26ai. Using an external AI service, you have to send your data elsewhere to encode it. With a DB-resident model, you can encode the semantics in place.This Oracle blog article features a link to download an `all-MiniLM-L12-v2` model you can try to get started. As a prerequisite to installing a model, the parsing schema of your APEX workspace requires the `CREATE MINING MODEL` privilege. Then use the following PL/SQL block to load the ONNX model after substituting *`TODO`* with the URL for the model file from the blog article:

```
begin
    dbms_vector.load_onnx_model(
        model_name => 'DOC_MODEL',
        model_data => apex_web_service.make_rest_request_b(
                          p_http_method => 'GET',
                          p_url => 'TODO')),
                                   -- ^ URL from Blog Article Here
        metadata => json(q'~
                    {
                       "function" : "embedding",
                       "embeddingOutput" : "embedding" ,
                       "input":{"input": ["DATA"]}
```

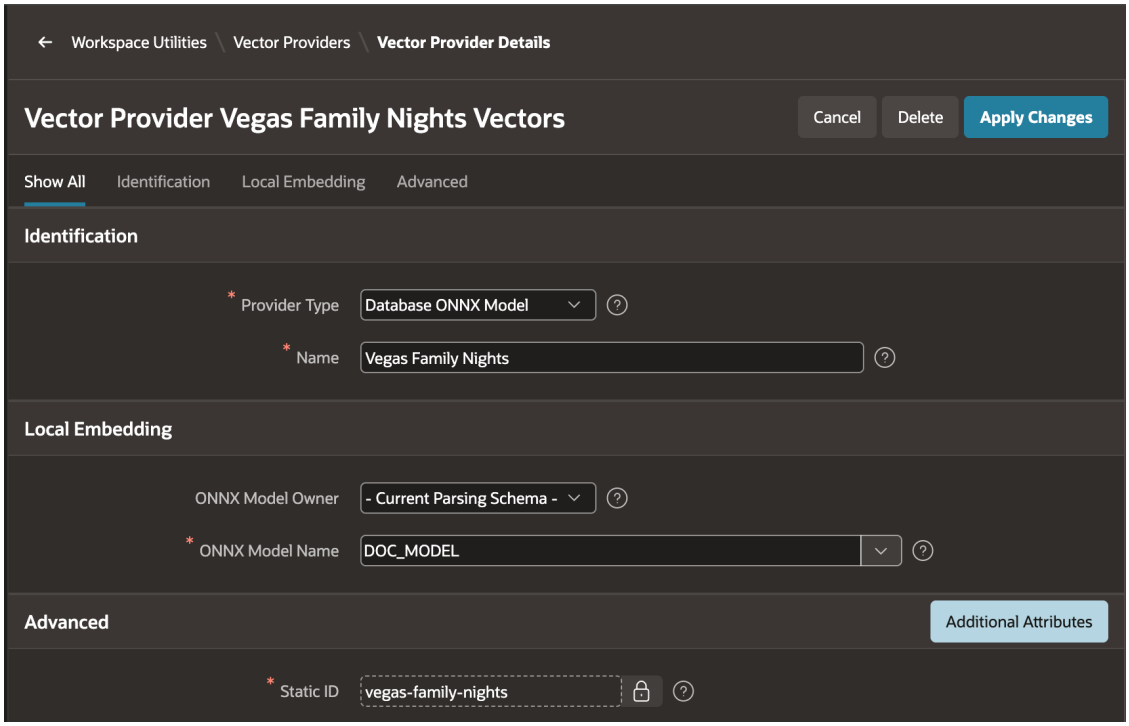

Copyright © 2026, Oracle and/or its affiliates.

```
                    }~')));
end;
```

Defining a Vector Provider

A vector embedding represents a model's understanding of the input as coordinates in a multi-dimensional space of traits. APEX simplifies working with an external or DB-resident model that provides vector embeddings to capture data semantics. You supply the info it needs to *use* the model by defining a vector provider in the APEX Builder under *Workspace* > *All Workspace Utilities* > *Vector Providers*. After defining a vector provider, you reference it in calls to the `GET_VECTOR_EMBEDDINGS` function in the `APEX_AI` package, or when you define search configurations that power a *Search* region in your page.

Figure 12-24 Creating a Vector Provider Based on a Database-Resident ONNX Model



Generating the Vector Embeddings for Your Data

To generate a vector for text input, use the `GET_VECTOR_EMBEDDINGS` function in the `APEX_AI` package. Pass it the text to encode and the static ID (e.g. `vegas-family-nights`) of the vector provider to use.

For example, to encode the meaning of the activity's title, description, image description, and tags, use an `UPDATE` statement like the one below. It uses the indicated vector provider to compute the `SEMANTIC_CONTENT` vector for each row based on the contents of each activity's `TITLE`, `DESCRIPTION`, `IMAGE_DESCRIPTION`, and `TAGS` columns.

```
begin
   apex_util.set_workspace('COMPANION');
   update aiw_activities
      set semantic_content =
            apex_ai.get_vector_embeddings(
```


Copyright © 2026, Oracle and/or its affiliates.

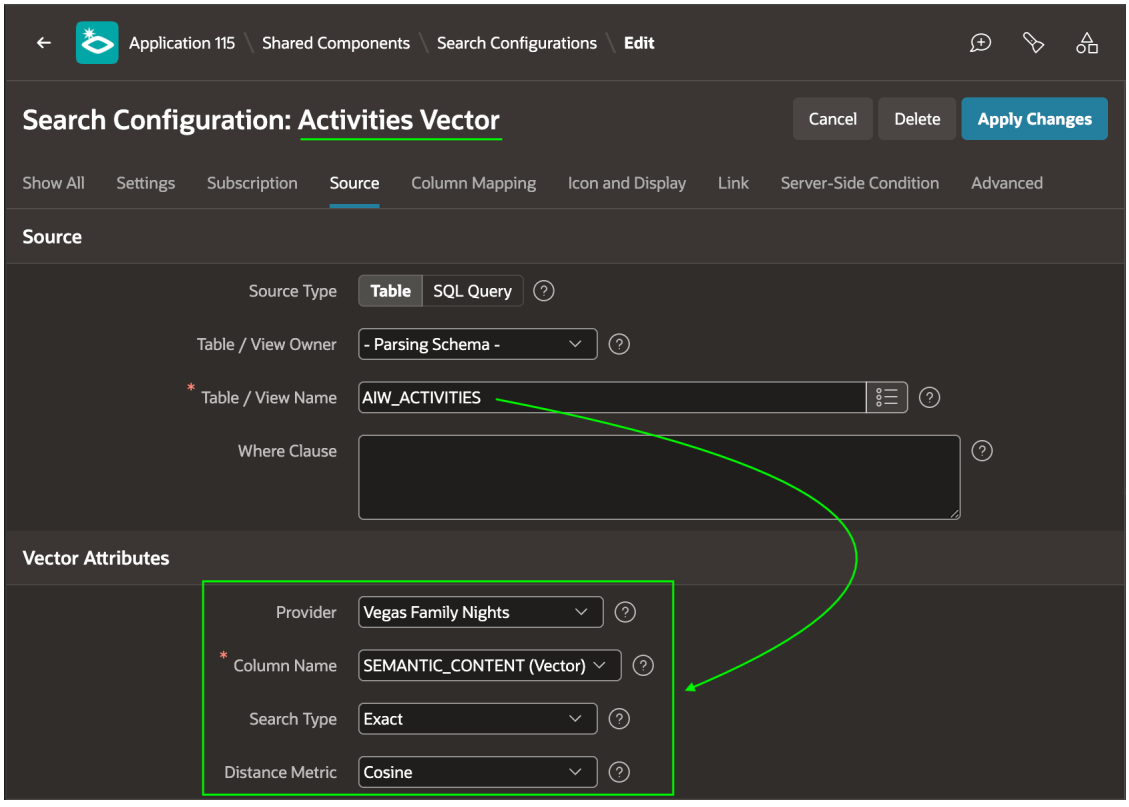
```
                p_value =>
                  'Title: ' || title || chr(13) ||
                  'Description: ' || description || chr(13) ||
                  'Feature Image Description: ' || image_description ||
                  'Tags: ' || tags,
                p_service_static_id =>'vegas-family-nights');
end;
```

Enabling Searching by Meaning in a Page

To let end users search by meaning using Vector Search, start by defining an *Activities Vector* search configuration under *Shared Components*. Choose the type: *Oracle AI Vector Search*. You configure the vector column as shown below. For `AIW_ACTIVITIES`, it's the `SEMANTIC_CONTENT` vector column you added and populated with vector embeddings for your existing activity data.

> **⚠ Caution**
>
> Choose the same *vector provider* you used to encode the semantic content in your existing data. This ensures the database can compare a user's search to your existing data's vector.

Figure 12-25 Defining a Vector Search Type Search Configuration on the Activities Table

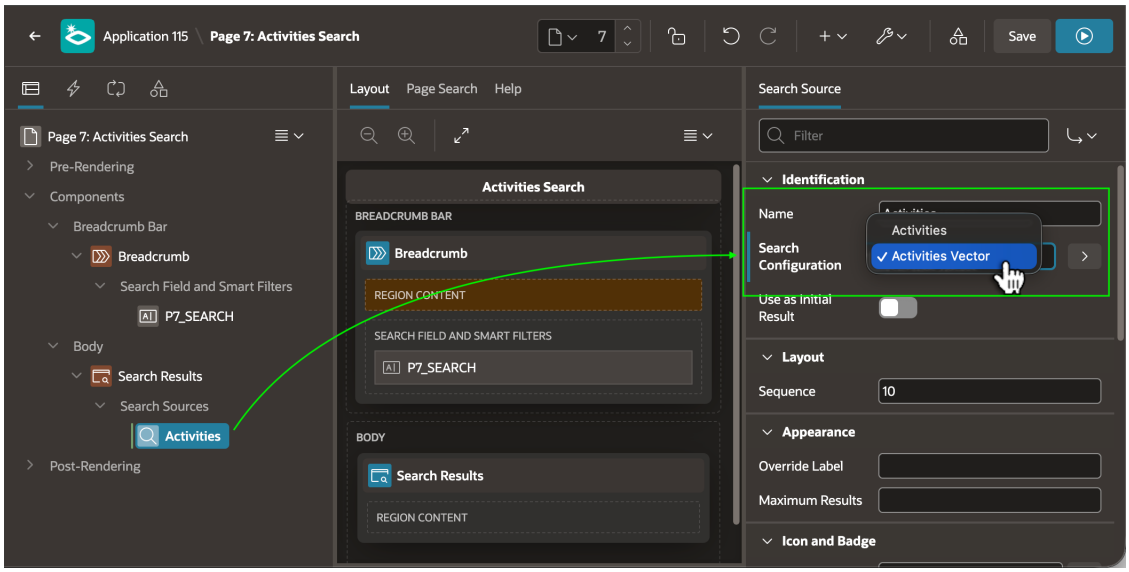


Then, reference the *Activities Vector* search configuration as the source of a Search region. Your existing search page already has one, but it currently uses a *standard* search

Copyright © 2026, Oracle and/or its affiliates.

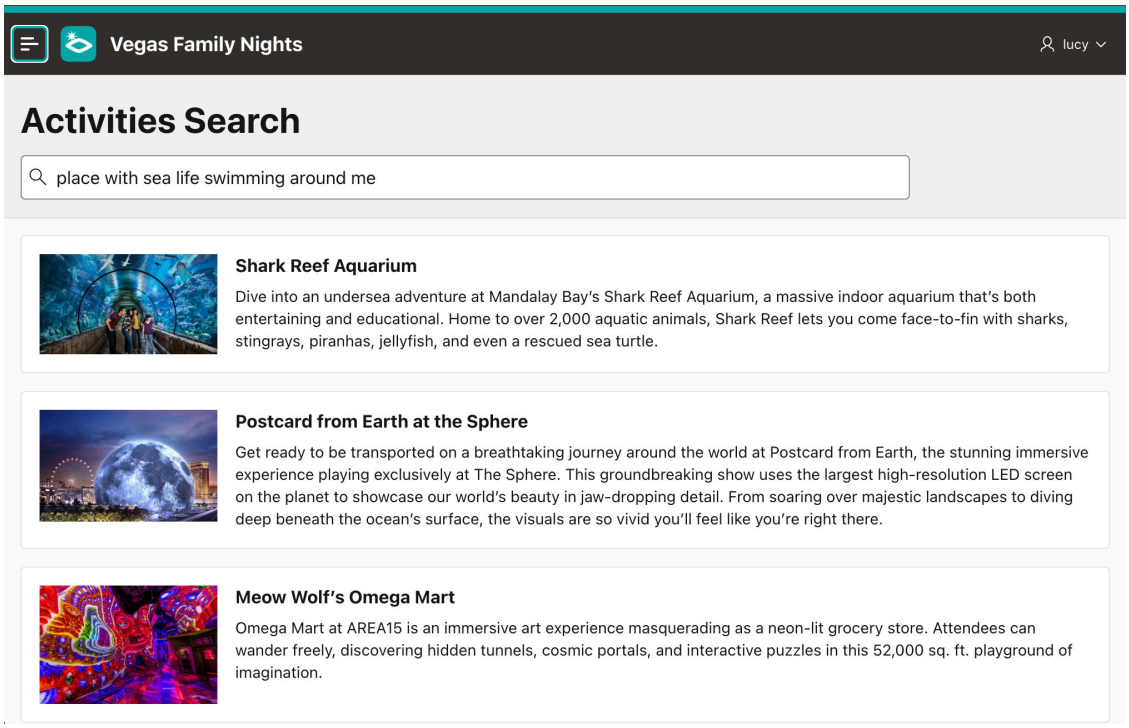
configuration that searches over regular column data. As shown below, just switch the Search Source to use the *Activities Vector* search configuration instead.

Figure 12-26 Configuring a Search Region to Use a Vector-Aware Search Configuration



With this minimal effort, your page now offers Vegas visitors an internet-style "search by meaning" experience over family-friendly activities. At runtime, APEX uses the `GET_VECTOR_EMBEDDINGS` function in the `APEX_AI` package to convert the user's search text into a vector that encodes the meaning of their search. Then it runs a query to find the activities whose vector is closest in *meaning* to the user's search. As shown below, now Lucy finds what she's looking for. The **Shark Reef Aquarium** activity is the first hit.

Figure 12-27 Lucy Now Finds Meaningful Results, Even When Words Differ



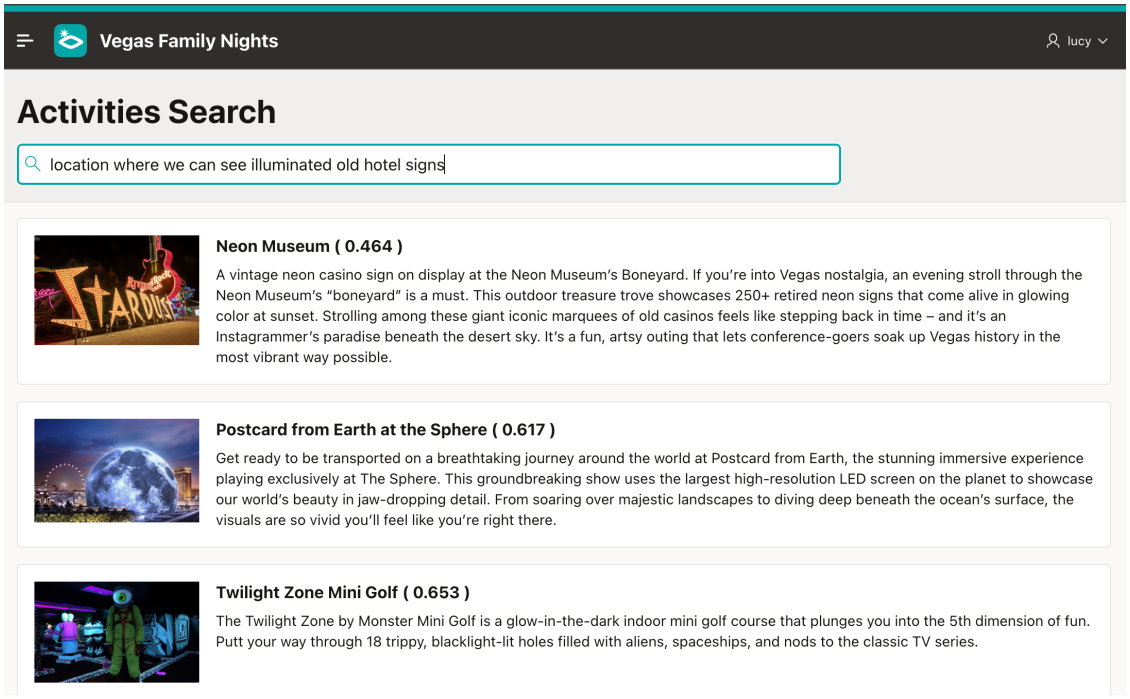
Copyright © 2026, Oracle and/or its affiliates.

Understanding Vector Distance

When a user enters a search, APEX first computes the vector that encodes the meaning of their search terms. Then it queries the `AIW_ACTIVITIES` table, with an `ORDER BY` clause to sort results by the *vector distance* between the user's search vector and the activity vectors in your table. Oracle 26ai's `VECTOR_DISTANCE()` function computes how close in meaning an activity is to the search vector. By also using `FETCH FIRST N ROWS ONLY` in the query, APEX finds only the "top N" results most similar in meaning to the user's search.

By temporarily enabling the **Custom Layout** switch in the **Appearance** section of the Search region's **Attributes** tab, you can add a reference to `&DISTANCE.` to display the vector distance computation for the results shown. Trying Lucy's other search, the results with distance displaying next to the title appear below. Notice the first result is spot on. When Lucy's searches for "*location where we can see illuminated old hotel signs*", the **Neon Museum** comes up first with a semantic distance of `0.464`.

Figure 12-28 Trying Another Search, Temporarily Studying Vector Distance of Results



The results also show two other activities that are semantically father away from the meaning of her search. *The Sphere* is a giant spherical projection screen inside and out, so it technically is a kind of *illuminated sign*. And the *Twilight Zone Mini Golf* description mentions "glow-in-the-dark" and "backlight-lit", so it also seems somewhat illumination-related. Sometimes the semantic relevance of a result may be less obvious to the end user, to the point of possibly being confusing: *Hmmmm. Why mini golf?*

To find only the most relevant results with the *strongest* semantic connection, you can refine your vector search strategy. First, find the result *closest* in meaning to the user's search. Then, only include *other* results whose semantic distance from the search vector falls within a reasonable threshold around the *best* result. Think of this approach as a "best-score proximity cutoff".

Copyright © 2026, Oracle and/or its affiliates.

Applying a Best-Score Proximity Cutoff Approach

To implement a best-score proximity cutoff, you first need to find the best score: the *closest* result's distance from the search vector. This is the *minimum* vector distance between the user's search vector and the activity rows' `SEMANTIC_CONTENT` vectors. Assuming a 10% cutoff from the best result and that `p_search_text` contains the user's search text, the query looks like this. It computes 10% more than the minimum vector distance between the user's search vector and activities' `SEMANTIC_CONTENT` vector. Notice `doc_model` is the name of the ONNX model installed in the database, used to produce the

```
select min(
        vector_distance(
          apex_ai.get_vector_embeddings(
            p_value             => p_search_text,
            p_service_static_id => 'vegas-family-nights'),
          a.semantic_content)) * 1.10 /* 10% more than minimum */
  from aiw_activities a
```

To make this best activity distance calculation more maintainable, you can create a scalar SQL macro like the `BEST_ACTIVITY_DISTANCE` one below. It lets you define a reusable fragment of SQL syntax that produces a scalar result.

Like a PL/SQL function, a macro can accept parameters with default values. The macro returns text the database includes into a SQL statement at runtime, wherever a scalar value is allowed. For example, in a `WHERE` clause predicate. The macro text result can include verbatim references to the macro parameter names, which the database handles as bind variables at query execution time.

```
create or replace function best_activity_distance(
    p_search_text in varchar2,
    p_factor      in number default 1.10
) return varchar2
    sql_macro(scalar)
is
begin
    return q'[
        (select min(
                  vector_distance(
                    apex_ai.get_vector_embeddings(
                      p_value             => p_search_text,
                      p_service_static_id => 'vegas-family-nights'),
                    a.semantic_content)) * p_factor
           from aiw_activities a)
    ]';
end;
```


Copyright © 2026, Oracle and/or its affiliates.

> **Tip**
>
> Although the scalar macro returns SQL text, it does ***not*** copy the user's search text or factor value into that SQL. The returned expression references the macro parameters `p_search_text` and `p_factor`. Oracle expands those references using the argument expressions from the calling SQL statement. A bind argument remains a bind variable in the expanded expression. A literal or SQL expression remains a literal or expression. In each case, the argument supplies a value to the expression; it does not become executable SQL text. The database still parses, type-checks, and executes the expanded expression as normal.

With the `BEST_ACTIVITY_DISTANCE` macro defined, you can apply it to vector search results. The `APEX_SEARCH` package contains a `SEARCH` function that returns the results of a Search Configuration for a given user's search text. To limit the results to those within 10% of the "most similar in meaning" one, use a query like the one below.

Notice it includes the search configuration's vector `DISTANCE` score column, then limits results with a `WHERE` clause to rows whose semantic distance falls within the best-score proximity cutoff. Since it omits the second parameter to `best_activity_distance`, that defaults to a 10% range. A *Content Row* region's **Order By Clause** gets configured as a separate property highlighted below.

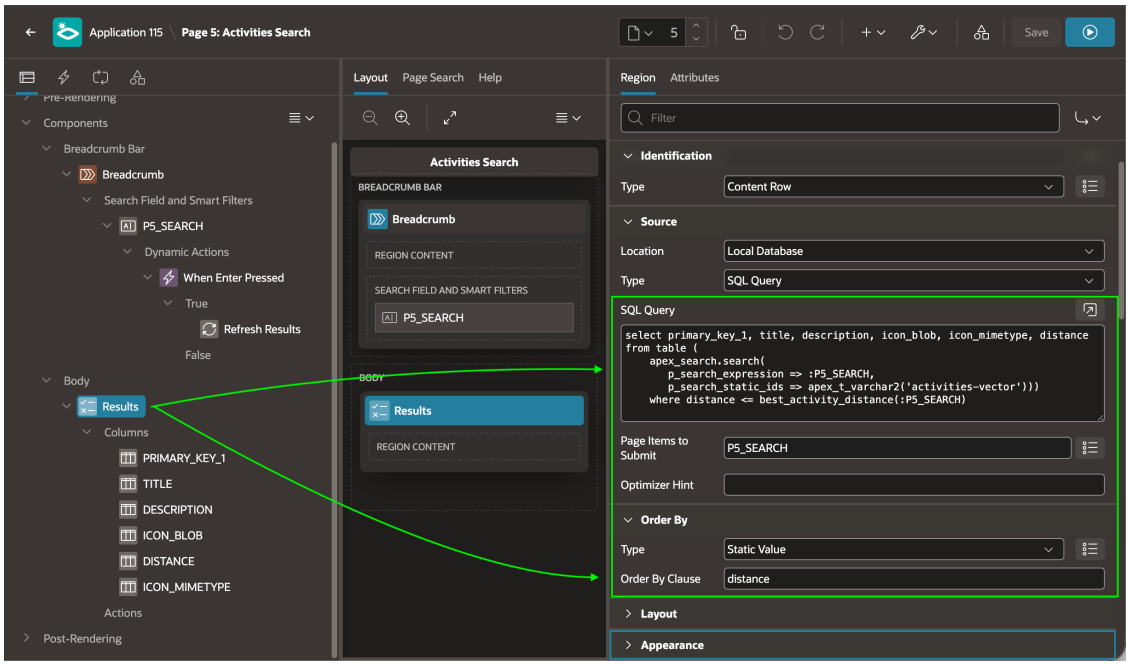
```
select primary_key_1, title, description, icon_blob, icon_mimetype, distance
  from table(
         apex_search.search(
             p_search_expression => :P5_SEARCH,
             p_search_static_ids => apex_t_varchar2('activities-vector')))
 where distance <= best_activity_distance(:P5_SEARCH)
```

The figure shows this query in use in a **Results** *Content Row* region. Users type their search into a `P5_SEARCH` field in the *Search Field* slot of the Breadcrumb bar. A *When Enter Pressed* dynamic action refreshs the *Results* region when the user pressed the [*Enter*] key. The page uses the technique explained in Including BLOB Images in Content Row to size the avatar images in the results.

Notice that in addition to configuring the query, **Page Items to Submit** is set to `P5_SEARCH` because the query references it as bind variable. Also observe the **Order By Clause** is the static `distance` column to sort on vector search distance, closest results first.

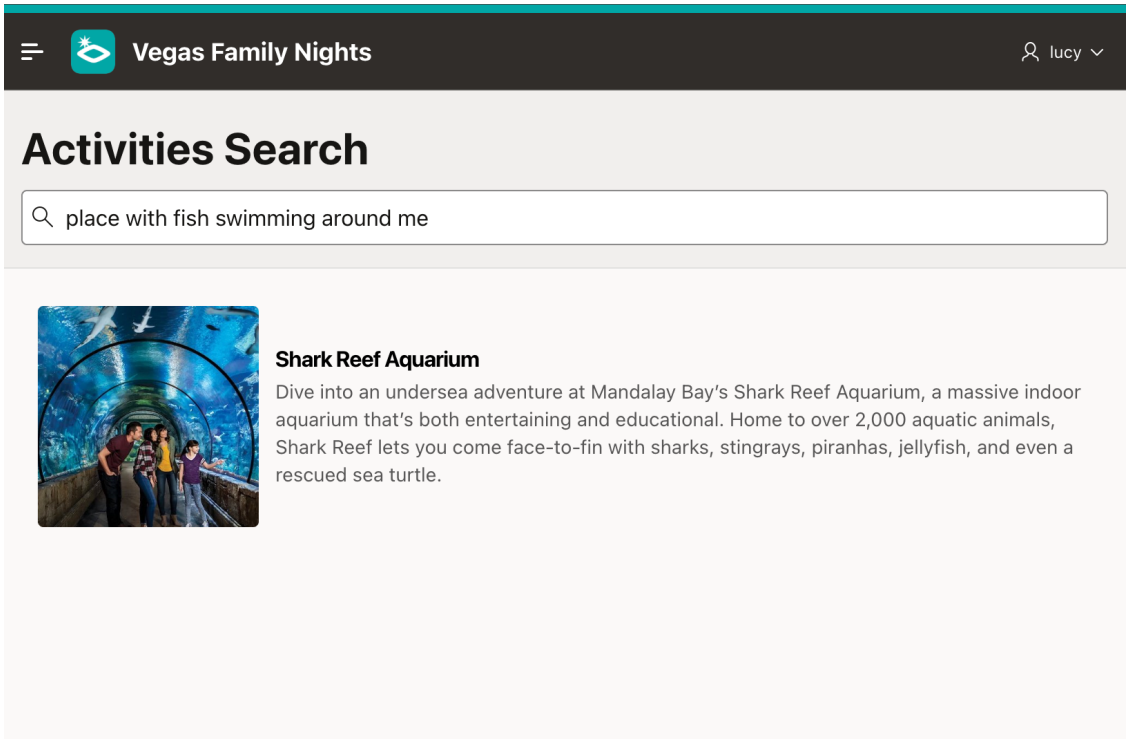
Copyright © 2026, Oracle and/or its affiliates.

Figure 12-29 Applying Best Score Proximity Cutoff to Search Configuration Results in a Content Row



The result shows how Lucy now sees only the most relevant results for her search.

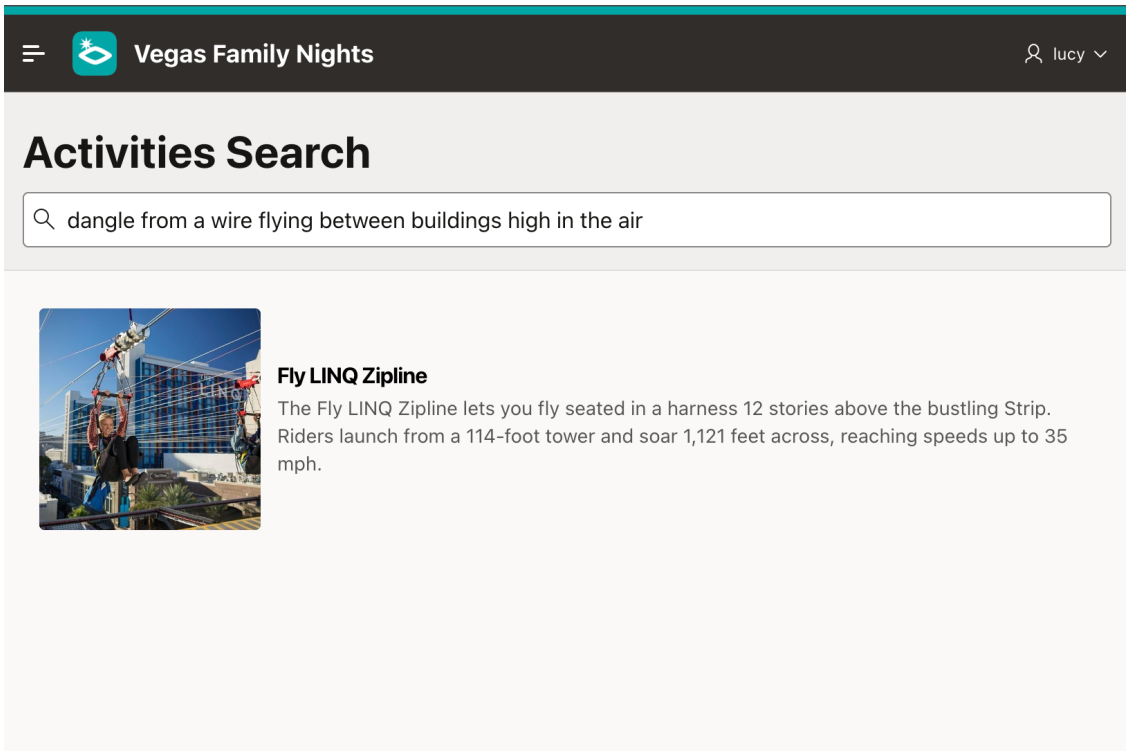
Figure 12-30 Lucy Now Sees Only the Most Relevant Results



Copyright © 2026, Oracle and/or its affiliates.

When she searches for any activities where she would "*dangle from a wire flying between buildings high in the air*", your app now presents the obvious winner for the family activity on Saturday: Fly LINQ Zipline.

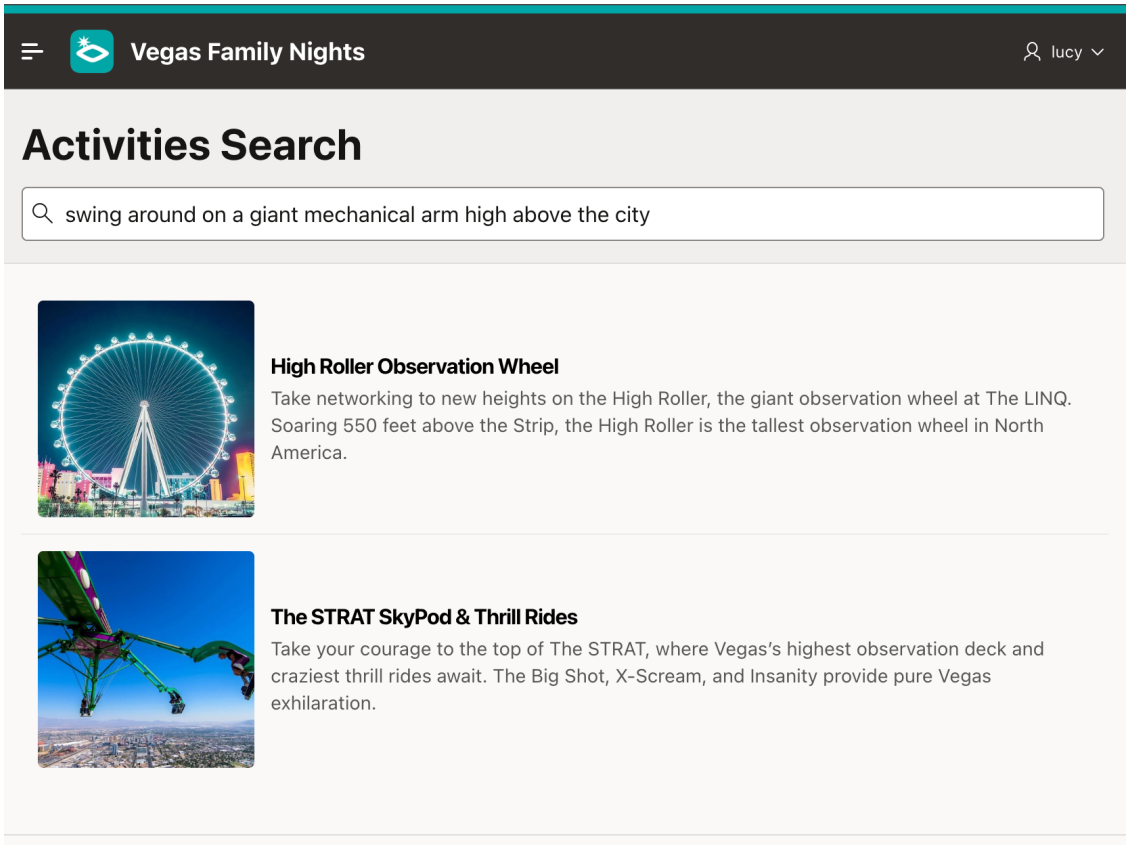
Figure 12-31 Searching by Meaning, Lucy Finds Exactly What She's Looking For



The result is not necessarily just a single activity. If Lucy wants to "*swing around on a giant mechanical arm high above the city*" two very relevant results appear. The kids decide to start slow with the view from atop the Observation Wheel, then graduate to something even more thrilling.

Copyright © 2026, Oracle and/or its affiliates.

Figure 12-32 Multiple Activities Land Inside the Best Score Proximity Cutoff



For more information about using the Search region and search configurations, see Unified Search Results Across Sources.

Copyright © 2026, Oracle and/or its affiliates.

13

Integrating Remote Data

The fastest way to work with data in Oracle APEX is to use tables or views in the local database. But many applications also need to integrate with remote data sources.

APEX makes it easy to work with remote Oracle and MySQL databases, REST APIs that follow common patterns such as Fusion Applications, ORDS, and OData services, as well as any other REST API. To simplify working with a set of REST APIs that use custom conventions, create a custom REST Data Source Adapter plug-in to encapsulate their rules and automate their use in your apps.

- Using Remote Oracle and MySQL Data
  Oracle REST Data Services' *REST-Enabled SQL* feature lets authenticated remote applications access data in an Oracle database or Oracle Cloud MySQL database system.
- Configuring Secure Web Credentials
  Use APEX Web Credentials to securely store the authentication details required for REST-Enabled SQL services and other REST API access.
- Using REST APIs as Data Sources
  To work with data from REST APIs, use a *REST Data Source*. Each one you create can support one or more standard REST operations to `GET`, `PUT`, `POST`, `PATCH`, or `DELETE` data from a REST API whose endpoint URL you configure.
- Invoking a REST Operation Declaratively
  Use the *Invoke API* page process to call a REST Data Source operation with no code.
- Using REST APIs in Code
  Sometimes your applications need to work with REST APIs in code. Using REST Data Sources is the maximally-declarative approach. In contrast, the `MAKE_REST_REQUEST` function in the `APEX_WEB_SERVICE` package provides additional control but requires writing more code.
- Extra Fusion Apps, ORDS, OData Features
  When related REST APIs follow common conventions for data filtering, sorting, attribute selection, object modification, and error reporting, Oracle APEX can provide powerful additional low-code functionality to your apps.
- Additional REST Data Source Features
  REST Data Sources can cache responses, synchronize reference data, add computed columns, apply declarative filtering and sorting, combine local and remote data, and support custom adapter plug-ins.
- Debugging REST API Errors
  If you work with REST APIs, one of them might return an error at runtime. Use APEX's built-in debug tracing facility to diagnose the problem and identify a fix.

13.1 Using Remote Oracle and MySQL Data

Oracle REST Data Services' *REST-Enabled SQL* feature lets authenticated remote applications access data in an Oracle database or Oracle Cloud MySQL database system.

In Oracle APEX, you can reference a REST-Enabled SQL service by providing the endpoint URL and authentication credentials. Then you can use data in the remote database in any

Copyright © 2026, Oracle and/or its affiliates.

APEX region just like a local table. Oracle APEX handles all the details of interacting with the remote database.

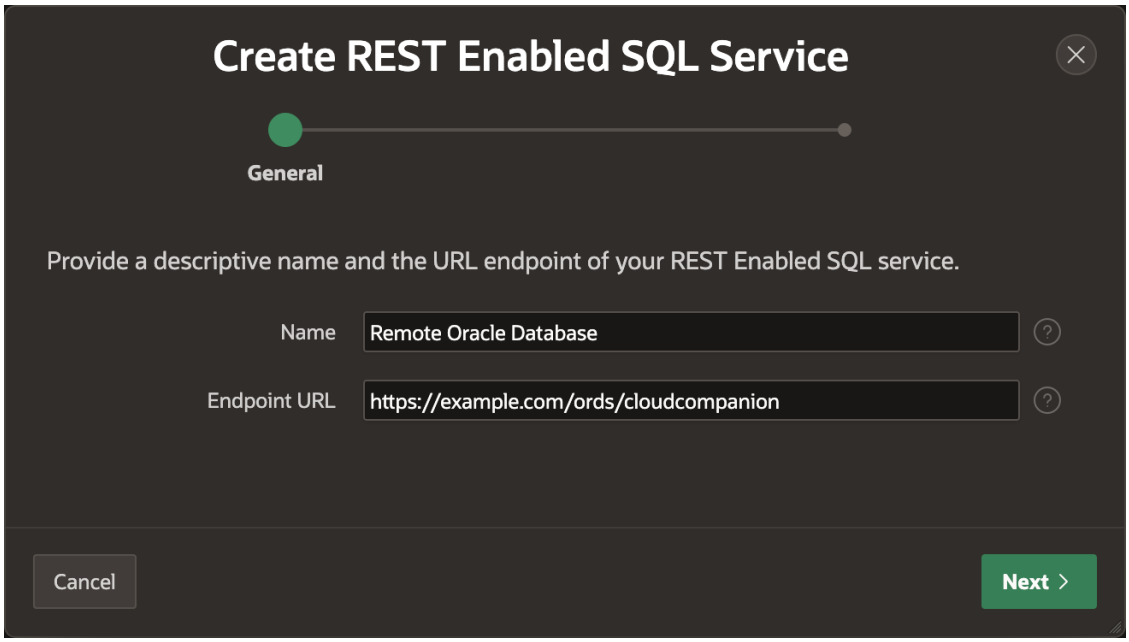
- Referencing a REST-Enabled SQL Service
  Define a REST-Enabled SQL reference to work with the data in a REST-Enabled SQL Service.
- REST-Enabled SQL Service URL
  The format of the URL clients like APEX need to work with your REST-Enabled SQL Service depends on where its running.
- Using a REST-Enabled SQL Service
  Once you've created a REST-Enabled SQL Service reference at workspace level, you can use it almost anywhere you can configure a data source.

13.1.1 Referencing a REST-Enabled SQL Service

Define a REST-Enabled SQL reference to work with the data in a REST-Enabled SQL Service.

A REST-Enabled SQL reference is a workspace-level definition. Once created, you can use it in any application.As shown below, the *Create REST Enabled SQL Service* wizard in *Workspace Utilities* lets you assign a meaningful name to the endpoint.

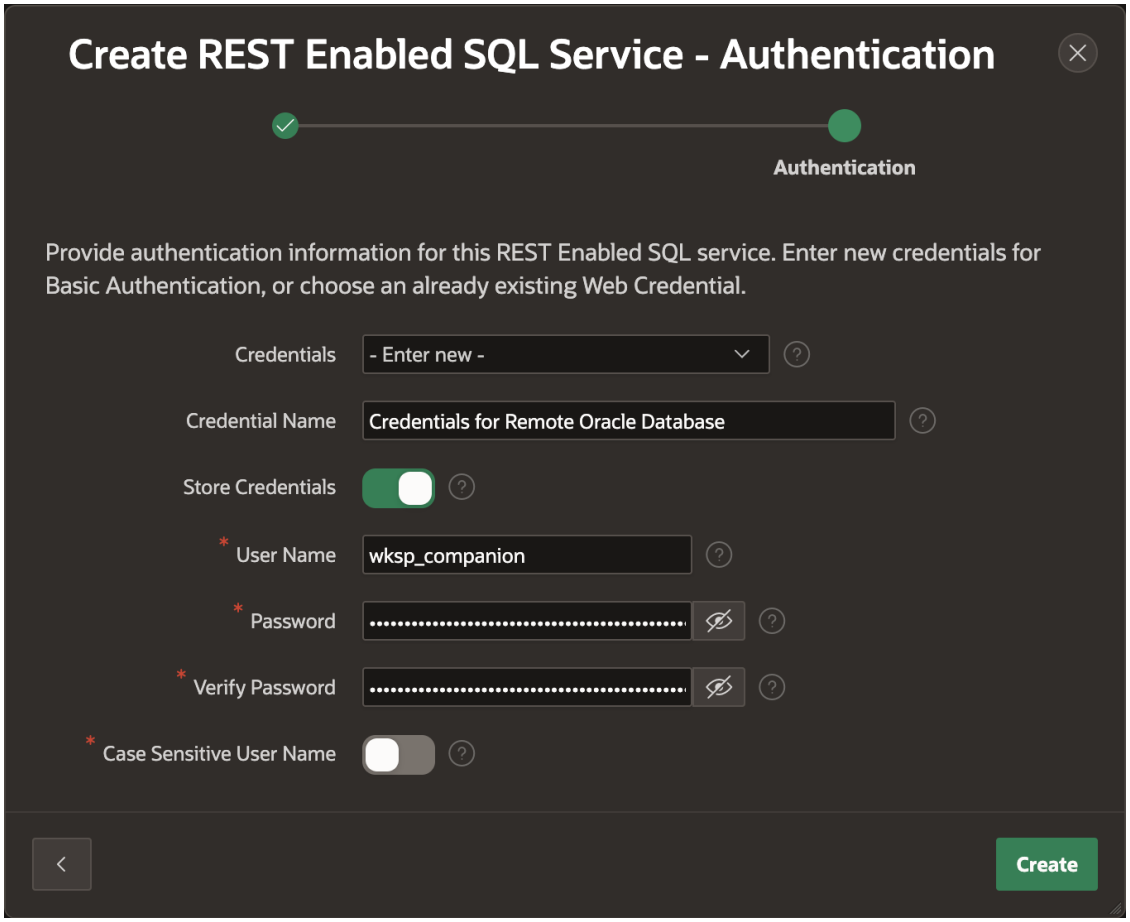
Figure 13-1 Referencing a REST-Enabled SQL Service in Workspace Utilities



As shown below, you can enter new authentication credentials on the second step of the wizard, or choose a credential you might have defined previously. Most REST-Enabled SQL services use basic authentication over HTTPS. This includes ones on-premises and in Autonomous AI Database.

Copyright © 2026, Oracle and/or its affiliates.

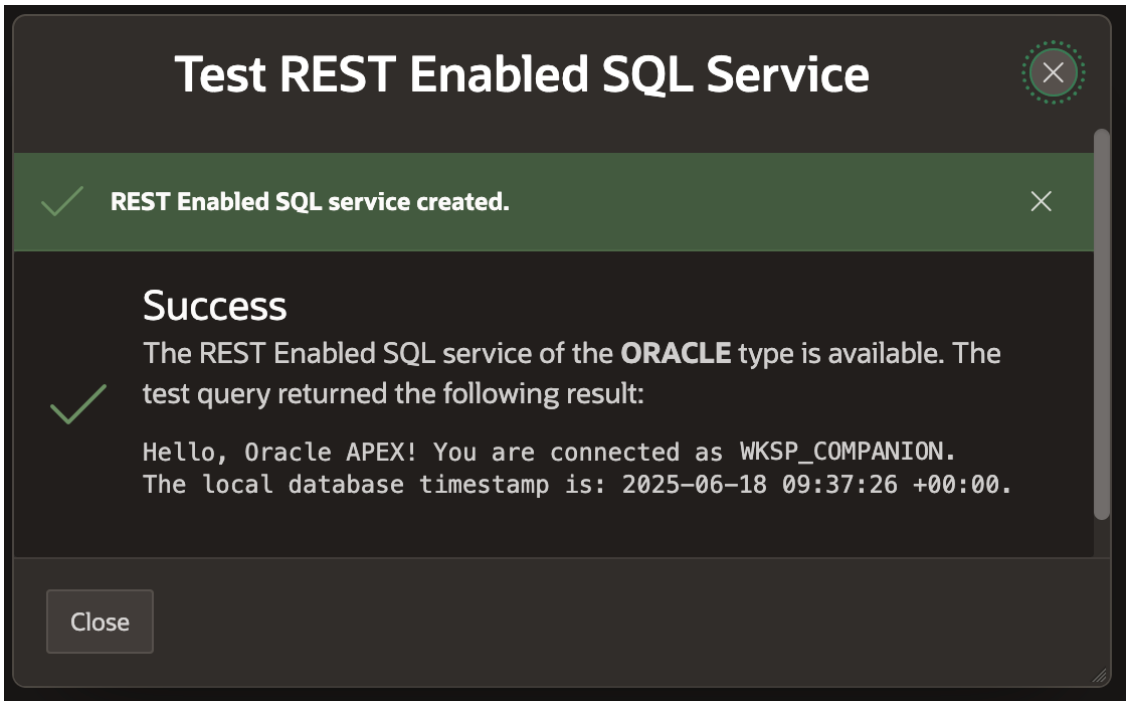
Figure 13-2 Providing Credentials for REST-Enabled SQL Service



When you click *(Create)* APEX validates the endpoint URL and authentication to confirm everything is working. As shown below, if all is well, you'll see a confirmation to that effect.

Copyright © 2026, Oracle and/or its affiliates.

Figure 13-3 Confirmation of Successfully Authenticated REST-Enabled SQL Connection



13.1.2 REST-Enabled SQL Service URL

The format of the URL clients like APEX need to work with your REST-Enabled SQL Service depends on where its running.

For most REST-Enabled SQL services – including those running on-premises or in Autonomous AI Database – use an endpoint URL like the following in APEX. It includes the ORDS schema alias you chose when enabling REST access, and *omits* the trailing `/_/sql` suffix that APEX adds automatically:

`https://domainname/ords/your_schema_alias`

To enable your OCI MySQL Database System for REST-Enabled SQL access, just define a Database Tools Connection for it. When using an Oracle Cloud Infrastructure (OCI) Database Tools Connection for your REST-Enabled SQL service, the service URL includes the Oracle Cloud ID of your connection. This OCID will be a long string like:

`ocid1.databasetoolsconnection.oc1.eu-frankfurt-1.xxxxxx`

Note the *eu-frankfurt-1* OCI region identifier. Your region may be different. The REST-Enabled SQL service URL to use in Oracle APEX looks like this:

`https://sql.dbtools.yourregion.oci.oraclecloud.com/20201005/ords/connectionocid`

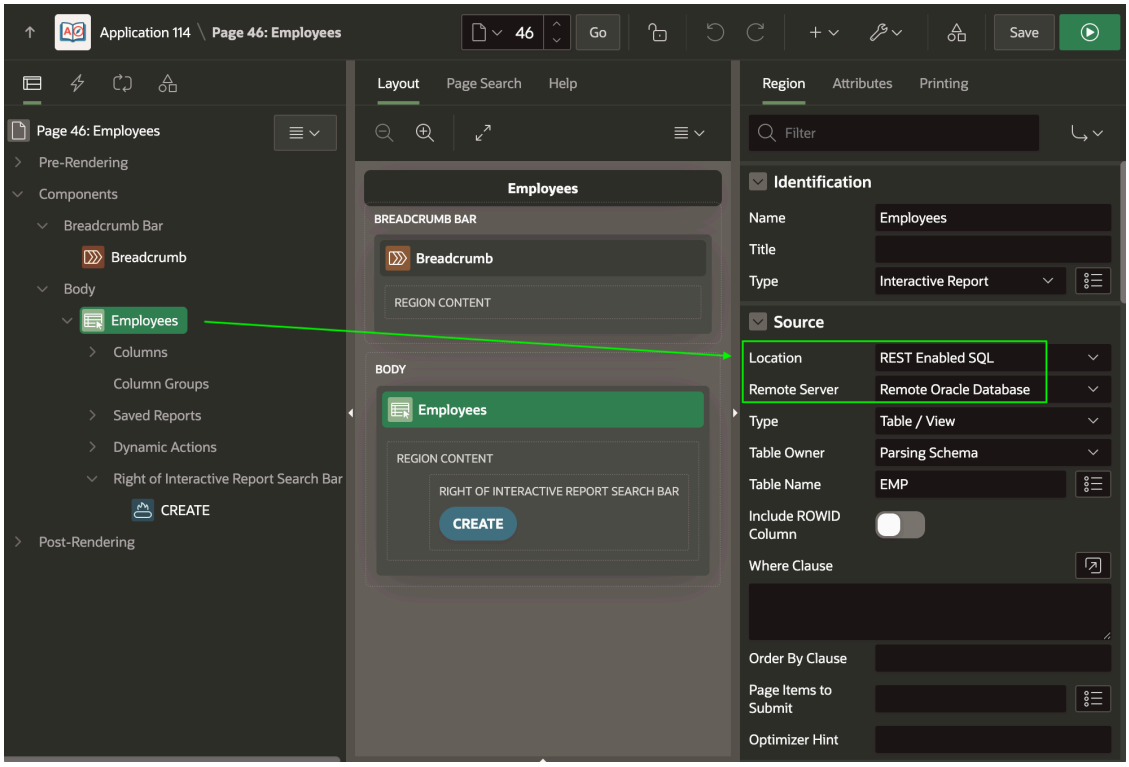
Copyright © 2026, Oracle and/or its affiliates.

13.1.3 Using a REST-Enabled SQL Service

Once you've created a REST-Enabled SQL Service reference at workspace level, you can use it almost anywhere you can configure a data source.

You can use a REST-Enabled SQL Service reference in any APEX region, List of Values, or other dynamic list use cases. You do this by choosing a source *Location* of *REST-Enabled SQL*. For example, as shown below, the *Employees* Interactive Report region uses the `EMP` table in the *Remote Oracle Database* defined above.

Figure 13-4 Using a Table in a Remote Database with REST-Enabled SQL



All the *Source* > *Type* options are available against a remote *Oracle* database, so you can also use a *SQL Statement* and *Function Body Returning SQL Query* as well. For a remote *Oracle* database, all read and write operations are supported. In contrast, remote MySQL databases are read-only. While you can display MySQL data in any APEX region including *Form* and *Interactive Grid*, the latter two regions' automatic DML page processes are not yet supported.

13.2 Configuring Secure Web Credentials

Use APEX Web Credentials to securely store the authentication details required for REST-Enabled SQL services and other REST API access.

Once you have saved a credential secret, it is encrypted and there is no way to retrieve it in clear text. APEX supports many different kinds of stored credentials, including:

Copyright © 2026, Oracle and/or its affiliates.

Table 13-1 Supported Web Credential Types

Web Credential Type	Used For
Basic Authentication	Most REST-Enabled SQL access and some REST APIs
OAuth2	• Client credentials flow • Password flow • Authorization code flow when used with *Social Sign-in* Authentication Scheme
OCI Native Authentication	• Database Tools connection REST-Enabled SQL service • Other OCI REST APIs
HTTP Header	REST services requiring auth token or API Key in a header
URL Query String	REST services requiring an API key in a URL parameter
Key Pair	Push notifications and user assertion signing

Basic Authentication and OAuth2 can optionally delegate to a securely stored credential defined in the Oracle database as well.

> **Tip**
>
> To define REST-Enabled SQL Service for an OCI Database Tools connection, start by defining an appropriate *OCI Native Authentication* web credential. Then choose this existing credential from the list in the REST-Enabled SQL Service wizard.

13.3 Using REST APIs as Data Sources

To work with data from REST APIs, use a *REST Data Source*. Each one you create can support one or more standard REST operations to `GET`, `PUT`, `POST`, `PATCH`, or `DELETE` data from a REST API whose endpoint URL you configure.

At runtime, APEX automatically uses the web credential you associate with it to authenticate access. If the service supports returning data in pages, you can choose from a number of common pagination strategies.

You can specify various kinds of parameters at the service or operation level to get data into and out of an API call. To simplify using the data the API returns in your pages, you specify *Data Profile* columns that let APEX treat the data as a rowset. In many cases, App Builder can discover these column descriptions automatically, which you can then further adjust to fit your requirements.

Once you've defined a REST Data Source, you can use it almost anywhere in your application. This means in any APEX region, List of Values, or other dynamic list use cases. You do this by choosing a source *Location* of *REST Source*. When useful, you can easily augment the data retrieved from the REST API to perform additional filtering, ordering, and joins to combine remote data with local data. To improve performance, you can take advantage of various caching options as well as local synchronization of less-frequently-changing data on a schedule you define.

If your REST Data Source supports appropriate operations enabling insert, updates, and deletes, your pages can also use it to create and modify data as well. Some API suites define convention-based enhancements for filtering, sorting, and modifying data. APEX natively supports these features for Oracle Fusion Applications, Oracle REST Data Services, and

Copyright © 2026, Oracle and/or its affiliates.

OData. For other situations, you can create a custom REST adapter plug-in to provide the same seamless, extended data access for any other API suite you may need to integrate with.

- Using a Simple HTTP REST Data Source
  The *Simple HTTP* service is the most generic and flexible REST Data Source type APEX offers. By understanding its capabilities and how to configure them, you can confidently integrate with any necessary API.
- Service URL Uses Remote Server
  When working with REST APIs, you often use several endpoints from the same remote server. These service URLs share a base URL, including the domain name and any common path segments. Only the *trailing* part of the Service URL differs.
- Data Profile Helps Turn JSON Into Rows
  Each REST Data Source includes a *Data Profile*. Its details let APEX automatically convert a REST API's JSON response into rows like a database query returns.
- Retrieving Remote Data Page by Page
  To configure pagination behavior on your *Simple HTTP* REST Data Source, use the *Settings* tab on the edit page.
- REST Operations and Database Actions
  You can define one or more operations for each REST Data Source. APEX queries, inserts, updates, or deletes rows using the operation whose *Database Action* is appropriate to each task. It uses the operation's *HTTP Method* when sending the HTTP request to the REST API endpoint URL.
- Configuring a Request Body Template
  Your operations for *Insert row* and *Update row* need to send data to the REST API in the request body.
- Supply and Return Values with Parameters
  Just as PL/SQL procedures can define `IN`, `OUT`, and `IN/OUT` parameters, so can REST Data Sources. Define a parameter at the operation level if it's specific to that action, or at the data source level if it's relevant to *all* operations.

13.3.1 Using a Simple HTTP REST Data Source

The *Simple HTTP* service is the most generic and flexible REST Data Source type APEX offers. By understanding its capabilities and how to configure them, you can confidently integrate with any necessary API.

If the service you need to use supports a `GET` method, then the *Create REST Data Source* wizard gives you a head start by automatically discovering the data profile columns based on the JSON response payload. If not, you can always create and configure the REST Data Source manually.

Imagine you need to integrate with a REST API for working with employees with the endpoint URL:

```
https://example.com/ords/cloudcompanion/emp
```

For simplicity, assume the endpoint allows *public* access, knowing you can configure any kind of authentication to access the REST APIs you need to work with. You can examine the payload it returns using a command line tool like `curl`:

```
$ curl -L https://example.com/ords/cloudcompanion/emp
```


Copyright © 2026, Oracle and/or its affiliates.

This example service responds with the following JSON payload in which ▒ replaces repeating elements. There are 14 employees total: notice a `count` value of 14 and `hasMore` value of `false`.

```
{
  "items": [
    {
      "empno": 7839,
      "ename": "KING",
      "job": "PRESIDENT",
      "mgr": null,
      "hiredate": "1981-11-17T00:00:00Z",
      "sal": 5000,
      "comm": null,
      "deptno": 10,
      "links": [
        {
          "rel": "self",
          "href": "https://example.com/ords/cloudcompanion/emp/7839"
        }
      ]
    },
    ▒
  ],
  "hasMore": false,
  "limit": 25,
  "offset": 0,
  "count": 14,
  "links": [
    {
      "rel": "self",
      "href": "https://example.com/ords/cloudcompanion/emp/"
    },
    ▒
  ]
}
```

Before defining a REST Data Source for the endpoint, consult the documentation for the service to understand whether it supports retrieving data in pages. This service supports using the `limit` query string parameter to set a page size, and the `offset` parameter to indicate a number of rows to skip.

This means you can retrieve just the first two employees using the `curl` command:

```
$ curl -L "https://example.com/ords/cloudcompanion/emp?limit=2"
```

To fetch employee rows 3 and 4 use a `curl` command that skips the first 2 and retrieves up to two more:

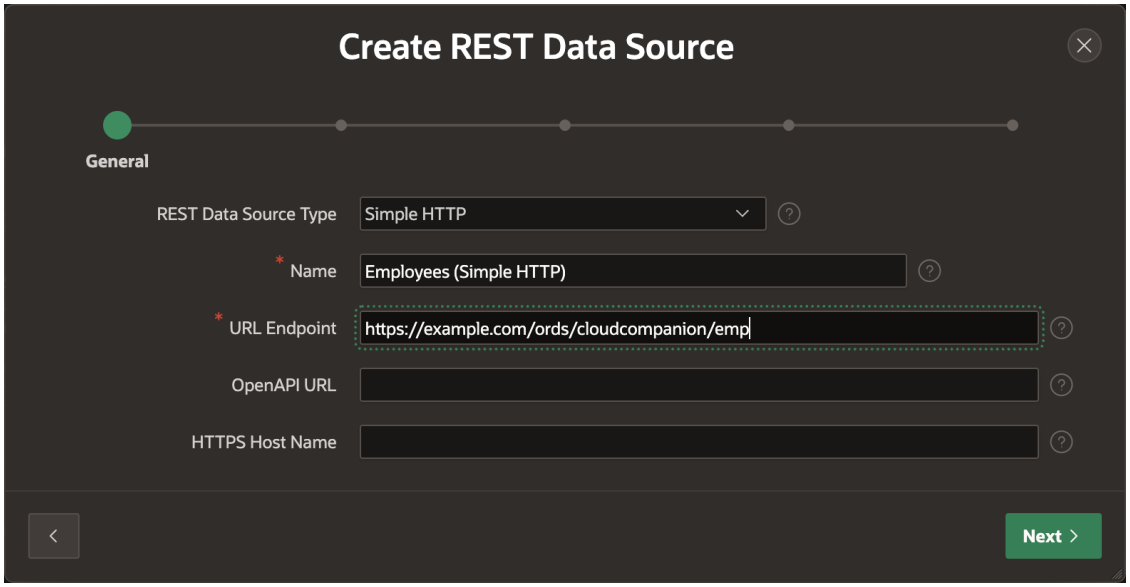
```
$ curl -L "https://example.com/ords/cloudcompanion/emp?offset=2&limit=2"
```

To create a REST Data Source to work with this API in your APEX application, as shown below, just use *Create REST Data Source* wizard from the *Shared Components* > *REST Data*

Copyright © 2026, Oracle and/or its affiliates.

Sources page. To get started, all you need to enter is the endpoint URL and proceed to the subsequent wizard steps.

Figure 13-5 Defining a New REST Data Source Using Its Endpoint URL



13.3.2 Service URL Uses Remote Server

When working with REST APIs, you often use several endpoints from the same remote server. These service URLs share a base URL, including the domain name and any common path segments. Only the *trailing* part of the Service URL differs.

For example, our *Employees* service uses the URL:

```
https://example.com/ords/cloudcompanion/emp
```

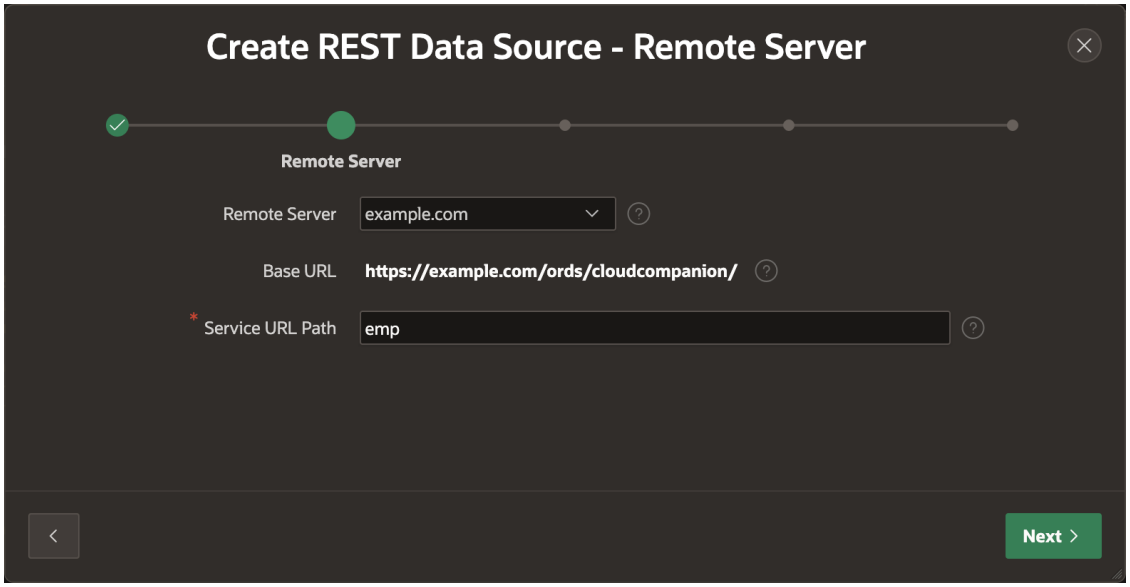
A REST API from the same organization for departments data might have the slightly *different* URL:

```
https://example.com/ords/cloudcompanion/dept
```

APEX supports this common usage pattern by storing the *base* URL in a *Remote Server* definition. When you create a new REST Data Source, APEX notices if the endpoint URL matches the base URL of an *existing* Remote Server. If not, it helps you define a new Remote Server to reuse later. As shown below, APEX detects the *Employees* service endpoint URL matches an existing *example.com* Remote Server, and assigns the trailing `emp` portion of the URL to the *Service URL Path* field.

Copyright © 2026, Oracle and/or its affiliates.

Figure 13-6 Service Endpoint URL Base URL Comes from Remote Server Definition



At runtime, APEX concatenates a REST Data Source's Remote Server *Base URL* and its *Service URL Path* to determine the full URL to use. APEX keeps these concepts separate to simplify another common requirement of REST integration. In your development environment, your application's REST Data Sources may work against a set of *test* endpoints. It is highly likely that in your production environment the application will need to work against the *live* endpoints that will reside on different *production* domains. APEX lets you set the base URL for each Remote Server differently in each environment, making it trivially easy to have your application talk to the *live* REST endpoints in production. The different Remote Server base URL values stay "sticky" to each environment, so when you update your production environment with a new version of the application, your REST Data Sources continue to interact with the correct production REST APIs.

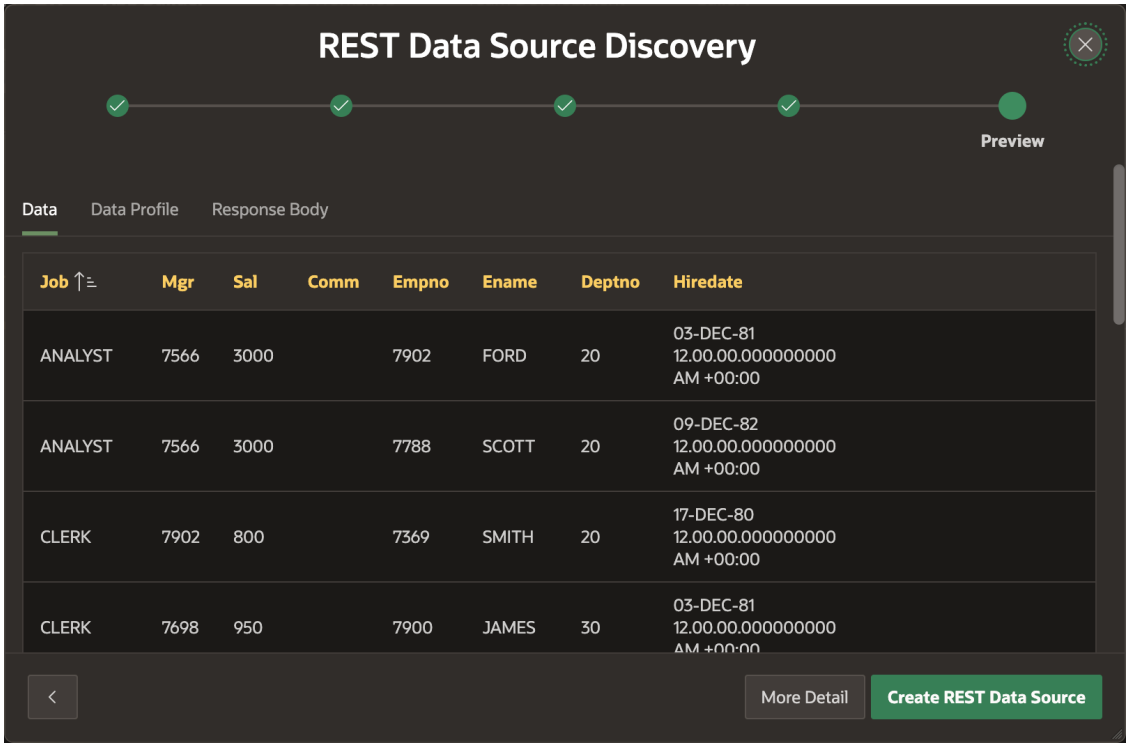
13.3.3 Data Profile Helps Turn JSON Into Rows

Each REST Data Source includes a *Data Profile*. Its details let APEX automatically convert a REST API's JSON response into rows like a database query returns.

If your endpoint supports `GET`, APEX can analyze a response payload to discover a starting set of Data Profile columns. As shown below, after this discovery step, the wizard displays the data the service returns in rows and columns. It uses the suggested Data Profile column information in the process.

Copyright © 2026, Oracle and/or its affiliates.

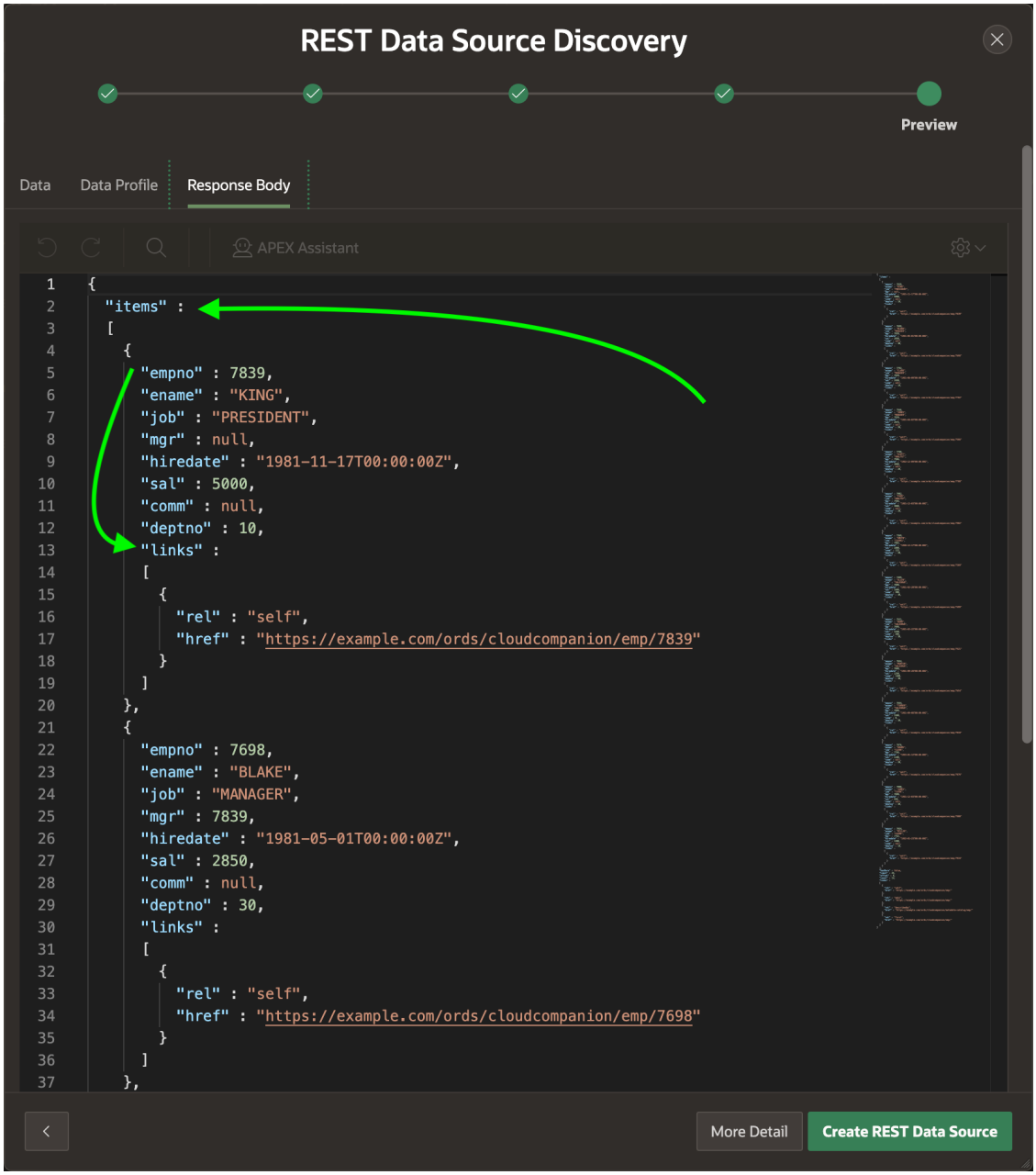
Figure 13-7 Preview of Data Rows Returned from REST Data Source



To peek at the JSON response APEX analyzed, click on the *Response Body* tab. As shown below, the *Employees* service returns a top-level `items` array of JSON objects containing employee information. Each employee JSON object contains a nested `links` array.

Copyright © 2026, Oracle and/or its affiliates.

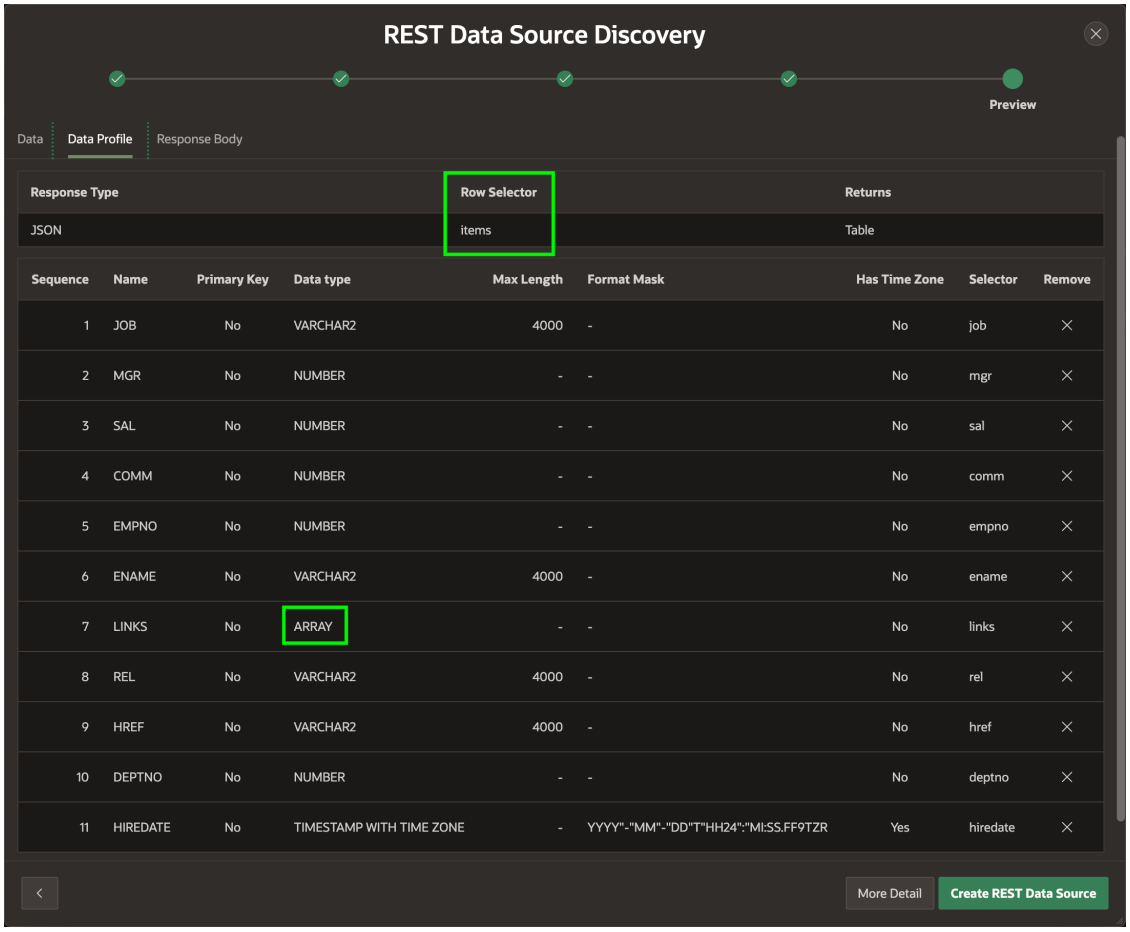
Figure 13-8 Inspecting REST API Response: items Array and Nested links Array



On the *Data Profile* tab, as shown below, you see APEX identified the `items` property as the *Row Selector*. This JSON path expression identifies the root JSON structure to convert into rows. The table below lists each discovered data profile column's name, data type, and JSON selector. APEX uses these details to extract each column's value from the JSON payload. Notice the `LINKS` column is of type `ARRAY`. The `REL` and `HREF` columns are part of this nested `links` array. You can click the "x" in the *Remove* column to omit any columns you don't want, or you can edit the Data Profile later in App Builder.

Copyright © 2026, Oracle and/or its affiliates.

Figure 13-9 Data Profile Columns Discovered Automatically by the Wizard



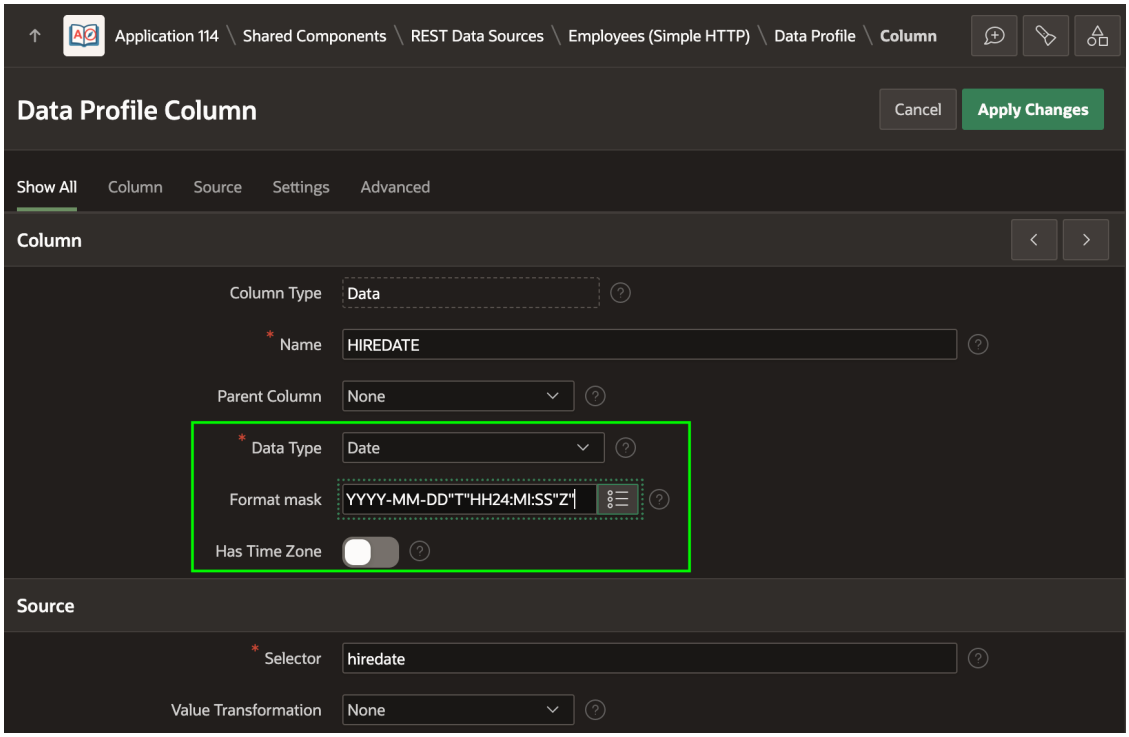
Tip

Many regions and all DML modification operations depend on identifying the primary key of the row being updated. Whenever relevant, make sure to update the data profile column that serves as the primary key to enable its *Primary Key* switch. This ensures any region columns will automatically recognize it as the primary key column so you don't have to remember to enable that switch each time you use the REST Data Source in a region.

For a column like `HIREDATE` if automatic discovery picks the wrong data type, you can adjust it later in the REST Data Source edit page. For example, say the `hiredate` property in the remote REST service is a value with only date and time information. If it does not include sub-second precision or a time zone, as shown below you can first edit the Data Profile columns, then edit the specific `HIREDATE` Data Profile column to adjust the data type to `DATE` and change its format mask to the one for an ISO 8601 standard date/time used by this REST API: `YYYY-MM-DD"T"HH24:MI:SS"Z"`

Copyright © 2026, Oracle and/or its affiliates.

Figure 13-10 Adjusting the HIREDATE Data Profile Column's Data Type



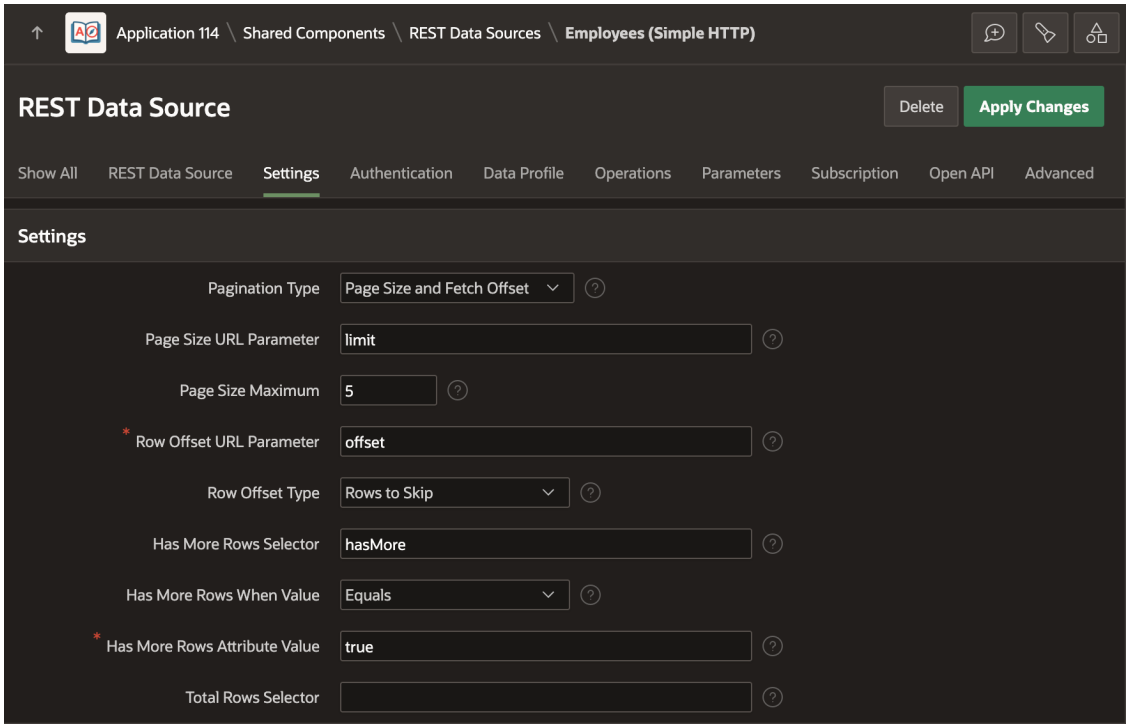
13.3.4 Retrieving Remote Data Page by Page

To configure pagination behavior on your *Simple HTTP* REST Data Source, use the *Settings* tab on the edit page.

As shown below, choose an appropriate *Pagination Type* from the list. This identifies the basic strategy your REST API uses to retrieve data page by page. Then configure the specific names your REST API uses in the different roles of your selected pagination strategy. This example *Employees* service supports specifying a page size with a `limit` parameter and a fetch `offset`. The `hasMore` property value of `true` lets APEX know there are more rows to retrieve.

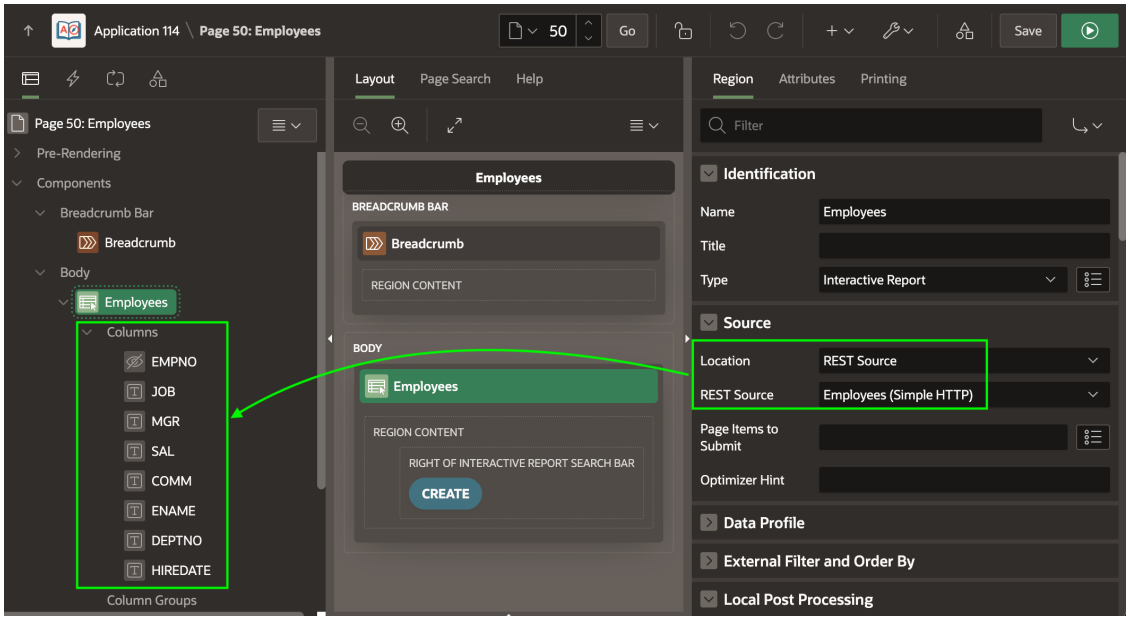
Copyright © 2026, Oracle and/or its affiliates.

Figure 13-11 Configuring a REST Data Source's Pagination Strategy



After configuring this information, APEX automatically uses the configured pagination strategy whenever you use this REST Data Source in regions that let users page through rows. For example, as shown below, an Interactive Report can use the *Employees (Simple HTTP)* REST Data Source by setting the *Location* property of the region to *REST Source*. Notice the Data Profile column names appear as the region's *Columns* in the rendering tree.

Figure 13-12 Interactive Report Using a REST Data Source



Copyright © 2026, Oracle and/or its affiliates.

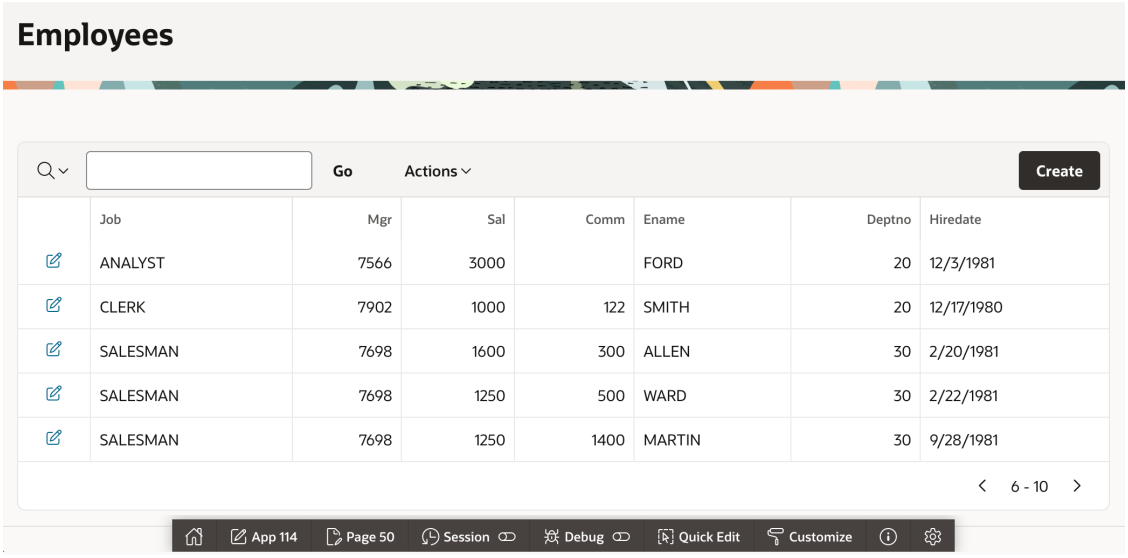
At runtime, if you use the developer toolbar to enable debug mode at the *Info* level, you'll see in the debug trace that the APEX engine requests the first five employee rows in the line:

```
making GET request to https://example.com/ords/cloudcompanion/emp/?
offset=5&limit=5
```

If you click to see the next page of five rows, you'll see the following in the debug log:

```
making GET request to https://example.com/ords/cloudcompanion/emp/?
offset=5&limit=5
```

Figure 13-13 Paging Through REST Data 5 Rows at a Time in an Interactive Report



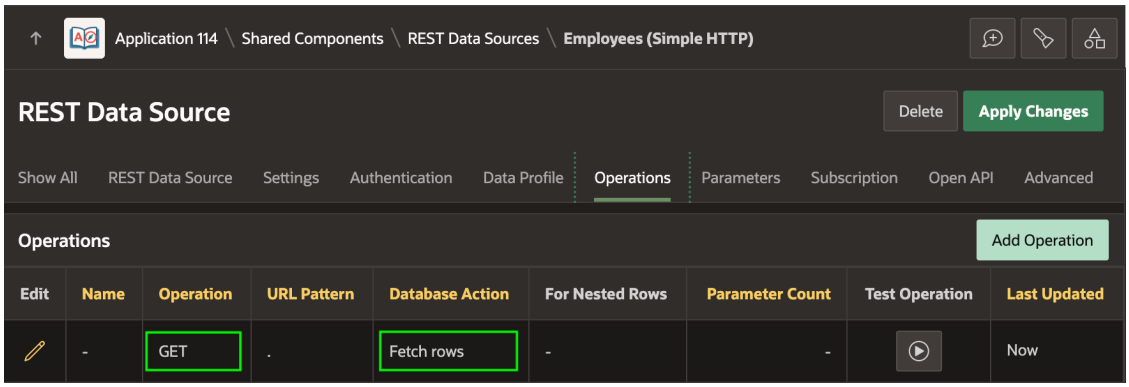
13.3.5 REST Operations and Database Actions

You can define one or more operations for each REST Data Source. APEX queries, inserts, updates, or deletes rows using the operation whose *Database Action* is appropriate to each task. It uses the operation's *HTTP Method* when sending the HTTP request to the REST API endpoint URL.

As shown below, a *Simple HTTP* REST Data Source you create with automatic discovery starts with a `GET` operation for the *Fetch Rows* database action. Until you assign a more descriptive operation name, APEX uses its *method* as an alternative name. Defined this way, the REST Data Source fully supports read-only use in any region or List of Values.

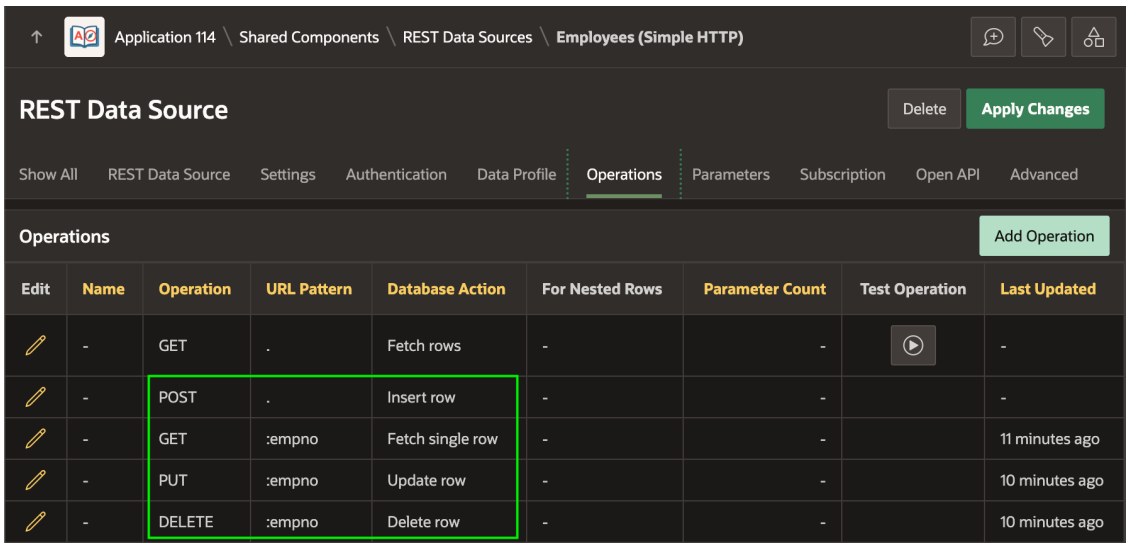
Copyright © 2026, Oracle and/or its affiliates.

Figure 13-14 Fetch Rows GET Operation for Discovered Simple HTTP REST Data Source



However, to *modify* data with a Form or Interactive Grid you need to add additional *Fetch single row*, *Insert row*, *Update row*, and *Delete row* operations as shown below.

Figure 13-15 Complete Set of Operations to Support All Database Actions



For example, to enforce lost update protection the *Automatic DML* page processes for Form and Interactive Grid regions use the *Fetch single row* database action to find a row to display by primary key. As shown below, the `GET` operation for this *Fetch single row* database action defines an additional *URL Pattern* to add onto the *REST Source Base URL*.

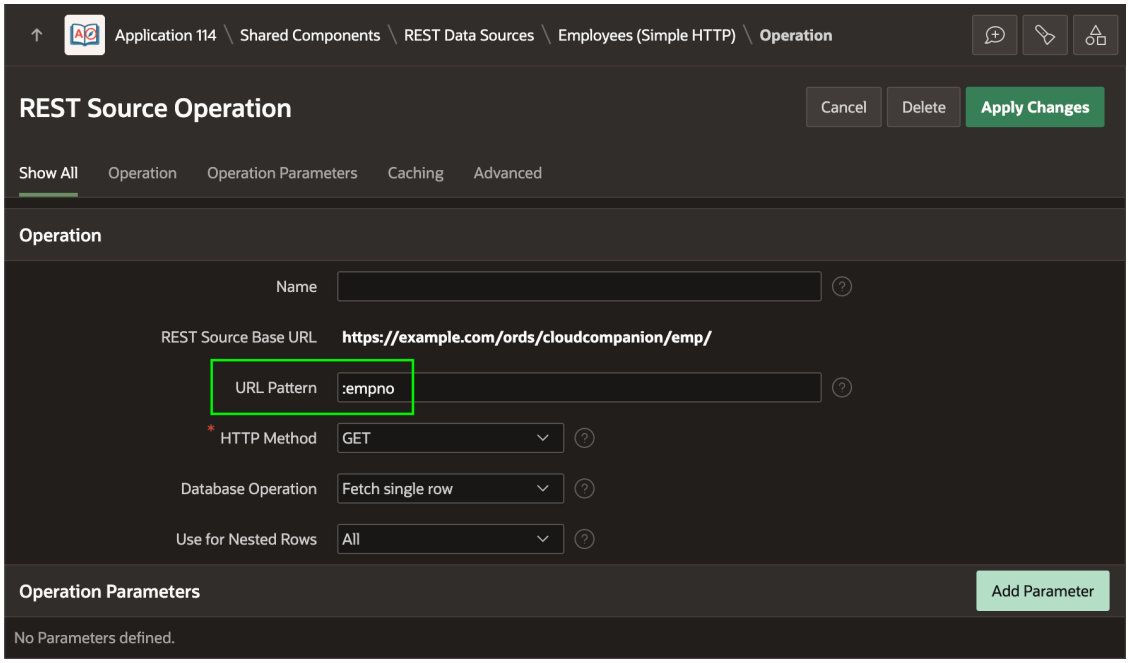
It's very useful that *Simple HTTP* REST Data Source operations can reference the case-sensitive *Selector* name of primary key columns as *virtual* parameters in the URL pattern. The lowercase `empno` name is the JSON selector for the primary key `EMPNO` column in this REST Data Source's Data Profile. So notice below the *URL Pattern* references the primary key value of the current row being modified using the `:empno` bind variable notation. The `PUT` and `DELETE` operations do the same.

Copyright © 2026, Oracle and/or its affiliates.

> **Tip**
>
> Using `{empno}` syntax is also allowed.

> **Caution**
>
> Do not define a URL pattern parameter for the primary key in the operation URL. Doing so overrides the implicit URL Pattern parameter, such as `empno`, from the primary key Data Profile column's JSON selector name.

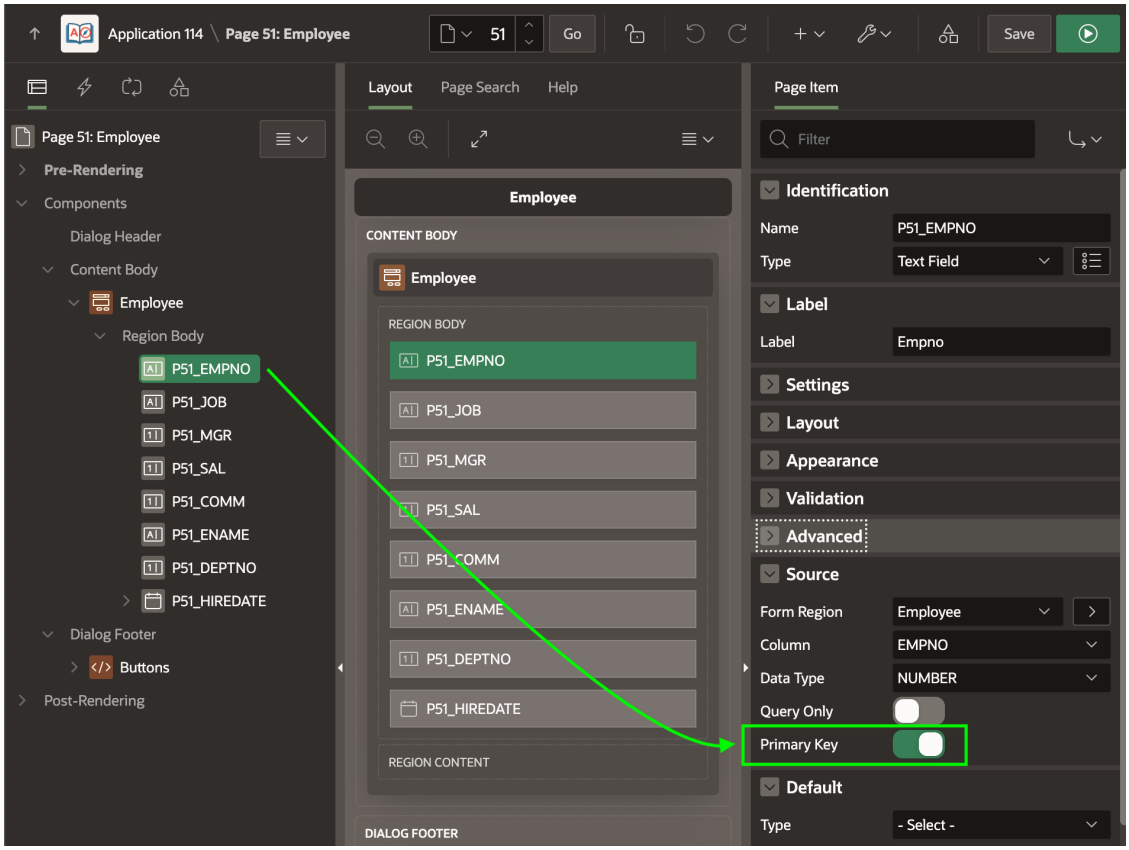
Figure 13-16 Fetch Single Row Operation to Retrieve a Single Row by Primary Key



As shown below, the `P51_EMPNO` page item in the *Employee* form region enables the *Primary Key* switch. Page Designer should infer the setting automatically from the Data Profile of the REST Data Source the Form region references. But it's always good to double-check. This ensures that the appropriate REST Data Source operation URL referencing the value of the `:empno` primary key data profile column selector as an *implicit* parameter will use the correct value.

Copyright © 2026, Oracle and/or its affiliates.

Figure 13-17 Ensuring the Primary Key Column is Enabled



At runtime, suppose you have left the default *Prevent Lost Updates* switch enabled setting on the *Form Automatic DML* page process that saves the form's changes on submit. If the value of `P51_EMPNO` is `7369` when the user updates or deletes a row, that page process uses the *Fetch single row* operation of the *Employees (Simple HTTP)* REST Data Source to retrieve the data for this employee using an HTTP `GET` request to the REST API. In the debug log you would see the following message showing the URL that substitutes the value `7369` for the `:empno` URL *implicit* URL pattern parameter:

```
making GET request to https://example.com/ords/cloudcompanion/emp/7369
```

13.3.6 Configuring a Request Body Template

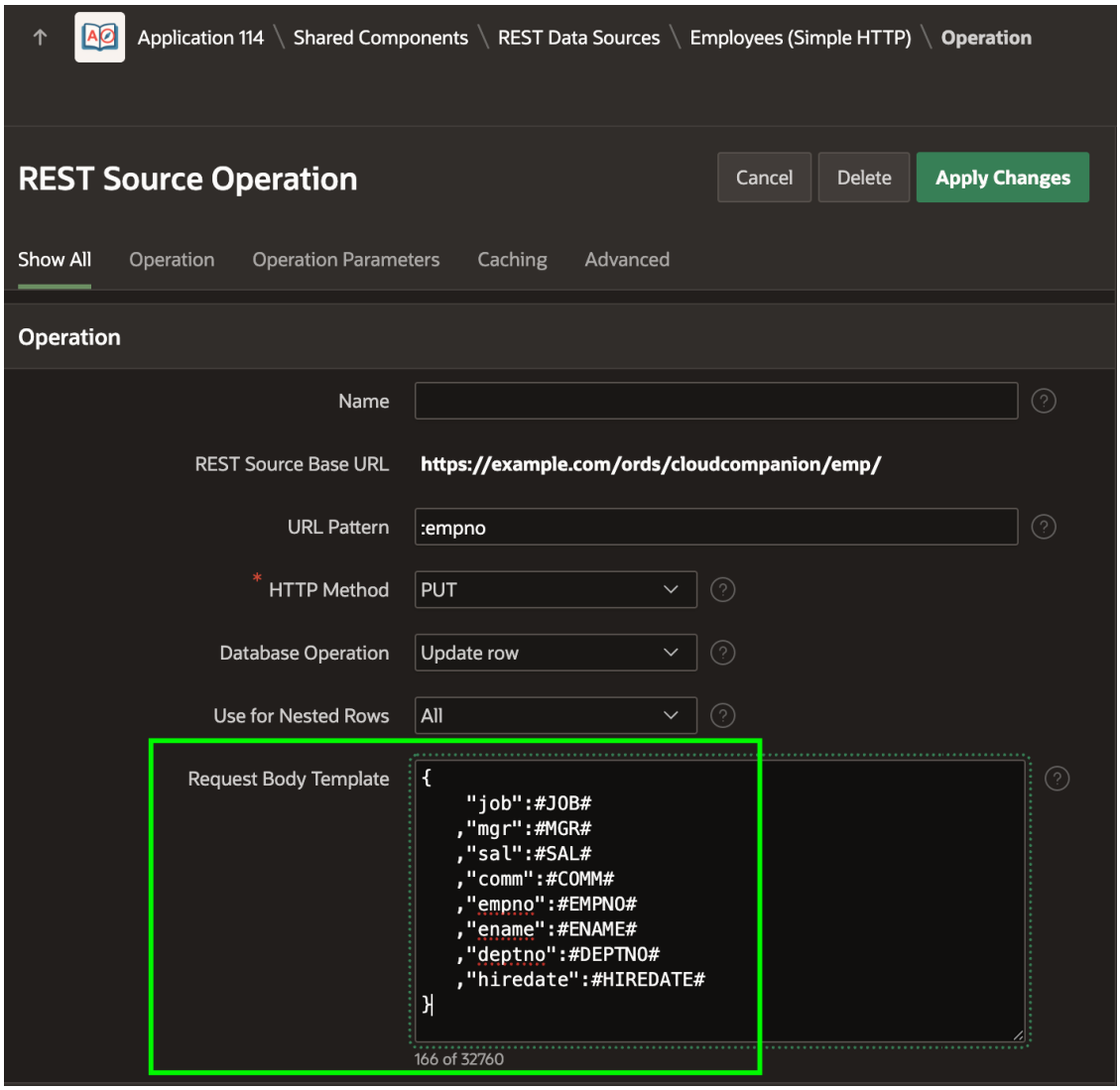
Your operations for *Insert row* and *Update row* need to send data to the REST API in the request body.

As shown below, you configure the contents of this payload using the operation's *Request Body Template*. It can contain a mix of JSON syntax and *`#NAME#`* substitutions. The names of region data source columns are available to reference automatically, as well as any parameters of type *Request Body* you might need to define. Notice the column name substitutions are *not* surrounded by double quotes. At runtime APEX replaces the substitutions with appropriately escaped values that contain the double quotes if needed. If the REST API you work with requires passing a null value using the JSON `null` keyword instead of empty double quotes, then reference the substitution like `#COMM!NULL#` using the `!NULL` modifier.

Copyright © 2026, Oracle and/or its affiliates.

The figure shows the Request Body Template for the Employees PUT operation. It's a JSON object template including a mix of JSON properties and data profile column substitutions.

Figure 13-18 Request Body Template in the PUT / Update Row Operation



To build the request body template dynamically, define a *Request Body* parameter, for example named PAYLOAD, and configure the *Request Body Template* like this:

```
#PAYLOAD!RAW#
```

The PAYLOAD parameter will appear in any context where you use the REST Data Source and you can assign a dynamic value to the parameter using a PL/SQL expression. The !RAW modifier asks APEX to treat the value of the parameter verbatim, avoiding the normal JSON escaping it performs when substituting parameter or data source column values.

Copyright © 2026, Oracle and/or its affiliates.

13.3.7 Supply and Return Values with Parameters

Just as PL/SQL procedures can define `IN`, `OUT`, and `IN/OUT` parameters, so can REST Data Sources. Define a parameter at the operation level if it's specific to that action, or at the data source level if it's relevant to *all* operations.

You can configure a default value for each one, mark it as required if relevant, and indicate if the value is static. Any parameter whose value is not static can be set on the page or List of Values where you use the REST Data Source so different usages can pass in appropriate values. In Page Designer, they appear beneath a *Parameters* node indented inside the region in the rendering tree. The table below describes the most common parameter types.

Table 13-2 Most Common Parameter Types

Type	Direction	Description
URL Pattern	`IN`	Use in URL Pattern as `:Name` or `{Name}`
URL Query String	`IN`	Included in URL query string as `Name=Val`
HTTP Header	`IN`, `OUT`, `IN/OUT`	Set and/or return HTTP header value
Request or Response Body	`IN`, `IN/OUT`	Use in Request Body Template as `#Name#`, `#Name!NULL#`, or `#Name!RAW#`
Request or Response Body	`OUT`, `IN/OUT`	Return entire response body
Data Profile Column	`OUT`	Return data profile column from 1-row response

For example, to send a `Content-Type` header with the fixed value `application/json`, use an *HTTP Header* type parameter with `IN` direction marked as static.

In contrast, for an HTTP header or query string parameter related to authentication and authorization, define an appropriate *Web Credential* of the appropriate type instead. Then reference that web credential in your REST Data Source. This ensures security-related values are obfuscated in debug logs and that the credential secret is securely stored and handled.

13.4 Invoking a REST Operation Declaratively

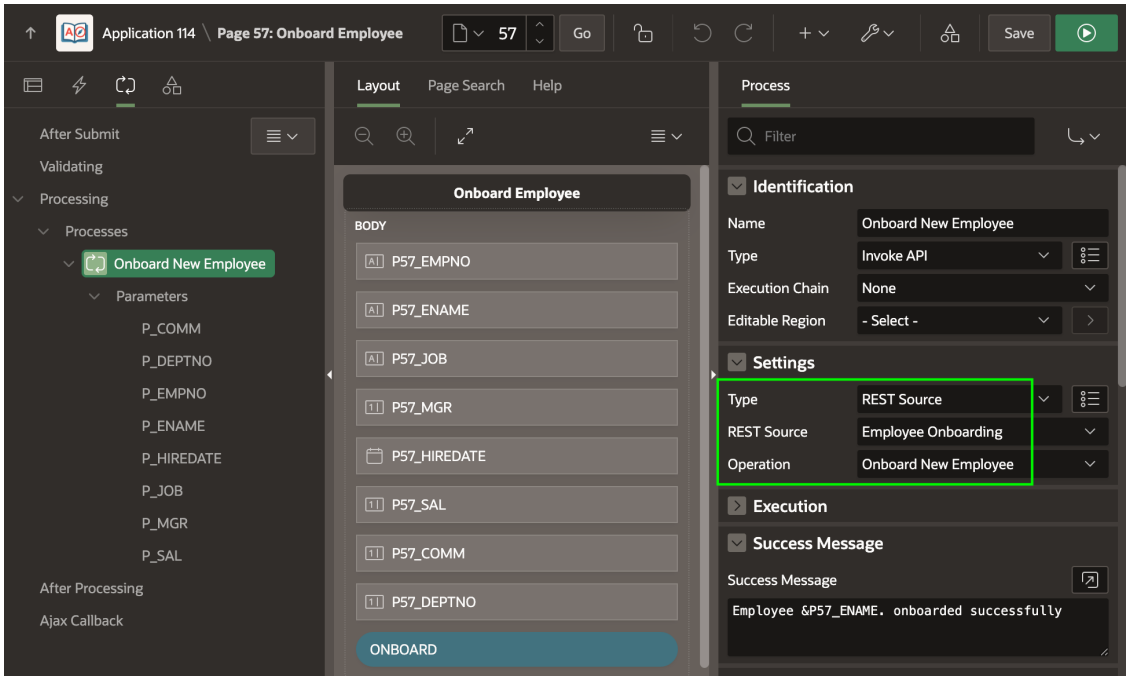
Use the *Invoke API* page process to call a REST Data Source operation with no code.

As shown below, set the API *Type* to *REST Source*, pick a REST Data Source, and choose an operation. Then configure parameter values to provide input values or map output values to page items. Your workflows can do the same with the *Invoke API* activity configuring parameters using workflow variables.

As shown below, the *Onboard New Employee* page process type is *Invoke API*. It is configured to invoke the *Onboard New Employee* operation of the *Employee Onboarding* REST Data Source.

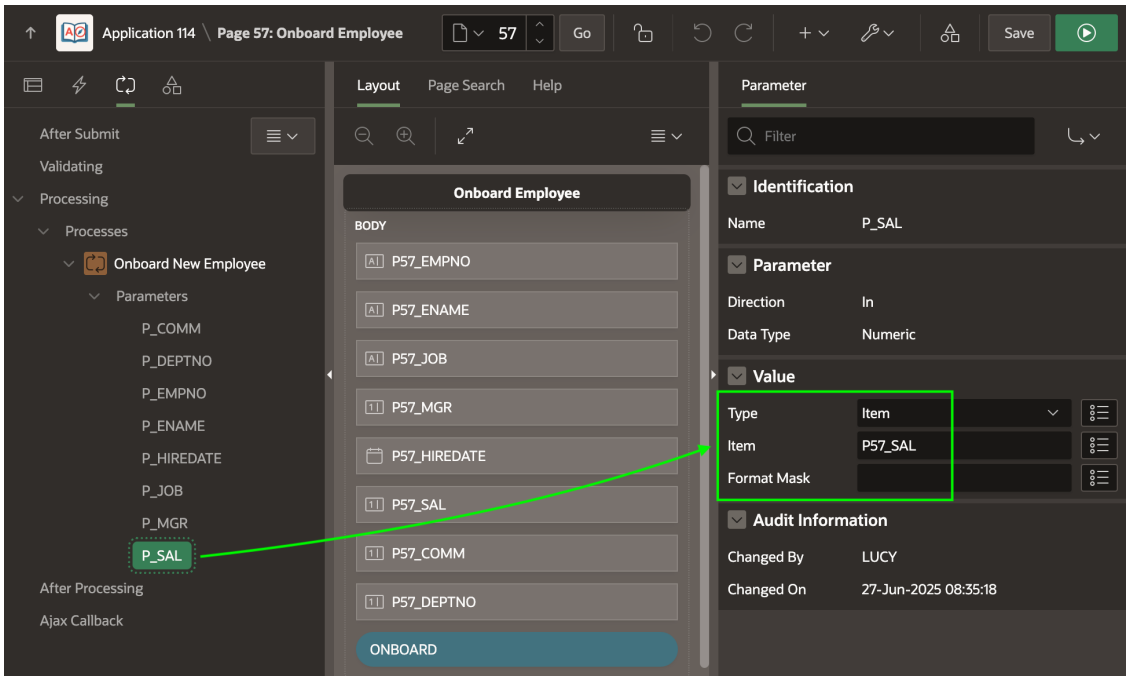
Copyright © 2026, Oracle and/or its affiliates.

Figure 13-19 Calling a REST Data Source Operation with Invoke API Page Process



Indented below the *Onboard New Employee* page process, notice the *Parameters* heading. Each updatable operation parameter appears there. As shown below, you can source the value of a parameter like `P_SAL` from a page item like `P57_SAL`.

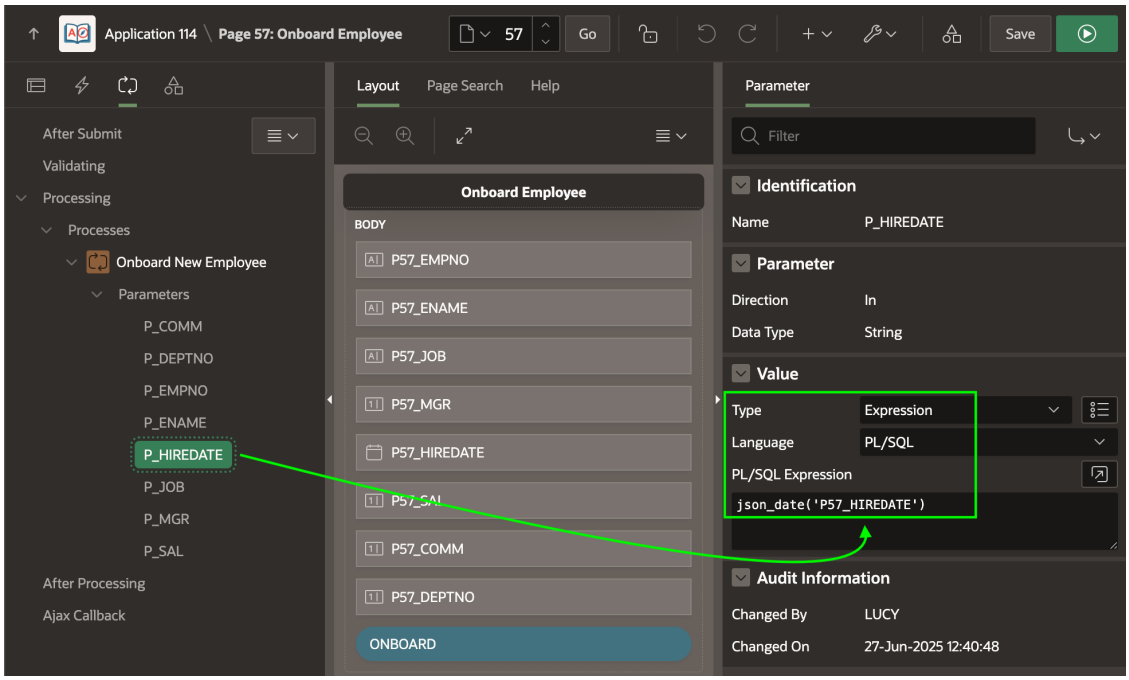
Figure 13-20 Configuring REST Data Source Parameter Using a Page Item



Copyright © 2026, Oracle and/or its affiliates.

When useful, as shown below you can also provide a parameter's value using an expression. For example, as shown below the value of the `P_HIREDATE` parameter is the result of calling an application-specific helper function `json_date()`. It accepts the name of a page item or workflow version variable and returns a date value for substitution into a JSON payload. Its result is the unquoted four characters `null` if the item's value is null or the double-quoted date in the ISO 8601 format. Many REST APIs expect their date values in this standard `2025-06-28T13:25:32Z` format.

Figure 13-21 Configuring Invoke API Page Process Parameters



For reference, the `json_date()` helper function looks like this:

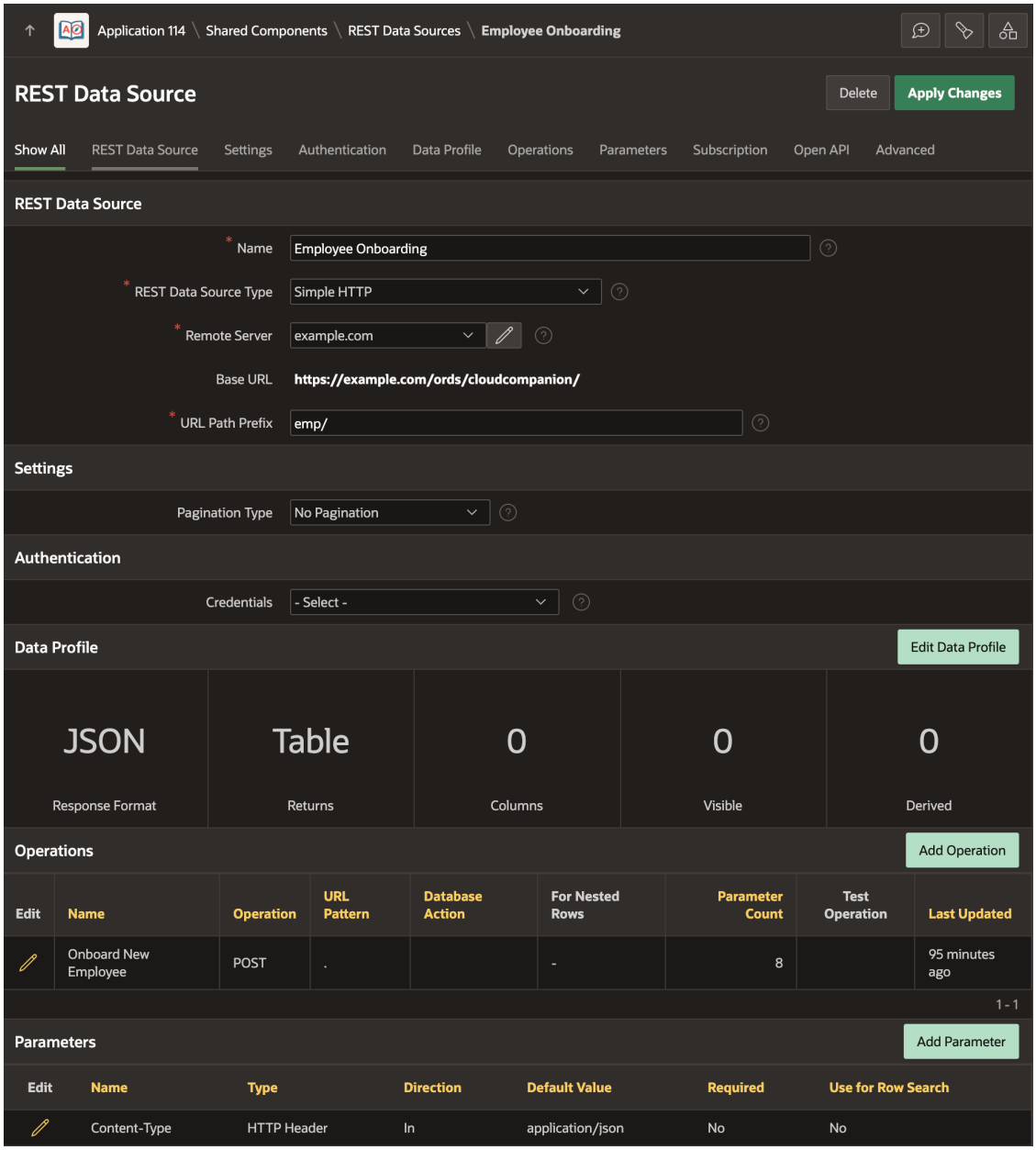
```
create or replace function json_date(
    p_item in varchar2)
    return    varchar2
is
    l_value varchar2(255) := apex_session_state.get_timestamp(p_item);
begin
    return case
            when l_value is null then 'null'
            else
                apex_string.format(
                    '"%s"',
                    to_char(
                        apex_session_state.get_timestamp(p_item),
                        apex_json.c_date_iso8601))
           end;
end;
```

As shown below, the *Employee Onboarding* REST Data Source has just the one *Onboard New Employee* operation the page invokes. If a REST API you're integrating with returns no response payload, or you have no need to reference values from it, then no Data Profile is

Copyright © 2026, Oracle and/or its affiliates.

needed. Notice that this REST Data Source has zero (0) Data Profile columns. It defines a single, static *HTTP Header* type parameter to send the `Content-Type` header with the value `application/json`.

Figure 13-22 REST Data Source with an Operation Used by Invoke API



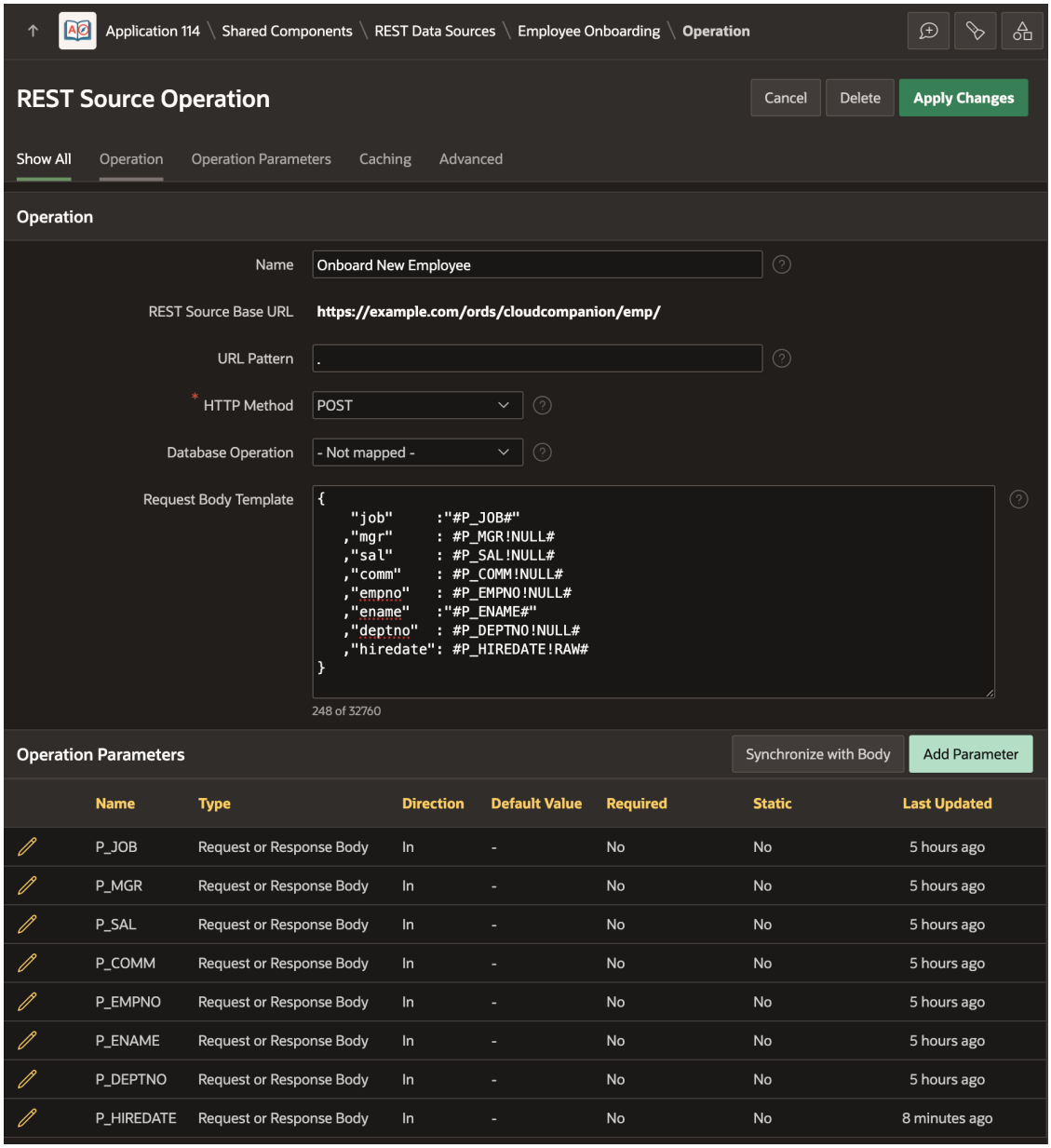
Studying the *Onboard New Employee* operation, notice two key differences. In contrast with operations the Form and Interactive Grid DML page processes use based on *Database Action* value, instead here:

- Each payload value is explicitly defined as a *Request Body* parameter, and
- Value substitutions do not automatically include double quotes.

Notice the operation parameters below corresponding to each value the *Request Body Template* requires.

Copyright © 2026, Oracle and/or its affiliates.

Figure 13-23 REST Data Source Operation for Invoke API



Notice how the `P_JOB` and `P_ENAME` parameter substitutions include explicit double quotes in the *Request Body Template*. The page supplying the parameter values computes `P_HIREDATE` using the application-specific `json_date()` function. That returns either the unquoted four characters `null` if the date value is null or a value like `"2025-06-29T00:00:00Z"` that is already double-quoted. The `!RAW` modifier on the `#P_HIREDATE!RAW#` substitution ensures that APEX does *not* perform further JSON escaping on the parameter value when building the payload at runtime.

```
{
    "job"      :"#P_JOB#"
   ,"mgr"      : #P_MGR!NULL#
   ,"sal"      : #P_SAL!NULL#
   ,"comm"     : #P_COMM!NULL#
```


Copyright © 2026, Oracle and/or its affiliates.

```
   ,"empno"    : #P_EMPNO!NULL#
   ,"ename"    :"#P_ENAME#"
   ,"deptno"   : #P_DEPTNO!NULL#
   ,"hiredate": #P_HIREDATE!RAW#
}
```

13.5 Using REST APIs in Code

Sometimes your applications need to work with REST APIs in code. Using REST Data Sources is the maximally-declarative approach. In contrast, the `MAKE_REST_REQUEST` function in the `APEX_WEB_SERVICE` package provides additional control but requires writing more code.

- Processing REST Source Rows in Code
  Use the `APEX_EXEC` package to process rows from REST Data Sources programmatically.
- Invoking a REST Operation Programmatically
  You can invoke a REST Data Source operation programmatically, treating it like a function call instead of a row set.
- Calling REST APIs Without a Data Source
  Use the `APEX_WEB_SERVICE` package to call a REST API that requires a binary payload or returns a binary response. Binary payloads require this approach. You can also use the package for general REST calls, but then you must code many tasks that a REST Data Source handles declaratively.

13.5.1 Processing REST Source Rows in Code

Use the `APEX_EXEC` package to process rows from REST Data Sources programmatically.

If the REST Data Source defines operations supporting all *Database Action* values, then you can both query *and* modify data. Using appropriate package functions, you open a "context" for working with rows. When querying rows, you iterate through the results in a loop. When modifying rows you add rows to the context and then execute the DML operations for those rows. When done, you close the context to release its resources.

- Querying REST Source Rows in Code
  Use `OPEN_REST_SOURCE_QUERY` in the `APEX_EXEC` package to query rows from a REST Data Source. It automatically executes the operation with the *Fetch rows* Database Action.
- Modifying REST Source Rows in Code
  Use `OPEN_REST_SOURCE_DML_CONTEXT` in the `APEX_EXEC` package to modify a row from a REST Data Source.

13.5.1.1 Querying REST Source Rows in Code

Use `OPEN_REST_SOURCE_QUERY` in the `APEX_EXEC` package to query rows from a REST Data Source. It automatically executes the operation with the *Fetch rows* Database Action.

To define the data source columns whose values you want to retrieve, use a variable of type `APEX_EXEC.T_COLUMNS` and call `ADD_COLUMN` to add one or more column names to it. Then pass it as the `p_columns` argument value.

To set operation parameters, use a variable of type `APEX_EXEC.T_PARAMETERS`, call `ADD_PARAMETER` to add one or more parameters to the list, then pass it as the `p_parameters` value. Use a `while` loop with `NEXT_ROW` as the loop condition to iterate through the results. When done, or when an exception occurs, call `CLOSE` to clean up context resources.

Copyright © 2026, Oracle and/or its affiliates.

A simple example that retrieves and iterates over all rows from the *Employees (Simple HTTP)* REST Data Source is below. Notice the code references the REST Data Source using its Static ID `employees_simple_http`.

```
declare
    l_params   apex_exec.t_parameters;
    l_cols     apex_exec.t_columns;
    l_ctx      apex_exec.t_context;
    l_empno    number;
    l_ename    varchar2(255);
    l_hiredate date;
    l_empinfo  apex_t_varchar2;
begin
    apex_exec.add_column(
        p_columns     => l_cols,
        p_column_name => 'EMPNO');
    apex_exec.add_column(
        p_columns     => l_cols,
        p_column_name => 'ENAME');
    apex_exec.add_column(
        p_columns     => l_cols,
        p_column_name => 'HIREDATE');
    apex_exec.add_parameter(
        p_parameters => l_params,
        p_name       => 'MY_PARAM',
        p_value      => 1234);
    l_ctx := apex_exec.open_rest_source_query(
                p_static_id  => 'employees_simple_http',
                p_columns    => l_cols,
                p_parameters => l_params);
    while apex_exec.next_row(l_ctx) loop
        l_empno    := apex_exec.get_number(l_ctx,'EMPNO');
        l_ename    := apex_exec.get_varchar2(l_ctx,'ENAME');
        l_hiredate := apex_exec.get_date(l_ctx,'HIREDATE');
        -- Do something here with current row values
    end loop;
    apex_exec.close(l_ctx);
exception
    when others then
        apex_exec.close(l_ctx);
        raise;
end;
```

`OPEN_REST_SOURCE_QUERY` accepts other optional arguments that let you:

- Order retrieved rows by passing a `APEX_EXEC.T_ORDER_BYS` list to `p_order_bys`
- Filter rows by passing a `APEX_EXEC.T_FILTERS` list to `p_filters`
- Retrieve only a `p_max_rows`-sized page of rows starting with row number `p_first_row`

For more information, see OPEN_REST_SOURCE_QUERY Function in *Oracle APEX API Reference*.

Be aware that by default *Simple HTTP* REST Data Sources perform ordering and filtering in the APEX database after retrieving all rows from the REST API. Other convention-based REST Data Source types automatically delegate filtering, sorting, and pagination to the REST API for

Copyright © 2026, Oracle and/or its affiliates.

an optimized data exchange. APEX offers these smarter REST Data Source types for Fusion Applications, Oracle REST Data Services, and OData.

13.5.1.2 Modifying REST Source Rows in Code

Use `OPEN_REST_SOURCE_DML_CONTEXT` in the `APEX_EXEC` package to modify a row from a REST Data Source.

To define the data source columns whose values you want to work with, use a variable of type `APEX_EXEC.T_COLUMNS` and call `ADD_COLUMN` to add one or more column names to it. Make sure to pass the `p_data_type` parameter of each column, too, as well as passing `true` for `p_is_primary_key` on at least one of the columns. Then pass this column list as the `p_columns` parameter value in your call to open the DML context.

To supply parameter values, use a variable of type `APEX_EXEC.T_PARAMETERS`, call `ADD_PARAMETER` to add one or more parameters to it, then pass the parameters variable as the value of `p_parameters`.

Once you create a "context" to work with rows, call the `ADD_DML_ROW` procedure to add a new current row to the context, passing a constant of type `APEX_EXEC.T_DML_OPERATION` as the value of the `p_operation` parameter. To set the column values on the current row, call `SET_VALUE`. To process the row you added to the context, call `EXECUTE_DML`. When done, or when an exception occurs, call `CLOSE` to clean up context resources.

For more information, see OPEN_REST_SOURCE_DML_CONTEXT Function in *Oracle APEX API Reference*.

- Inserting a REST Source Row in Code
  Review an example of inserting data using a REST Data Source.
- Updating a REST Source Row in Code
  Review an example of updating a row using a REST Data Source.
- Deleting a REST Source Row in Code
  Review an example of deleting a row from a REST Data Source.

13.5.1.2.1 Inserting a REST Source Row in Code

Review an example of inserting data using a REST Data Source.

A simple example that inserts two employees using the *Employees (Simple HTTP)* REST Data Source is below. Notice the code references the REST Data Source using its Static ID `employees_simple_http`.

```
declare
    l_cols     apex_exec.t_columns;
    l_dml_ctx  apex_exec.t_context;
begin
    apex_exec.add_column(
        p_columns        => l_cols,
        p_column_name    => 'EMPNO',
        p_data_type      => apex_exec.c_data_type_number,
        p_is_primary_key => true);
    apex_exec.add_column(
        p_columns        => l_cols,
        p_column_name    => 'MGR',
        p_data_type      => apex_exec.c_data_type_number);
    apex_exec.add_column(
```


Copyright © 2026, Oracle and/or its affiliates.

```
            p_columns        => l_cols,
            p_column_name    => 'SAL',
            p_data_type      => apex_exec.c_data_type_number);
        apex_exec.add_column(
            p_columns        => l_cols,
            p_column_name    => 'COMM',
            p_data_type      => apex_exec.c_data_type_number);
        apex_exec.add_column(
            p_columns        => l_cols,
            p_column_name    => 'DEPTNO',
            p_data_type      => apex_exec.c_data_type_number);
        apex_exec.add_column(
            p_columns     => l_cols,
            p_column_name => 'ENAME',
            p_data_type   => apex_exec.c_data_type_varchar2);
        apex_exec.add_column(
            p_columns     => l_cols,
            p_column_name => 'JOB',
            p_data_type   => apex_exec.c_data_type_varchar2);
        apex_exec.add_column(
            p_columns     => l_cols,
            p_column_name => 'HIREDATE',
            p_data_type   => apex_exec.c_data_type_date);
        l_dml_ctx := apex_exec.open_rest_source_dml_context(
                        p_static_id  => 'employees_simple_http',
                        p_columns    => l_cols);
        -- Insert LUCY
        apex_exec.add_dml_row(l_dml_ctx,apex_exec.c_dml_operation_insert);
        apex_exec.set_value(l_dml_ctx,'EMPNO',1234);
        apex_exec.set_value(l_dml_ctx,'ENAME','LUCY');
        apex_exec.set_value(l_dml_ctx,'HIREDATE',date '2025-06-27');
        -- Insert LEO
        apex_exec.add_dml_row(l_dml_ctx,apex_exec.c_dml_operation_insert);
        apex_exec.set_value(l_dml_ctx,'EMPNO',1235);
        apex_exec.set_value(l_dml_ctx,'ENAME','LEO');
        apex_exec.set_value(l_dml_ctx,'HIREDATE',date '2025-05-14');
        -- Process the rows in the DML context
        apex_exec.execute_dml(l_dml_ctx);
        apex_exec.close(l_dml_ctx);
    exception
        when others then
            apex_exec.close(l_dml_ctx);
            raise;
    end;
```

13.5.1.2.2 Updating a REST Source Row in Code

Review an example of updating a row using a REST Data Source.

A simple example that updates the `HIREDATE` of one employee using the *Employees (Simple HTTP)* REST Data Source is below. Notice the code references the REST Data Source using its Static ID `employees_simple_http` for the DML context. To support lost update protection, the code needs to fetch the existing *Employee* row first before proceeding to update the necessary fields in it. To perform this fetch by primary key, the code uses a *second* REST Data Source with Static ID `employees_simple_http_by_pk` to retrieve the single row to be updated.

Copyright © 2026, Oracle and/or its affiliates.

After using this second REST Data Source to retrieve the existing *Employees* row using a query context, it:

- Opens the DML context for `employees_simple_http`
- Adds a DML row to the context with the update operation
- Sets the row version checksum on the DML row using the existing row to compute it
- Calls the `SET_VALUES` procedure to copy all the existing row values into the DML row

Then it proceeds to update any values that need to be changed. Finally, it calls `EXECUTE_DML` to process the DML context. It closes both query and DML contexts to release resources when completed or in case of an exception.

```
declare
    l_params   apex_exec.t_parameters;
    l_cols     apex_exec.t_columns;
    l_ctx      apex_exec.t_context;
    l_dml_ctx  apex_exec.t_context;
begin
    apex_exec.add_column(
        p_columns        => l_cols,
        p_column_name    => 'EMPNO',
        p_data_type      => apex_exec.c_data_type_number,
        p_is_primary_key => true);
    apex_exec.add_column(
        p_columns        => l_cols,
        p_column_name    => 'MGR',
        p_data_type      => apex_exec.c_data_type_number);
    apex_exec.add_column(
        p_columns        => l_cols,
        p_column_name    => 'SAL',
        p_data_type      => apex_exec.c_data_type_number);
    apex_exec.add_column(
        p_columns        => l_cols,
        p_column_name    => 'COMM',
        p_data_type      => apex_exec.c_data_type_number);
    apex_exec.add_column(
        p_columns        => l_cols,
        p_column_name    => 'DEPTNO',
        p_data_type      => apex_exec.c_data_type_number);
    apex_exec.add_column(
        p_columns     => l_cols,
        p_column_name => 'ENAME',
        p_data_type   => apex_exec.c_data_type_varchar2);
    apex_exec.add_column(
        p_columns     => l_cols,
        p_column_name => 'JOB',
        p_data_type   => apex_exec.c_data_type_varchar2);
    apex_exec.add_column(
        p_columns     => l_cols,
        p_column_name => 'HIREDATE',
        p_data_type   => apex_exec.c_data_type_date);
    apex_exec.add_parameter(
        p_parameters => l_params,
        p_name       => 'empno_update',
        p_value      => 7788);
```

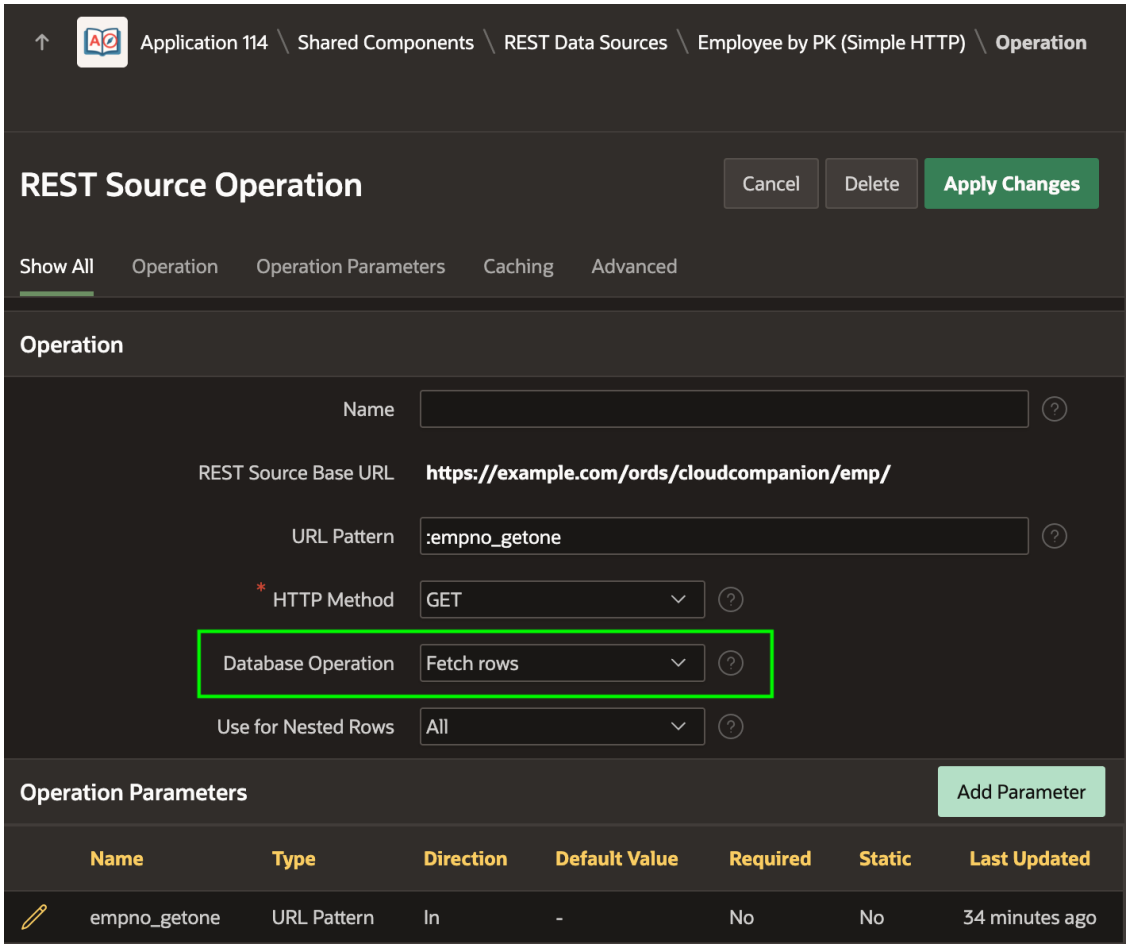

Copyright © 2026, Oracle and/or its affiliates.

```
    -- Add parameter used by employee_by_pk_simple_http
    -- Fetch rows operation
    apex_exec.add_parameter(
        p_parameters => l_params,
        p_name       => 'empno_getone',
        p_value      => 7788);
    l_ctx := apex_exec.open_rest_source_query(
                p_static_id  => 'employee_by_pk_simple_http',
                p_columns    => l_cols,
                p_parameters => l_params);
    if apex_exec.next_row(l_ctx) then
        l_dml_ctx := apex_exec.open_rest_source_dml_context(
                        p_static_id            => 'employees_simple_http',
                        p_columns              => l_cols,
                        p_lost_update_detection =>
apex_exec.c_lost_update_implicit);
        -- Update SCOTT
        apex_exec.add_dml_row(l_dml_ctx,apex_exec.c_dml_operation_update);
        -- Checksum used for lost-update protection
        apex_exec.set_row_version_checksum(
            p_context   => l_dml_ctx,
            p_checksum  => apex_exec.get_row_version_checksum( p_context =>
l_ctx ));
        apex_exec.set_values(
            p_context         => l_dml_ctx,
            p_source_context  => l_ctx );
        -- Update the hiredate of the SCOTT employee
        apex_exec.set_value(l_dml_ctx,'HIREDATE',date'2024-07-14');
        -- Process the DML context
        apex_exec.execute_dml(l_dml_ctx);
    end if;
    apex_exec.close(l_ctx);
    apex_exec.close(l_dml_ctx);
exception
    when others then
        apex_exec.close(l_ctx);
        apex_exec.close(l_dml_ctx);
        raise;
end;
```

The *Employee by PK (Simple HTTP)* REST Data Source, with Static ID `employee_by_pk_simple_http` has just a single `GET` operation with *Database Action* of *Fetch rows*. As shown below, it's configured to use the URL Pattern parameter `empno_getone` in the URL to retrieve a single employee.

Copyright © 2026, Oracle and/or its affiliates.

Figure 13-24 Fetch Rows GET Operation to Retrieve One Employee by PK



At runtime, by setting the `empno_getone` parameter to a value like `7788` as the example above is doing, if debug tracing is enabled you would see the APEX engine requests the URL:

```
making GET request to https://example.com/ords/cloudcompanion/emp/7788
```

When requesting a single employee from the *Employees* service, it responds with a JSON payload like the following that contains just a single object instead of an `items` array with one or more objects in it:

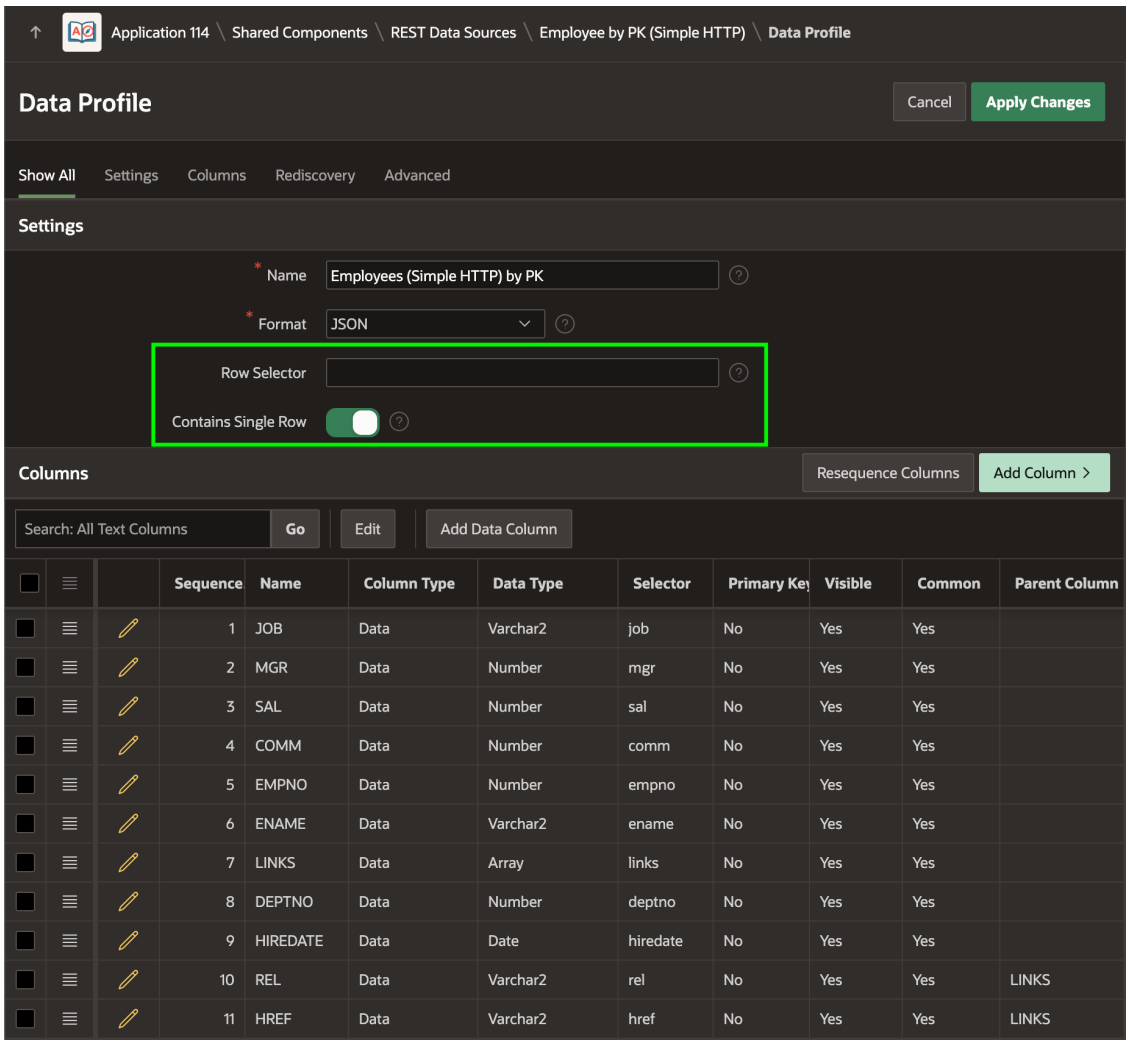
```
{
    "empno": 7788,
    "ename": "SCOTT",
    "job": "ANALYST",
    "mgr": 7566,
    "hiredate": "1982-12-09T00:00:00Z",
    "sal": 3000,
    "comm": null,
    "deptno": 20,
    "links": [
        {
            "rel": "self",
```


Copyright © 2026, Oracle and/or its affiliates.

```
            "href": "https://example.com/ords/cloudcompanion/emp/7788"
        },
        ▒
    ]
}
```

This means the *Employee by PK (Simple HTTP)* REST Data Source needs a Data Profile that caters for this different, single-row response. As shown below, the Data Profile for this REST Data Source omits the *Row Selector* and enables the *Contains Single Row* switch.

Figure 13-25 Data Profile for Employee by PK (Simple HTTP) REST Data Source



13.5.1.2.3 Deleting a REST Source Row in Code

Review an example of deleting a row from a REST Data Source.

A simple example that deletes one employee from the *Employees (Simple HTTP)* REST Data Source is below. Notice the code references the REST Data Source using its Static ID `employees_simple_http` and that it only needs to define the primary key column since the

Copyright © 2026, Oracle and/or its affiliates.

corresponding operation for the *Delete row* database action has no payload containing attribute values.

```
declare
    l_cols     apex_exec.t_columns;
    l_dml_ctx  apex_exec.t_context;
begin
    apex_exec.add_column(
        p_columns        => l_cols,
        p_column_name    => 'EMPNO',
        p_data_type      => apex_exec.c_data_type_number,
        p_is_primary_key => true);
    l_dml_ctx := apex_exec.open_rest_source_dml_context(
                    p_static_id  => 'employees_simple_http',
                    p_columns    => l_cols);
    -- Delete SMITH
    apex_exec.add_dml_row(l_dml_ctx,apex_exec.c_dml_operation_delete);
    apex_exec.set_value(l_dml_ctx,'EMPNO',7369);
    -- Process the DML context
    apex_exec.execute_dml(l_dml_ctx);
    apex_exec.close(l_dml_ctx);
exception
    when others then
        apex_exec.close(l_dml_ctx);
        raise;
end;
```

13.5.2 Invoking a REST Operation Programmatically

You can invoke a REST Data Source operation programmatically, treating it like a function call instead of a row set.

Consider a company selling books that offers a REST API at the following endpoint URL:

```
https://example.com/ords/cloudcompanion/orders/singleBookCreate
```

It lets a caller `POST` a JSON payload like the following to order some quantity of a single book:

```
{
   "isbn": "978-1565926912",
   "quantity": 1
}
```

The API responds with an order confirmation payload containing:

```
{
    "orderNumber": 12345,
    "estimatedDelivery": "2025-07-06T00:00:00"
}
```

After configuring a REST Data Source with appropriate Data Profile columns and a `POST` operation having the required *Request Body Template*, you invoke it programmatically using the `EXECUTE_REST_SOURCE` procedure in the `APEX_EXEC` package.

Copyright © 2026, Oracle and/or its affiliates.

- Configuring REST Source for Invocation
  When a REST API has no GET method or OpenAPI description to automatically discover, you can define the operations and data profile columns manually.
- Invoking REST Operation in Code
  Use EXECUTE_REST_SOURCE in the APEX_EXEC package to invoke a REST operation in code.

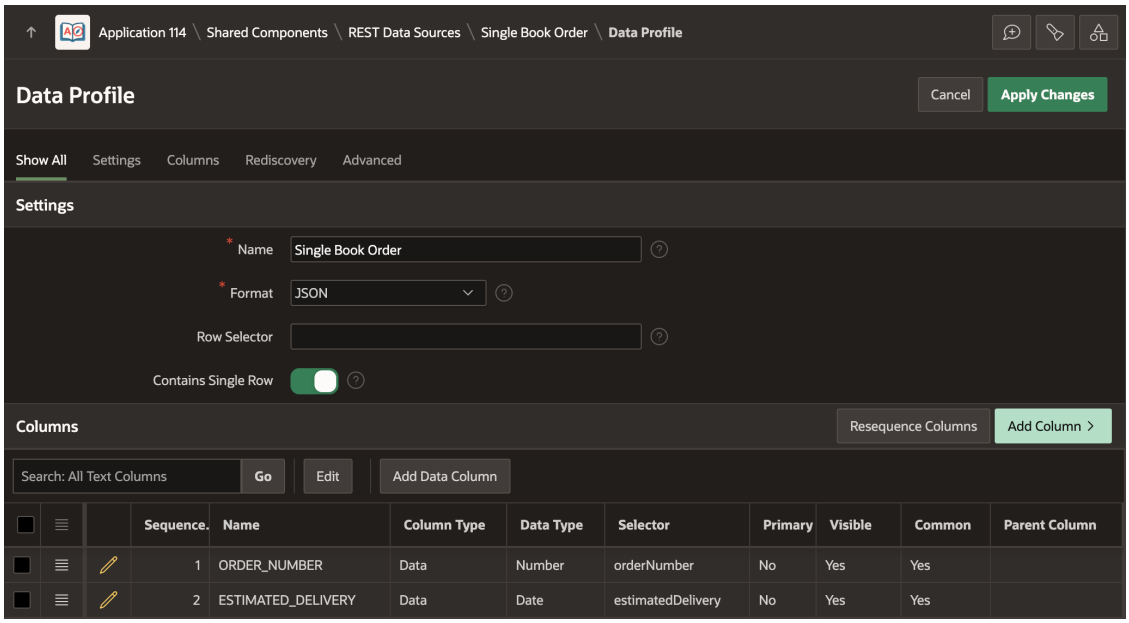
13.5.2.1 Configuring REST Source for Invocation

When a REST API has no GET method or OpenAPI description to automatically discover, you can define the operations and data profile columns manually.

After entering the endpoint URL in the wizard, on a following wizard step clicking the *(Create REST Source Manually)* button creates the data source you can adjust further in the edit pages.

When you create a REST Data Source manually, the wizard creates *placeholder* operations and data profile columns. Start by adjusting the Data Profile. You can delete the third placeholder data profile column, and edit the two existing ones to adjust the column name, data type, and JSON Selector as appropriate for the current API. As shown below, to access the two values in the response payload the Data Profiles columns are updated to reflect a numeric ORDER_NUMBER column with selector orderNumber and a date column ESTIMATED_DELIVERY.

Figure 13-26 Data Profile for Single Book Create REST API Response Payload

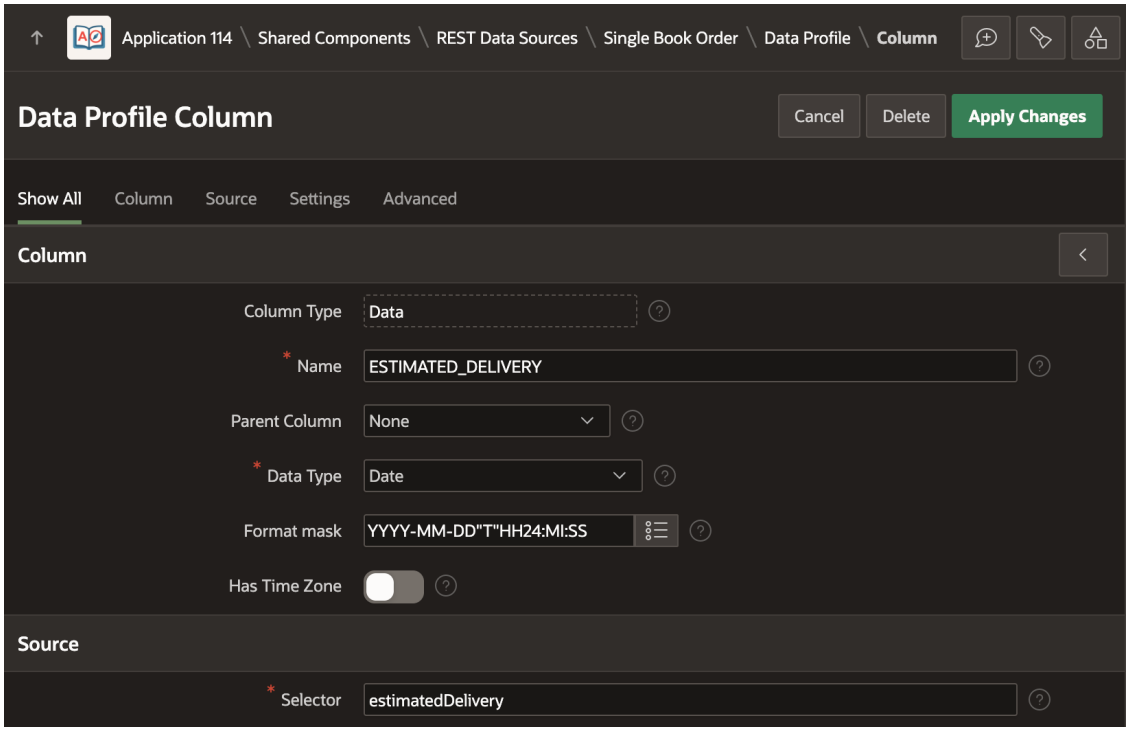


In the response payload example above, notice this particular service returns the date value in the ISO 8601 format without the "Zulu" (UTC) timezone letter Z at the end: 2025-07-06T00:00:00 So as shown below, the format mask used for the ESTIMATED_DELIVERY data profile column is:

```
YYYY-MM-DD"T"HH24:MI:SS
```


Copyright © 2026, Oracle and/or its affiliates.

Figure 13-27 Data Profile Column Configuration for Estimated Delivery Date



Next focus your attention on the operations. Since this service only requires a `POST` operation, you can delete the `GET`, `PUT`, and `DELETE` operations. This leaves a `POST` operation you can adjust as shown below to have the:

- Name *Single Book Create*
- Static ID `single_book_create`
- Appropriate *Request Body Template*

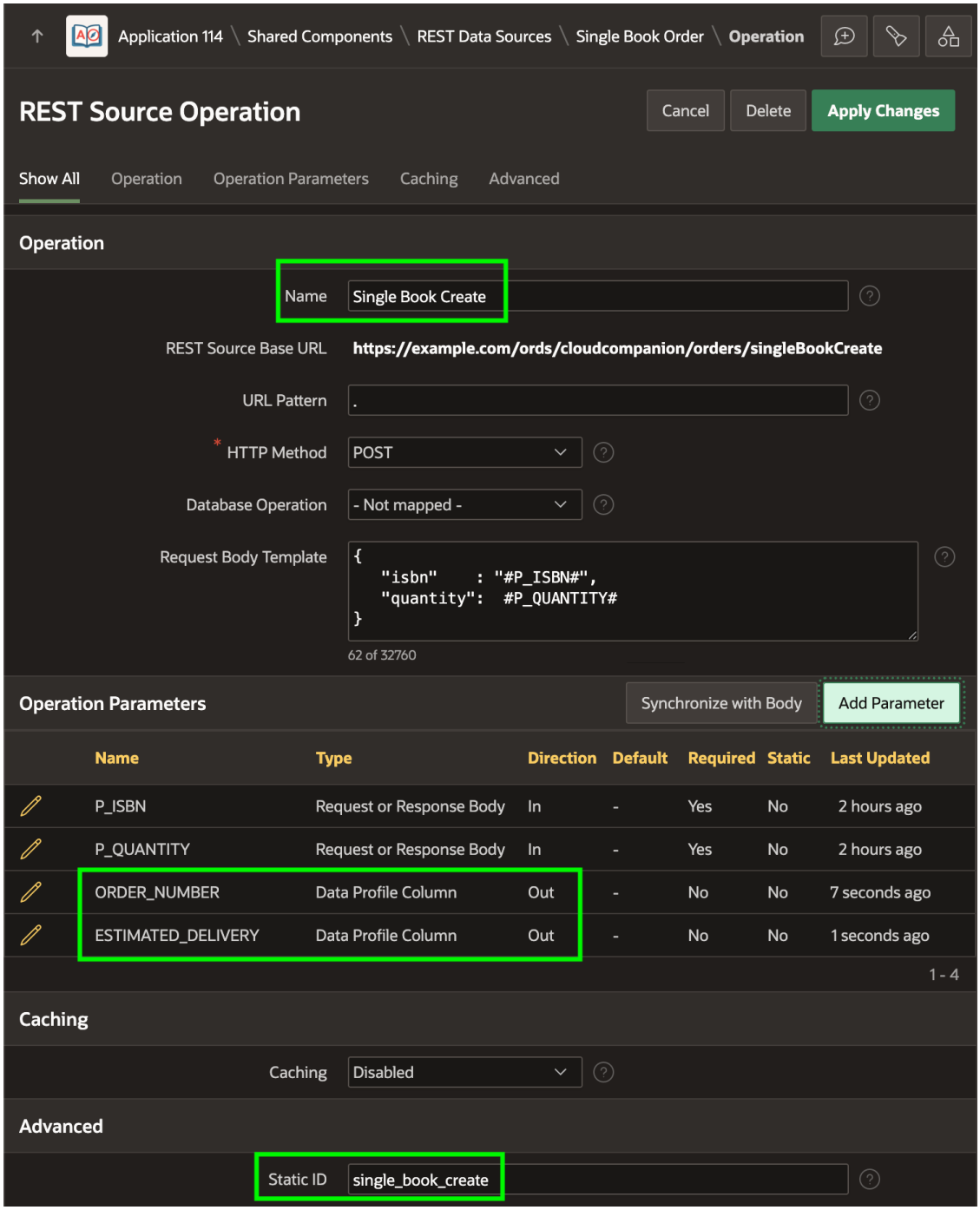
Notice the operation defines two required *Request Body* parameters `P_ISBN` and `P_QUANTITY` with appropriate data type and references them in the *Request Body Template*. When you invoke an operation using *Invoke API* or programmatically, include the double quotes in the payload template around property value substitutions that need them. For example, the `#P_ISBN#` substitution does this:

```
{
   "isbn"    : "#P_ISBN#",
   "quantity":  #P_QUANTITY#
}
```

You can also include two *Data Profile Column* type parameters to access the API response payload values as out parameters.

Copyright © 2026, Oracle and/or its affiliates.

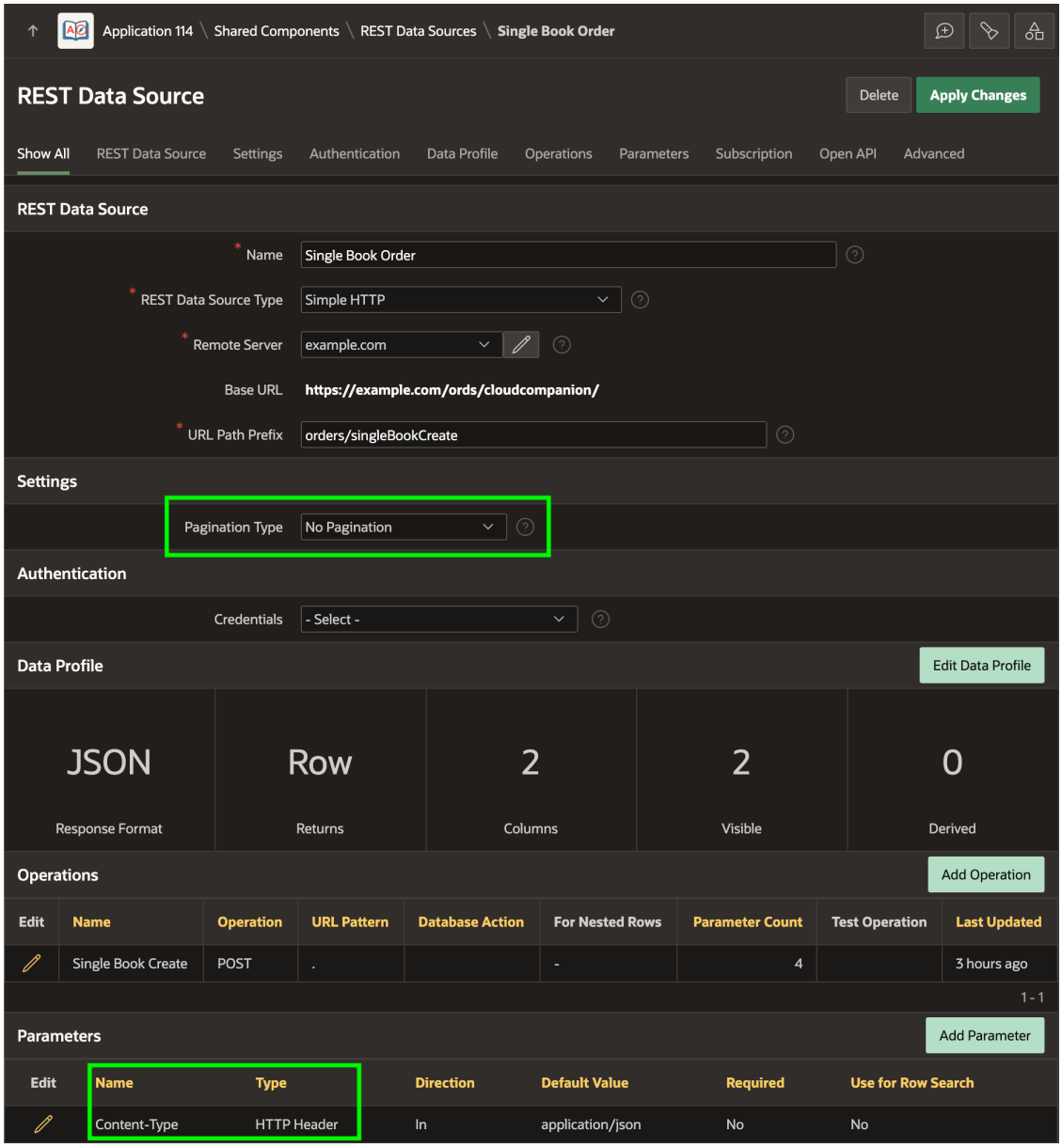
Figure 13-28 POST Operation for Single Book Create REST API



After configuring Data Profile columns and `POST` operation, the *Single Book Order* REST Data Source appears below. Some REST APIs expect the `Content-Type` header with the value `application/json` so you can add an appropriate static *Header*-type parameter to send when needed. Since the service returns a single JSON object response instead of an array, notice as well that *Pagination Type* is set to *No Pagination*.

Copyright © 2026, Oracle and/or its affiliates.

Figure 13-29 Resulting REST Data Source for Simple Book Order



13.5.2.2 Invoking REST Operation in Code

Use `EXECUTE_REST_SOURCE` in the `APEX_EXEC` package to invoke a REST operation in code.

To supply parameter values, use a variable of type `APEX_EXEC.T_PARAMETERS`, call `ADD_PARAMETER` to add one or more parameters to it, then pass the parameters variable as the value of `p_parameters`. After invoking the REST Data Source operation, use `GET_PARAMETER_VARCHAR2` or `GET_PARAMETER_CLOB` to retrieve the out parameters by name.

Copyright © 2026, Oracle and/or its affiliates.

A simple example of invoking the *Simple Book Create* operation of the *Single Book Order* REST Data Source appears below. Notice it uses the Static ID of the data source and the operation.

```
declare
    l_params             apex_exec.t_parameters;
    l_new_order_number   number;
    l_estimated_delivery date;
begin
    -- Set inbound parameter values
    apex_exec.add_parameter(
        p_parameters => l_params,
        p_name       => 'P_ISBN',
        p_value      => '978-1565926912');
    apex_exec.add_parameter(
        p_parameters => l_params,
        p_name       => 'P_QUANTITY',
        p_value      => 1);
    -- Invoke using static ids of data source and operation
    apex_exec.execute_rest_source(
        p_static_id           => 'single_book_order',
        p_operation_static_id => 'single_book_create',
        p_parameters          => l_params);
    -- Reference the values from the response as out parameters
    l_new_order_number       := apex_exec.get_parameter_varchar2(
                                    l_params,
                                    'ORDER_NUMBER');
    l_estimated_delivery := to_date(apex_exec.get_parameter_varchar2(
                                     l_params,
                                     'ESTIMATED_DELIVERY'));
    :P55_SUCCESS_MESSAGE := apex_string.format(
                                'New Order %s should deliver %s',
                                l_new_order_number,
                                to_char(l_estimated_delivery,
                                        'DD-MON-YYYY'));
end;
```

13.5.3 Calling REST APIs Without a Data Source

Use the `APEX_WEB_SERVICE` package to call a REST API that requires a binary payload or returns a binary response. Binary payloads require this approach. You can also use the package for general REST calls, but then you must code many tasks that a REST Data Source handles declaratively.

For more information, see MAKE_REST_REQUEST Function in *Oracle APEX API Reference*.

> **Tip**
>
> To pass a binary request payload, use `p_body_blob` instead of `p_body`. To receive a binary response, use `MAKE_REST_REQUEST_B` instead of `MAKE_REST_REQUEST`.

Copyright © 2026, Oracle and/or its affiliates.

For easier comparison, the example below calls the same *Single Book Order* endpoint without using a REST Data Source.

```
declare
    c_endpoint_url constant varchar2(200) :=
       'https://example.com/ords/cloudcompanion/orders/singleBookCreate';
    l_new_order_number      number;
    l_estimated_delivery    date;
    l_request               json_object_t;
    l_response_payload      clob;
    l_response              json_object_t;
begin
    -- Build the request body JSON
    l_request := json_object_t();
    l_request.put('isbn','978-1565926912');
    l_request.put('quantity',1);
    -- POST the JSON request body to the REST API endpoint URL
    l_response_payload :=
        apex_web_service.make_rest_request(
            p_http_method          => 'POST',
            p_url                  => c_endpoint_url,
            p_body                 => l_request.to_clob,
            p_credential_static_id => null /* Non-null for auth case */,
            p_token_url            => null /* Non-null for OAuth */);
    -- If HTTP status code indicates a success and response is not empty
    if apex_web_service.g_status_code = 200
       and l_response_payload is not null
       and dbms_lob.getlength(l_response_payload) > 0
    then
        -- Work with the response body as JSON
        l_response := json_object_t(l_response_payload);
        -- Reference the values from the response JSON
        l_new_order_number   := l_response.get_number('orderNumber');
        l_estimated_delivery := l_response.get_date('estimatedDelivery');
        :P55_SUCCESS_MESSAGE := apex_string.format(
                                    'New Order %s should deliver %s',
                                    l_new_order_number,
                                    to_char(l_estimated_delivery,
                                            'DD-MON-YYYY'));
    end if;
end;
```

13.6 Extra Fusion Apps, ORDS, OData Features

When related REST APIs follow common conventions for data filtering, sorting, attribute selection, object modification, and error reporting, Oracle APEX can provide powerful additional low-code functionality to your apps.

In addition to the *Simple HTTP* REST Data Source type, APEX offers native support for the following convention-based REST API types:

- *Oracle Cloud Applications (SaaS)* – Most existing Fusion Applications APIs
- *Oracle Cloud Applications (BOSS)* – Emerging `/api/boss` Fusion Applications APIs
- *Oracle REST Data Services* – Auto-REST enabled tables/views and collection query APIs

Copyright © 2026, Oracle and/or its affiliates.

- *OData* – Used by Microsoft Graph API, Dynamics 365, SAP, and others

Their goal is to make integrating remote data as easy as working with a local database table. When you create a REST Data Source using one of these types, the wizard automatically discovers all supported operations and Data Profile columns based on understanding their extended custom metadata. At runtime, when a region *uses* one of these REST Data Sources, the APEX engine dynamically adds any necessary request parameters and builds request body payloads. No need to define any of the common parameters or worry about maintaining request body templates.

APEX also automatically delegates filtering and sorting to the backend system when using these types of REST Data Sources. This is important for optimized application performance because it leverages backend indexes to quickly return only relevant rows to the APEX application. By default, *Simple HTTP* data sources must retrieve all rows to the APEX engine database to then perform filtering and sorting there instead.

In addition, *Oracle Cloud Applications* REST Data Sources automatically:

- Request only the attributes used by the current region, reducing payload sizes,
- Report business object validation errors as APEX error messages, and
- Support bulk API DML functionality, useful for Interactive Grid use cases.

> **Tip**
>
> To download a complete catalog of Fusion Apps REST APIs for use in your apps, visit https://github.com/oracle/apex/tree/rest-source-catalogs/fusion

13.7 Additional REST Data Source Features

REST Data Sources can cache responses, synchronize reference data, add computed columns, apply declarative filtering and sorting, combine local and remote data, and support custom adapter plug-ins.

Any of your REST Data Sources can take advantage of several additional features:

- To improve performance, cache query responses in various ways in the server.
- To speed up lookups for remote reference data, synchronize it to a local cache table.
- To enrich remote data, add computed columns that reference other ones.
- To add default filtering or sorting, use declarative syntax where available
- To combine local and remote data, and filter, sort, or aggregate it, explore local post-processing.

Finally, if you work with REST APIs that follow a set of conventions APEX doesn't support natively, you can create a custom plug-in to add a new REST Data Source type that lets your teammates automatically benefit from all the declarative smarts you build into it.

- Caching a GET Operation's Response
  To improve performance and reduce unnecessary HTTP calls, you can enable server-side caching for selected REST Data Source `GET` operations.
- Synchronizing Data Locally
  Use local data synchronization to cache less-frequently changing REST Data Source data for faster List of Values, lookup, and region access.

Copyright © 2026, Oracle and/or its affiliates.

- Adding Computed Data Profile Columns
  Add computed columns to enrich response payload data, then use their values in regions, Lists of Values, or programmatic code.
- Configuring Default Filtering and Sorting
  Configure default filter and sort expressions wherever a REST Data Source type supports them.
- Filter, Sort, Join, or Aggregate Remote Data
  Complement APEX's JSON row and column extraction with SQL to filter, sort, join, or aggregate remote REST API data as needed.
- Creating a Custom REST Source Plug-in
  If you work with REST APIs that follow a set of conventions APEX doesn't support natively, create a custom REST Data Source plug-in.

13.7.1 Caching a GET Operation's Response

To improve performance and reduce unnecessary HTTP calls, you can enable server-side caching for selected REST Data Source `GET` operations.

This lets your application reuse previously retrieved results instead of re-fetching them from the remote service. Caching options let you control whether results are shared across all users, stored per user, or scoped just to a specific session. You can also configure when cached content becomes invalid using a simple duration in minutes or a more advanced DBMS_SCHEDULER calendaring expression. For example, you might cache responses until the top of the hour or midnight, or keep them for 15 minutes. Setting the invalidation to **0** means the data is cached only during the current page execution. That can be useful if multiple regions on the same page use the same REST Data Source. APEX manages this caching entirely on the server, so it does not rely on the browser's cache.

> **Tip**
>
> If your application *changes* remote data that affects a cached REST Data Source, you can call the `PURGE_REST_SOURCE_CACHE()` procedure in the `APEX_EXEC` package. This invalidates the cache for a selected REST Data Source. Consider invoking it as part of the processing where your app modifies the remote data.

13.7.2 Synchronizing Data Locally

Use local data synchronization to cache less-frequently changing REST Data Source data for faster List of Values, lookup, and region access.

For less-frequently changing remote data, APEX can automatically create and manage a local cache table for a REST Data Source. You can configure a synchronization schedule, or call the `SYNCHRONIZE_DATA` procedure in the `APEX_REST_SOURCE_SYNC` package to refresh the data on-demand. Then enable the List of Values *Use Synchronization Table?* switch or a region's *Use Local Table* switch in the *REST Synchronization* section. When this option is enabled, APEX automatically uses the locally cached data instead of requesting the data over REST on each access.

13.7.3 Adding Computed Data Profile Columns

Add computed columns to enrich response payload data, then use their values in regions, Lists of Values, or programmatic code.

Copyright © 2026, Oracle and/or its affiliates.

Your computed columns use a SQL expression that can reference the names of other data profile columns. For example, imagine you're working with a service whose response payload contains `firstName` and `lastName` properties like this:

```
{
   ▒
  "firstName": "Jordan",
  "lastName": "Jones"
   ▒
}
```

Assume your REST Data Source's Data Profile already contains the columns:

- `FIRST_NAME` sourced from the `firstName`, and
- `LAST_NAME` using the selector `lastName`

Then, in the Data Profile editor you can click *(Add Column >)* and choose the *Column Type* of *SQL Expression*. You can call your new column `FULL_NAME` and enter a valid *SQL Expression* like the following to concatenate the first and last name values together:

```
FIRST_NAME||' '||LAST_NAME
```

Other types of computed columns include *Lookup* and *SQL Query (return single value)*. Once defined, the computed column is a read-only value you can reference anywhere you use the REST Data Source throughout your application.

13.7.4 Configuring Default Filtering and Sorting

Configure default filter and sort expressions wherever a REST Data Source type supports them.

For example, the Oracle Cloud Applications, ORDS, and OData REST Data Source types all support their own declarative filter and order by syntax. By enabling the *External Filter and Order By* switch on a region or List of Value, you can specify either or both using an appropriately structured clause.

For example, imagine working with a REST API from the Fusion Applications *Customer Experience* pillar. In a region using a REST Data Source based on the *Sales Opportunities* business object endpoint, you might want to only show open opportunities with a win probability greater than 75%. To achieve this, you can specify the following *External Filter* clause on the region:

```
StatusCode = 'OPEN' and WinProb >= 75
```

Then, say you want those results ordered descending by win probability and then by the primary contact. To make that happen, just specify the *External Order By* clause of:

```
WinProb:desc,PrimaryContactPartyName:asc
```

For more information on external filter and order by syntax for Fusion Applications REST APIs, see REST Data Source Runtime Features for Oracle Cloud SaaS Apps and BOSS REST Data Source Runtime Features in *Oracle APEX App Builder User's Guide*.

Copyright © 2026, Oracle and/or its affiliates.

When configuring the same features for ORDS or OData REST Data Source, check with the documentation of those services to understand the filtering and sorting syntax they expect.

13.7.5 Filter, Sort, Join, or Aggregate Remote Data

Complement APEX's JSON row and column extraction with SQL to filter, sort, join, or aggregate remote REST API data as needed.

- Understanding the JSON_TABLE Query
  The APEX engine turns a REST Data Source's JSON response payload into rows and column using the Data Profile information.
- Locally Filtering and Sorting Remote Data
  If the REST API you use doesn't let you filter or sort the data, you can do it in the local APEX database instead. Since these follow-up actions occur *after* the APEX engine retrieves the JSON response from the REST API, it's known as *Local Post-Processing*.
- Using Custom SQL Involving Remote Data
  Set a region or LOV's *Local Post-Processing* to *SQL Query* to query remote data from the `#APEX$SOURCE_DATA#` inline view.

13.7.5.1 Understanding the JSON_TABLE Query

The APEX engine turns a REST Data Source's JSON response payload into rows and column using the Data Profile information.

It dynamically produces a `SELECT` statement using the `JSON_TABLE()` operator to map JSON data into relational query results. For example, in a REST Data Source based on an *Employees* REST API, assume its Data Profile defines three columns:

Table 13-3 Data Profile Columns

Name	Data Type	Format Mask	Selector
EMPNO	NUMBER		empno
ENAME	VARCHAR2		ename
HIREDATE	DATE	YYYY-MM-DD"T"HH24:MI:SS"Z"	hiredate

At runtime, the APEX engine retrieves a JSON response payload that looks like this:

```
{
    "items": [
        {
            "empno": 7839,
            "ename": "KING",
            "job": "PRESIDENT",
            "mgr": null,
            "hiredate": "1981-11-17T00:00:00Z",
            "sal": 5000,
            "comm": null,
            "deptno": 10
        },
        ▒
        {
            "empno": 7788,
            "ename": "SCOTT",
```


Copyright © 2026, Oracle and/or its affiliates.

```
            "job": "ANALYST",
            "mgr": 7566,
            "hiredate": "1982-12-09T00:00:00Z",
            "sal": 3000,
            "comm": null,
            "deptno": 20
        }
    ]
}
```

It then uses the above `response_payload` CLOB in a SQL statement like the following:

```
select "EMPNO",
       "ENAME",
       "HIREDATE"
from (
    select *
    from (
    /* Cast extracted data to declared data type here */
    select to_number("EMPNO")                    as "EMPNO",
           "ENAME"                               as "ENAME",
           to_date("HIREDATE",
                   'YYYY-MM-DD"T"HH24:MI:SS"Z"') as "HIREDATE"
        from
            /* Use the row selector and column selectors here */
            json_table ( response_payload format json,'$."items"[*]'
                columns (
                    "EMPNO"    varchar2 ( 4000 ) path '$."empno"',
                    "ENAME"    varchar2 ( 4000 ) path '$."ename"',
                    "HIREDATE" varchar2 ( 4000 ) path '$."hiredate"'
                )
            )
    )
)
```

This statement produces the relational result:

```
     EMPNO ENAME      HIREDATE
---------- ---------- ---------
      7839 KING       17-NOV-81
      7698 BLAKE      01-MAY-81
      7782 CLARK      09-JUN-81
      7566 JONES      02-APR-81
      7788 SCOTT      09-DEC-82
```

13.7.5.2 Locally Filtering and Sorting Remote Data

If the REST API you use doesn't let you filter or sort the data, you can do it in the local APEX database instead. Since these follow-up actions occur *after* the APEX engine retrieves the JSON response from the REST API, it's known as *Local Post-Processing*.

When you set this property of your region or List of Values to *Where/Order By Clause*, then additional *Where Clause* and *Order by Type* properties appear. If you choose a *Static Value*

Copyright © 2026, Oracle and/or its affiliates.

type of order by clause, then can just type it in yourself. Otherwise the order by clause is determined by the *Order by Item* you configure.

The `WHERE` clause and `ORDER BY` clauses you configure for Local Post-Processing are added to the outermost SQL query in the APEX engine's `JSON_TABLE()` statement. As a consequence, these clauses can only reference column names that are included in that outermost `SELECT` list. If you need to filter on or order by a column that you didn't *initially* include in a region, add the additional column and leave it hidden.

For example, to present a list of *Employees* REST Data Source hired after October 1981, ordered descending by hire date, set the Local Post-Processing `WHERE` clause to:

```
HIREDATE > DATE'1981-10-31'
```

Then set the *Static Value* type `ORDER BY` clause to:

```
HIREDATE DESC
```

The result is an augmented `JSON_TABLE()` SQL statement that looks like this:

```
select "EMPNO",
       "ENAME",
       "HIREDATE"
from (
    /* Rest of APEX JSON_TABLE() query omitted for brevity */
)
where HIREDATE > DATE'1981-10-31'
order by HIREDATE DESC
```

This additional configuration results in retrieving the original set of *Employees* from the remote REST API, but then displaying only those in the desired hire date range, ordered as indicated.

13.7.5.3 Using Custom SQL Involving Remote Data

Set a region or LOV's *Local Post-Processing* to *SQL Query* to query remote data from the `#APEX$SOURCE_DATA#` inline view.

At runtime, the APEX engine substitutes `#APEX$SOURCE_DATA#` for the entire `JSON_TABLE()` query it normally uses to process remote REST data. The *SQL Query* property that appears will *initially* contain a statement that produces the same results as before:

```
select EMPNO,
       ENAME,
       HIREDATE
  from #APEX$SOURCE_DATA#
```

But you can edit this SQL statement to add additional `WHERE` clauses, `JOIN` clauses, additional `SELECT` list columns or expressions, or really anything that produces a resulting valid `SELECT` statement.

Copyright © 2026, Oracle and/or its affiliates.

For example, to join the `EMPNO` column value with local tables `EMP_TRAINING` and `EMP_COURSE` to produce a list of courses for retrieved employees having a status of completed or certified, you could write the query:

```
select r.EMPNO,
       r.ENAME,
       r.HIREDATE,
       c.NAME as COURSE_NAME,
       t.STATUS
  from #APEX$SOURCE_DATA# r
  join EMP_TRAINING t
    on r.EMPNO = t.EMPNO
  join EMP_COURSE c
    on t.COURSE_ID = c.ID
 where t.STATUS in ('COMPLETED','CERTIFIED')
```

There are no strict requirements beyond the statement's:

- Including a reference to `#APEX$SOURCE_DATA#`
- Being syntactically correct `SQL`

This means your Local Post-Processing SQL statement can also compute aggregates, rename columns, or simply do whatever your use case requires. If you need to dynamically construct your SQL statement, you can use the *PL/SQL Function Body Returning SQL Query* type instead.

13.7.6 Creating a Custom REST Source Plug-in

If you work with REST APIs that follow a set of conventions APEX doesn't support natively, create a custom REST Data Source plug-in.

This kind of extension adds a new REST Data Source type to the list in the Create REST Data Source wizard. By choosing your new kind of data source, your teammates can automatically benefit from all the declarative smarts you build into it.

Creating a REST Data Source plug-in requires writing PL/SQL code that implements one or more contract procedures the APEX engine defines. The high-level contracts this type of plug-in can implement are:

Table 13-4 REST Data Source Plug-in Contract

Procedure	Purpose
Capabilities	Defines if server-side filtering, sorting, and pagination are supported
Discover	Determine operations, parameters, and data profile columns based on custom service metadata
Fetch	Retrieve data with optional support for server-side filtering, sorting, and pagination
DML	Save changes implied by DML rows from form, grid, or `APEX_EXEC`
Execute	Handle the *Invoke API* or `EXECUTE_REST_SOURCE` use case

You can find a basic, working example of a REST Source plug-in to study and experiment with on the Oracle APEX GitHub repository. After choosing the APEX release you're using, see the `plugins/rest-source` subdirectory. It implements the *Capabilities*, *Discover*, and *Fetch* contracts.

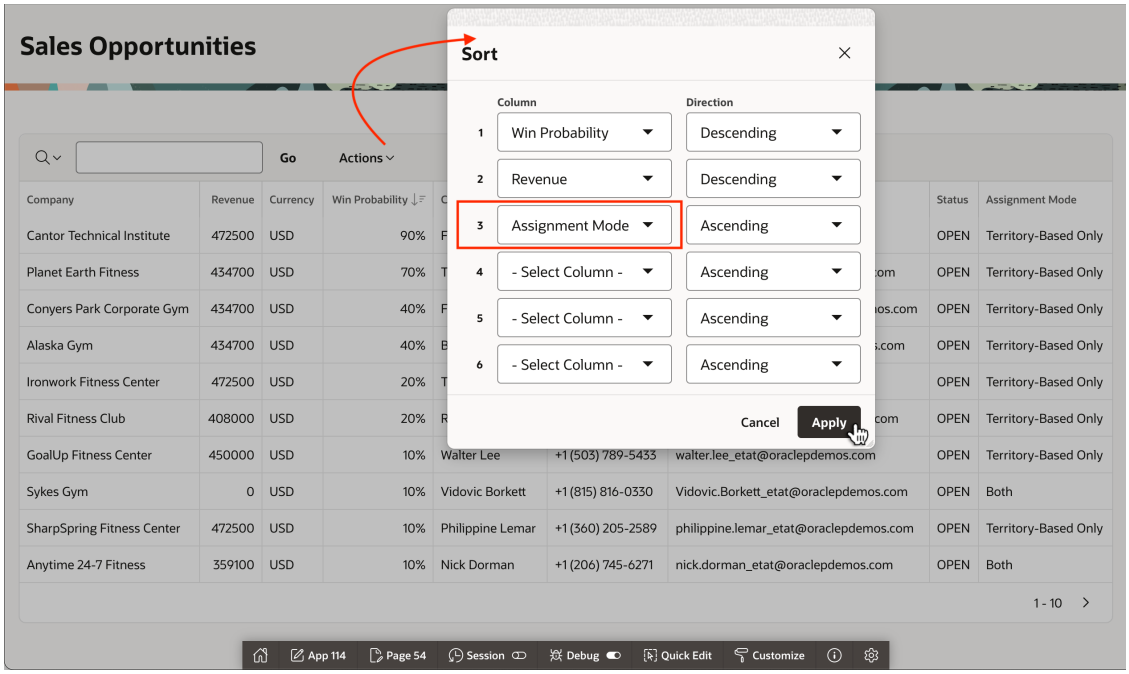
Copyright © 2026, Oracle and/or its affiliates.

13.8 Debugging REST API Errors

If you work with REST APIs, one of them might return an error at runtime. Use APEX's built-in debug tracing facility to diagnose the problem and identify a fix.

The figure below shows an Interactive Report that uses a REST Data Source based on the Fusion Applications *Sales Opportunities* REST API. The user chooses *Actions* > *Data* > *Sort* to configure their preferred sorting order. Notice that they choose the *Assignment Mode* column for the third sorting column. When they click *(Apply)* to refresh the interactive report with the data sorted in this new way, they receive a runtime error.

Figure 13-30 Interactive Report Based on Fusion Apps Sales Opportunities REST Data Source



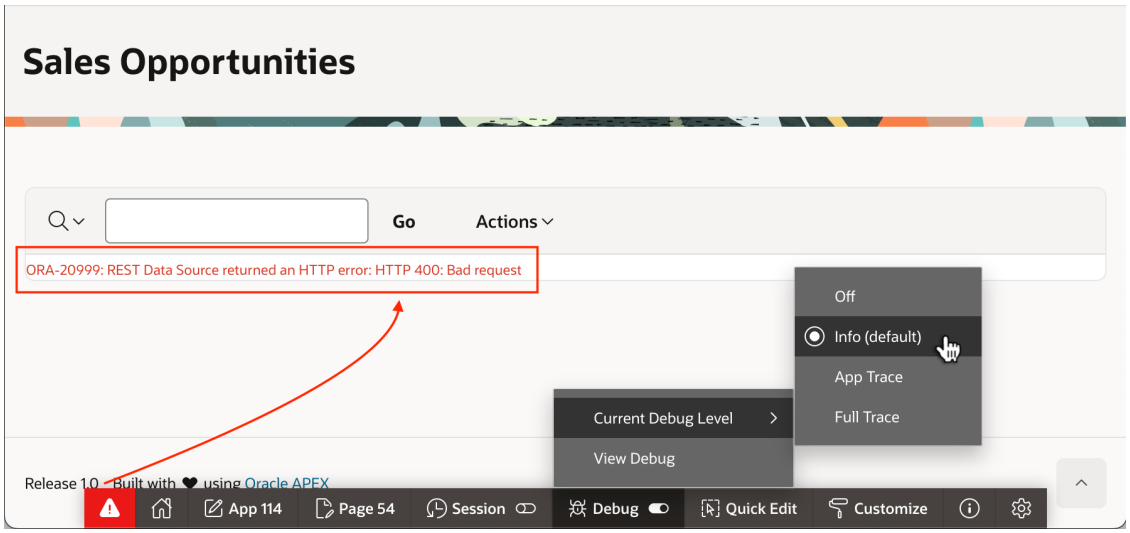
The error displays as shown below with the error message:

```
ORA-20999: REST Data Source returned an HTTP error: HTTP 400: Bad request
```

When running your application from App Builder, by default, a developer toolbar appears at the bottom of the page. Notice when an error occurs that it displays a warning icon. By choosing *Debug* > *Current Debug Level* > *Info*, you can enable debug tracing. When you set the debug level, the current page refreshes to capture a debug trace. While debug tracing is enabled, each subsequent HTTP request the browser makes to your APEX application produces a distinct debug trace log entry. To browse the latest debug trace entries, choose *Debug* > *View Debug* from the developer toolbar.

Copyright © 2026, Oracle and/or its affiliates.

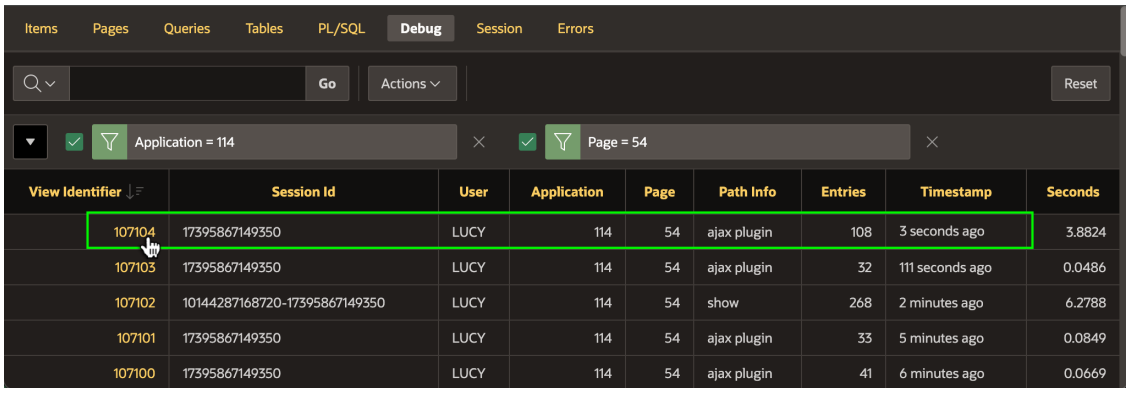
Figure 13-31 Enable Debug Tracing to Diagnose HTTP 400 Error from REST Data Source



As shown below, the debug viewer opens in a new window and displays recent debug request traces in an Interactive Report. By default they should be sorted descending by *Timestamp* so the most recent entries are first. Notice the viewer's Interactive Report region already has an application id and page id filter applied to focus your attention on the most likely trace entries you want to see. However, you can use standard features of the Interactive Report to disable a filter or remove it if you need to see a more complete set of trace log entries.

In addition to the *Timestamp* column, the *Path Info* can also help identify the right request to examine in more detail. *Path Info* will be `show` for an initial page render, `accept` *`REQUESTVAL/BUTTONNAME`* for a page submit, and one of `ajax`, `ajax plugin`, or `ajax process` for partial page requests that APEX components make to refresh a section of a page. In this case when the user applies a new sort condition the Interactive Report native page item plug-in does a partial refresh so *Path Info* is `ajax plugin`. The figure highlights the most recent request debug trace record. Click on the view identifier link to drill down to inspect the trace detail details for that request.

Figure 13-32 Identifying Most Recent Debug Trace and Drilling Down for Details



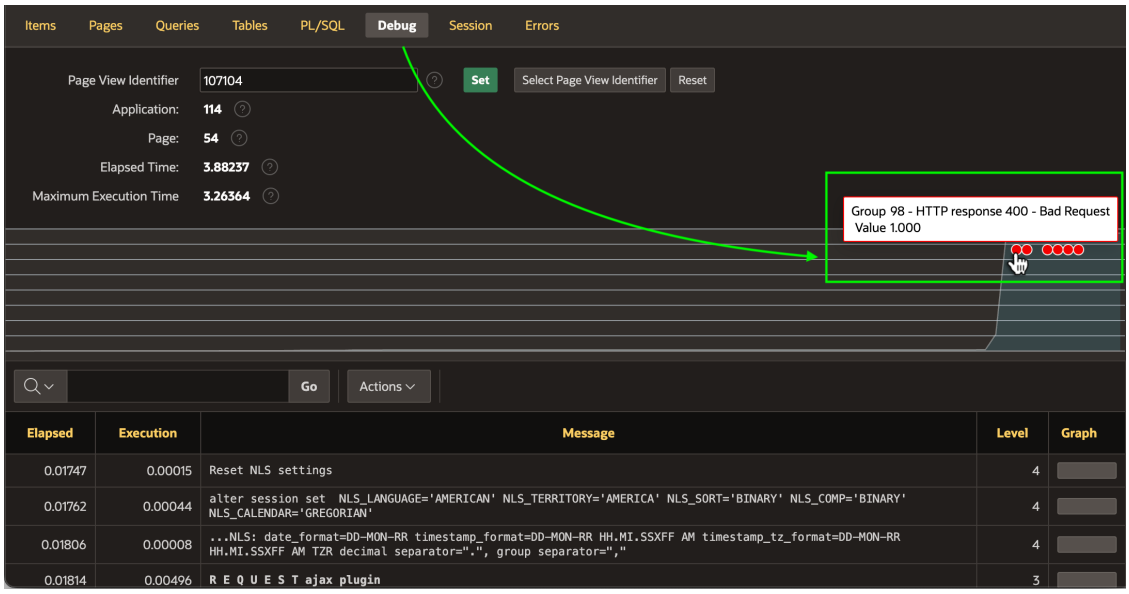
View Identifier	Session Id	User	Application	Page	Path Info	Entries	Timestamp	Seconds
107104	17395867149350	LUCY	114	54	ajax plugin	108	3 seconds ago	3.8824
107103	17395867149350	LUCY	114	54	ajax plugin	32	111 seconds ago	0.0486
107102	10144287168720-17395867149350	LUCY	114	54	show	268	2 minutes ago	6.2788
107101	17395867149350	LUCY	114	54	ajax plugin	33	5 minutes ago	0.0849
107100	17395867149350	LUCY	114	54	ajax plugin	41	6 minutes ago	0.0669

The debug viewer for a single request shows the complete logging detail. The *Info* level of debug tracing is usually enough to understand most problems. On occasion, when you require

Copyright © 2026, Oracle and/or its affiliates.

additional detail, choose a more granular debug trace level. A timeline accompanies the page to help understand at what point during the request work is performed that takes time and where errors occur. As shown below, errors display as red dots on the timeline. Hover over a dot to see an error message summary, and click a dot to navigate to that line of the debug trace log.

Figure 13-33 Quickly Navigating to an Error on the Debug Trace Timeline

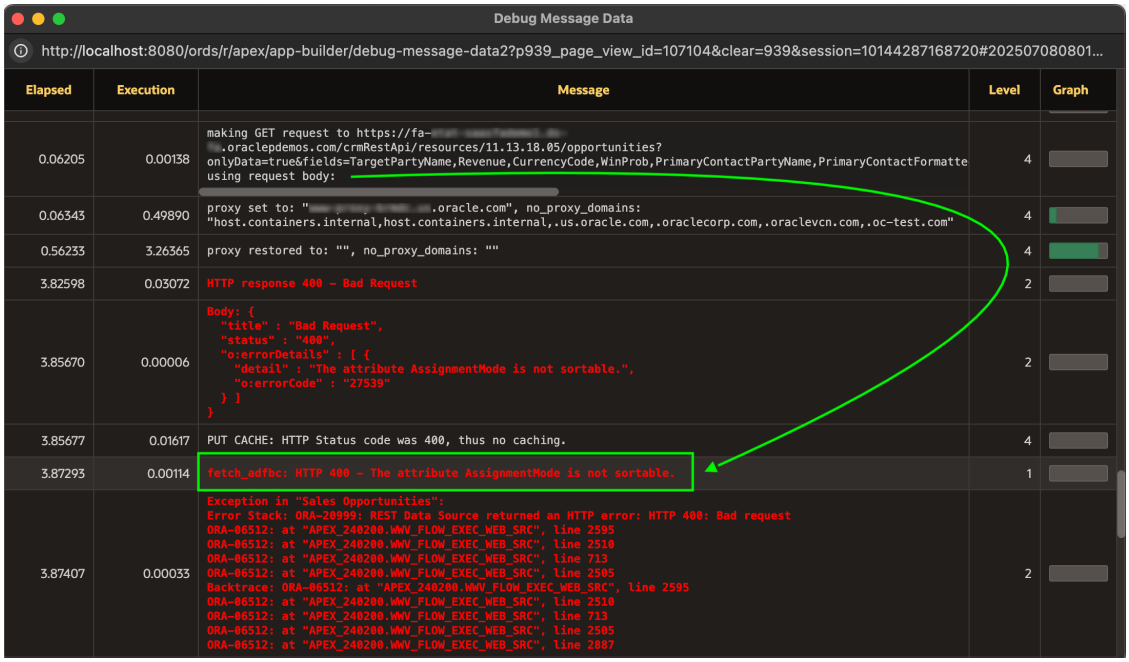


As shown below, the HTTP 400 error the *Sales Opportunity* REST API returns contains an error response payload. It contains the error detail that appears in the log explaining that the `AssignmentMode` attribute is not sortable.

```
fetch_adfbc: HTTP 400 - The attribute AssignmentMode is not sortable
```


Copyright © 2026, Oracle and/or its affiliates.

Figure 13-34 Examining First Error and the Preceding REST API Call



Scroll up in the log messages to find the preceding REST API call. It's identified with the word `making` and the HTTP method used for the request. The request looks like the following, that is slightly abridged and reformatted for readability:

```
making GET request to https://server/crmRestApi/resources/11.13.18.05/
opportunities
?onlyData=true
&fields=TargetPartyName,Revenue,CurrencyCode,WinProb,PrimaryContactPartyName,.
..
&orderBy=WinProb:desc,Revenue:desc,AssignmentMode:asc
&limit=500, using request body:
```

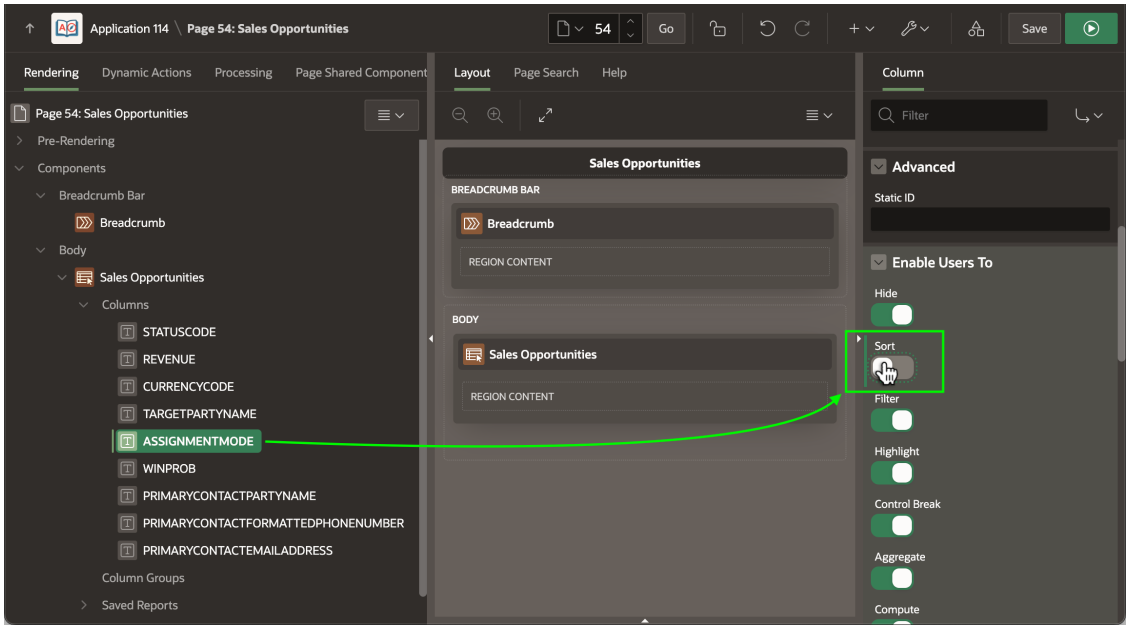
Notice APEX delegated the end user's sort configuration into an appropriate `orderBy` query string sort expression so the Fusion Applications REST API could perform the requested sort on the server side:

```
:
&orderBy=WinProb:desc,Revenue:desc,AssignmentMode:asc
:
```

The error message clearly explains that the server cannot sort on this attribute, so you've identified the problem. To remedy the situation, you just need to adjust one setting on the `ASSIGNMENTMODE` column in the Interactive Report to disable sorting. As shown below, you do this with a single click on the *Enable Users To* > *Sort* switch after selecting the column in the *Rendering* tab.

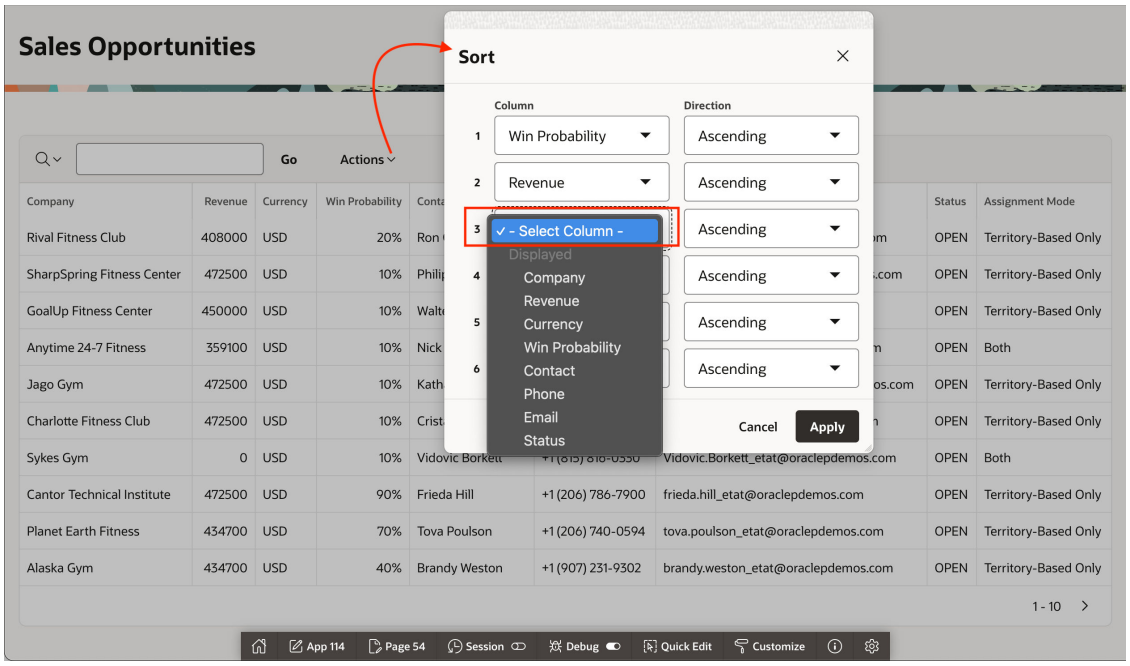
Copyright © 2026, Oracle and/or its affiliates.

Figure 13-35 Disabling Sorting on Assignment Mode Column



After diagnosing the problem using the debug trace log, and adjusting the page to avoid the issue in the future, as shown below, you can see the end user no longer has *Assignment Mode* available in the list of columns they can sort on. You can disable the debug tracing using the developer toolbar until the next time it proves useful. The figure shows the end user Interactive Report *Action* > *Sort* dialog where the end user configures the attributes to user to order the results.

Figure 13-36 Problem Fixed: Users Can No Longer Choose the Non-Sortable Column



Copyright © 2026, Oracle and/or its affiliates.

Copyright © 2026, Oracle and/or its affiliates.

14

Exposing APIs for Integration

Expose REST APIs for external integrations while reusing business logic and securing access.

A PL/SQL package provides an application programming interface (API) for any solution in your workspace. However, when *external systems* need to integrate with your app, you need to provide them a *REST* API instead. It's a web service that another program can call over the network, and uses JavaScript Object Notation (JSON) as a simple way to send and receive data.

Some integrations only need to read data, while others may require creating, updating, or deleting it. If you let an outside system *change* your application data, you *must* apply the same validation and defaulting business logic used in your web user interface to keep bad data out of your system of record. In either case, you need to ensure only *intended* clients can access your services.

This section's example code helps you witness how two key Oracle 26ai features can cut the effort to expose APIs in half.

- Reviewing Action Items API Requirements
  Review why an external system needs REST APIs to integrate with your APEX app.
- Reviewing REST API Basics
  Review the HTTP, JSON, SQL, and PL/SQL basics behind REST APIs.
- Planning Your ORDS Service Module
  Plan your ORDS service module schema alias, URL templates, and handlers.
- Using SQL Developer Web REST Designer
  Use SQL Developer Web REST Designer to create and navigate ORDS service definitions.
- Creating and Testing Read-Only APIs
  Create and test read-only REST APIs for collections and single rows.
- Enforcing Business Logic in REST APIs
  Apply the same business rules to REST API changes that your APEX pages use.
- Controlling REST API Behavior Completely
  For *complete* control over your ORDS REST APIs, create a full set of *PL/SQL* handlers.
- Layering Packages for Modularity
  Layer PL/SQL packages so pages and REST APIs share checks while handlers stay simple.
- Simplifying REST APIs in Oracle 26ai
  Simplify REST API development in Oracle 26ai with JSON type and duality views.
- Securing APIs with Role-Based Access Control
  Secure REST APIs with roles and OAuth 2.0 clients for authorized partner access.

14.1 Reviewing Action Items API Requirements

Review why an external system needs REST APIs to integrate with your APEX app.

Copyright © 2026, Oracle and/or its affiliates.

An external support ticketing system needs to integrate with your existing application managing action items and the team of staff members assigned to work on each one. Review the app's functionality and how the external tool needs to integrate with it.

- Exploring Action Item Teams Example
  Explore an action items app that manages teams and enforces team membership rules.
- Understanding the Integration Scenario
  Understand how an external ticketing system uses REST APIs to manage action items in your APEX app.

14.1.1 Exploring Action Item Teams Example

Explore an action items app that manages teams and enforces team membership rules.

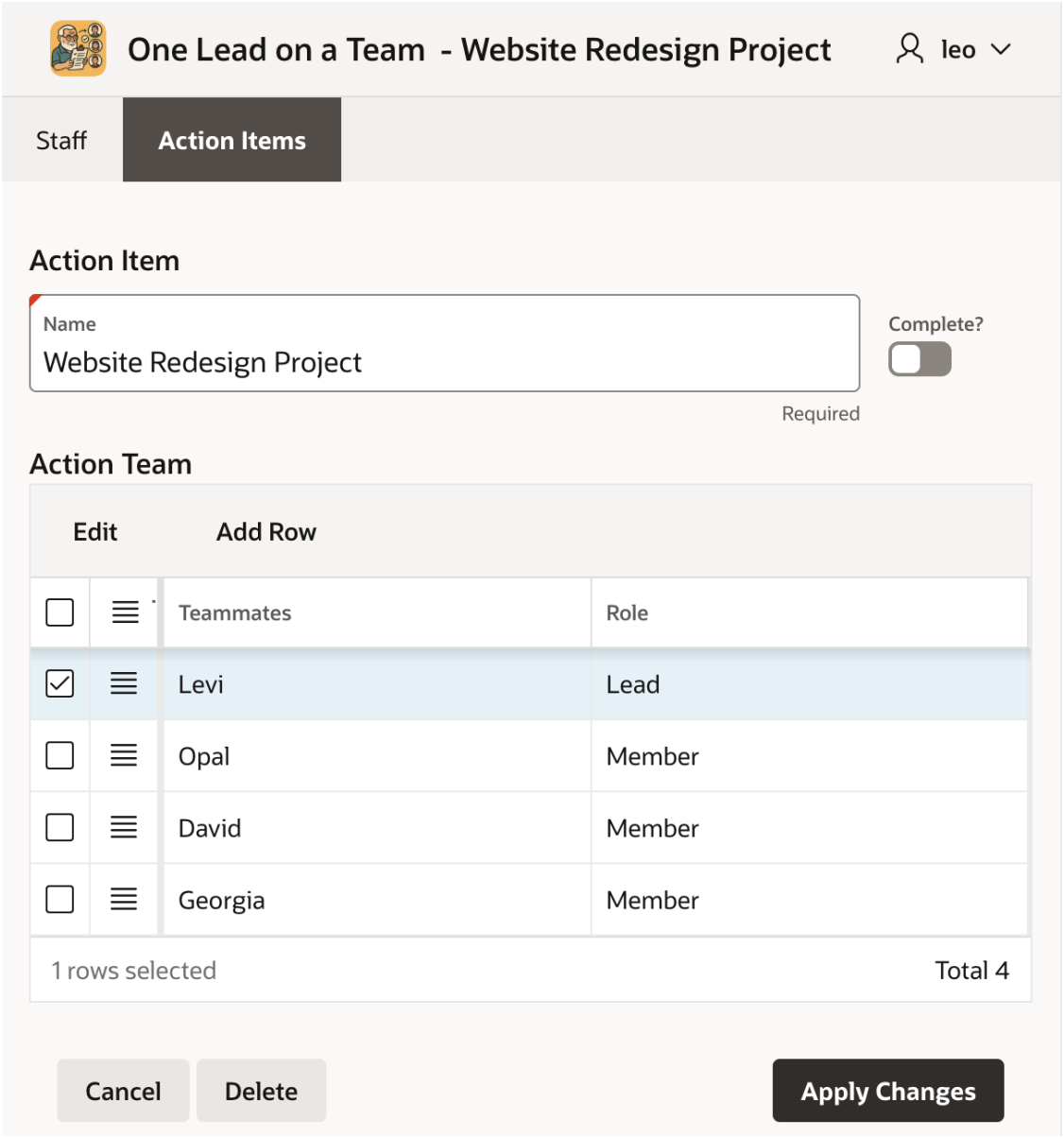
Your application manages action items, staff members, and the team assigned to work on an action item. Each staff member on a particular action item team plays the role of *lead* or a regular team *member*. Your app enforces business rules like:

- Every action item team needs a lead,
- There can only be one lead on a team, and
- No duplicate team members allowed.

The figure below shows a page from your app that all staff use daily. It shows a two-tab navigation structure with a tab to edit Staff members and a tab to edit **Action Items**. On the **Action Items** tab, the user can list, create, edit, and delete action teams of staff members assigned to work on an action item. The Website Redesign Project being edited in the figure has team members Levi, Opal, David, and Georgia. Levi's role is **Lead**, and the others' role is **Member**.

Copyright © 2026, Oracle and/or its affiliates.

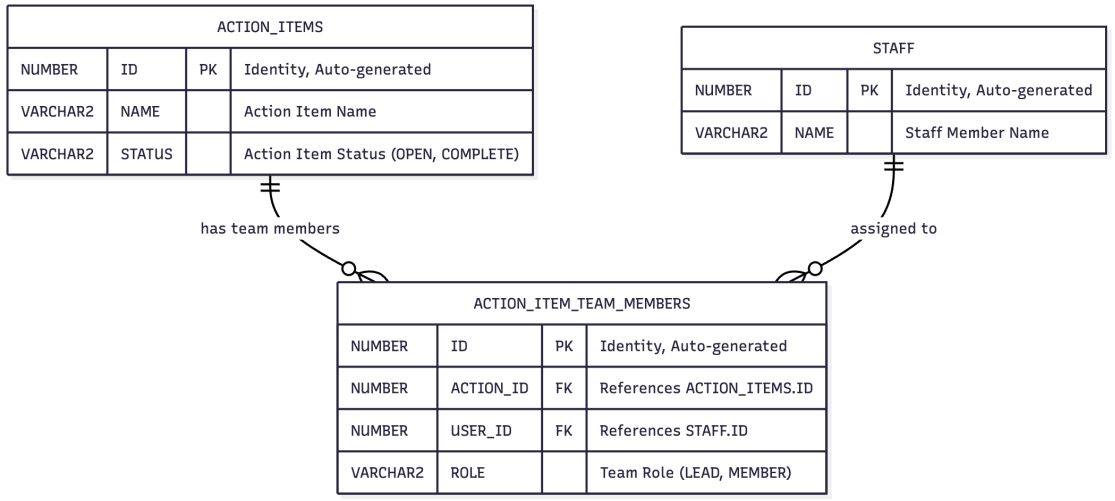
Figure 14-1 Application Page for Editing an Action Item and Assigned Team Members



The simple underlying data model appears below. Just three tables: `ACTION_ITEMS`, `STAFF`, and `ACTION_ITEM_TEAM_MEMBERS`. The entity relationship diagram shows that an action item has one or more team members, each of whom is a staff member.

Copyright © 2026, Oracle and/or its affiliates.

Figure 14-2 Simple Schema for Action Items and Team Members Assigned to Them



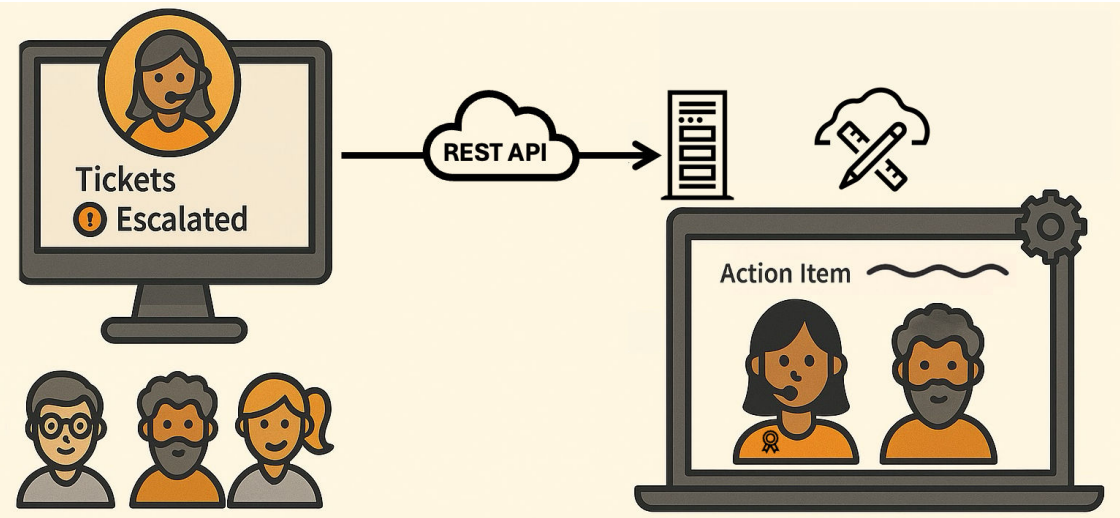
14.1.2 Understanding the Integration Scenario

Understand how an external ticketing system uses REST APIs to manage action items in your APEX app.

Your support team uses an open-source ticketing system outside your APEX environment. When a customer escalates a ticket that requires development work, that system uses a REST API to create an action item in your app. It assigns the support agent as the lead and adds key engineers as team members. If the ticket changes, the system updates the action item. If the ticket closes without requiring development work, the system deletes it.

The diagram illustrates the REST API interaction between the external ticketing system and the action item system built with APEX.

Figure 14-3 Open Source Ticketing System Needs to Integrate with Your Action Item App



Copyright © 2026, Oracle and/or its affiliates.

14.2 Reviewing REST API Basics

Review the HTTP, JSON, SQL, and PL/SQL basics behind REST APIs.

Before learning how to expose a REST API for your application, it's good to review the basic "ingredients" involved in the recipe:

- Key HTTP methods that REST APIs use,
- The basic syntax of JSON documents,
- SQL features to construct JSON from relational data, and
- PL/SQL's native JSON types, and
- Simplifications JSON Relational Duality Views enable in Oracle 26ai.

- Reviewing HTTP Methods for REST APIs
  Review how HTTP methods map to REST operations on business entities.
- Inspecting JSON Syntax
  Review JSON objects, arrays, and value types used in REST API payloads.
- Constructing JSON in SQL
  Use `JSON_OBJECT` and `JSON_ARRAYAGG` SQL functions to construct JSON from relational data.
- Working with JSON in PL/SQL
  Use PL/SQL JSON types to parse, build, and serialize JSON data.

14.2.1 Reviewing HTTP Methods for REST APIs

Review how HTTP methods map to REST operations on business entities.

The Hypertext Transfer Protocol (HTTP) defines methods that state a client's purpose in requesting a server URL. In RESTful web services, the four methods below map to the Create, Read, Update, and Delete (CRUD) operations on *resources*. Each resource is a business entity like an *Action Item*.

Table 14-1 HTTP Methods and REST Operations

Method	Business Entity Operation
`POST`	Create a new entity
`GET`	Read an entity collection or a specific entity
`PUT`	Update an existing entity with a new version
`DELETE`	Delete an entity

Tip

`PATCH` is a less frequently used method for performing a *partial* update to a business entity. Services supporting the `PATCH` "verb" interpret the client's JSON payload as an incremental change to apply to the existing object.

Copyright © 2026, Oracle and/or its affiliates.

14.2.2 Inspecting JSON Syntax

Review JSON objects, arrays, and value types used in REST API payloads.

JavaScript Object Notation (JSON) is a lightweight, text-based format for representing structured data. It's easy for humans to read and write, and simple for programs to parse and generate. JSON describes data using two main structures:

- *Objects* – collections of name/value pairs, enclosed in curly braces `{}`, and
- *Arrays* – ordered lists of values, enclosed in square brackets `[]`.

A JSON object begins with `{` and ends with `}`. Inside, one or more *name/value pairs* appear, separated by commas:

- You always write names – also called *properties* or *keys* – in double quotes `"…"`.
- A value *follows* each name, separated by a colon.

For example, the object below represents a team member named Georgia. The keys are `name`, `role`, `user_id`, and `active`. The values `"Georgia"` and `"LEAD"` are strings, while `5` is a number and `true` is a *boolean* (true or false) value.

```
{
  "name": "Georgia",
  "role": "LEAD",
  "user_id": 5,
  "active": true
}
```

The *order* of the properties in a JSON object is *not* significant, so the following object is the same *logical* object as above, even though their two text representations are not exactly equal.

```
{
  "user_id": 5,
  "role": "LEAD",
  "name": "Georgia",
  "active": true
}
```

A JSON array starts with `[` and ends with `]`. Its elements can be strings, numbers, booleans, `null`, objects, or other arrays. Commas separate multiple elements. For example, an array of strings looks like:

```
["apple","banana","cherry"]
```

> **Tip**
>
> In *contrast* to the ordering of object properties, array element order *is* significant. However, some apps may *ignore* that ordering.

Copyright © 2026, Oracle and/or its affiliates.

Arrays often hold objects. For example, this is an array of two team members:

```
[
  { "name": "Georgia", "role": "LEAD",   "user_id": 5, "active": true},
  { "name": "Karl",    "role": "MEMBER", "user_id": 8, "active": true}
]
```

In JSON, a value can be one of the types below.

Table 14-2 JSON Types and Their Representations

Type	Textual Representation
String	`"some text"`
Number	42 or `3.14` (no quotes)
Boolean	`true` or `false`
Null	`null`
Object	`{…}`
Array	`[…]`

For better readability, use any spaces, tabs, or line breaks you need. Parsers ignore this whitespace.

14.2.3 Constructing JSON in SQL

Use `JSON_OBJECT` and `JSON_ARRAYAGG` SQL functions to construct JSON from relational data.

For example, the following query retrieves one JSON object per `ACTION_ITEMS` row. Notice you can include a column as is, with a *default* property name, or explicitly *specify* a column's property name using the *`value`* keyword. Study the team member `name`, `user_id`, and `role`. For consistency, it's fine to use *`value`* even if you choose the *same* lowercase version of the column name that *would* have been the default.

```
select json_object(
          '_id' value ai.id,
          ai.name,
          ai.status,
          'team' value (select json_arrayagg(
                                 json_object(
                                    'team_member_id' value tm.id,
                                    'name'           value s.name,
                                    'user_id'        value tm.user_id,
                                    'role'           value tm.role))
                          from action_item_team_members tm
                          join staff s
                            on tm.user_id = s.id
                         where tm.action_id = ai.id))
  from action_items ai
```

Each *row* of the result, looks like the following:

```
{
  "_id": 14,
```


Copyright © 2026, Oracle and/or its affiliates.

```
  "name": "End of Year Party",
  "status: "OPEN",
  "team": [
    {
      "team_member_id": 23,
      "name": "David",
      "user_id": 4,
      "role": "MEMBER"
    },
    {
      "team_member_id": 24,
      "name": "Georgia",
      "user_id": 5,
      "role": "LEAD"
    },
    {
      "team_member_id": 25,
      "name": "Jane",
      "user_id": 13,
      "role": "MEMBER"
    }
  ]
}
```

When using `JSON_ARRAYAGG` to collect one or more elements into an array, you can include an `ORDER BY` clause to sort the array elements. For example, to sort the team members *first* by `role`, *then* by `name`, you can write:

```
select json_object(
          '_id' value ai.id,
          ai.name,
          ai.status,
          'team' value (select json_arrayagg(
                                 json_object(
                                    'team_member_id' value tm.id,
                                    'name'           value s.name,
                                    'user_id'        value tm.user_id,
                                    'role'           value tm.role)
                                 order by tm.role, s.name)
                          from action_item_team_members tm
                          join staff s
                            on tm.user_id = s.id
                         where tm.action_id = ai.id))
  from action_items ai
```

With this query, the equivalent row in the result shown above looks like this instead:

```
{
  "_id": 14,
  "name": "End of Year Party",
  "status": "OPEN",
  "team": [
    {
      "team_member_id": 24,
      "name": "Georgia",
```


Copyright © 2026, Oracle and/or its affiliates.

```
      "user_id": 5,
      "role": "LEAD"
    },
    {
      "team_member_id": 23,
      "name": "David",
      "user_id": 4,
      "role": "MEMBER"
    },
    {
      "team_member_id": 25,
      "name": "Jane",
      "user_id": 13,
      "role": "MEMBER"
    }
  ]
}
```

14.2.4 Working with JSON in PL/SQL

Use PL/SQL JSON types to parse, build, and serialize JSON data.

- Parsing JSON in PL/SQL
  Parse JSON text into PL/SQL JSON objects and arrays, then access their values by type.
- Treating an Element More Specifically
  Use TREAT to work with a generic JSON element as a more specific PL/SQL JSON type.
- Constructing JSON in PL/SQL
  Build JSON objects and arrays in PL/SQL by adding properties and array elements.
- Serializing JSON in PL/SQL
  Serialize PL/SQL JSON objects to text with `STRINGIFY` or `TO_CLOB`, depending on size.

14.2.4.1 Parsing JSON in PL/SQL

Parse JSON text into PL/SQL JSON objects and arrays, then access their values by type.

PL/SQL offers native types `JSON_OBJECT_T` and `JSON_ARRAY_T` for working programmatically with JSON data. Create an instance by passing JSON text in a `VARCHAR2` or `CLOB` to the type *constructor function*. Then as shown below you can access property values using *getter* functions, depending on the data type expected, and access array elements using the `get()` function.

```
declare
    l_actionitem json_object_t;
    l_team       json_array_t;
    l_member     json_object_t;
    ----------------------------
    function get_object(
        p_array in json_array_t,
        p_index in pls_integer)
        return     json_object_t
    is
    begin
        return treat(p_array.get(p_index) as json_object_t);
```


Copyright © 2026, Oracle and/or its affiliates.

```
    end get_object;
begin
   l_actionitem := json_object_t(q'~
                   {
                     "name": "Practice Using JSON in PL/SQL",
                     "team": [
                       {"role": "LEAD","name": "Georgia"},
                       {"name": "Usha","role": "MEMBER"}
                     ]
                   }
                   ~');
   l_team   := l_actionitem.get_array('team');
   l_member := treat(l_team.get(0) as json_object_t);
   -- All three print out "Georgia"
   dbms_output.put_line(l_member.get_string('name'));
   dbms_output.put_line(treat(l_team.get(0) as
json_object_t).get_string('name'));
   dbms_output.put_line(get_object(l_team,0).get_string('name'));
end;
```

Tip

When calling object functions on JSON types, ensure the variable is not null. Calling an object function on a null produces the error message:

```
ORA-30625: method dispatch on NULL SELF argument is disallowed
```

14.2.4.2 Treating an Element More Specifically

Use TREAT to work with a generic JSON element as a more specific PL/SQL JSON type.

The generic type `JSON_ELEMENT_T` is the "grandparent" in the PL/SQL *family* of JSON types. A JSON array can hold scalars, objects, or other arrays. In the tree below, if type B is indented inside type A, then you can say B *is a more specific kind of* A.

```
JSON_ELEMENT_T
├─── JSON_SCALAR_T
│    ├─── JSON_NUMBER_T
│    ├─── JSON_STRING_T
│    └─── JSON_BOOLEAN_T
├─── JSON_OBJECT_T
└─── JSON_ARRAY_T
```

Since PL/SQL functions of the same name *can't* return different types, `JSON_ARRAY_T.GET()` returns the *root* `JSON_ELEMENT_T` type. To explicitly treat the returned array element as one of the more specific types, use:

```
TREAT(… AS JSON_SPECIFIC_TYPE)
```


Copyright © 2026, Oracle and/or its affiliates.

A function like `GET_OBJECT` below can help save you typing `TREAT` a lot if you work frequently with arrays of objects.

```
-- Get object with index p_index in an array of objects
function get_object(
    p_array in json_array_t,
    p_index in pls_integer)
    return     json_object_t
is
begin
    return treat(p_array.get(p_index) as json_object_t);
end get_object;
```

14.2.4.3 Constructing JSON in PL/SQL

Build JSON objects and arrays in PL/SQL by adding properties and array elements.

You can construct JSON programmatically property by property, starting with empty JSON objects and arrays. As shown below, notice that the code *first* builds the team member array's contents, *then* puts the array in the containing object with a `team` property name.

```
declare
    l_actionitem     json_object_t := json_object_t();
    l_team           json_array_t  := json_array_t();
    l_member_usha    json_object_t;
    l_member_georgia json_object_t;
begin
   l_actionitem.put('name', 'Practice Using JSON in PL/SQL');
   l_member_georgia := json_object_t();
   l_member_georgia.put('role', 'LEAD');
   l_member_georgia.put('name', 'Georgia');
   l_team.append(l_member_georgia);
   l_member_usha := json_object_t();
   l_member_usha.put('name', 'Usha');
   l_member_usha.put('role', 'MEMBER');
   l_team.append(l_member_usha);
   -- append all array elements before adding array to object
   l_actionitem.put('team', l_team);
   -- Both print out "Georgia"
   dbms_output.put_line(l_member_georgia.get_string('name'));
   dbms_output.put_line(treat(l_team.get(0) as
json_object_t).get_string('name'));
end;
```

Tip

When calling object functions on JSON types, ensure the variable is not null. Calling an object function on a null produces the error message:

```
ORA-30625: method dispatch on NULL SELF argument is disallowed
```


Copyright © 2026, Oracle and/or its affiliates.

14.2.4.4 Serializing JSON in PL/SQL

Serialize PL/SQL JSON objects to text with `STRINGIFY` or `TO_CLOB`, depending on size.

For a variable of type `JSON_OBJECT_T`, two object functions let you "serialize" its data to textual format:

- `STRINGIFY` – when the text does not exceed 32767 characters, or
- `TO_CLOB` – for a JSON object of any size.

The following example containing a JSON object with a property having a value of length 32767 shows how `TO_CLOB` works correctly. However, uncommenting the call to `STRINGIFY` produces an error since the JSON object's text representation exceeds 32767 characters.

```
declare
    l_value varchar2(32767) := lpad('y',32767,'x');
    l_json  json_object_t   := json_object_t();
    ------------------------------------------------------
    procedure p(p_clob in clob)
    is
        l_offset integer := 1;
        l_chunk  varchar2(32767);
    begin
        while l_offset <= dbms_lob.getlength(p_clob) loop
        l_chunk := dbms_lob.substr(p_clob, 32767, l_offset);
        dbms_output.put_line(l_chunk);
        l_offset := l_offset + 32767;
        end loop;
    end p;
begin
  l_json.put('name',l_value);
  -- Gives ORA-40478: output value too large
  -- p(l_json.stringify);
  p(l_json.to_clob);
end;
```

> **Tip**
>
> When calling object functions on JSON types, ensure the variable is not null. Calling an object function on a null produces the error message:
>
> ```
> ORA-30625: method dispatch on NULL SELF argument is disallowed
> ```

14.3 Planning Your ORDS Service Module

Plan your ORDS service module schema alias, URL templates, and handlers.

Oracle REST Data Services (ORDS) lets you define web APIs using SQL and PL/SQL. Using Oracle 26ai's multilingual engine (MLE), you can also define them with server-side JavaScript.

Copyright © 2026, Oracle and/or its affiliates.

- Choosing a Schema Alias
  Choose a schema alias to define the leading segment of your ORDS service URLs.
- Identifying URL Templates in a Module
  Define URL templates in a module to identify the service endpoints clients call.
- Choosing GET Handlers to Read Data
  Use GET handlers to expose read-only access for an entity collection and a single entity.
- Creating a View for GET Handlers
  Create a view that returns action item data with nested team members for GET handlers.

14.3.1 Choosing a Schema Alias

Choose a schema alias to define the leading segment of your ORDS service URLs.

When you enable your schema to start creating APIs, you configure a *schema alias* to set the name that appears in your service URLs. For example, if your schema were named `WKSP_COMPANION`, you might choose `cloudcompanion` as its schema alias. When clients call your web apis, the URL begins with:

```
https://example.com/ords/cloudcompanion/...
```

14.3.2 Identifying URL Templates in a Module

Define URL templates in a module to identify the service endpoints clients call.

You specify service endpoints using *templates* to identify the app-specific part of the URL a client uses. Then, you configure a *handler* for one or more HTTP methods for each template. *Before* you define templates, you first create a *module* to contain them. You can name the module anything you like, but say you call it `v0`.

Then, in your `v0` module, you can use URL templates like:

- `/actionitems` – for working with action items in general, and
- `/actionitems/:id` – to work with a specific action item by unique identifier.

The URL a client app uses to call your web API starts with your domain, followed first by `/ords/schemaalias`, then by your module name and template. For example, to work with action items in general, the remote application uses the URL:

```
https://example.com/ords/cloudcompanion/v0/actionitems
```

To work instead with the specific action item with ID 1234, the client uses this URL instead:

```
https://example.com/ords/cloudcompanion/v0/actionitems/1234
```

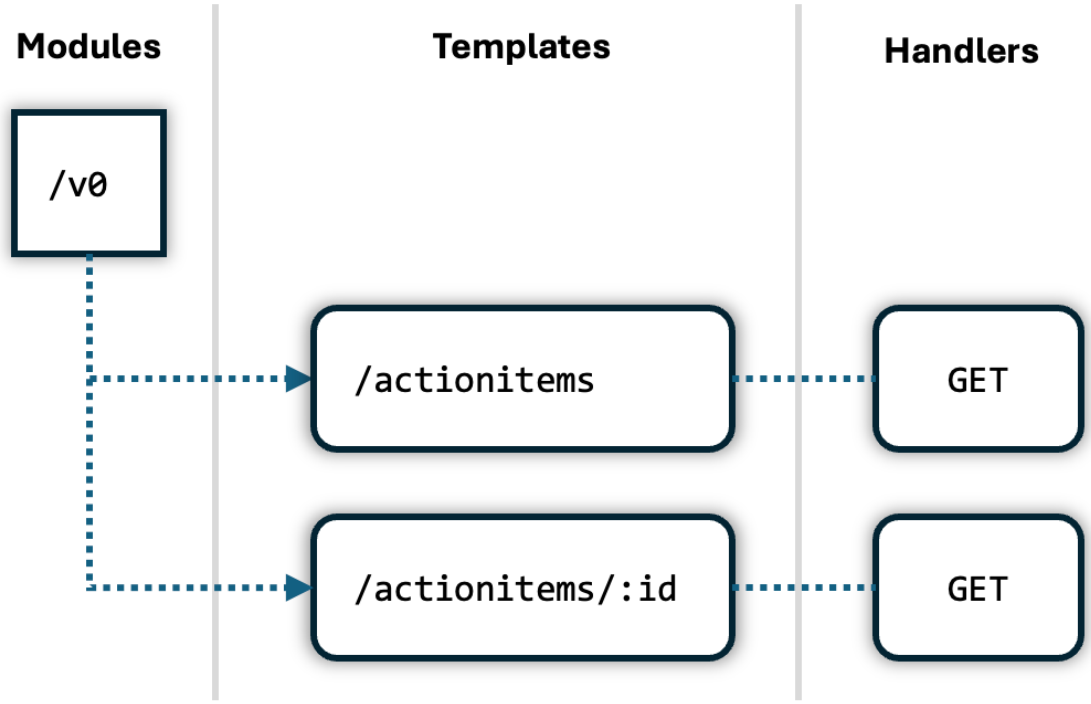
14.3.3 Choosing GET Handlers to Read Data

Use GET handlers to expose read-only access for an entity collection and a single entity.

Start by assuming your integration requires read-only access to action items. The HTTP `GET` method is the appropriate one to handle this case. The figure below shows the `v0` ORDS module, with its two URL path templates, and a `GET` method handler for each template.

Copyright © 2026, Oracle and/or its affiliates.

Figure 14-4 ORDS "v0" Module with Templates and Handlers for Action Items API



14.3.4 Creating a View for GET Handlers

Create a view that returns action item data with nested team members for GET handlers.

Start by defining a view the `/actionitems` template's `GET` handlers can use. This gives a meaningful name like `ACTION_ITEMS_WITH_TEAM_JV` to the query that retrieves the exact action item data you want the handler to return. Select top-level `ACTION_ITEMS` columns you want to include, then use a subquery with `JSON_ARRAYAGG` and `JSON_OBJECT` as shown below to include the JSON array of nested `ACTION_ITEM_TEAM_MEMBERS` data. Notice how the subquery joins this table with `STAFF` to include the staff member's name in the result. Your ORDS `GET` handlers can now reference this query to simplify their definition.

```
create view action_items_with_team_jv as
select ai.id,
       ai.name,
       ai.status,
       (select json_arrayagg(
                 json_object(
                   'team_member_id' value tm.id,
                   'name'           value s.name,
                   'user_id'        value tm.user_id,
                   'role'           value tm.role
                 )
               )
          from action_item_team_members tm
          join staff s on tm.user_id = s.id
         where tm.action_id = ai.id
```


Copyright © 2026, Oracle and/or its affiliates.

```
        ) as team
from action_items ai
```

14.4 Using SQL Developer Web REST Designer

Use SQL Developer Web REST Designer to create and navigate ORDS service definitions.

Access *SQL Developer Web* directly from App Builder to define your ORDS modules, templates, and handlers.

- Accessing SQL Developer Web REST Designer
  Open SQL Developer Web from App Builder to use its REST API designer.
- Navigating ORDS Service Definitions
  Navigate from modules to templates to handlers in SQL Developer Web REST Designer.

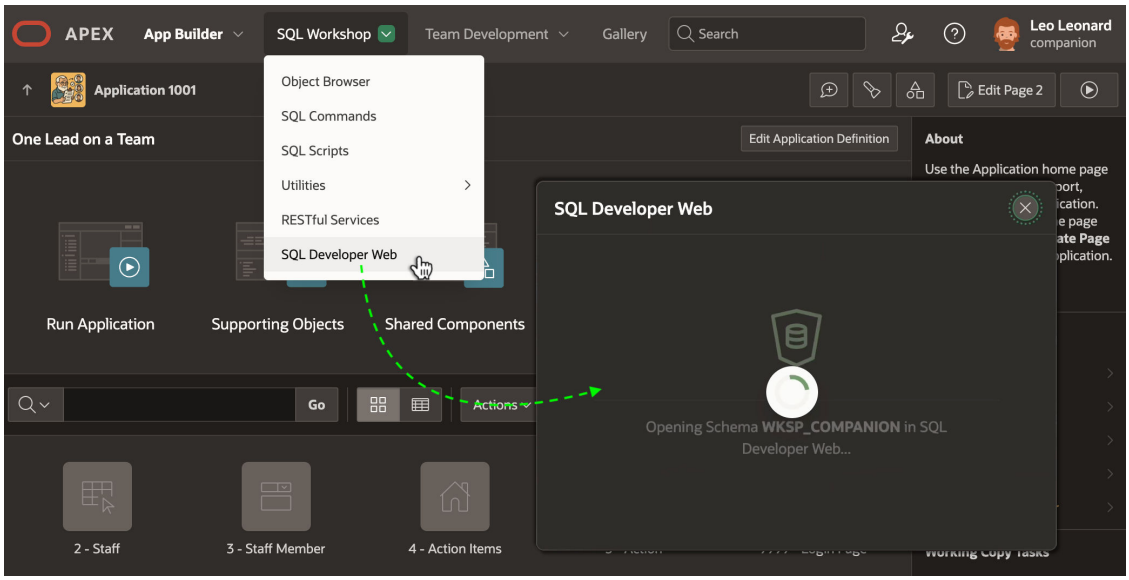
14.4.1 Accessing SQL Developer Web REST Designer

Open SQL Developer Web from App Builder to use its REST API designer.

After enabling your workspace schema for web API development, as shown below, launch **SQL Developer Web** any time from the *SQL Workshop* menu.

> **Tip**
>
> If nothing happens, your browser may be blocking pop-up windows. After allowing them for your APEX instance's domain, repeat the menu selection to try again.

Figure 14-5 Opening SQL Developer Web from the App Builder

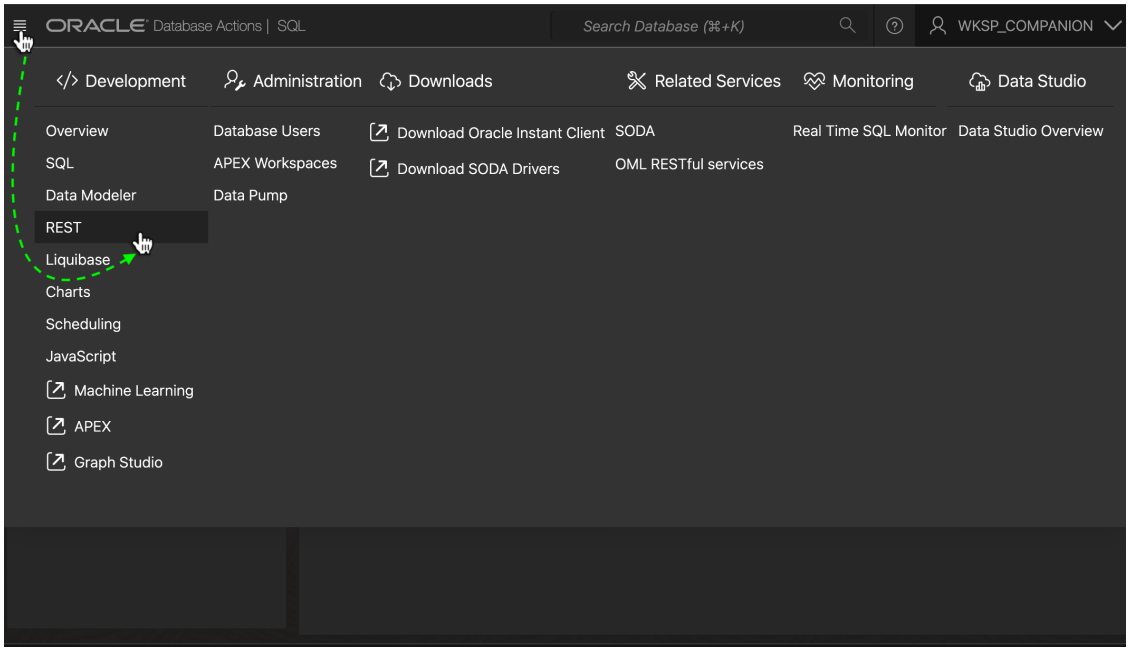


Copyright © 2026, Oracle and/or its affiliates.

> **⚠ Caution**
>
> If the *SQL Developer Web* entry is missing from your *SQL Workshop* menu, ask your APEX instance administrator to enable *SQL Developer Web* access in Administration Services.

When launched from App Builder, SQL Developer Web is connected to your workspace schema. As shown below, access the REST API designer using the *REST* entry on the "hamburger" menu.

Figure 14-6 Navigating to View and Define REST Services in SQL Developer Web



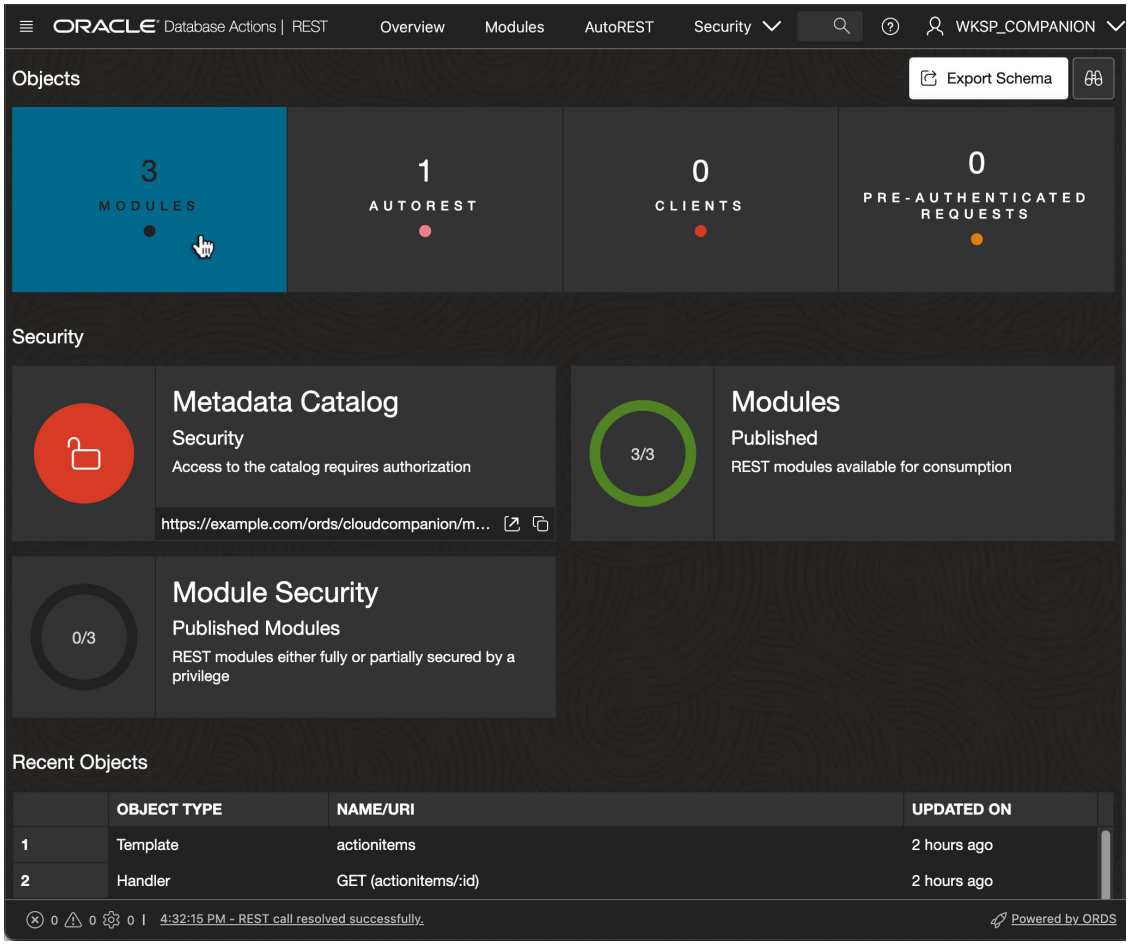
14.4.2 Navigating ORDS Service Definitions

Navigate from modules to templates to handlers in SQL Developer Web REST Designer.

When you access SQL Developer Web's REST Designer, as shown below, you start on the landing page. The dashboard at the top shows the count of your modules and other information. Click on the *Modules* tile to see the module list.

Copyright © 2026, Oracle and/or its affiliates.

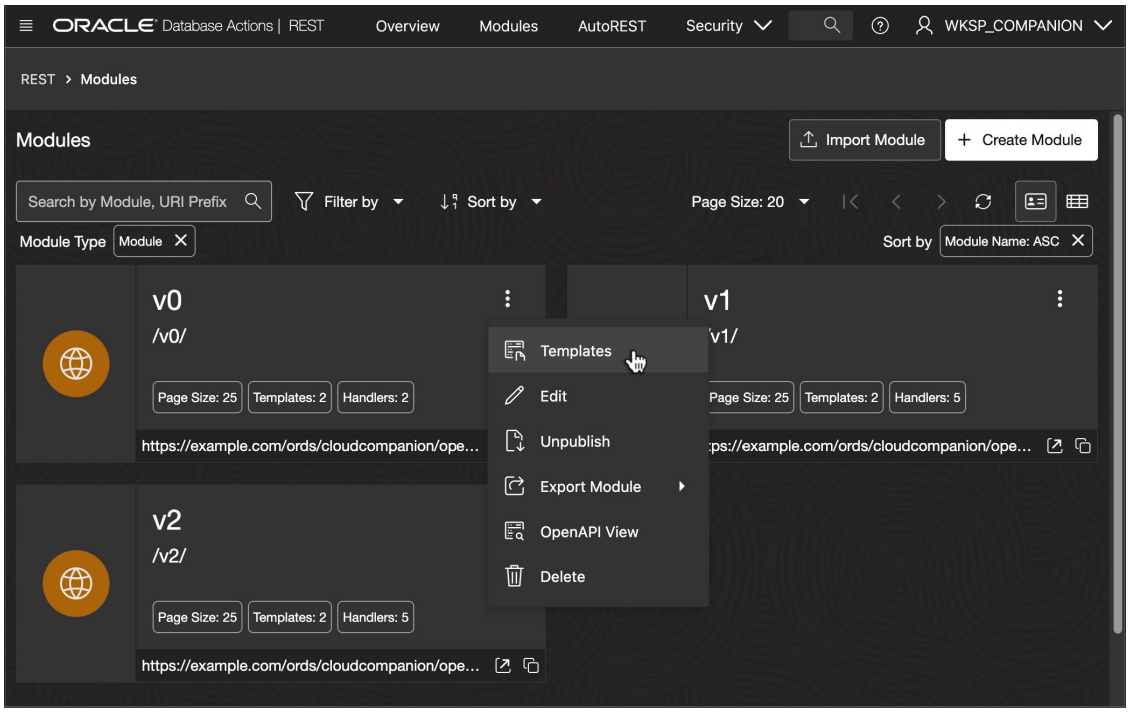
Figure 14-7 Navigating to View and Define REST Modules in SQL Developer Web



The *Modules* list displays a tile for each, with summary information. From here you can create a new module, or use the three-vertical-dots menu as shown below to access an existing module's list of *Templates*.

Copyright © 2026, Oracle and/or its affiliates.

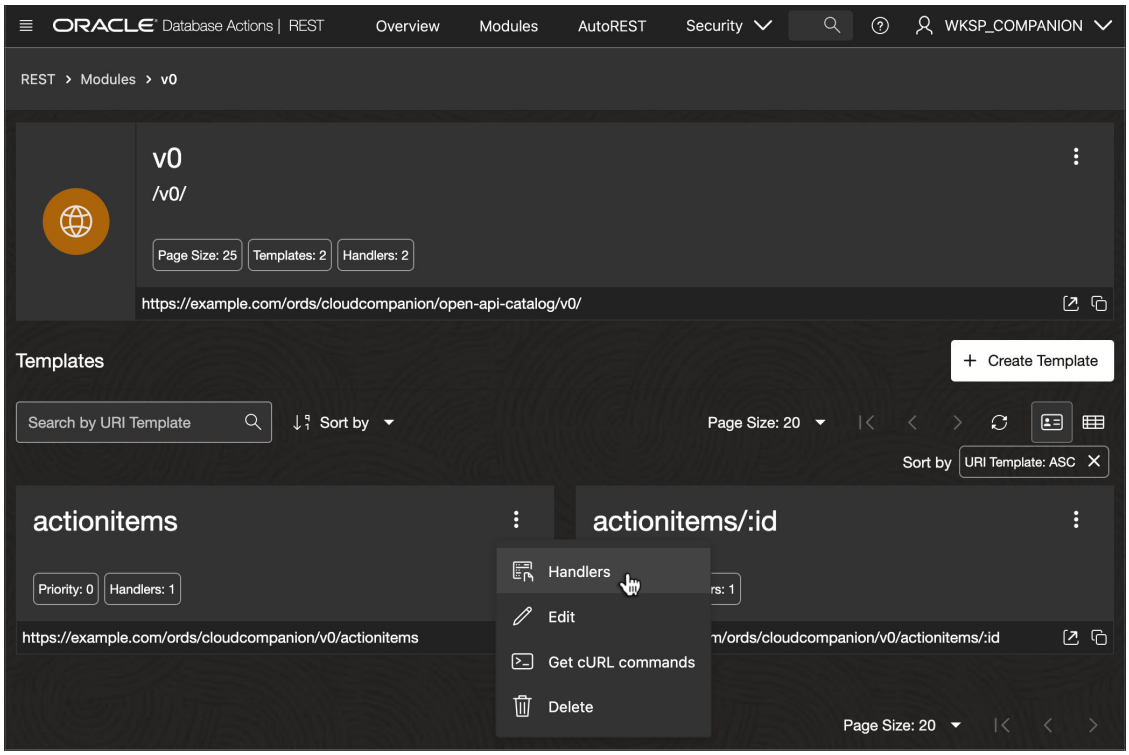
Figure 14-8 Navigating to View and Configure a Module's Templates in SQL Developer Web



The *Templates* list displays a tile for each, with the service URL and a button to quickly copy it to the clipboard. From here, you can create a new template or use the three-vertical-dots menu as shown below to access an existing template's list of *Handlers*.

Copyright © 2026, Oracle and/or its affiliates.

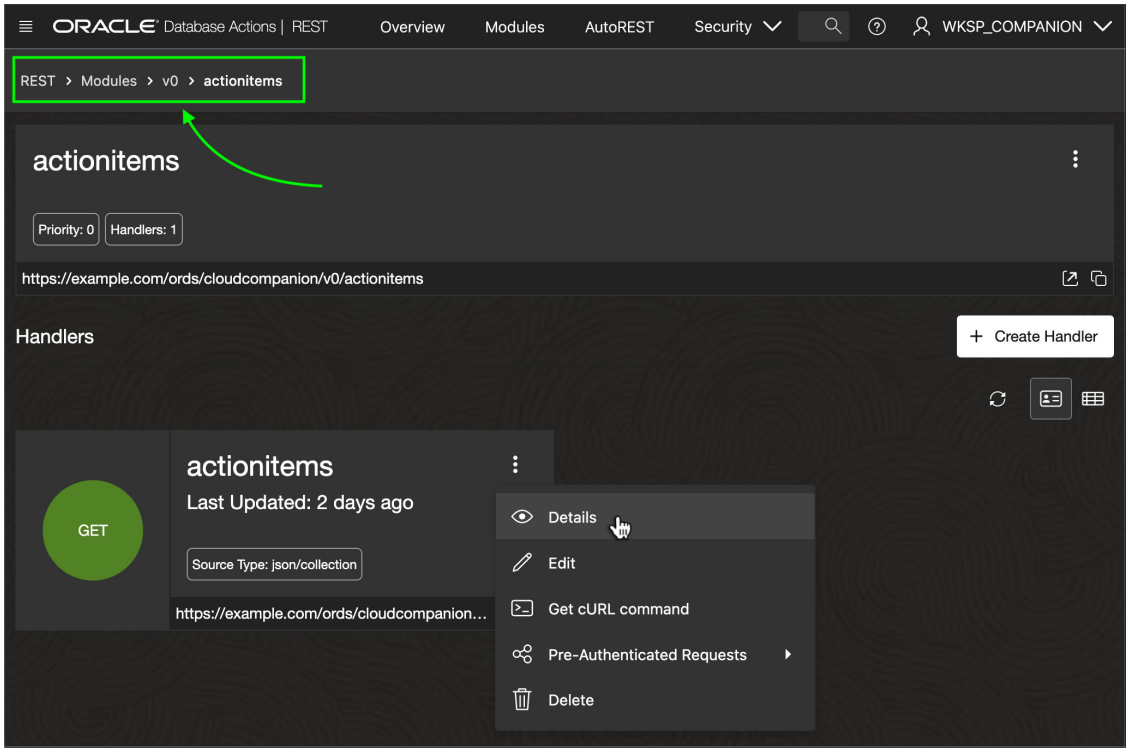
Figure 14-9 Navigating to View and Configure a Template's Handlers in SQL Developer Web



The *Handlers* list displays a tile for each. Use the three-vertical-dots menu as shown below to view or edit the handler's details. Also notice the "breadcrumb" bar with a hierarchy of links you can click to return to the `v0` module's list of templates, the *Modules* list, or the *REST* designer's landing page.

Copyright © 2026, Oracle and/or its affiliates.

Figure 14-10 Listing Handlers for a Template in SQL Developer Web



14.5 Creating and Testing Read-Only APIs

Create and test read-only REST APIs for collections and single rows.

When your use case calls for *reading* data, you can build REST APIs using SQL queries. A *Collection Query* handler lets clients filter, sort, and page through the results, while a *Collection Item* handler returns a single row. You can *test* your REST APIs using `curl` on the command line or tools with a graphical interface like *Postman*.

- Defining a Read-Only Collection Query
  Define a Collection Query GET handler that returns action items with nested team data.
- Testing Handler with curl or Postman
  Test a GET endpoint with `curl` and `jq` or with Postman to inspect the JSON response.
- Filtering/Paging Collection Query Endpoints
  Use query string parameters to filter and page through Collection Query results.
- Configuring a Collection Item Handler
  Define a *Collection Item* `GET` handler that returns one action item, given its unique id.

14.5.1 Defining a Read-Only Collection Query

Define a Collection Query GET handler that returns action items with nested team data.

Define the ORDS template handler for the `GET` method on the `/actionitems` template using the *Source Type* of *Collection Query*. This handler type uses a SQL query to specify the data to return. Having already created the `ACTION_ITEMS_WITH_TEAM_JV` view, the SQL is very simple as shown below. Just select the top-level attributes you want to include in the result. Notice two column *aliases* in use. They affect the JSON response:

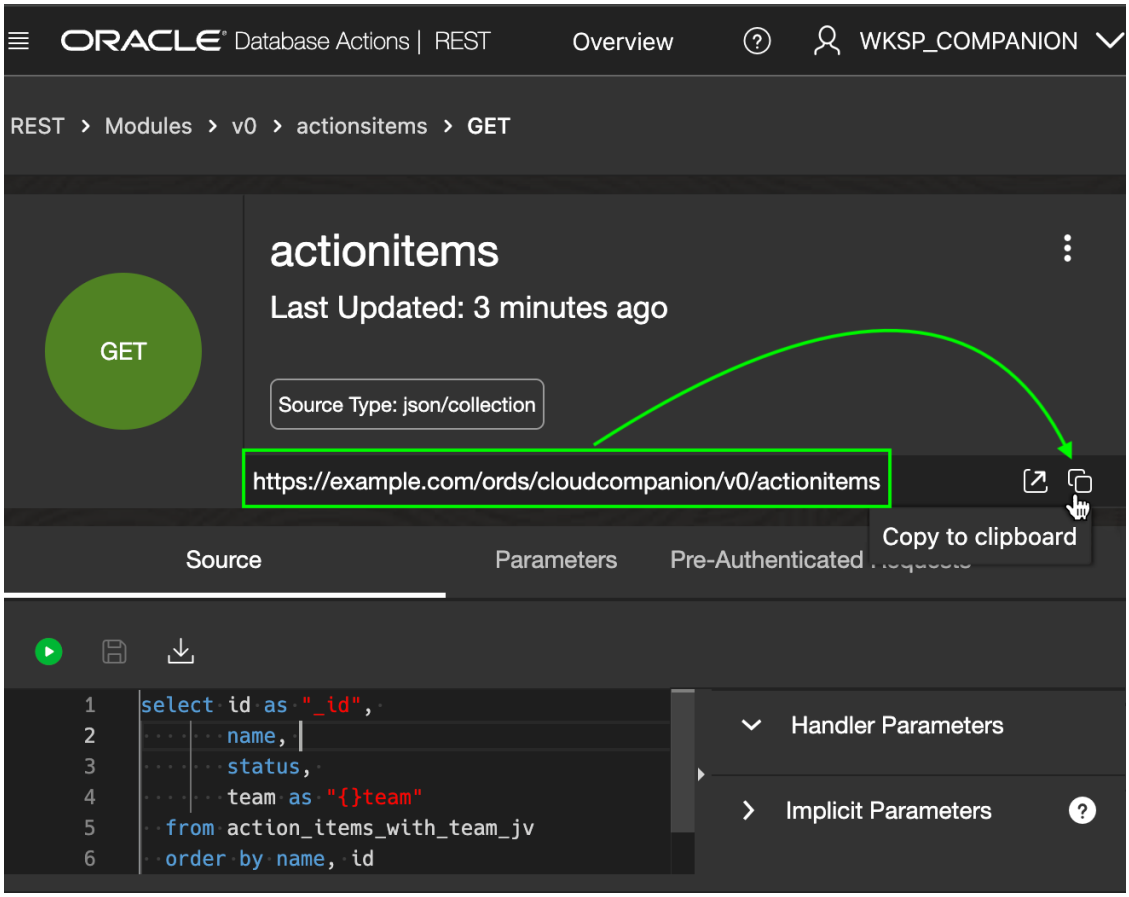
Copyright © 2026, Oracle and/or its affiliates.

- "_id" – ensures the ID attribute has the JSON property name _id
- "{}team" – Uses {} to indicate the team property is *already* in JSON format

```
select id as "_id",
       name,
       status,
       team as "{}team"
  from action_items_with_team_jv
order by name, id
```

Once defined, your collection query handler for the v0 module's /actionitems template is ready to test out. Click the *Copy to Clipboard* button as shown below to paste its URL into your favorite testing tool.

Figure 14-11 Defining a Collection Query GET Handler for /v0/actionitems Template



14.5.2 Testing Handler with curl or Postman

Test a GET endpoint with curl and jq or with Postman to inspect the JSON response.

Test your endpoint from the command line using curl. By default this utility uses the HTTP GET method, so the command is very simple:

```
$curl https://example.com/ords/cloudcompanion/v0/actionitems
```


Copyright © 2026, Oracle and/or its affiliates.

JSON parsers ignore whitespace, so as shown below, ORDS returns data without extra spaces or newlines. On your terminal, the JSON wraps across many lines.

Figure 14-12 Testing ORDS GET Handler on the Command Line with curl



Pipe `curl` output into `jq` using `|` to format it. It indents and color-codes the result for easier reading.

```
$ curl https://example.com/ords/cloudcompanion/v0/actionitems | jq
```


Copyright © 2026, Oracle and/or its affiliates.

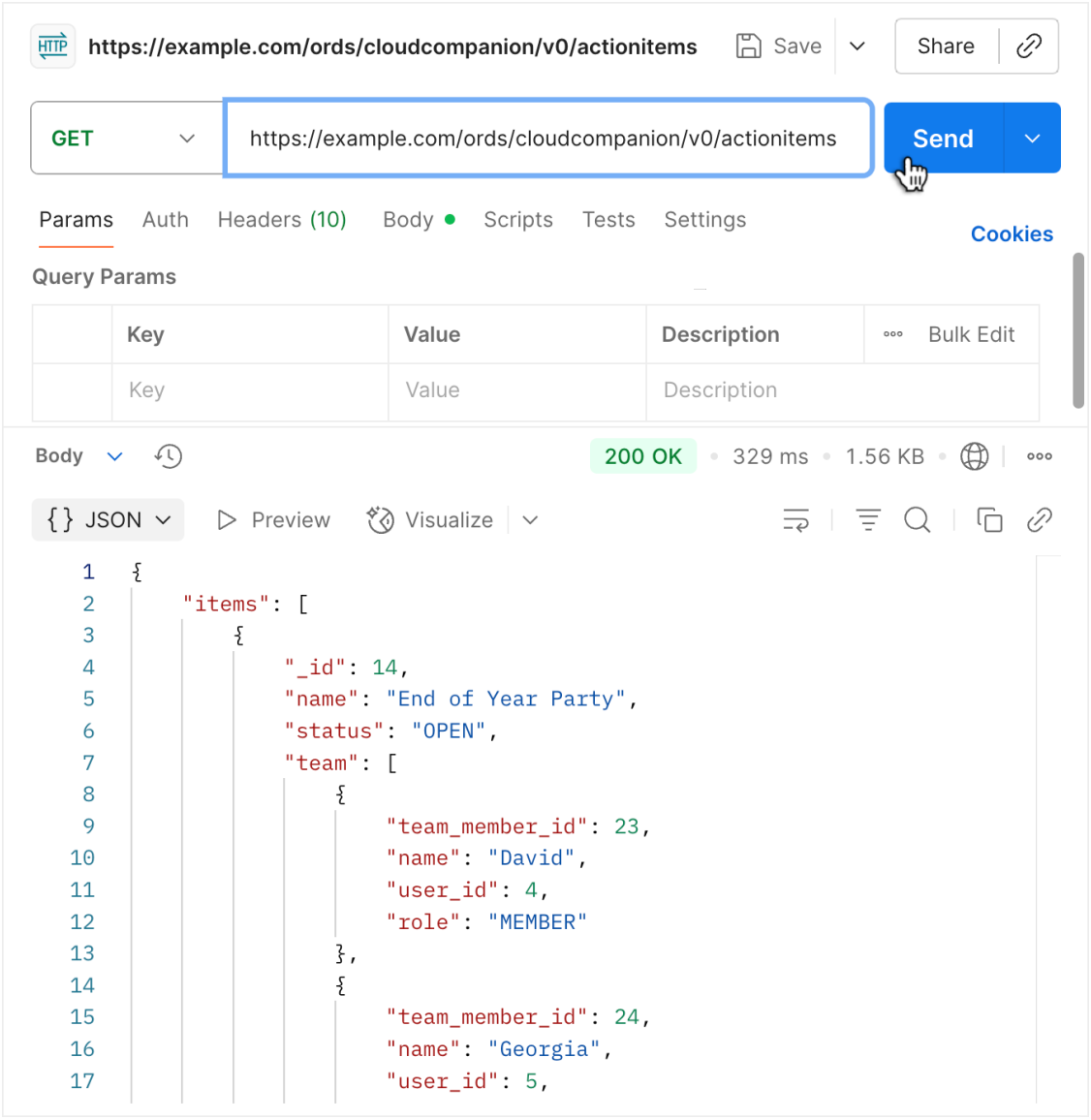
Figure 14-13 Formatting Output of curl Using jq Command Line Utility



Postman is another popular tool developers use for experimenting with web APIs. As shown below, pick the HTTP method and enter the URL, then click *(Send)* to see the nicely-formatted response.

Copyright © 2026, Oracle and/or its affiliates.

Figure 14-14 Using Postman to Access Your New Action Items REST API



14.5.3 Filtering/Paging Collection Query Endpoints

Use query string parameters to filter and page through Collection Query results.

ORDS *Collection Query* `GET` handlers support query filtering using the `q` query string parameter, whose value is an ORDS JSON filter predicate. For example, testing a `GET` request with the following URL finds action items whose name contains the word `Party`:

```
$ curl https://example.com/ords/v0/actionitems?q={"name":{"$like":"%Party%"}}
```

The response looks like the following if just one action item in the system matches.

```
{
    "items": [
        {
```


Copyright © 2026, Oracle and/or its affiliates.

```
                "_id": 14,
                "name": "End of Year Party",
                "status": "OPEN",
                "team": [
                    {
                        "team_member_id": 23,
                        "name": "David",
                        "user_id": 4,
                        "role": "MEMBER"
                    },
                    {
                        "team_member_id": 24,
                        "name": "Georgia",
                        "user_id": 5,
                        "role": "LEAD"
                    },
                    {
                        "team_member_id": 25,
                        "name": "Jane",
                        "user_id": 13,
                        "role": "MEMBER"
                    }
                ]
            }
    ],
    "hasMore": false,
    "limit": 25,
    "offset": 0,
    "count": 1,
    "links": [
      ▒
    ]
}
```

Notice the following properties in the JSON response:

```
{
      ▒
    "hasMore": false,
    "limit": 25,
    "offset": 0,
    "count": 1,
      ▒
}
```

An ORDS Collection Query handler's response includes the following top-level properties:

- `count` – rows in this response "pageful"
- `limit` – maximum rows in a single response, also called the "page size"
- `offset` – rows that were skipped before returning the current "page"
- `hasMore` – `true` if there are more rows to retrieve, `false` otherwise

ORDS Collection Query GET handlers support paging through data by adding the query string parameters:

Copyright © 2026, Oracle and/or its affiliates.

- `limit` – maximum rows in a single response, also called the "page size"
- `offset` – rows that were skipped before returning the current "page"

The default value of `offset` is `0` if not specified, so to fetch the first 10 actions items, use:

```
$ curl https://example.com/ords/v0/actionitems?limit=10
```

If the response includes a `hasMore` value of `true`, then get the second page of 10 action items with:

```
$ curl https://example.com/ords/v0/actionitems?offset=10&limit=10
```

> **Tip**
>
> When configuring Collection Query `GET` handlers, always include an `ORDER BY` clause in your query to ensure a predictable result when paging.

You can combine filtering with paging, too. For example, to retrieve the first 5 action items whose name starts with `Feature:`, use:

```
$ curl https://ords/v0/actionitems?limit=5&q={"name":{"$like":"Feature:%"}}
```

If the response includes a `hasMore` value of `true`, then get the second page of 5 action items with:

```
$ curl https://ords/v0/actionitems?offset=5&limit=5&q={"name":
{"$like":"Feature:%"}}
```

14.5.4 Configuring a Collection Item Handler

Define a *Collection Item* `GET` handler that returns one action item, given its unique id.

Define the ORDS template handler for the `GET` method on the `/actionitems/:id` template using the *Source Type* of *Collection Item*. This handler type's SQL query returns a single row by referencing a value provided in the URL. The `:id` in the template defines an implicit variable. As shown below, your query uses its value with bind variable notation. Then, select the top-level attributes you want to include in the result.

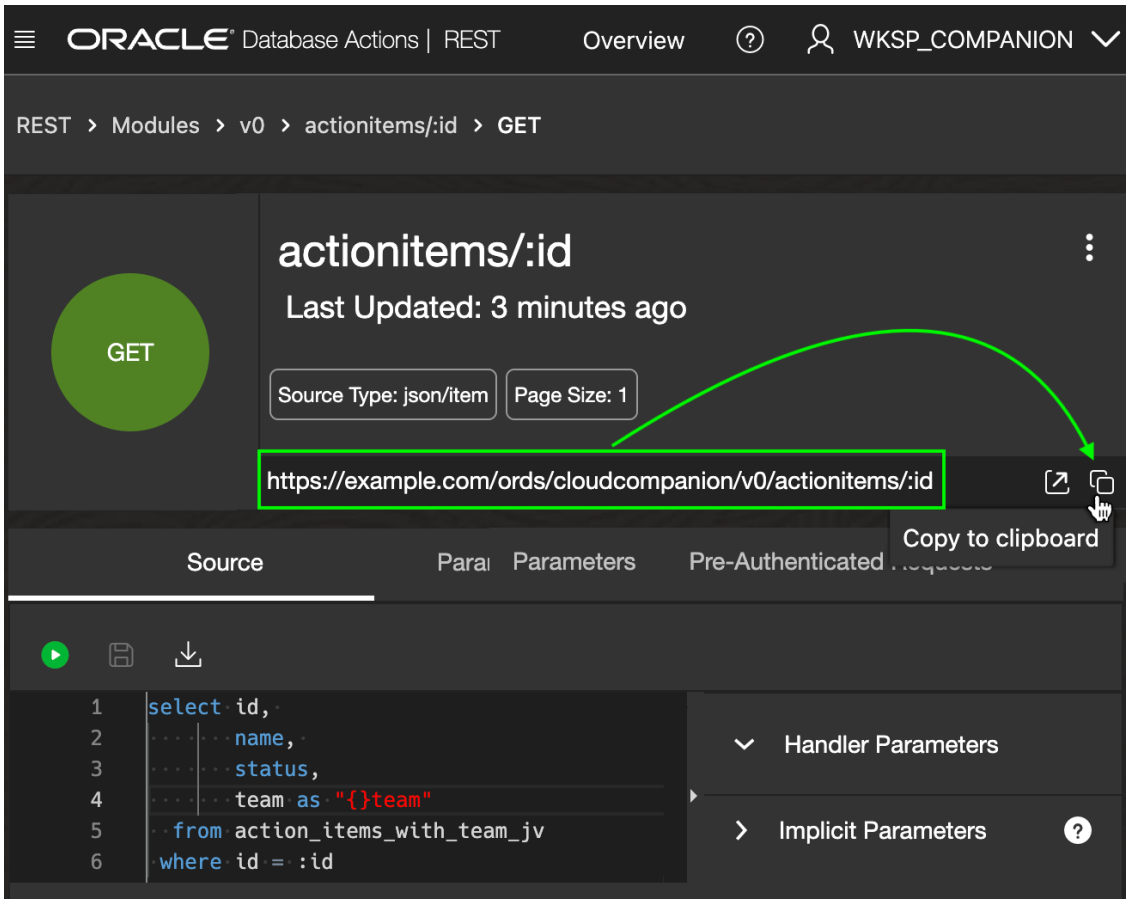
```
select id as "_id",
       name,
       status,
       team as "{}team"
  from action_items_with_team_jv
 where id = :id
```


Copyright © 2026, Oracle and/or its affiliates.

> **Tip**
>
> Notice two column *aliases* in use. They affect the JSON response:
>
> - `"_id"` – ensures the ID attribute has the JSON property name `_id`
> - `"{}team"` – Uses `{}` to indicate the `team` property is *already* in JSON format

Once defined, your collection item handler for the `v0` module's `/actionitems/:id` template is ready to test out. Click the *Copy to Clipboard* button as shown below to paste its URL into your favorite testing tool.

Figure 14-15 Defining an Collection Item GET Handler for /v0/actionitems/:id Template



For example, using `curl` to request action item number 14 produces the response below. Since the result is a *single* action item, notice the response has:

- No `items` array
- No pagination properties: `count`, `hasMore`, `limit`, or `offset`.

```
$ curl https://example.com/ords/cloudcompanion/v0/actionitems/14 | jq
{
    "id": 14,
    "name": "End of Year Party",
```


Copyright © 2026, Oracle and/or its affiliates.

```
    "status": "OPEN",
    "team": [
        {
            "team_member_id": 23,
            "name": "David",
            "user_id": 4,
            "role": "MEMBER"
        },
        {
            "team_member_id": 24,
            "name": "Georgia",
            "user_id": 5,
            "role": "LEAD"
        },
        {
            "team_member_id": 25,
            "name": "Jane",
            "user_id": 13,
            "role": "MEMBER"
        }
    ],
    "links": [
        {
            "rel": "collection",
            "href": "https://example.com/ords/cloudcompanion/v0/actionitems/"
        }
    ]
}
```


14.6 Enforcing Business Logic in REST APIs

Apply the same business rules to REST API changes that your APEX pages use.

If you let an outside system *change* your application data, you *must* apply the same validation and defaulting business logic used in your web user interface to keep bad data out of your system of record.

- Reviewing Rules for Data Changes
  Review the defaulting and validation rules the example app enforces for action item teams.
- Defining Text Messages for Errors
  Define Text Messages for translatable errors your pages and PL/SQL can share.
- Using Text Messages in Pages and PL/SQL
  Use Text Message substitution syntax in pages and `APEX_LANG.GET_MESSAGE` in PL/SQL.
- Handling Errors in a Common Way
  Centralize error lookup, reporting, and logging so pages and APIs handle errors consistently.
- Coding Business Rules for Reuse
  Put shared business rules in a package so APEX pages and REST APIs can reuse the same logic.
- Experiencing Validation Failure in UI
  Show validation failures in the UI using the same shared business logic your REST APIs use.

Copyright © 2026, Oracle and/or its affiliates.

- Receiving Validation Error in REST API
  REST API clients see the same validation errors that end users see in the app.

14.6.1 Reviewing Rules for Data Changes

Review the defaulting and validation rules the example app enforces for action item teams.

For any action item and its related action item team, your application enforces the following business rules:

- If no role is specified, default it to `MEMBER`
- Team member role on an action item team must be either `LEAD` or `MEMBER`
- Each action item team *must* have a lead
- Only one lead is allowed per action item team
- No duplicate staff members are allowed on the same team.

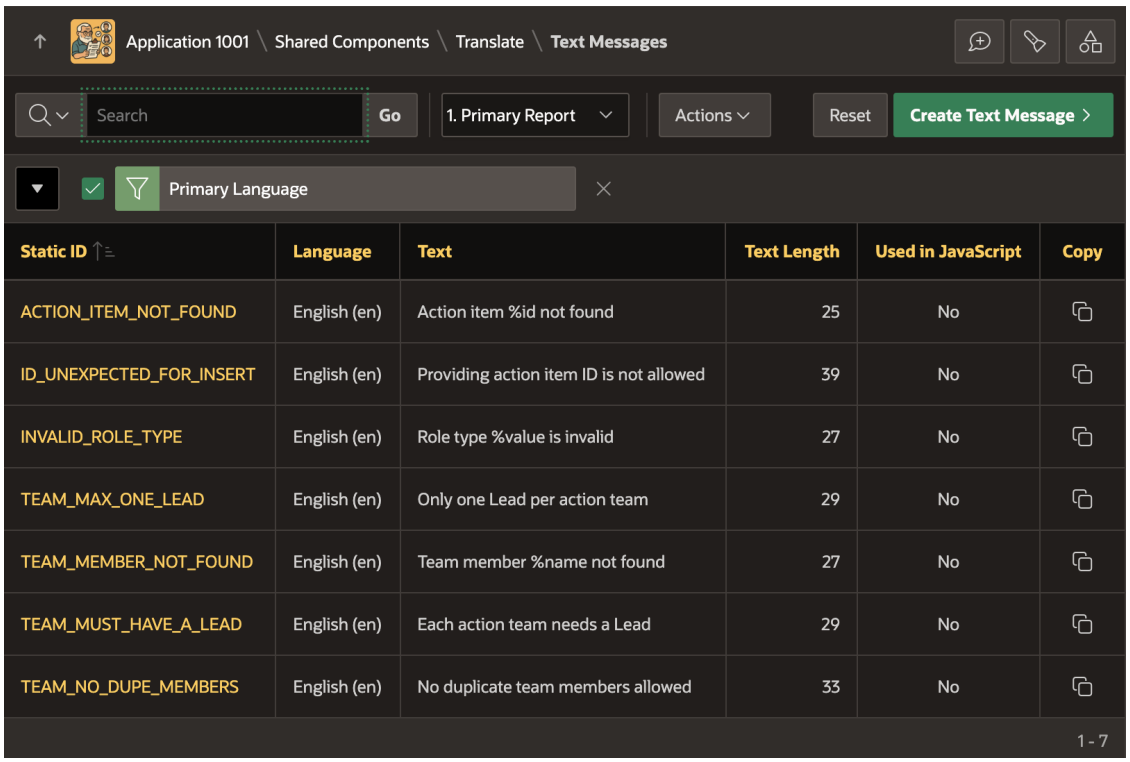
14.6.2 Defining Text Messages for Errors

Define Text Messages for translatable errors your pages and PL/SQL can share.

To make user-visible error messages in your business logic translatable, create shared component *Text Messages* for each one. As shown below, each has a:

- Static ID (e.g. `TEAM_MEMBER_NOT_FOUND`),
- language code (e.g. `en`), and
- text that can include named substitution tokens (e.g. `%name`).

Figure 14-16 Shared Component Text Messages Define Translatable Error Messages



Copyright © 2026, Oracle and/or its affiliates.

Your app pages and process messages access a string using this substitution syntax:

```
&{TEAM_MEMBER_NOT_FOUND}.
```

In PL/SQL code, `GET_MESSAGE` in the `APEX_LANG` package gets a string by its static ID in the user's language. If the message includes named tokens, pass an `APEX_T_VARCHAR2` string list with each token name followed by its string value.

```
l_message := apex_lang.get_message(
                p_name   => 'TEAM_MEMBER_NOT_FOUND',
                p_params => apex_t_varchar2('name',l_team_member.name));
```

14.6.3 Using Text Messages in Pages and PL/SQL

Use Text Message substitution syntax in pages and `APEX_LANG.GET_MESSAGE` in PL/SQL.

Your app pages and process messages access a string with its Statid ID using this substitution syntax:

```
&{TEAM_MAX_ONE_LEAD}.
```

If the message includes named tokens, then provide their values like this:

```
&{TEAM_MAX_ONE_LEAD name="&P3_NAME."}.
```

In PL/SQL code, `GET_MESSAGE` in the `APEX_LANG` package gets a string by its unique name. If the message includes named tokens, pass an `APEX_T_VARCHAR2` string list with each token name followed by its string value.

```
l_message := apex_lang.get_message(
                p_name   => 'TEAM_MEMBER_NOT_FOUND',
                p_params => apex_t_varchar2('name',l_team_member.name));
```

14.6.4 Handling Errors in a Common Way

Centralize error lookup, reporting, and logging so pages and APIs handle errors consistently.

Use a package like `APP_COMMON` below to centralize application-wide common error handling logic. This simplifies reporting and logging errors from a single location. Key functions include:

- `error_message` – to return an error message in the current language based on a Static ID
- `report_if_error` – to optionally raise an error if the message passed in is not null
- `report_error_key` – to report an error in the current language based on a Static ID

```
package app_common is

    c_workspace constant varchar2(40) := 'COMPANION';
    c_app_id    constant number       := 1001;

    ------------------------------------------------------------
    -- Return error message with indicated key
    ------------------------------------------------------------
```


Copyright © 2026, Oracle and/or its affiliates.

```
    function error_message(
        p_message_key in varchar2,
        p_params      in apex_t_varchar2 default null)
        return           varchar2;
    ------------------------------------------------------------
    -- Report error message with indicated behavior if non-null
    ------------------------------------------------------------
    procedure report_if_error(
        p_error_message in varchar2);
    ------------------------------------------------------------
    -- Report error message with indicated message key
    ------------------------------------------------------------
    procedure report_error_key(
        p_error_message_key in varchar2,
        p_params            in apex_t_varchar2 default null);
end app_common;
```

The `APP_COMMON` package body appears below. Notice its `is_apex_call` that detects whether the error is being reported while running in the context of an APEX application or not. The `report_if_error` procedure uses this to decide whether to add the error to the APEX error stack using `APEX_ERROR.ADD_ERROR`, or whether to raise an exception containing the error message instead. The code uses `APEX_LANG.GET_MESSAGE` to get the error message by Static ID in the user's current language.

```
package body app_common is
    ------------------------------------------
    function is_apex_call
    return boolean
    is
    begin
        return apex_application.g_flow_id is not null;
    end is_apex_call;
    ------------------------------------------
    function error_message(
        p_message_key in varchar2,
        p_params      in apex_t_varchar2 default null)
        return           varchar2
    is
    begin
        apex_util.set_workspace(c_workspace);
        return apex_lang.get_message(
                p_name           => p_message_key,
                p_params         => p_params,
                p_application_id => c_app_id);
    end error_message;
    ------------------------------------------
    procedure report_if_error(
        p_error_message in varchar2)
    is
        l_error varchar2(4000);
    begin
        if p_error_message is not null then
            if is_apex_call then
                apex_error.add_error (
                    p_message          => p_error_message,
```


Copyright © 2026, Oracle and/or its affiliates.

```
                    p_display_location => apex_error.c_on_error_page);
            else
                raise_application_error(-20001,p_error_message);
            end if;
        end if;
    end report_if_error;
    ------------------------------------------
    procedure report_error_key(
        p_error_message_key in varchar2,
        p_params            in apex_t_varchar2 default null)
    is
    begin
        report_if_error(
            p_error_message => error_message(p_error_message_key,p_params));
    end report_error_key;
end app_common;
```

14.6.5 Coding Business Rules for Reuse

Put shared business rules in a package so APEX pages and REST APIs can reuse the same logic.

To reuse common *action item* business logic in your app and REST APIs, put it in a separate package. For example, `ACTION_ITEMS_COMMON` below has validation functions for action items. They return either a:

- `BOOLEAN` – with `false` representing invalid, or
- `VARCHAR2` – with `null` representing success, or an error message on failure.

The validation checks include functions:

- `action_exists` – checks whether an action items with a given ID exists
- `ensure_null_id_for_new_action` – asserts a new action item's ID is null
- `ensure_action_exists` – returns message if action does *not* exist
- `one_lead_per_action_team` – returns message if action item team has no leads or more than one lead
- `no_duplicate_team_members` – returns message if duplicate team members exists

```
package action_items_common is
    ------------------------------------------------------------------------
    -- Return true if the action item with provide id exists
    ------------------------------------------------------------------------
    function action_exists(
        p_action_id in number)
    return boolean;
    ------------------------------------------------------------------------
    -- Ensure id passed in is null, or return error message
    ------------------------------------------------------------------------
    function ensure_null_id_for_new_action(
        p_id   in number)
        return    varchar2;
    ------------------------------------------------------------------------
    -- Ensure action item exists, or return error message
    ------------------------------------------------------------------------
```


Copyright © 2026, Oracle and/or its affiliates.

```
    function ensure_action_exists(
        p_action_id in number)
    return varchar2;
    ------------------------------------------------------------------------
    -- Ensure action item has exactly one Lead, or return error message
    ------------------------------------------------------------------------
    function one_lead_per_action_team(
        p_action_id in number)
    return varchar2;
    ------------------------------------------------------------------------
    -- Ensure action item has no duplicate members, or return error message
    ------------------------------------------------------------------------
    function no_duplicate_team_members(
        p_action_id in number)
    return varchar2;
    ------------------------------------------------------------------------
    -- Ensure role value is legal, or return error message
    ------------------------------------------------------------------------
    function is_valid_role(
        p_role_value in varchar2)
        return          varchar2;
end action_items_common;
```

You can study the simple code that *implements* the variable functions and procedures below. Notice how validations can easily enforce conditions about the state of a parent Action Item and its child Action Item Team Members using simple SQL statements. Two key examples to review:

- To count team leads, `one_lead_per_action_team` uses: `COUNT(*)` with `WHERE ROLE='LEAD'`
- To find duplicate team members, `no_duplicate_team_members` uses: `GROUP BY` and `HAVING COUNT(*) > 1`

> **Tip**
>
> These kinds of simple SQL-based aggregate checks are possible because both your APEX application and REST API PL/SQL can perform checks ***after*** the data changes are saved but ***before*** they are committed. Signaling an error in either context is enough to rollback the transaction and let the user know the problem with a helpful error message in their current language.

```
package body action_items_common is
    ------------------------------------------------------------
    -- Constant for error message keys
    ------------------------------------------------------------
    c_msg_team_must_have_lead   constant varchar2(25) :=
'TEAM_MUST_HAVE_A_LEAD';
    c_msg_team_max_one_lead     constant varchar2(25) := 'TEAM_MAX_ONE_LEAD';
    c_msg_team_no_duplicates    constant varchar2(25) :=
'TEAM_NO_DUPE_MEMBERS';
    c_msg_action_item_not_found constant varchar2(25) :=
'ACTION_ITEM_NOT_FOUND';
    c_msg_illegal_role_value    constant varchar2(25) := 'INVALID_ROLE_TYPE';
```


Copyright © 2026, Oracle and/or its affiliates.

```
    c_msg_no_id_for_new_action  constant varchar2(25) :=
'ID_UNEXPECTED_FOR_INSERT';

    c_no_error                  constant varchar2(1)  := null;
    -----------------------------------
    function ensure_null_id_for_new_action(
        p_id   in number)
        return    varchar2
    is
    begin
        if p_id is not null then
            return app_common.error_message(c_msg_no_id_for_new_action);
        end if;
        return c_no_error;
    end ensure_null_id_for_new_action;
    -----------------------------------
    function ensure_action_exists(
        p_action_id in number)
    return varchar2
    is
    begin
        if not action_exists(p_action_id) then
            return app_common.error_message(c_msg_action_item_not_found,
                    apex_t_varchar2('id',p_action_id));
        end if;
        return c_no_error;
    end ensure_action_exists;
    -----------------------------------
    function action_exists(
        p_action_id in number)
    return boolean
    is
    begin
        for j in (select null
                    from action_items
                   where id = p_action_id)
        loop
            return true;
        end loop;
        return false;
    end action_exists;
    -----------------------------------
    function one_lead_per_action_team(
        p_action_id in number)
        return         varchar2
    is
        l_lead_count number;
    begin
        -- Ensure the action id still exists since it might
        -- have been deleted. If it exists, perform the check
        -- on the children rows.
        for j in (select id
                    from action_items
                   where id = p_action_id) loop
            select count(*)
            into l_lead_count
```


Copyright © 2026, Oracle and/or its affiliates.

```
            from action_item_team_members
            where action_id = p_action_id
            and role = 'LEAD';
            if l_lead_count != 1 then
                return app_common.error_message(
                      case
                           when l_lead_count < 1
                           then c_msg_team_must_have_lead
                           else c_msg_team_max_one_lead
                       end);
            end if;
        end loop;
        return c_no_error;
    end;

    -----------------------------------
    function no_duplicate_team_members(
        p_action_id in number)
        return         varchar2
    is
    begin
        -- Ensure the action id still exists since it might
        -- have been deleted. If it exists, perform the check
        -- on the children rows.
        for j in (select id
                    from action_items
                   where id = p_action_id) loop
            for k in (select user_id, count(*)
                        from action_item_team_members
                       where action_id = p_action_id
                       group by user_id
                       having count(*) > 1
                       fetch first row only) loop
                return app_common.error_message(c_msg_team_no_duplicates);
                exit;
            end loop;
        end loop;
        return c_no_error;
    end;

    -----------------------------------
    function is_valid_role(
        p_role_value in varchar2)
        return          varchar2
    is
    begin
        if p_role_value not in ('MEMBER','LEAD') then
            return app_common.error_message(
                    c_msg_illegal_role_value,
                    apex_t_varchar2('value',p_role_value));
        end if;
        return c_no_error;
    end is_valid_role;
end action_items_common;
```


Copyright © 2026, Oracle and/or its affiliates.

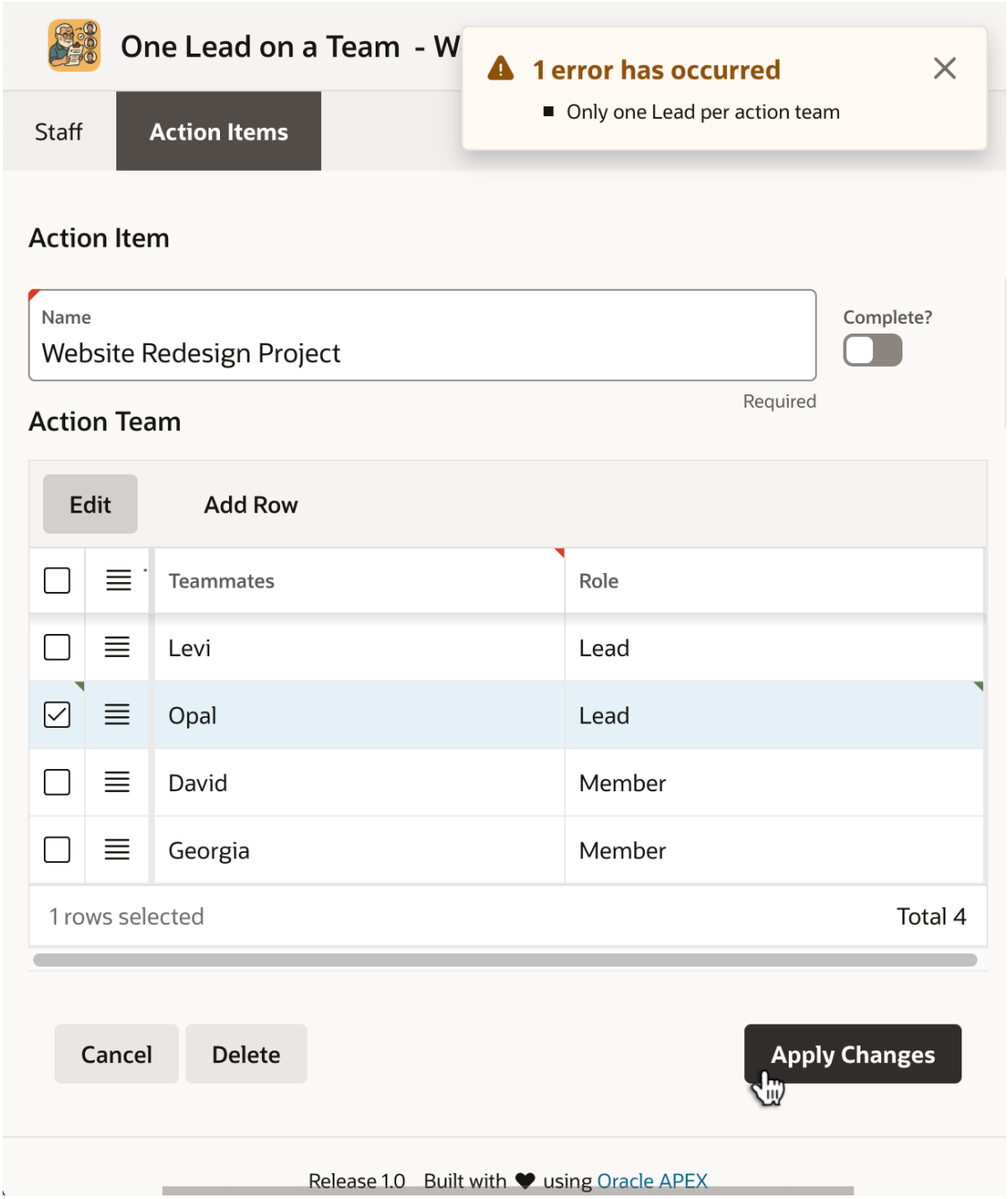
14.6.6 Experiencing Validation Failure in UI

Show validation failures in the UI using the same shared business logic your REST APIs use.

After you set up the shared Action Item logic in the `ACTION_ITEMS_COMMON` package, app pages use it declaratively through *Invoke API* page processes or validations. The figure below shows what users see when saving an action item team with two leads. In the figure, the user clicked (**Apply Changes**) on the Action Item edit page for the *Website Redesign Project*. The Action Team grid below shows both Levi and Opal as having the role of *Lead*. The error notification area appears with an orange triangle containing an exclamation point and the title "*1 error has occurred*". The application specific error appears below that as a bulleted message: "*Only one Lead per action team*".

Copyright © 2026, Oracle and/or its affiliates.

Figure 14-17 Editing an Action Item and Assigned Team Members in a Page



14.6.7 Receiving Validation Error in REST API

REST API clients see the same validation errors that end users see in the app.

Your REST APIs URL template handlers call common Action Item logic in the `ACTION_ITEMS_COMMON` package. If a REST API client tries to create a new action item with duplicate team members, the call fails. The behavior is the same for `curl`, Postman, or another system.

Copyright © 2026, Oracle and/or its affiliates.

As shown below, HTTP error `400` signals an application-specific failure, and the JSON response explains why. By using shared logic and error handling, REST clients see the same messages app users see for the same invalid data. The figure shows the result of using the Postman REST API testing tool to POST a request to the endpoint:

```
https://example.com/ords/cloudcompanion/v1/actionitems
```

The example request's body contains the following JSON payload where both Levi and Opal team members have the `role` of `LEAD`.

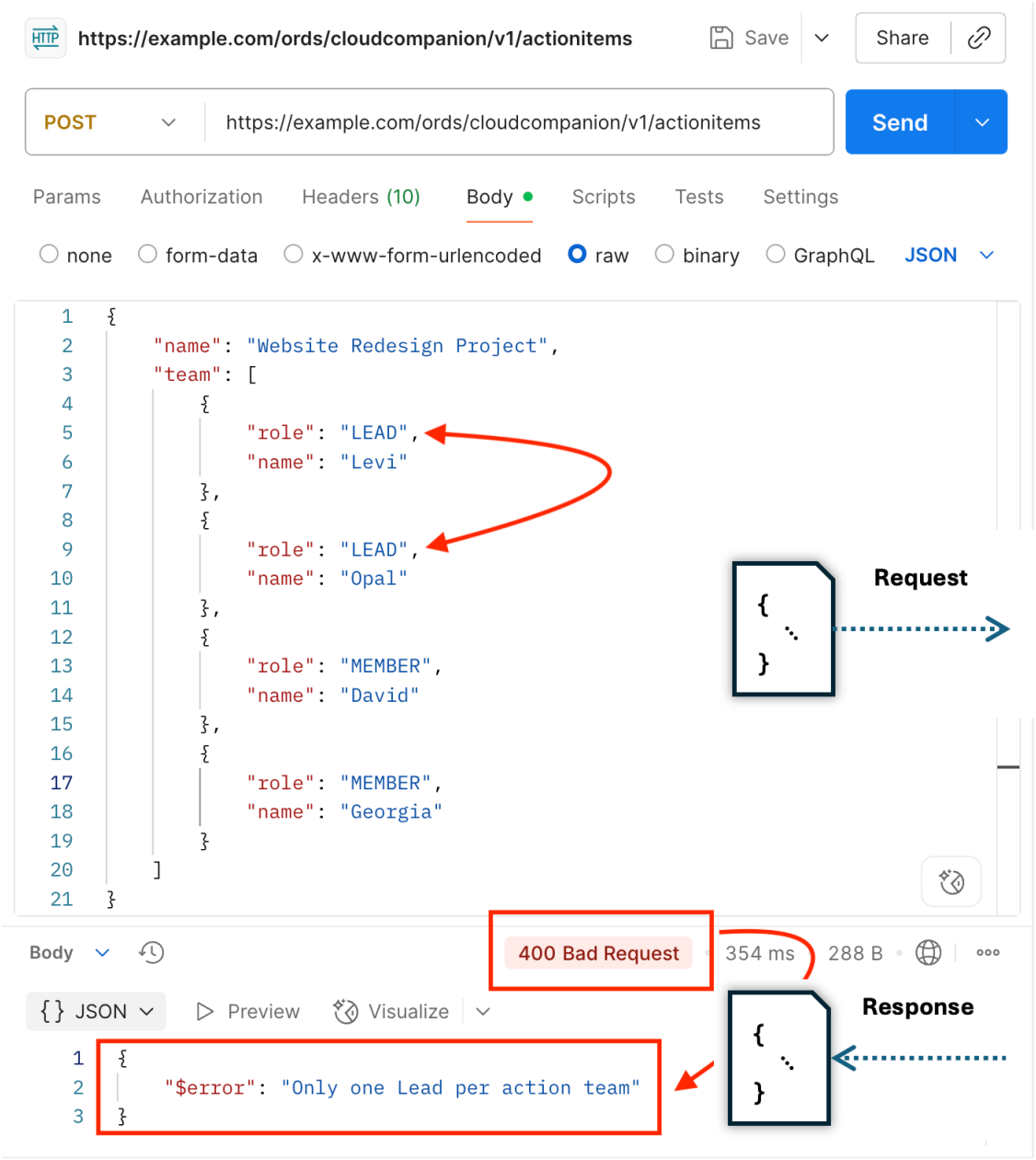
```
{
  "name": "Website Redesign Project",
  "team": [
     {"role":"LEAD",  "name":"Levi"},
     {"role":"LEAD",  "name":"Opal"},
     {"role":"MEMBER","name":"David"},
     {"role":"MEMBER","name":"Georgia"},
  ]
}
```

The status `400` response body includes the JSON payload:

```
{"$error":"Only one lead per action team"}
```


Copyright © 2026, Oracle and/or its affiliates.

Figure 14-18 Working with Action Items and Team Members Using Postman



14.7 Controlling REST API Behavior Completely

For *complete* control over your ORDS REST APIs, create a full set of *PL/SQL* handlers.

> **Tip**
>
> Your module templates can mix *Collection Query* and *Collection Item* `GET` handlers with *PL/SQL* handlers for `POST`, `PUT`, and `DELETE` operations. However, the `/v1` and `/v2` modules show how to manage even the *`GET`* handlers using custom PL/SQL when that level of flexibility might be useful to your application.

Copyright © 2026, Oracle and/or its affiliates.

- Providing Complete Set of ORDS Handlers
  Understand the full set of operations your REST API can implement for maximum control.
- Using Handler Bind Variables
  Use handler bind variables to read URL values, request bodies, parameters, and headers.
- Keeping PL/SQL Handlers Maintainable
  Keep ORDS PL/SQL handlers small by delegating REST logic to package procedures.
- Defining Explicit Handler Parameters
  Define explicit handler parameters for request headers, query strings, and other inputs.
- Reusing Common REST API Handler Code
  Reuse common REST handler code to send JSON responses, errors, and optional debug details.
- Reading and Writing Binary Content
  Read binary content with a *Media Resource* handler and store uploads using the `:body` bind variable.

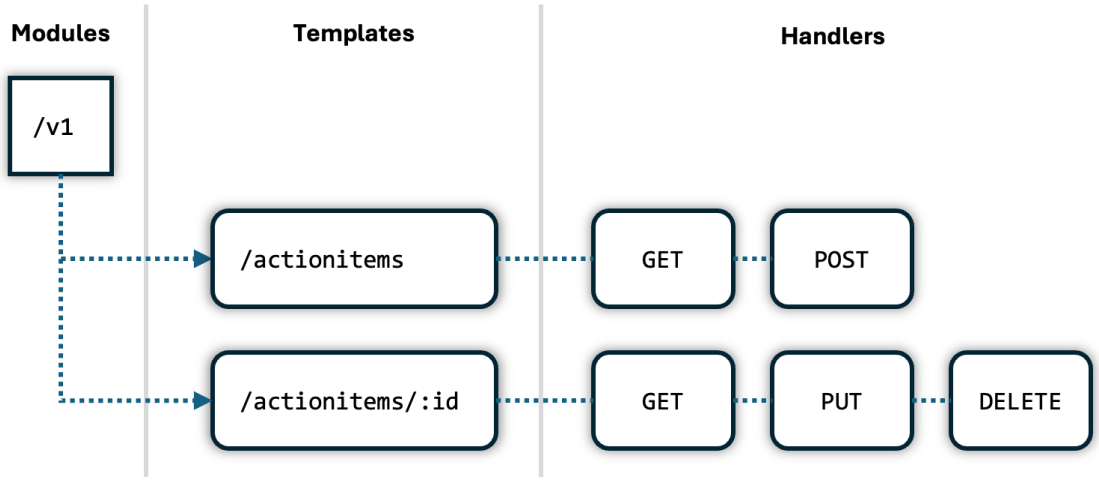
14.7.1 Providing Complete Set of ORDS Handlers

Understand the full set of operations your REST API can implement for maximum control.

When your integration requirements go beyond the read-only use cases, consider creating handlers for all five of the operations a client may need to perform using your business entity REST API. For example, the v1 module shown in the diagram below has handlers for:

- *Listing/Searching Action Items* – `GET` on `/v1/actionitems`
- *Creating a New Action Item* – `POST` on `/v1/actionitems`
- *Getting a Specific Action Item by ID* – `GET` on `/v1/actionitems/:id`
- *Updating a Specific Action Item by ID* – `PUT` on `/v1/actionitems/:id`
- *Deleting a Specific Action Item by ID* – `DELETE` on `/v1/actionitems/:id`

Figure 14-19 ORDS "v1" Module with Templates and Handlers for Action Items API



Copyright © 2026, Oracle and/or its affiliates.

14.7.2 Using Handler Bind Variables

Use handler bind variables to read URL values, request bodies, parameters, and headers.

Your handler can use bind variables to access any of the following information in the incoming request:

- *URL path variables* – like `:id` from `/v1/actionitems/:id` template
- *Implicit parameters* like:
  - *Request body* – using `:body` (BLOB) or `:body_text` (CLOB)
- *Handler parameters* you configure explicitly:
  - Query string parameters – e.g. `/v1/actionitems?skip=5&pagesize=5`
  - HTTP Headers – e.g. `:debug` to reference a header named `app$debug`
  - Request payload properties

> ⚠ **Caution**
>
> Your handler can only reference *either* the `:body` *or* `:body_text` variable, and you can reference the one you pick only a *single time*. Any attempt beyond the first one to read a body variable returns `null`. So, if you need to reference either one in *multiple* places, assign the bind variable to a local PL/SQL variable *first*, then reference the variable anywhere you need to.

14.7.3 Keeping PL/SQL Handlers Maintainable

Keep ORDS PL/SQL handlers small by delegating REST logic to package procedures.

To simplify maintenance, organize your REST API code in PL/SQL packages. This lets each PL/SQL handler make a one-line call to a package procedure. The figure below shows the `PUT` handler for the template `/actionitems/:id` template, which updates a specific action item by ID:

```
begin
   action_items_api_rest.update_object(:id,:body_text,:debug);
end;
```

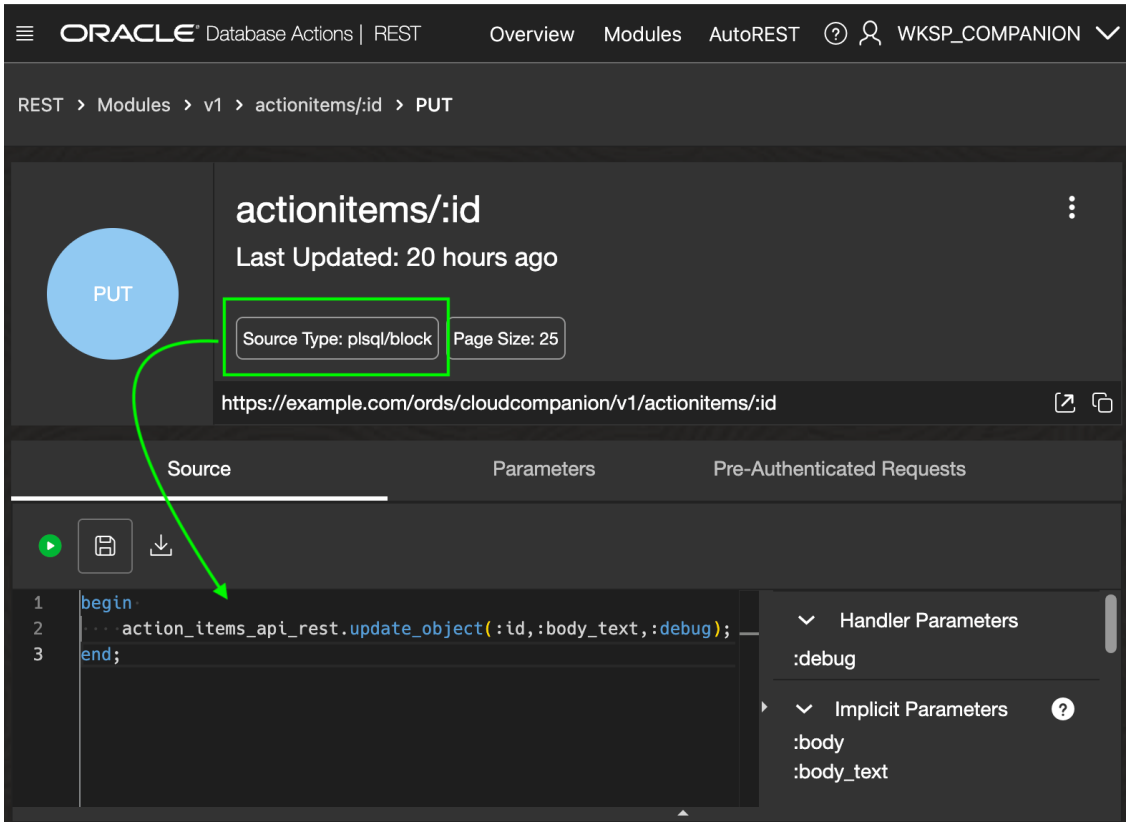
Notice the one-line PL/SQL block references three parameters:

- *`:id`* – from the URL pattern parameter in the `/actionitems/:id` template
- *`:body_text`* – implicit parameter to pass the JSON document CLOB the client sends
- *`:debug`* – explicitly defined parameter for the HTTP header named `app$debug`

The figure shows the `v1` ORDS module's `actionsitems/:id` template's `PUT` handler PL/SQL block explained above in context in the SQL Developer Web REST Designer. Notice the `:debug` listed in the *Handler Parameters* section and `:body_text` in the *Implicit Parameters* sections on the right.

Copyright © 2026, Oracle and/or its affiliates.

Figure 14-20 An ORDS Template Handler Can Use PL/SQL for Complete Control



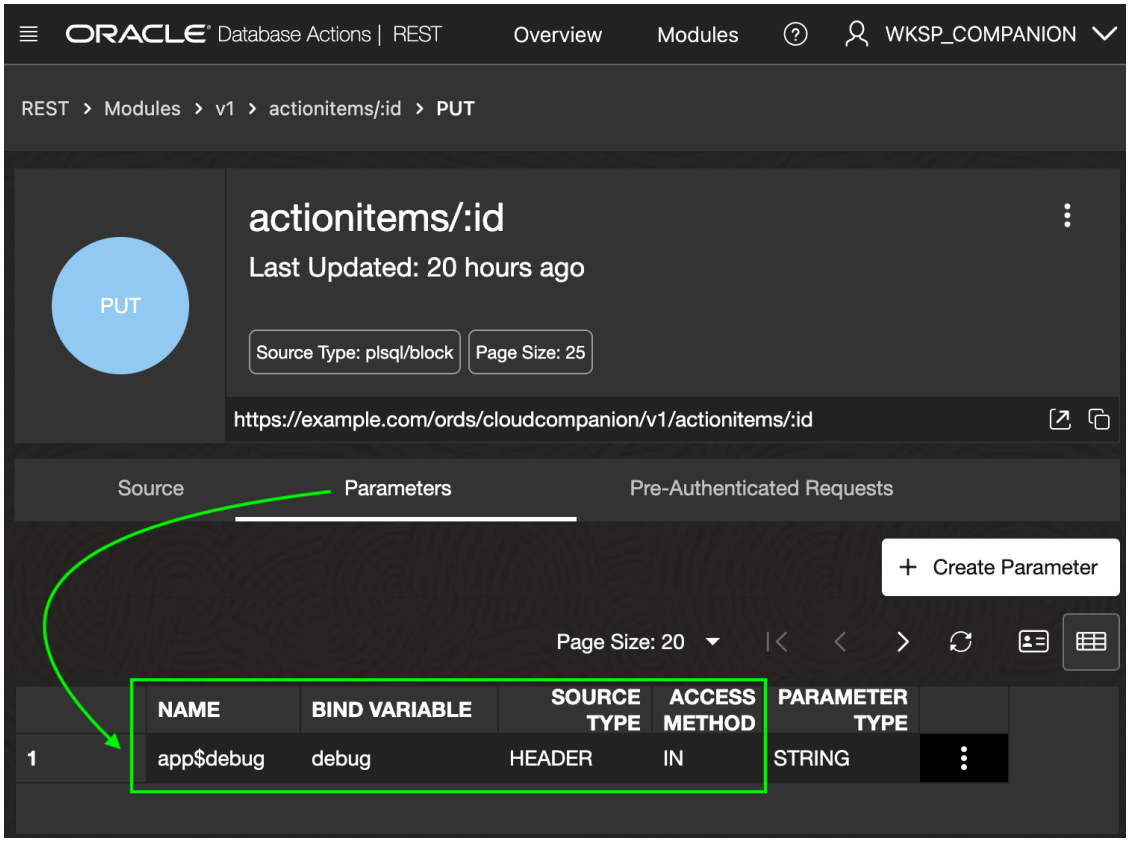
14.7.4 Defining Explicit Handler Parameters

Define explicit handler parameters for request headers, query strings, and other inputs.

The figure below shows the *Parameters* tab of the PL/SQL `PUT` handler's details page. Notice how it defines the bind variable name `debug` for an incoming HTTP Header named `app$debug`. Use this tab to create any explicit parameters your handler needs, defining a parameter name, bind variable name, source type, access method, and data type for each one.

Copyright © 2026, Oracle and/or its affiliates.

Figure 14-21 Defining a Custom Parameter for an ORDS Handler



14.7.5 Reusing Common REST API Handler Code

Reuse common REST handler code to send JSON responses, errors, and optional debug details.

Every PL/SQL handler performs the following set of steps:

- If necessary:
  - *processes* incoming request by referencing bind variables
  - *prepares* response to return to the client
- Then always:
  - Sets the MIME Type of the response, and any other response headers
  - Sets the status code if the default 200 status is not appropriate
  - Closes the header
  - Writes the response to the output buffer using `HTP.P` or `HTP.PRN`

To simplify handler implementation, consider using a package like `APP_COMMON_REST` below that provides procedures for:

- `commit_and_send_response` – committing the transaction and writing out the response
- `handle_error` – rolling back the transaction, signalling an error, and returning error details

Copyright © 2026, Oracle and/or its affiliates.

- `set_debug` – to enable including additional debug info useful to developers.

```
package app_common_rest as
    ------------------------------------------------------------
    -- Set debug mode for error reporting
    ------------------------------------------------------------
    procedure set_debug(
        p_value in varchar2);
    ------------------------------------------------------------
    -- Commit and write out JSON response
    ------------------------------------------------------------
    procedure commit_and_send_response(
        p_json json_element_t default null);
    ------------------------------------------------------------
    -- Handle error by sending HTTP error with $error payload
    ------------------------------------------------------------
    procedure handle_error(
        p_error_message   in varchar2,
        p_http_error_code in number default 400);
end app_common_rest;
```

The package body follows for you to study. Notice it uses:

- `owa_util.mime_header` – to set standard `Content-Type` MIME type header
- `owa_util.status_line` – to set a non-default HTTP response status code
- `htp.p` – to write out the response body as serialized JSON text.

If the `g_debug_mode` flag is enable by a call to `set_debug`, then notice also that the `handle_error` adds an additional `debug$error` property whose value is a JSON array containing the lines of the PL/SQL call stack info from `format_error_stack` and `format_error_backtrace` in the `dbms_utility` package.

```
create or replace package body app_common_rest as
    g_debug_mode boolean := false;
    ----------------------------------
    procedure set_debug(
        p_value in varchar2)
    is
    begin
        if upper(p_value) = 'TRUE' then
            g_debug_mode := true;
        end if;
    end set_debug;
    ----------------------------------
    procedure commit_and_send_response(
        p_json in json_element_t default null)
    is
        l_has_json boolean := p_json is not null;
    begin
        commit;
        owa_util.mime_header('application/json', l_has_json);
        if p_json is not null then
            htp.p(p_json.stringify);
        else
            owa_util.status_line(204, 'No content', true);
```


Copyright © 2026, Oracle and/or its affiliates.

```
            end if;
        end;
        ------------------------------------------------------------
        -- Handle error by sending HTTP 400 with $error payload
        ------------------------------------------------------------
        procedure handle_error(
            p_error_message   in varchar2,
            p_http_error_code in number default 400)
        is
            l_json_err json_object_t := json_object_t();
            l_json_arr json_array_t;
        begin
            owa_util.mime_header('application/json', p_error_message is null);
            owa_util.status_line(p_http_error_code, 'Error', true);
            if p_error_message is not null then
                l_json_err.put('$error',
                               regexp_replace(p_error_message,'^ORA-[0-9]+:\s'));
                if g_debug_mode then
                    l_json_arr := json_array_t();
                    for j in (select rownum, column_value
                                from table(apex_string.split(
                                    dbms_utility.format_error_stack()
                                    || chr(10)
                                    || dbms_utility.format_error_backtrace())))
                    loop
                        if j.column_value is not null then
                            l_json_arr.append(j.column_value);
                        end if;
                    end loop;
                    l_json_err.put('debug$error', l_json_arr);
                end if;
                htp.p(l_json_err.stringify);
            end if;
            rollback;
        end;
    end app_common_rest;
```

14.7.6 Reading and Writing Binary Content

Read binary content with a *Media Resource* handler and store uploads using the `:body` bind variable.

When writing an ORDS handler that needs to return binary content, the simple way is to use a *Media Resource* type handler. Write a query with two columns that returns a single row. The first column provides the MIME type value, and the second column is the BLOB content to return. Imagine a template `/attachments/:id` in a module that needs to return an attachment by id. After choosing *Media Resource* handler type, write the query:

```
select mime_type, file_contents
  from attachments
 where id = :id
```

When writing PL/SQL handlers that *accept* a binary payload, use the `:body` implicit variable to reference it. For example, suppose you have a template `/attachments` and you write a `POST`

Copyright © 2026, Oracle and/or its affiliates.

handler to store an uploaded attachment. You can reference the implicit `:content_type` and `:body` parameters to insert the attachment like this:

```
begin
   your_package.insert_attachment(
       p_mime_type     => :content_type,
       p_file_contents => :body);
   commit;
end;
```

> ⚠ **Caution**
>
> Your handler can only reference the `:body` variable a *single time*. Any attempt beyond the first one to read it returns `null`. So, if you need to reference it in *multiple* places, assign the bind variable to a local PL/SQL variable *first*, then reference the variable anywhere you need to.

14.8 Layering Packages for Modularity

Layer PL/SQL packages so pages and REST APIs share checks while handlers stay simple.

Layer your PL/SQL packages with care. This keeps ORDS handlers simple and lets pages and REST APIs run the same checks, no matter who sends the data.

- Studying API Handler Package Layers
  Study how ORDS handlers call a REST layer that delegates to shared core API logic.
- Examining Core Action Item API Logic
  The `ACTION_ITEMS_API` package is the heart of the Action Items REST API.

14.8.1 Studying API Handler Package Layers

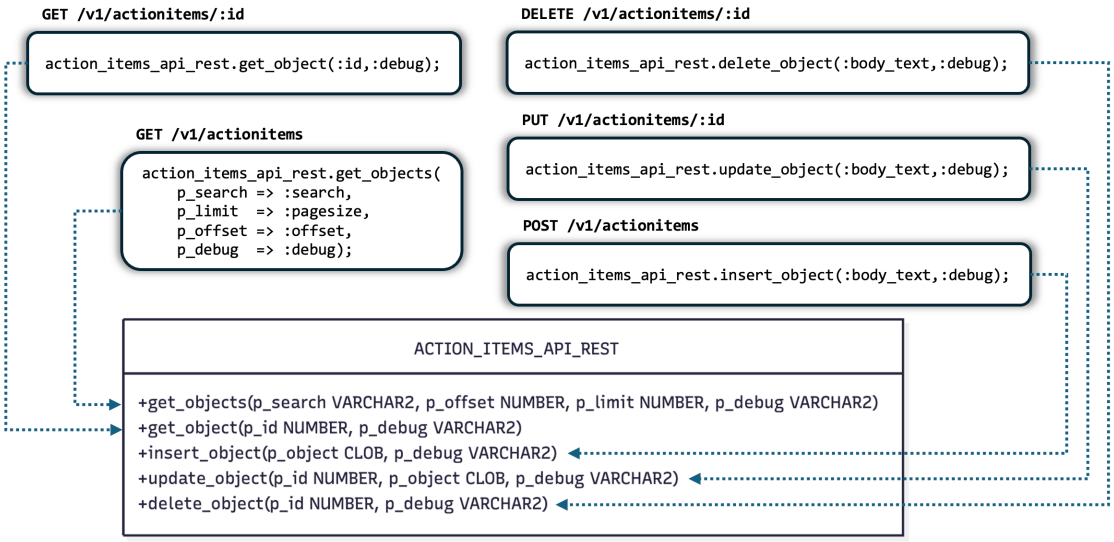
Study how ORDS handlers call a REST layer that delegates to shared core API logic.

The figure below illustrates how the five `v1` module template handlers call appropriate procedures in the `ACTIONS_ITEMS_API_REST` package. Each handler uses a combination of ORDS implicit and explicit parameter bind variables to pass incoming request information to the corresponding package procedure. The digram shows the following ORDS handlers and the `ACTION_ITEMS_API_REST` procedure each invokes:

- `GET /v1/actionitems → get_objects`
- `GET /v1/actionitems/:id → get_object`
- `POST /v1/actionitems → insert_object`
- `PUT /v1/actionitems/:id → update_object`
- `DELETE /v1/actionitems/:id → delete_object`

Copyright © 2026, Oracle and/or its affiliates.

Figure 14-22 Each ORDS Handler Call an ACTION_ITEMS_API_REST Procedure



The package specification for `ACTION_ITEMS_API_REST` appears below.

```
package action_items_api_rest as
    -----------------------------------
    procedure get_objects(
        p_search in varchar2,
        p_offset in number   default 0,
        p_limit  in number   default 10,
        p_debug  in varchar2 default null);
    -----------------------------------
    procedure get_object(
        p_id in number,
        p_debug  in varchar2 default null);
    -----------------------------------
    procedure insert_object(
        p_object in clob,
        p_debug  in varchar2 default null);
    -----------------------------------
    procedure update_object(
        p_id     in number,
        p_object in clob,
        p_debug  in varchar2 default null);
    -----------------------------------
    procedure delete_object(
        p_id     in number,
        p_debug  in varchar2 default null);
end action_items_api_rest;
```

As shown below, the `ACTION_ITEMS_API_REST` package is a REST-specific layer on top of the `ACTIONS_ITEM_API` package that contains the core of the API logic. Its simple procedures essentially convert the incoming JSON `CLOB` payloads into a `JSON_OBJECT_T` types to pass to the core Action Items API package. Then they use the common functionality in

Copyright © 2026, Oracle and/or its affiliates.

APP_COMMON_REST to write out the response or error details. This layering keeps the ORDS handler PL/SQL blocks extremely simple.

Notice in the diagram how the ACTION_ITEMS_API package reuses application-wide common code from APP_COMMON and Action Item specific common validation logic from ACTION_ITEMS_COMMON. This layering maximizes reuse so your application pages and REST APIs apply the same checks to modified data, regardless of how the users or external systems submit it.

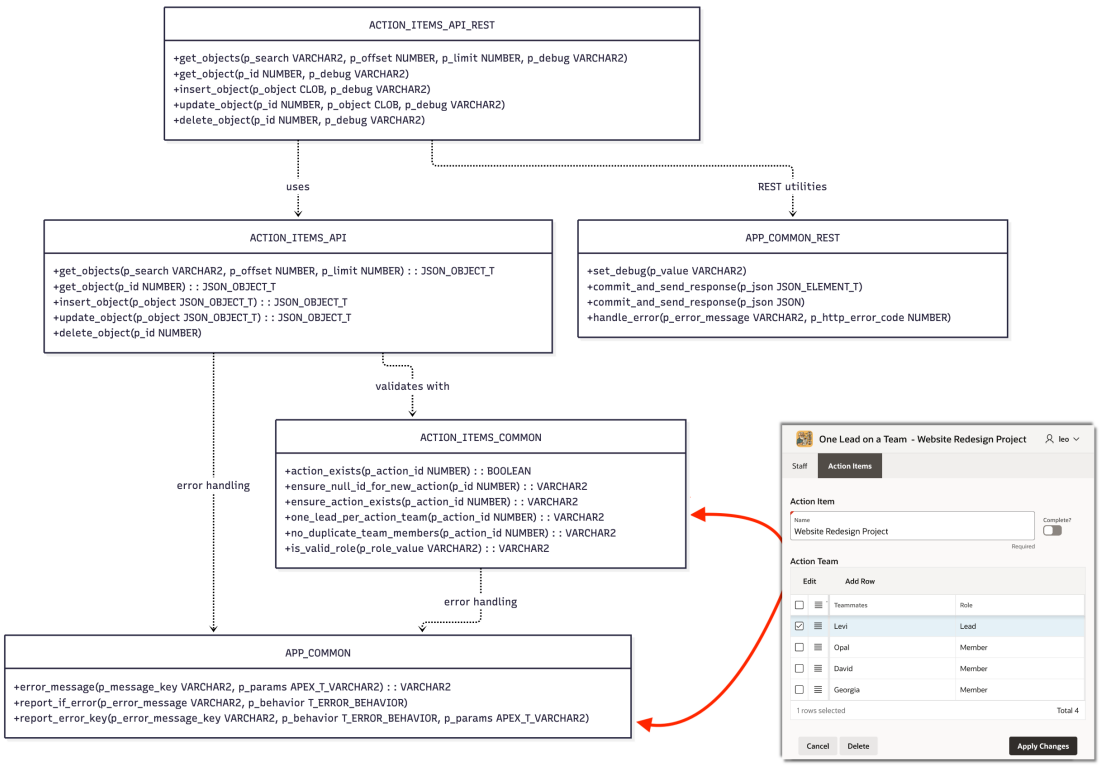
The figure illustrates a package dependency diagram showing:

- ACTION_ITEMS_API_REST uses
  - ACTION_ITEMS_API for logic
  - APP_COMMON_REST as a REST services helper
- ACTION_ITEMS_API uses
  - ACTION_ITEMS_COMMON for validation
  - APP_COMMON for error handling
- ACTION_ITEMS_COMMON also uses APP_COMMON for error handling

In addition, the diagram shows that the *Action Items* edit page uses the same packages:

- ACTION_ITEMS_COMMON for validation
- APP_COMMON for error handling

Figure 14-23 Layered Package Design for Sharing Business Logic with a Page



Copyright © 2026, Oracle and/or its affiliates.

14.8.2 Examining Core Action Item API Logic

The `ACTION_ITEMS_API` package is the heart of the Action Items REST API.

It implements the five public procedures the ORDS template handlers call via `ACTION_ITEMS_API_REST`:

- `get_object` – to retrieve a single action item by ID
- `get_objects` – to list or search action items with paging support
- `insert_object` – to create a new action item
- `update_object` – to update an existing action item by ID
- `delete_object` – to delete an existing action item by ID.

Each procedure uses other interesting private helper routines whole roles are important to understand before digging into the full code of the package body.

- Constructing or Querying an Action Item
  Compare programmatic and SQL-based ways to build action item JSON.
- Retrieving Action Items with Paging
  Retrieve matching action items in a predictable order and return paged JSON results.
- Modifying Action Items with Pre/Post Logic
  Coordinate JSON parsing, business logic, and table changes for action item CRUD operations.
- Inspecting the Core Action Item API Body
  Inspect the full core API package body that coordinates JSON parsing, data changes, and validation.

14.8.2.1 Constructing or Querying an Action Item

Compare programmatic and SQL-based ways to build action item JSON.

For educational purposes, the `GET_OBJECT` procedure shows two different ways to return the JSON representation of an action item. As shown below, if the `c_get_object_approach` constant equals `c_approach_programmatic` then it calls `GET_OBJECT_PROGRAMMATICALLY`. If the constant is `c_approach_sql` instead, then it calls `GET_OBJECT_USING_SQL`.

```
-- in package action_items_api
function get_object(
    p_id in number)
return json_object_t
is
    c_get_object_approach constant t_get_object_approach := c_approach_sql;
    l_ret json_object_t;
begin
    case c_get_object_approach
        when c_approach_programmatic then
            l_ret := get_object_programmatically(p_id);
        when c_approach_sql then
            l_ret := json_object_t(get_object_using_sql(p_id));
    end case;
    return l_ret;
end;
```


Copyright © 2026, Oracle and/or its affiliates.

GET_OBJECT_PROGRAMMATICALLY shown below selects the data from ACTION_ITEMS and ACTION_ITEMS_TEAM_MEMBERS tables into appropriate PL/SQL record structures. Then it uses the PL/SQL JSON_OBJECT_T and JSON_ARRAY_T types to create the action item JSON property by property. Notice that the code *first* builds up the contents of the JSON array, and then adds the array to the action item JSON object with a property name of team at the end.

```
-- in package action_items_api
function get_object_programmatically(
    p_id in number)
return json_object_t
is
    l_action_item_rec action_items%rowtype;
    l_action_item_obj json_object_t := json_object_t();
    l_team_members_arr json_array_t := json_array_t();
    l_team_members     t_action_item_team_members :=
t_action_item_team_members();
    l_team_member_obj  json_object_t;
    l_team_member_name staff.name%type;
begin
    app_common.report_if_error(
        action_items_common.ensure_action_exists(p_id));
    select *
      into l_action_item_rec
      from action_items
     where id = p_id;
    l_action_item_obj.put('_id',l_action_item_rec.id);
    l_action_item_obj.put('name',l_action_item_rec.name);
    l_action_item_obj.put('status',l_action_item_rec.status);
    select *
      bulk collect into l_team_members
      from action_item_team_members
     where action_id = p_id;
    for j in 1..l_team_members.count loop
        l_team_member_obj := json_object_t();
        l_team_member_obj.put('team_assignment_id',l_team_members(j).id);
        l_team_member_obj.put('role',l_team_members(j).role);
        l_team_member_obj.put('staff_id',l_team_members(j).user_id);
        -- Lookup staff name using id
        select name
          into l_team_member_name
          from staff
         where id = l_team_members(j).user_id;
        l_team_member_obj.put('name',l_team_member_name);
        -- Append team member object to the team array
        l_team_members_arr.append(l_team_member_obj);
    end loop;
    l_action_item_obj.put('team',l_team_members_arr);
    return l_action_item_obj;
end get_object_programmatically;
```

As shown below, GET_OBJECT_USING_SQL uses the JSON_OBJECT and JSON_ARRAYAGG functions to create the action item JSON object with its nested team members. Notice the RETURNING

Copyright © 2026, Oracle and/or its affiliates.

CLOB clauses on the array function and on the outer use of JSON_OBJECT to ensure the JSON construction is not limited to 4000 characters.

```
-- in package action_items_api
function get_object_using_sql(
    p_id in number)
return clob
is
    l_action_item_json_clob clob;
begin
    app_common.report_if_error(
        action_items_common.ensure_action_exists(p_id));
    select json_object(
      '_id' value ai.id,
      'name' value ai.name,
      'status' value ai.status,
      'team' value (
        select json_arrayagg(
          json_object(
            'team_assignment_id' value tm.id,
            'role' value tm.role,
            'staff_id' value tm.user_id,
            'name' value s.name
          )
        returning clob /* json_arrayagg */)
        from action_item_team_members tm
        left join staff s on tm.user_id = s.id
        where tm.action_id = ai.id
      )
    returning clob /* json_object */)
    into l_action_item_json_clob
    from action_items ai
    where ai.id = p_id;
    return l_action_item_json_clob;
end get_object_using_sql;
```

14.8.2.2 Retrieving Action Items with Paging

Retrieve matching action items in a predictable order and return paged JSON results.

The GET_OBJECTS procedure accepts a search string, offset, and limit as parameters as shown below. Using a cursor FOR loop, it select the action item's primary key ID column. Interesting things to notice in the loop's query include:

- COUNT(*) OVER() analytic function computes the total number of matching rows
- WHERE clause filters action item name using search criteria (matching anything by default)
- ORDER BY UPPER(NAME) guarantees predictable order for paging
- OFFSET … ROWS skips indicated number of rows (skipping 0 by default)
- FETCH FIRST … ROWS retrieves up to indicated row limit page size (25 by default).

Inside the loop, for each matching action item it calls GET_OBJECT to generate the JSON action item to add to an items array. This ensures consistent results across contexts and keeps the JSON construction logic in one place. The final JSON object GET_OBJECTS returns includes these properties:

Copyright © 2026, Oracle and/or its affiliates.

- `items` *array* containing the matching action item objects
- `hasMore` *boolean* indicating whether there are more rows to return
- `totalRows` *number* indicating how many action items the client *could* retrieve.

```
-- in package action_items_api
function get_objects(
    p_search in varchar2,
    p_offset in number default null,
    p_limit  in number default null)
return json_object_t
is
    l_offset pls_integer := nvl(p_offset,0);
    l_limit  pls_integer := nvl(p_limit,25);
    l_ret json_object_t := json_object_t();
    l_items json_array_t := json_array_t();
    l_row_count  pls_integer := 0;
    l_total_rows pls_integer := 0;
begin
    for j in (select id, count(*) over() as total_count
                from action_items
               where upper(name) like upper('%'||p_search||'%')
               order by upper(name)
               offset l_offset rows
               fetch first l_limit rows only)
    loop
        if l_total_rows = 0 then
            l_total_rows := j.total_count;
        end if;
        l_row_count := l_row_count + 1;
        l_items.append(get_object(j.id));
    end loop;
    l_ret.put('items',l_items);
    l_ret.put('hasMore', l_offset + l_row_count < l_total_rows);
    l_ret.put('totalRows',l_total_rows);
    return l_ret;
end get_objects;
```

14.8.2.3 Modifying Action Items with Pre/Post Logic

Coordinate JSON parsing, business logic, and table changes for action item CRUD operations.

The `INSERT_OBJECT`, `UPDATE_OBJECT`, and `DELETE_OBJECT` functions coordinate the processing of the incoming JSON document to:

- Populate PL/SQL records with the JSON action item and team member data
- Apply before save and after save logic to compute defaults and validate what's needed
- Save the valid data to `ACTION_ITEMS` and `ACTION_ITEM_TEAM_MEMBERS` tables.
- Populating PL/SQL Records from JSON
  Parse action item JSON into PL/SQL records for the parent row and team members.
- Saving Action Item Data to Tables
  Save action item records and team members with insert, update, or delete logic.

Copyright © 2026, Oracle and/or its affiliates.

- Defaulting and Validating Pre/Post-Save
  Apply defaults before saving, then validate saved parent and child rows before commit.
- Coordinating the API Helper Code
  Coordinate JSON parsing, pre-save logic, table changes, post-save checks, and response generation.

14.8.2.3.1 Populating PL/SQL Records from JSON

Parse action item JSON into PL/SQL records for the parent row and team members.

The package passes action item data around in a PL/SQL record variable of type `ACTION_ITEMS%ROWTYPE`. Similarly, it passes action item team member data around in a `T_ACTION_ITEM_TEAM_MEMBERS` type, defined as a table of `ACTION_ITEM_TEAM_MEMBERS%ROWTYPE`. The `ACTION_ITEM_JSON_TO_RECORD` procedure populates the PL/SQL record structures with the action item JSON payload data, using built-in functions on the `JSON_OBJECT_T` and `JSON_ARRAY_T` types to access the data.

```
-- in package action_items_api
procedure action_item_json_to_record(
    p_object        in json_object_t,
    p_action_item  out action_items%rowtype,
    p_action_item_team_members out t_action_item_team_members)
is
    l_team_members_arr json_array_t;
    l_team_member_obj  json_object_t;
    l_team_member_rec  action_item_team_members%rowtype;
    l_team_member_name staff.name%type;
begin
    p_action_item.id     := p_object.get_number('_id');
    p_action_item.name   := p_object.get_string('name');
    p_action_item.status := p_object.get_string('status');
    p_action_item_team_members := t_action_item_team_members();
    l_team_members_arr := p_object.get_array('team');
    for j in 0..l_team_members_arr.get_size - 1 loop
        -- Fill in foreign key attribute from parent
        l_team_member_rec := null;
        l_team_member_rec.action_id := p_action_item.id;
        l_team_member_obj := treat(l_team_members_arr.get(j) as
json_object_t);
        l_team_member_rec.id      :=
l_team_member_obj.get_number('team_assignment_id');
        l_team_member_rec.user_id := l_team_member_obj.get_string('staff_id');
        -- if user_id is null, try to use name to look it up since name is
unique
        if l_team_member_rec.user_id is null then
            l_team_member_name := l_team_member_obj.get_string('name');
            if l_team_member_name is not null then
                begin
                    select id
                      into l_team_member_rec.user_id
                      from staff
                     where name = l_team_member_name;
                exception
                    when no_data_found then
                        app_common.report_error_key(
                            'TEAM_MEMBER_NOT_FOUND',
```


Copyright © 2026, Oracle and/or its affiliates.

```
                                apex_t_varchar2('name',l_team_member_name));
                    end;
                end if;
            end if;
            l_team_member_rec.role :=
substr(l_team_member_obj.get_string('role'),1,6);
            p_action_item_team_members.extend;
            p_action_item_team_members(p_action_item_team_members.count)
                := l_team_member_rec;
    end loop;
end action_item_json_to_record;
```

14.8.2.3.2 Saving Action Item Data to Tables

Save action item records and team members with insert, update, or delete logic.

The `SAVE` procedure stores the data from an action item into `ACTION_ITEMS` and `ACTION_ITEM_TEAM_MEMBERS` tables. Depending on the operation type passed in, it performs appropriate `INSERT`, `UPDATE`, or `DELETE` statements against these tables. The update logic checks which team members are still in the action item and inserts, updates, or deletes as needed.

```
-- in package action_items_api
procedure save(
    p_action_item              in out action_items%rowtype,
    p_action_item_team_members in out t_action_item_team_members,
    p_operation                in     t_operation)
is
    l_new_ids                 apex_t_number;
    l_action_item_member_ids apex_t_number := apex_t_number();
begin
    case p_operation
        when c_operation_insert then
            -- insert the action item
            insert into action_items(name)
            values (p_action_item.name)
            returning id into p_action_item.id;
            -- Set foreign key attribute for new child rows
            for i in 1..p_action_item_team_members.count loop
                p_action_item_team_members(i).action_id := p_action_item.id;
            end loop;
            -- insert all the team members
            forall i in 1..p_action_item_team_members.count
                insert into action_item_team_members
                values p_action_item_team_members(i)
                returning id bulk collect into l_new_ids;
            -- Populate the system-assigned id
            for i in 1..p_action_item_team_members.count loop
                p_action_item_team_members(i).id := l_new_ids(i);
            end loop;
        when c_operation_update then
            update action_items
            set name = p_action_item.name,
                status = p_action_item.status
            where id = p_action_item.id;
            -- Insert/update child rows that need inserting, populating
```


Copyright © 2026, Oracle and/or its affiliates.

```
assigned id
            for i in 1..p_action_item_team_members.count loop
                if p_action_item_team_members(i).id is null then
                    -- id is null? Needs an insert
                    insert into action_item_team_members
                    values p_action_item_team_members(i)
                    returning id into p_action_item_team_members(i).id;
                else
                    -- id is not null, needs an update
                    update action_item_team_members
                    set role = p_action_item_team_members(i).role,
                        user_id = p_action_item_team_members(i).user_id
                    where id = p_action_item_team_members(i).id
                      and action_id = p_action_item_team_members(i).action_id;
                end if;
                -- track which ids we updated and inserted so we can delete
others
                l_action_item_member_ids.extend;
                l_action_item_member_ids(l_action_item_member_ids.count)
                    := p_action_item_team_members(i).id;
            end loop;
            -- Delete any existing team members who are not among the updates
            delete from action_item_team_members
            where action_id = p_action_item.id
            and id not in (select column_value
                           from table(l_action_item_member_ids));
        when c_operation_delete then
            delete from action_items
             where id = p_action_item.id;
    end case;
end save;
```

14.8.2.3.3 Defaulting and Validating Pre/Post-Save

Apply defaults before saving, then validate saved parent and child rows before commit.

The `BEFORE_SAVE` and `AFTER_SAVE` procedures contain the defaulting and validation checks required, using procedures in the `ACTION_ITEM_COMMON` package. The steps performed in `BEFORE_SAVE` include:

- Ensuring an insert does not supply a value for the `ID` primary key
- Defaulting a team member's role to `MEMBER` if not supplied
- Validating the value of role is only either `LEAD` or `MEMBER`.

```
-- in package action_items_api
procedure before_save(
    p_action_item  in out action_items%rowtype,
    p_action_item_team_members in out t_action_item_team_members,
    p_operation    in     t_operation)
is
begin
    if p_operation = c_operation_insert then
        -- Providing an ID with insert not allowed
        app_common.report_if_error(
            action_items_common.ensure_null_id_for_new_action(
```


Copyright © 2026, Oracle and/or its affiliates.

```
                p_action_item.id));
    end if;
    if p_operation in (c_operation_insert,c_operation_update) then
        -- Default role to member if not supplied
        for j in 1..p_action_item_team_members.count loop
            if p_action_item_team_members(j).role is null then
                p_action_item_team_members(j).role := 'MEMBER';
            else
                -- Check role value is legal
                app_common.report_if_error(
                    action_items_common.is_valid_role(
                       p_action_item_team_members(j).role));
            end if;
        end loop;
    end if;
end before_save;
```

The checks enforcing conditions about the action item team membership go in the `AFTER_SAVE` so they can query the parent and child data *after* it's saved, but *before* it gets committed. The `AFTER_SAVE` validations include validating that an action team:

- Contains no duplicate team members
- Has *exactly* one lead.

```
-- in package action_items_api
procedure after_save(
    p_action_item              in action_items%rowtype,
    p_action_item_team_members in t_action_item_team_members,
    p_operation                in t_operation)
is
begin
    if p_operation in (c_operation_insert,c_operation_update) then
        app_common.report_if_error(
            action_items_common.no_duplicate_team_members(p_action_item.id));
        app_common.report_if_error(
            action_items_common.one_lead_per_action_team(p_action_item.id));
    end if;
end after_save;
```

14.8.2.3.4 Coordinating the API Helper Code

Coordinate JSON parsing, pre-save logic, table changes, post-save checks, and response generation.

The `INSERT_OBJECT`, `UPDATE_OBJECT`, and `DELETE_OBJECT` procedures coordinate the flow. The insert and update routines have the same sequence. They call:

- `ACTION_ITEM_JSON_TO_RECORD` to populate the PL/SQL record structures
- `BEFORE_SAVE` to default and validate anything not requiring aggregate SQL checks
- `SAVE` to store the action item and its action item team members into tables
- `AFTER_SAVE` to validate conditions requiring aggregate SQL checks

Copyright © 2026, Oracle and/or its affiliates.

- `GET_OBJECT` to return the latest state of the action item to the client.

```
-- in package action_items_api
function insert_object(
    p_object in json_object_t)
return json_object_t
is
    l_action_item_rec action_items%rowtype;
    l_action_item_team_members t_action_item_team_members;
begin
    action_item_json_to_record(
        p_object                  => p_object,
        p_action_item             => l_action_item_rec,
        p_action_item_team_members => l_action_item_team_members);

before_save(l_action_item_rec,l_action_item_team_members,c_operation_insert);
    save(l_action_item_rec,l_action_item_team_members,c_operation_insert);

after_save(l_action_item_rec,l_action_item_team_members,c_operation_insert);
    return get_object(l_action_item_rec.id);
end insert_object;
```

> **Tip**
>
> The `UPDATE_OBJECT` is identical, but passes the `c_operation_update` constant instead.

The `DELETE_OBJECT` routine receives no action item JSON, so there is no need to populate PL/SQL records. Instead, it creates an action item record with only the primary key `ID` field set. While this particular example contains no delete-time *before* or *after* save logic, if used it needs to rely only on accessing this primary key value. Finally, since the delete REST API operation requires no response JSON, the `DELETE_OBJECT` procedure does *not* call `GET_OBJECT` at the end.

```
-- in package action_items_api
procedure delete_object(
    p_id in number)
is
    l_action_item_rec action_items%rowtype;
    l_action_item_team_members t_action_item_team_members;
begin
    l_action_item_rec.id := p_id;

before_save(l_action_item_rec,l_action_item_team_members,c_operation_delete);
    save(l_action_item_rec,l_action_item_team_members,c_operation_delete);

after_save(l_action_item_rec,l_action_item_team_members,c_operation_delete);
end delete_object;
```

14.8.2.4 Inspecting the Core Action Item API Body

Inspect the full core API package body that coordinates JSON parsing, data changes, and validation.

Copyright © 2026, Oracle and/or its affiliates.

The full code of the `ACTION_ITEMS_API` package body appears below.

```
create or replace package body action_items_api as

    subtype t_get_object_approach is pls_integer range 1..2;
    c_approach_programmatic constant t_get_object_approach := 1;
    c_approach_sql          constant t_get_object_approach := 2;

    subtype t_operation is pls_integer range 1..3;
    c_operation_insert      constant t_operation := 1;
    c_operation_update      constant t_operation := 2;
    c_operation_delete      constant t_operation := 3;

    -----------------------------------
    procedure before_save(
        p_action_item  in out action_items%rowtype,
        p_action_item_team_members in out t_action_item_team_members,
        p_operation    in     t_operation)
    is
    begin
        if p_operation = c_operation_insert then
            -- Providing an ID with insert not allowed
            app_common.report_if_error(

action_items_common.ensure_null_id_for_new_action(p_action_item.id));
        end if;
        if p_operation in (c_operation_insert,c_operation_update) then
            -- Default role to member if not supplied
            for j in 1..p_action_item_team_members.count loop
                if p_action_item_team_members(j).role is null then
                    p_action_item_team_members(j).role := 'MEMBER';
                else
                    -- Check role value is legal
                    app_common.report_if_error(

action_items_common.is_valid_role(p_action_item_team_members(j).role));
                end if;
            end loop;
        end if;
    end before_save;
    -----------------------------------
    procedure save(
        p_action_item              in out action_items%rowtype,
        p_action_item_team_members in out t_action_item_team_members,
        p_operation                in     t_operation)
    is
        l_new_ids                apex_t_number;
        l_action_item_member_ids apex_t_number := apex_t_number();
    begin
        case p_operation
            when c_operation_insert then
                -- insert the action item
                insert into action_items(name)
                values (p_action_item.name)
                returning id into p_action_item.id;
                -- Set foreign key attribute for new child rows
```


Copyright © 2026, Oracle and/or its affiliates.

```
                for i in 1..p_action_item_team_members.count loop
                    p_action_item_team_members(i).action_id :=
p_action_item.id;
                end loop;
                -- insert all the team members
                forall i in 1..p_action_item_team_members.count
                    insert into action_item_team_members values
p_action_item_team_members(i)
                    returning id bulk collect into l_new_ids;
                -- Populate the system-assigned id
                for i in 1..p_action_item_team_members.count loop
                    p_action_item_team_members(i).id := l_new_ids(i);
                end loop;
            when c_operation_update then
                update action_items
                set name = p_action_item.name,
                    status = p_action_item.status
                where id = p_action_item.id;
                -- Insert/update child rows that need inserting, populating
assigned id
                for i in 1..p_action_item_team_members.count loop
                    if p_action_item_team_members(i).id is null then
                        -- id is null? Needs an insert
                        insert into action_item_team_members
                        values p_action_item_team_members(i)
                        returning id into p_action_item_team_members(i).id;
                    else
                        -- id is not null, needs an update
                        update action_item_team_members
                        set role = p_action_item_team_members(i).role,
                            user_id = p_action_item_team_members(i).user_id
                        where id = p_action_item_team_members(i).id
                          and action_id =
p_action_item_team_members(i).action_id;
                    end if;
                    -- track which ids we updated and inserted so we can
delete others
                    l_action_item_member_ids.extend;
                    l_action_item_member_ids(l_action_item_member_ids.count)
                        := p_action_item_team_members(i).id;
                end loop;
                -- Delete any existing team members who are not among the
updates
                delete from action_item_team_members
                where action_id = p_action_item.id
                and id not in (select column_value from
table(l_action_item_member_ids));
            when c_operation_delete then
                delete from action_items
                 where id = p_action_item.id;
        end case;
    end save;
    ----------------------------------
    procedure after_save(
        p_action_item              in action_items%rowtype,
        p_action_item_team_members in t_action_item_team_members,
```


Copyright © 2026, Oracle and/or its affiliates.

```
        p_operation               in t_operation)
    is
    begin
        if p_operation in (c_operation_insert,c_operation_update) then
            app_common.report_if_error(

action_items_common.no_duplicate_team_members(p_action_item.id));
            app_common.report_if_error(

action_items_common.one_lead_per_action_team(p_action_item.id));
        end if;
    end after_save;
    ----------------------------------
    procedure action_item_json_to_record(
        p_object        in json_object_t,
        p_action_item  out action_items%rowtype,
        p_action_item_team_members out t_action_item_team_members)
    is
        l_team_members_arr json_array_t;
        l_team_member_obj  json_object_t;
        l_team_member_rec  action_item_team_members%rowtype;
        l_team_member_name staff.name%type;
    begin
        p_action_item.id     := p_object.get_number('_id');
        p_action_item.name   := p_object.get_string('name');
        p_action_item.status := p_object.get_string('status');
        p_action_item_team_members := t_action_item_team_members();
        l_team_members_arr := p_object.get_array('team');
        for j in 0..l_team_members_arr.get_size - 1 loop
            -- Fill in foreign key attribute from parent
            l_team_member_rec := null;
            l_team_member_rec.action_id := p_action_item.id;
            l_team_member_obj := treat(l_team_members_arr.get(j) as
json_object_t);
            l_team_member_rec.id      :=
l_team_member_obj.get_number('team_assignment_id');
            l_team_member_rec.user_id :=
l_team_member_obj.get_string('staff_id');
            -- if user_id is null, try to use name to look it up since name
is unique
            if l_team_member_rec.user_id is null then
                l_team_member_name := l_team_member_obj.get_string('name');
                if l_team_member_name is not null then
                    begin
                        select id
                          into l_team_member_rec.user_id
                          from staff
                         where name = l_team_member_name;
                    exception
                        when no_data_found then
                            app_common.report_error_key(
                                'TEAM_MEMBER_NOT_FOUND',
                                apex_t_varchar2('name',l_team_member_name));
                    end;
                end if;
            end if;
```


Copyright © 2026, Oracle and/or its affiliates.

```
            l_team_member_rec.role :=
substr(l_team_member_obj.get_string('role'),1,6);
            p_action_item_team_members.extend;
            p_action_item_team_members(p_action_item_team_members.count) :=
l_team_member_rec;
        end loop;
    end action_item_json_to_record;
    -----------------------------------
    function get_object_using_sql(
        p_id in number)
    return clob
    is
        l_action_item_json_clob clob;
    begin
        app_common.report_if_error(
            action_items_common.ensure_action_exists(p_id));
        select json_object(
          '_id' value ai.id,
          'name' value ai.name,
          'status' value ai.status,
          'team' value (
            select json_arrayagg(
              json_object(
                'team_assignment_id' value tm.id,
                'role' value tm.role,
                'staff_id' value tm.user_id,
                'name' value s.name
              )
            returning clob /* json_arrayagg */)
            from action_item_team_members tm
            left join staff s on tm.user_id = s.id
            where tm.action_id = ai.id
          )
        returning clob /* json_object */)
        into l_action_item_json_clob
        from action_items ai
        where ai.id = p_id;
        return l_action_item_json_clob;
    end get_object_using_sql;
    -----------------------------------
    function get_object_programmatically(
        p_id in number)
    return json_object_t
    is
        l_action_item_rec action_items%rowtype;
        l_action_item_obj json_object_t := json_object_t();
        l_team_members_arr json_array_t := json_array_t();
        l_team_members     t_action_item_team_members :=
t_action_item_team_members();
        l_team_member_obj  json_object_t;
        l_team_member_name staff.name%type;
    begin
        app_common.report_if_error(
            action_items_common.ensure_action_exists(p_id));
        select *
          into l_action_item_rec
```


Copyright © 2026, Oracle and/or its affiliates.

```
            from action_items
           where id = p_id;
          l_action_item_obj.put('_id',l_action_item_rec.id);
          l_action_item_obj.put('name',l_action_item_rec.name);
          l_action_item_obj.put('status',l_action_item_rec.status);
          select *
            bulk collect into l_team_members
            from action_item_team_members
           where action_id = p_id;
          for j in 1..l_team_members.count loop
              l_team_member_obj := json_object_t();
              l_team_member_obj.put('team_assignment_id',l_team_members(j).id);
              l_team_member_obj.put('role',l_team_members(j).role);
              l_team_member_obj.put('staff_id',l_team_members(j).user_id);
              -- Lookup staff name using id
              select name
                into l_team_member_name
                from staff
               where id = l_team_members(j).user_id;
              l_team_member_obj.put('name',l_team_member_name);
              -- Append team member object to the team array
              l_team_members_arr.append(l_team_member_obj);
          end loop;
          l_action_item_obj.put('team',l_team_members_arr);
          return l_action_item_obj;
      end get_object_programmatically;
      ----------------------------------
      function get_object(
          p_id in number)
      return json_object_t
      is
          c_get_object_approach   constant t_get_object_approach :=
  c_approach_sql;
          l_ret json_object_t;
      begin
          case c_get_object_approach
              when c_approach_programmatic then
                  l_ret := get_object_programmatically(p_id);
              when c_approach_sql then
                  l_ret := json_object_t(get_object_using_sql(p_id));
          end case;
          return l_ret;
      end;
      ----------------------------------
      function get_objects(
          p_search in varchar2,
          p_offset in number default null,
          p_limit  in number default null)
      return json_object_t
      is
          l_offset pls_integer := nvl(p_offset,0);
          l_limit  pls_integer := nvl(p_limit,25);
          l_ret json_object_t := json_object_t();
          l_items json_array_t := json_array_t();
          l_row_count  pls_integer := 0;
          l_total_rows pls_integer := 0;
```


Copyright © 2026, Oracle and/or its affiliates.

```
    begin
        for j in (select id, count(*) over() as total_count
                    from action_items
                   where upper(name) like upper('%'||p_search||'%')
                   order by upper(name)
                   offset l_offset rows
                   fetch first l_limit rows only)
        loop
            if l_total_rows = 0 then
                l_total_rows := j.total_count;
            end if;
            l_row_count := l_row_count + 1;
            l_items.append(get_object(j.id));
        end loop;
        l_ret.put('items',l_items);
        l_ret.put('hasMore', l_offset + l_row_count < l_total_rows);
        l_ret.put('totalRows',l_total_rows);
        return l_ret;
    end get_objects;
    ----------------------------------
    function insert_object(
        p_object in json_object_t)
    return json_object_t
    is
        l_action_item_rec action_items%rowtype;
        l_action_item_team_members t_action_item_team_members;
    begin
        action_item_json_to_record(
            p_object                  => p_object,
            p_action_item             => l_action_item_rec,
            p_action_item_team_members => l_action_item_team_members);

before_save(l_action_item_rec,l_action_item_team_members,c_operation_insert);
        save(l_action_item_rec,l_action_item_team_members,c_operation_insert);

after_save(l_action_item_rec,l_action_item_team_members,c_operation_insert);
        return get_object(l_action_item_rec.id);
    end insert_object;
    ----------------------------------
    function update_object(
        p_object in json_object_t)
    return json_object_t
    is
        l_action_item_rec action_items%rowtype;
        l_action_item_team_members t_action_item_team_members;
    begin
        action_item_json_to_record(
            p_object                  => p_object,
            p_action_item             => l_action_item_rec,
            p_action_item_team_members => l_action_item_team_members);

before_save(l_action_item_rec,l_action_item_team_members,c_operation_update);
        save(l_action_item_rec,l_action_item_team_members,c_operation_update);

after_save(l_action_item_rec,l_action_item_team_members,c_operation_update);
        return get_object(l_action_item_rec.id);
```


Copyright © 2026, Oracle and/or its affiliates.

```
    end update_object;
    ----------------------------------
    procedure delete_object(
        p_id in number)
    is
        l_action_item_rec action_items%rowtype;
        l_action_item_team_members t_action_item_team_members;
    begin
        l_action_item_rec.id := p_id;

before_save(l_action_item_rec,l_action_item_team_members,c_operation_delete);
        save(l_action_item_rec,l_action_item_team_members,c_operation_delete);

after_save(l_action_item_rec,l_action_item_team_members,c_operation_delete);
    end delete_object;
end action_items_api;
```

14.9 Simplifying REST APIs in Oracle 26ai

Simplify REST API development in Oracle 26ai with JSON type and duality views.

If you use Oracle 26ai, creating REST APIs for your applications is dramatically easier thanks to significant new JSON capabilities in this release.

- Using JSON Type and Duality Views
  Use JSON type and duality views to work with relational data as JSON.
- Exploring ORDS AutoREST for Duality View
  Expose a JSON relational duality view quickly by enabling ORDS AutoREST access.
- Configuring ORDS Handlers for 26ai API
  Set up ORDS handlers to call an alternate API that uses 26ai JSON type and a duality view.
- Seeing How Duality View Simplifies Logic
  Compare the 26ai duality view API with the 19c version to see how much code it removes.

14.9.1 Using JSON Type and Duality Views

Use JSON type and duality views to work with relational data as JSON.

Oracle 26ai's `JSON` type further simplifies JSON data processing in SQL and PL/SQL. Its JSON Relational Duality Views let you define a declarative JSON structure for one or more related tables using a simplified syntax. This JSON view of the underlying relational data model can be read-only if desired, but you can also define it to support inserting, updating, and deleting data through the same view.

- Storing and Accessing JSON Type Data
  Oracle 26ai JSON type to store, query, and transform JSON, with easy access from JSON_OBJECT_T when needed.
- Defining JSON Relational Duality View
  Define a JSON relational duality view that maps related table data into a single JSON document.
- Querying JSON Relational Duality View
  Query a JSON relational duality view to retrieve related data as JSON documents.

Copyright © 2026, Oracle and/or its affiliates.

- Modifying JSON Using a Duality View
  Modify related table data by inserting, updating, or deleting JSON documents through a duality view.
- Flexing the Duality of Duality Views
  Work with the same data as JSON documents or relational rows, with changes reflected in both.
- Preventing Lost Updates with ETAG
  A duality view document's ETAG value prevents a stale update from overwriting newer changes.
- Comparing _id and RESID in Duality View
  Use RESID as an alternative document identifier for duality view updates and deletes.

14.9.1.1 Storing and Accessing JSON Type Data

Oracle 26ai JSON type to store, query, and transform JSON, with easy access from JSON_OBJECT_T when needed.

Oracle 26ai has a `JSON` data type for optimized storage and access of JSON data in SQL and PL/SQL. You can use `JSON` columns in relational tables, create JSON collection tables with their single `DATA` column of `JSON` type, and more easily and efficiently manipulate JSON in using SQL. For example, by working with the JSON data using a `JSON` type you can use the powerful `JSON_TRANSFORM` function to do bulk, conditional modification of JSON. Consider starting with a JSON document in a `JSON` variable like this:

```
l_actionitem json :=
             json(q'~
                {
                  "name": "Practice Using JSON in PL/SQL",
                  "team": [
                    {"name": "Georgia","role": "LEAD"},
                    {"name": "Quinn"},
                    {"name": "Zelda"}
                  ]
                }
              ~');
```

Using `JSON_TRANSFORM` you can default the `role` property to the value `MEMBER` on any `team` array object where `role` is `null` or is missing using the following declarative operation:

```
-- Default role to member if not supplied
select
   json_transform(l_actionitem,
      set '$.team[*]?(@.role == null || !exists(@.role)).role'
           = 'MEMBER')
    into l_actionitem;
```

You can use the full power of JSON path expressions to access any data in the document right from PL/SQL:

```
-- Print out the name of the LEAD team member
p(json_value(l_actionitem,'$.team[*]?(@.role == "LEAD").name'));
```


Copyright © 2026, Oracle and/or its affiliates.

You can loop over array objects and process them in a cursor for loop:

```
-- Process each team member object in a loop
for k in (select team_member
            from json_table(l_actionitem,'$.team[*]'
                   columns (team_member json path '$')))
loop
    p(json_value(k.team_member,'$.name')||','||
      json_value(k.team_member,'$.role'));
end loop;
```

You can selectively remove elements:

```
-- Delete team member Zelda using SQL
select json_transform(l_actionitem,
            remove '$.team[*]?(@.name == "Zelda" && @.role == "MEMBER")')
  into l_actionitem;
```

When needed, the `JSON` type interoperates seamlessly with the `JSON_OBJECT_T` API you may already be familiar with. Simply pass the `JSON` object to the `JSON_OBJECT_T` constructor function to work with it using that "interface" instead. Then later you can call `TO_JSON` on the `JSON_OBJECT_T` to get it back as a `JSON` type again. This example shows this back and forth in action:

```
-- Delete team member Quinn using PL/SQL JSON_OBJECT_T
l_actionitem_obj := json_object_t(l_actionitem);
l_teammembers    := l_actionitem_obj.get_array('team');
-- iterate backwards for deleting
for j in reverse 0..l_teammembers.get_size - 1  loop
    if get_object(l_teammembers,j).get_string('name') = 'Quinn' then
        l_teammembers.remove(j);
    end if;
end loop;
-- Show interoperability between JSON and JSON_OBJECT_T
l_actionitem := l_actionitem_obj.to_json;
```

The full code example follows below:

```
declare
    l_actionitem json :=
                 json(q'~
                    {
                      "name": "Practice Using JSON in PL/SQL",
                      "team": [
                        {"name": "Georgia","role": "LEAD"},
                        {"name": "Quinn"},
                        {"name": "Zelda"}
                      ]
                    }
                  ~');
    l_actionitem_obj json_object_t;
    l_teammembers    json_array_t;
    ----------------------------
    function get_object(
```


Copyright © 2026, Oracle and/or its affiliates.

```
        p_array in json_array_t,
        p_index in pls_integer)
        return     json_object_t
    is
    begin
        return treat(p_array.get(p_index) as json_object_t);
    end get_object;
    ----------------------------
    procedure p(p_str in varchar)
    is
    begin
        dbms_output.put_line(p_str);
    end p;
begin
    -- Default role to member if not supplied
    select json_transform(l_actionitem,
                set '$.team[*]?(@.role == null
                                ||!exists(@.role))
                                .role' = 'MEMBER')
    into l_actionitem;
    p(json_serialize(l_actionitem pretty));
    -- Print out the name of the LEAD team member
    p(json_value(l_actionitem,'$.team[*]?(@.role == "LEAD").name'));
    -- Process each team member object in a loop
    for k in (select team_member
                from json_table(l_actionitem,'$.team[*]'
                        columns (team_member json path '$')))
    loop
        p(json_value(k.team_member,'$.name')||','||
          json_value(k.team_member,'$.role'));
    end loop;
    -- Delete team member Zelda using SQL
    select json_transform(l_actionitem,
                remove '$.team[*]?(@.name == "Zelda" && @.role == "MEMBER")')
      into l_actionitem;
    p(json_serialize(l_actionitem pretty));
    -- Delete team member Quinn using PL/SQL JSON_OBJECT_T
    l_actionitem_obj := json_object_t(l_actionitem);
    l_teammembers    := l_actionitem_obj.get_array('team');
    -- iterate backwards for deleting
    for j in reverse 0..l_teammembers.get_size - 1  loop
        if get_object(l_teammembers,j).get_string('name') = 'Quinn' then
            l_teammembers.remove(j);
        end if;
    end loop;
    -- Show interoperability between JSON and JSON_OBJECT_T
    l_actionitem := l_actionitem_obj.to_json;
    p(json_serialize(l_actionitem pretty));
end;
```

Running the PL/SQL block above produces the following output:

```
{
  "name" : "Practice Using JSON in PL/SQL",
  "team" :
```


Copyright © 2026, Oracle and/or its affiliates.

```
  [
    {
      "name" : "Georgia",
      "role" : "LEAD"
    },
    {
      "name" : "Quinn",
      "role" : "MEMBER"
    },
    {
      "name" : "Zelda",
      "role" : "MEMBER"
    }
  ]
}
Georgia
Georgia,LEAD
Quinn,MEMBER
Zelda,MEMBER
{
  "name" : "Practice Using JSON in PL/SQL",
  "team" :
  [
    {
      "name" : "Georgia",
      "role" : "LEAD"
    },
    {
      "name" : "Quinn",
      "role" : "MEMBER"
    }
  ]
}
{
  "name" : "Practice Using JSON in PL/SQL",
  "team" :
  [
    {
      "name" : "Georgia",
      "role" : "LEAD"
    }
  ]
}
```

14.9.1.2 Defining JSON Relational Duality View

Define a JSON relational duality view that maps related table data into a single JSON document.

The following is the simple JSON Relational Duality View data definition language (DDL) statement for a JSON view of action items, their team members, and related staff names. Notice the ***`ACTION_ITEMS`*** table forms the "root" of JSON view, the `team` property automatically includes the data from the ***`ACTION_ITEMS_TEAM_MEMBERS`*** in an array, and it pulls in the `name` of the action team member from the ***`STAFF`*** table. The *`@insert`*, `@update`, and `@delete` annotations on a table indicate which DML operations are allowed. If *no annotation* is included on a table,

Copyright © 2026, Oracle and/or its affiliates.

as for STAFF below, then its data is read-only in the duality view. You don't need to express how the tables are related, because the duality view infers that from foreign key constraints. The *@unnest* annotation promotes the staff_id and name properties to be at the same level as team_assignment_id and role.

```
create json relational duality view action_items_dv as
action_items @insert @update @delete
{
  _id: id,
  name: name,
  status: status,
  team: action_item_team_members @insert @update @delete
  {
    team_assignment_id: id,
    role: role,
    staff @unnest
    {
      staff_id: id,
      name: name
    }
  }
};
```

14.9.1.3 Querying JSON Relational Duality View

Query a JSON relational duality view to retrieve related data as JSON documents.

A JSON Relational Duality View has a single JSON-type column named DATA, so query the view like this:

```
-- Retrieve 3 action item JSON docs by _id value
select data
  from action_items_dv
 where json_value(data,'$._id') in (14, 25, 27)
```

Each row of data retrieved looks like the following:

```
{
  "_id": 14,
  "name": "End of Year Party",
  "status": "OPEN",
  "team": [
    {
      "team_assignment_id": 23,
      "role": "MEMBER",
      "staff_id": 4,
      "name": "David"
    },
    {
      "team_assignment_id": 24,
      "role": "LEAD",
      "staff_id": 5,
      "name": "Georgia"
    },
```


Copyright © 2026, Oracle and/or its affiliates.

```
    {
      "team_assignment_id": 25,
      "role": "MEMBER",
      "staff_id": 13,
      "name": "Jane"
    }
  ],
  "_metadata": {
    "etag": "5891809CF5691D5400D4141A65123976",
    "asof": "000025C52A64CB47"
  }
}
```

14.9.1.4 Modifying JSON Using a Duality View

Modify related table data by inserting, updating, or deleting JSON documents through a duality view.

Since you include the `@insert`, `@update`, and `@delete` annotations when you define the duality view, you can use the view to modify action items and team members, too. For example, you can insert a new action item with its related action team members in a single `INSERT` statement like this:

```
declare
    l_json json :=
           json(q'~
            {
              "name": "Buy Talulah's Retirement Gift",
              "team": [
                {
                  "role": "LEAD",
                  "name": "Abigail"
                },
                {
                  "role": "MEMBER",
                  "name": "Chaz"
                }
              ]
            }
       ~');
begin
    insert into action_items_dv(data)
    values (l_json)
    returning data into l_json;
    dbms_output.put_line(json_serialize(l_json pretty));
end;
```

This single `INSERT` statement into the `ACTION_ITEMS_DV` JSON Relational Duality View handles:

- Looking up the `staff_id` for each team member since `name` is unique in the `STAFF` table,
- Inserting the new action item into the `ACTION_ITEMS` table, and
- Inserting two new action item team members into the `ACTION_ITEM_TEAM_MEMBERS` table.

Copyright © 2026, Oracle and/or its affiliates.

It returns the JSON document below:

```
{
  "_id" : 58,
  "name" : "Buy Talulah's Retirement Gift",
  "status": "OPEN",
  "team" :
  [
    {
      "team_assignment_id" : 127,
      "role" : "MEMBER",
      "staff_id" : 23,
      "name" : "Chaz"
    },
    {
      "team_assignment_id" : 128,
      "role" : "LEAD",
      "staff_id" : 18,
      "name" : "Abigail"
    }
  ],
  "_metadata" :
  {
    "etag" : "48A8D3D393B6653A5D2B0359EDAD2D21",
    "asof" : "000025C52A6F11EE"
  }
}
```

14.9.1.5 Flexing the Duality of Duality Views

Work with the same data as JSON documents or relational rows, with changes reflected in both.

Duality gives you a *JSON-first* way to work with your relational data, with both approaches fully functional and always in sync. Tools that prefer JSON use *that* view, while familiar relational ones keep working as before. For example, suppose you just inserted a new action item – *Buy Talulah's Retirement Gift* – with two team members as a JSON document into the duality view. As soon as the *`INSERT`* transaction commits, the query below immediately returns the team member data for that action item.

```
select ai.name as action_name, s.name, tm.role
  from action_item_team_members tm
  join action_items ai
    on ai.id = tm.action_id
  join staff s
    on s.id = tm.user_id
 where ai.id = 58
 order by tm.role, s.name
```

It produces the relational row set result:

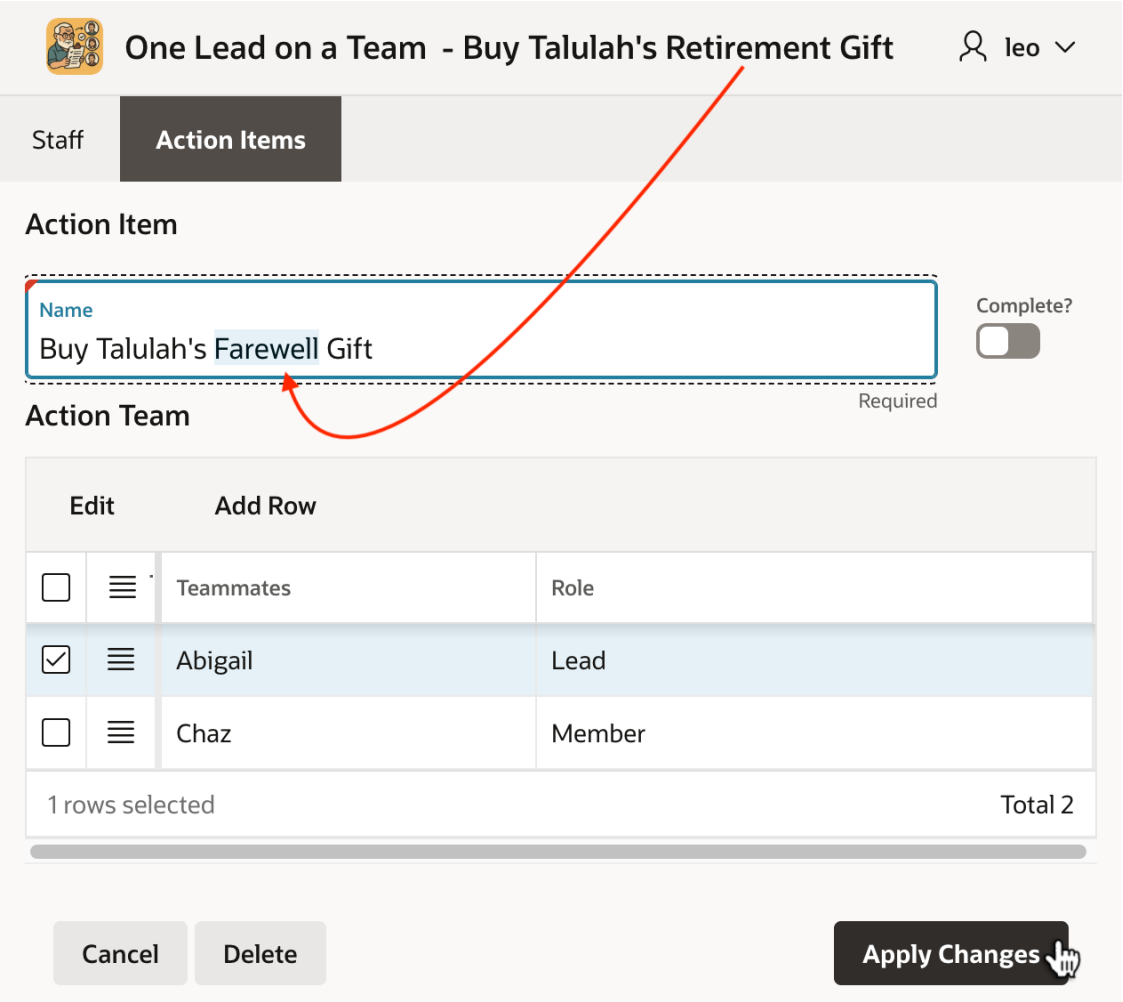
```
ACTION_NAME                    NAME      ROLE
-----------------------------  --------  ------
```


Copyright © 2026, Oracle and/or its affiliates.

```
Buy Talulah's Retirement Gift  Abigail   LEAD
Buy Talulah's Retirement Gift  Chaz      MEMBER
```

Duality means apps using the relational tables keep working normally. For example, imagine end user Leo uses your existing application page shown below to update the new action item's title from *Retirement* to *Farewell*. When he clicks *(Apply Changes)*, the data updates. Any subsequent request of this action item through the duality view instantly reflects the change in the JSON, too.

Figure 14-24 Leo Updates the Title of the New Action Item



14.9.1.6 Preventing Lost Updates with ETAG

A duality view document's ETAG value prevents a stale update from overwriting newer changes.

Suppose you update the same action item through the JSON Relational Duality view, using the *original* JSON document you retrieved *prior* to Leo's update. When you try this, you see a valuable native behavior. For example, say you learn Chaz is busy, so you swap them with

Copyright © 2026, Oracle and/or its affiliates.

Usha using the block of PL/SQL below. As before, it uses a simple `UPDATE` statement to set the duality view's `DATA` column to the new JSON document.

```
declare
    l_json json :=
           -- Modified original JSON document
           -- to swap Chaz with Usha
           json(q'~
            {
              "_id" : 58,
              "name" : "Buy Talulah's Retirement Gift",
              "team" :
              [
                {
                  "role" : "MEMBER",
                  "name" : "Usha"
                },
                {
                  "team_assignment_id" : 128,
                  "role" : "LEAD",
                  "staff_id" : 18,
                  "name" : "Abigail"
                }
              ],
              "_metadata" :
              {
                "etag" : "48A8D3D393B6653A5D2B0359EDAD2D21",
                "asof" : "000025C52A6F11EE"
              }
            }
       ~');
begin
    update action_items_dv
       set data =  l_json
     where json_value(data,'$._id') = 58
    returning data into l_json;
    dbms_output.put_line(json_serialize(l_json pretty));
end;
```

Your attempted update fails with the following error:

```
ORA-42699: Cannot update JSON Relational Duality View 'ACTION_ITEMS_DV':
  The ETAG of document with ID 'FB03C13B00' in the database did not match
  the ETAG passed in.
```

The "`_metadata`" property in the original document contained the ETAG value shown below:

```
"_metadata" : {
   "etag" : "48A8D3D393B6653A5D2B0359EDAD2D21",
   "asof" : "000025C52A6F11EE"
}
```

The ETAG is a computed checksum the Duality View uses to prevent lost updates. When Leo changes the action item's title using your APEX page, it alters the checksum for the row with

Copyright © 2026, Oracle and/or its affiliates.

`_id = 58`. Since your attempted update includes the *old* ETAG, it no longer matches the current value, so the update fails. To fix this, retrieve the latest version of the action item JSON, review the changes, and try the update again.

> **Tip**
>
> When defining your duality view, by default all participating columns are included in the ETAG checksum computation. You can add `@nocheck` annotations to columns you *don't* want to participate.

14.9.1.7 Comparing _id and RESID in Duality View

Use RESID as an alternative document identifier for duality view updates and deletes.

Your attempted update failed before with the following error due to an ETAG mismatch:

```
ORA-42699: Cannot update JSON Relational Duality View 'ACTION_ITEMS_DV':
  The ETAG of document with ID 'FB03C13B00' in the database did not match
  the ETAG passed in.
```

The `ID` referenced in the error message is the *resource id* whose value you can retrieve using the `RESID` pseudo-column. This `RAW`-typed value is *not included* in the JSON document, so REST client applications aren't aware of it unless it is passed out of band in an HTTP header.

To use it as an alternative for the `_id` property in PL/SQL, query it along with the `DATA` column when retrieving the JSON document into a `RAW(4000)` variable. Then use the following predicate in your `UPDATE` or `DELETE` statement:

```
WHERE RESID = l_resid_var_raw
```

instead of:

```
WHERE JSON_VALUE(data,'$._id') = l_id_value
```

14.9.2 Exploring ORDS AutoREST for Duality View

Expose a JSON relational duality view quickly by enabling ORDS AutoREST access.

Once you've created a JSON Relational Duality View, the *fastest* way to expose it to REST clients is to enable AutoREST access on it using SQL Developer Web.

- Enabling AutoREST for a Duality View
  Enable AutoREST on a duality view and choose the object alias clients use in the URL.
- Exercising Duality View AutoREST Endpoint
  Test an AutoREST duality view endpoint to see the database defaults and checks it enforces.
- Enforcing More Complex Business Logic
  Combine PL/SQL handlers with Duality Views to apply custom logic with much simpler code.

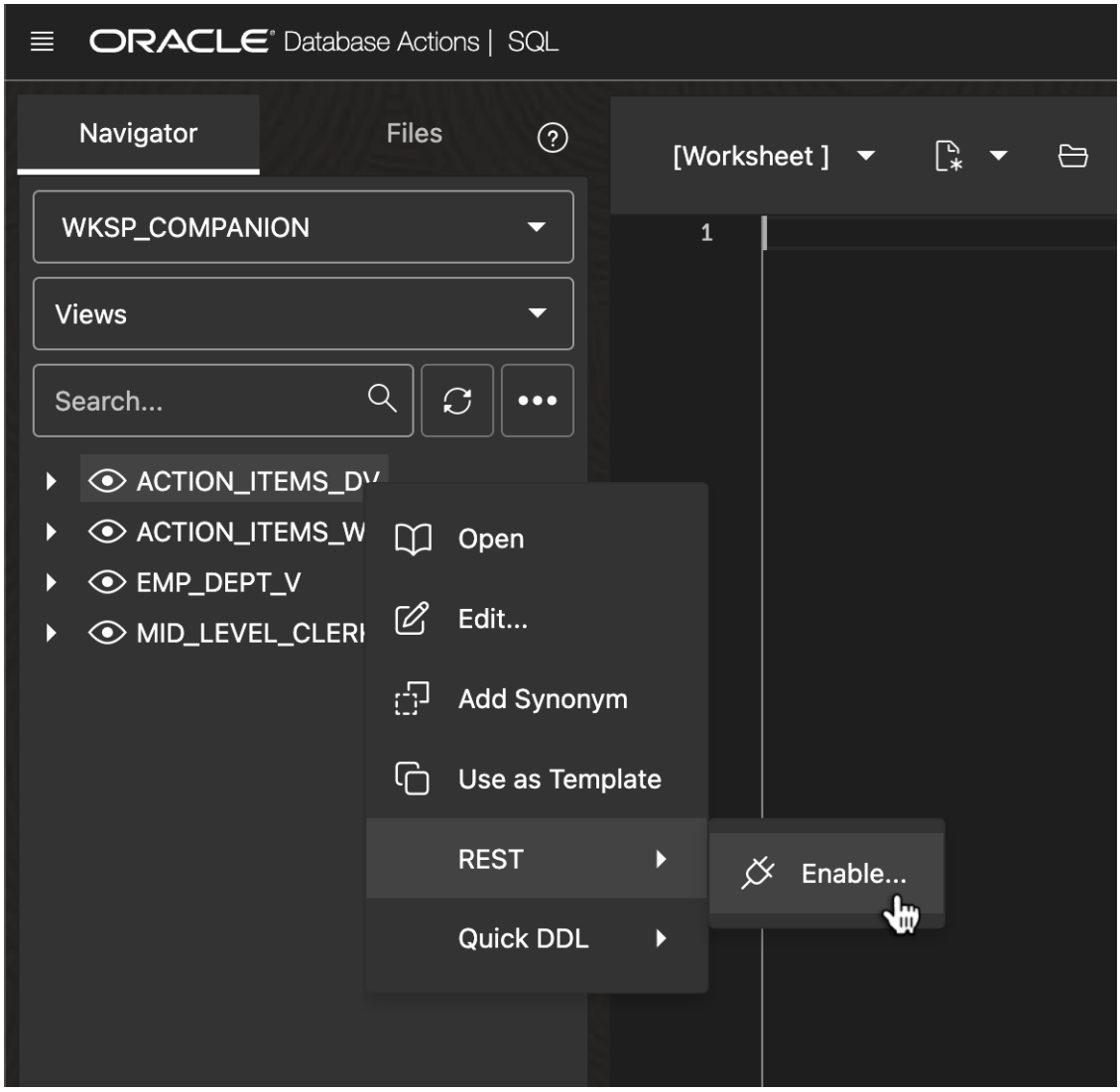
Copyright © 2026, Oracle and/or its affiliates.

14.9.2.1 Enabling AutoREST for a Duality View

Enable AutoREST on a duality view and choose the object alias clients use in the URL.

As shown below, in the SQL Worksheet navigator, enable AutoREST for a duality view like `ACTION_ITEMS_DV` using the *Enable...* option in its *REST* context menu.

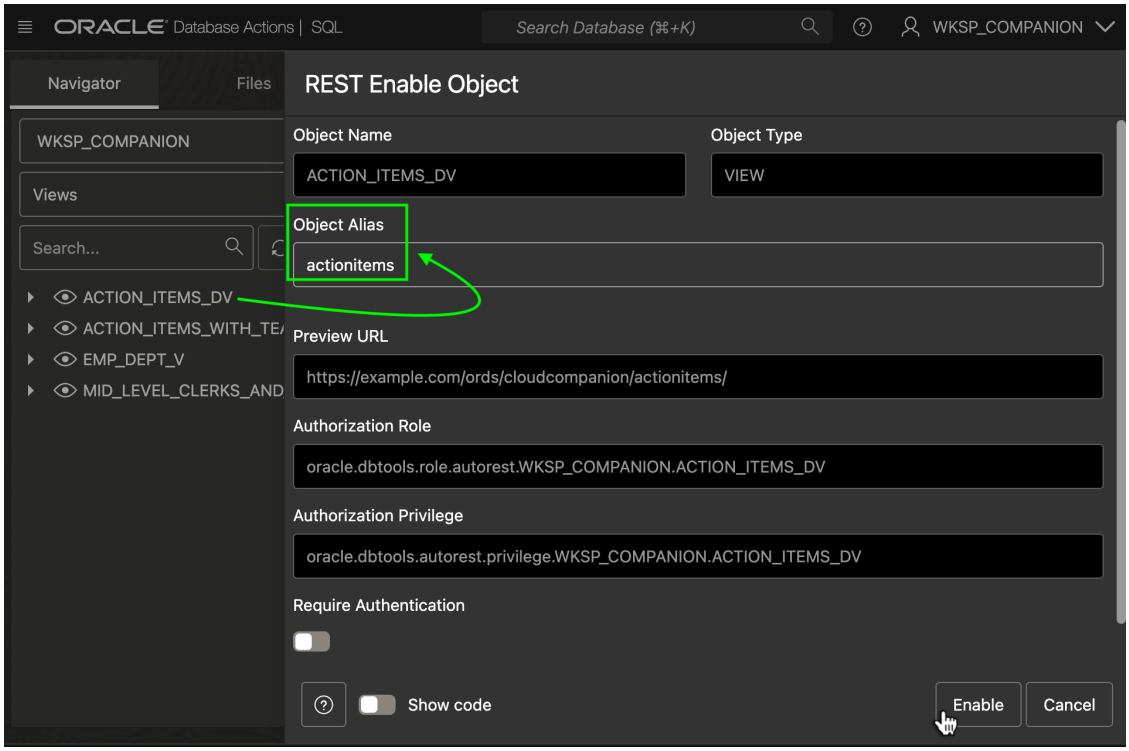
Figure 14-25 AutoREST-Enabling the ACTION_ITEMS_DV Duality View in SQL Developer Web



Study the proposed *Object Alias* and adjust as needed. It's the name clients see in the URL when accessing the AutoREST-enabled duality view REST endpoint. For example, say you change it to `actionitems`.

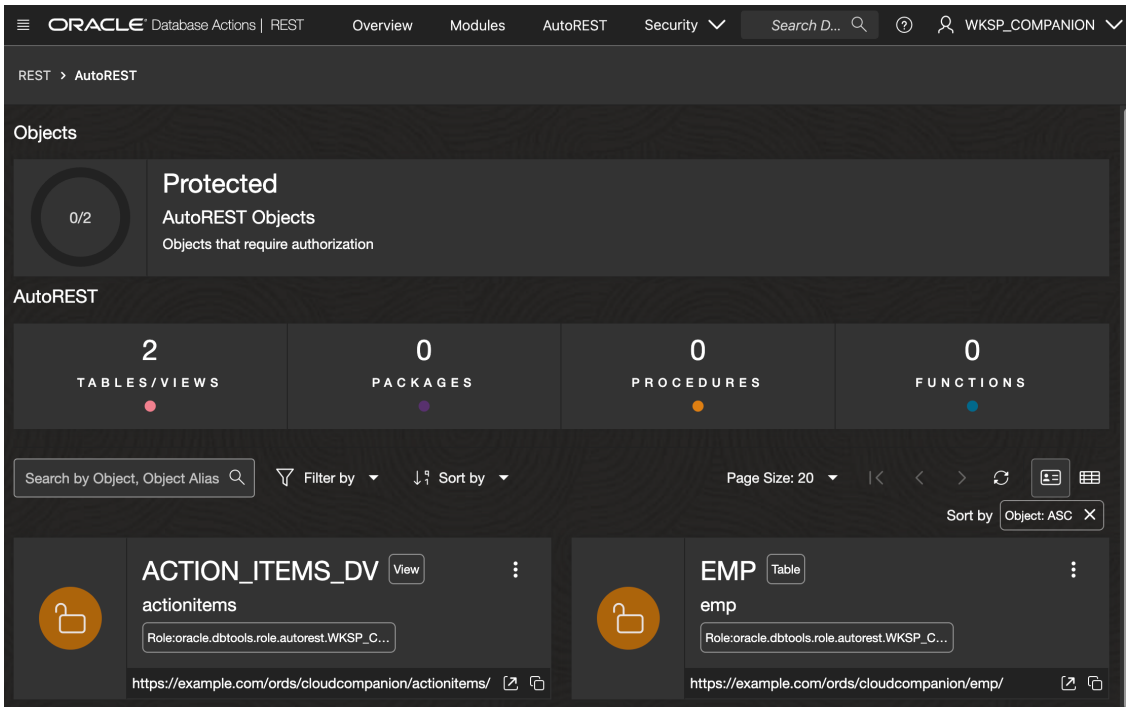
Copyright © 2026, Oracle and/or its affiliates.

Figure 14-26 Configuring Object Alias for AutoREST-Enabled Duality View



Once enabled, as shown below, your duality view appears in the list of *AutoREST*-enabled objects.

Figure 14-27 Browsing AutoREST-Enabled Objects in SQL Developer Web



Copyright © 2026, Oracle and/or its affiliates.

> **Tip**
>
> Once enabled, use the duality view's context menu in the SQL Worksheet navigator tree to access other options.

14.9.2.2 Exercising Duality View AutoREST Endpoint

Test an AutoREST duality view endpoint to see the database defaults and checks it enforces.

AutoREST is the *simplest* way to enable a REST API when the built-in defaulting and checks the database performs meet your needs. By default, a duality view's inherits its underlying tables':

- defaulted column values
- referential integrity checks
- check constraint enforcement, and
- table trigger logic.

It also *only* allows the DML operations you enabled via `@insert`, `@update`, or `@delete` annotations. These native database features extend to your AutoREST-enabled API for the duality view.

Applying these rules to action items, consider the following example. Say you define table-level checks:

- On the `ACTION_ITEMS` table:
  - check constraint `STATUS IN ('OPEN','COMPLETE')`
  - default value on `STATUS` of `'OPEN'`
- On the `ACTION_ITEM_TEAM_MEMBERS` table:
  - check constraint `ROLE IN ('LEAD','MEMBER')`
  - default value on `ROLE` of `'MEMBER'`

These defaults and checks are enforced.

Consider another example where you include a non-existent staff member name `Leopoldo` when `POST`-ing a new action item to the AutoREST-enabled duality view endpoint:

```
{
    "name": "New Product Promotion Deck",
    "team": [
        {
            "role": "LEAD",
            "name": "Usha"
        },
        {"name": "Leopoldo"}
    ]
}
```


Copyright © 2026, Oracle and/or its affiliates.

You get the error:

```
ORA-40937: Cannot insert into table 'STAFF' in JSON Relational Duality View
           'ACTION_ITEMS_DV': Missing INSERT annotation
```

The duality view's natural behavior while processing action team members explains this result:

- If action team staff member id is supplied, use it,
- Otherwise, lookup staff member `ID` in `STAFF` using unique key `NAME` if it exists
- Otherwise, insert new staff member to get new `ID` to reference.

However, the `ACTION_ITEMS_DV` duality view definition *does not allow* inserts into `STAFF`.

14.9.2.3 Enforcing More Complex Business Logic

Combine PL/SQL handlers with Duality Views to apply custom logic with much simpler code.

For example, via an AutoREST-enabled `ACTION_ITEMS_DV` duality view endpoint, the document below would insert successfully with no team lead, and staff member Georgia enrolled twice on the team.

```
{
    "name": "New Product Pitch Deck",
    "team": [
        {"name": "David"},
        {"name": "Georgia"},
        {"name": "Jane"},
        {"name": "Georgia"}
    ]
}
```

Emerging duality view *directives* for validation, augmentation, and notification are designed to extend what a view can enforce declaratively and may cover many common cases. The example in the next section illustrates the *full control* approach, routing REST API changes through a PL/SQL "gatekeeper" package. It uses `ACTION_ITEMS_DV` and applies the application's defaulting and business rules, such as "One Lead per Team", "Team Must Have a Lead", and "No Duplicates".

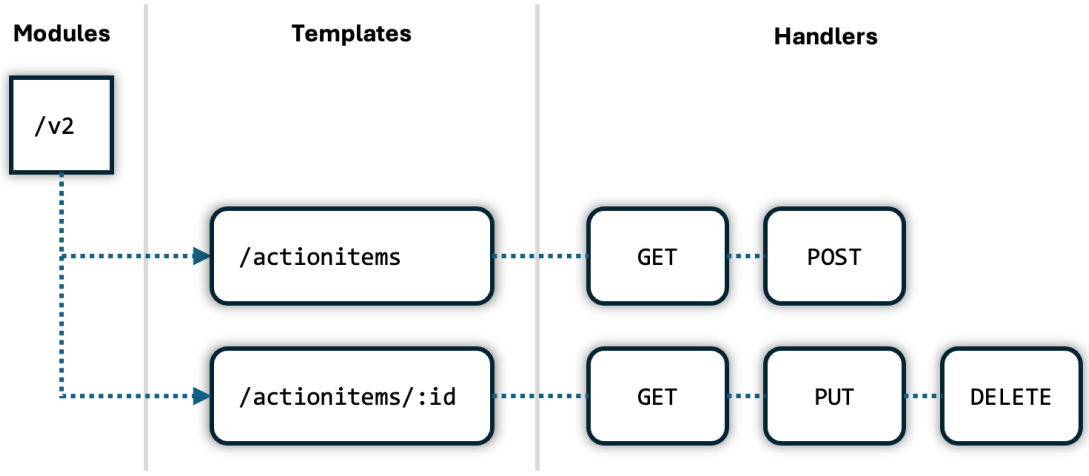
14.9.3 Configuring ORDS Handlers for 26ai API

Set up ORDS handlers to call an alternate API that uses 26ai JSON type and a duality view.

The `v2` module has the same template and handlers as the `v1` module. However, its handlers use 26ai JSON type and Duality Views to simplify the implementation. The diagram shows the module, its templates, and their handlers using the segment of the REST API URL they contribute. The `v2` module has an `/actionitems` template with `GET` and `POST` handlers. Its `/actionitems/:id` template has `GET`, `PUT`, and `DELETE` handlers.

Copyright © 2026, Oracle and/or its affiliates.

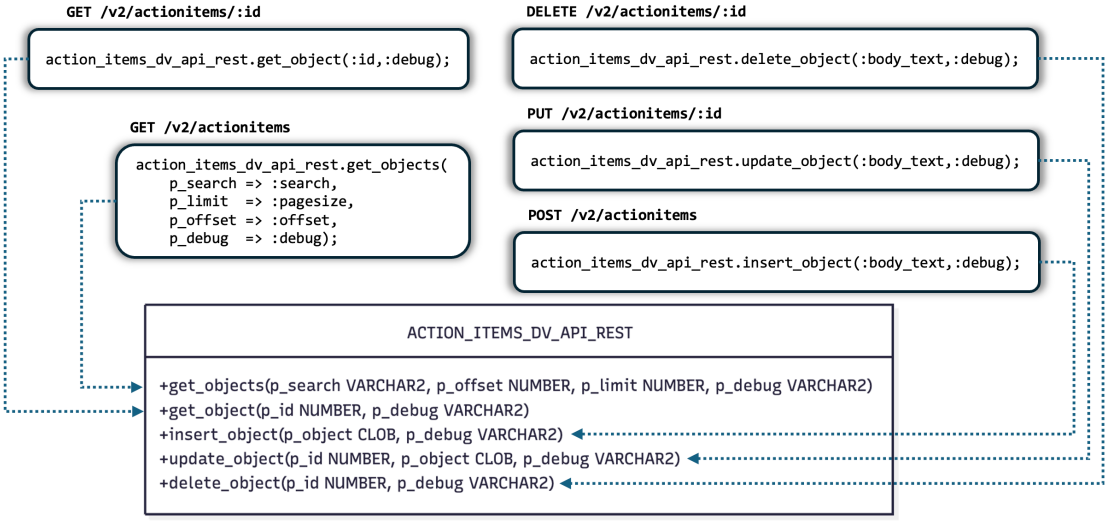
Figure 14-28 ORDS "v2" Module for Action Items API Using JSON Relational Duality View



Similar to the v1 module handlers, each v2 handler shown below calls the indicated procedure in the ACTION_ITEMS_DV_API_REST package:

- GET /v2/actionitems → get_objects
- GET /v2/actionitems/:id → get_object
- POST /v2/actionitems → insert_object
- PUT /v2/actionitems/:id → update_object
- DELETE /v2/actionitems/:id → delete_object

Figure 14-29 Each ORDS Handler Call an ACTION_ITEMS_API_REST Procedure



The ACTION_ITEMS_DV_API_REST accepts the CLOB JSON payload and constructs a JSON type to pass to the main ACTION_ITEMS_DV_API package. The diagram below shows the similar package layering the v1 module uses.

Copyright © 2026, Oracle and/or its affiliates.

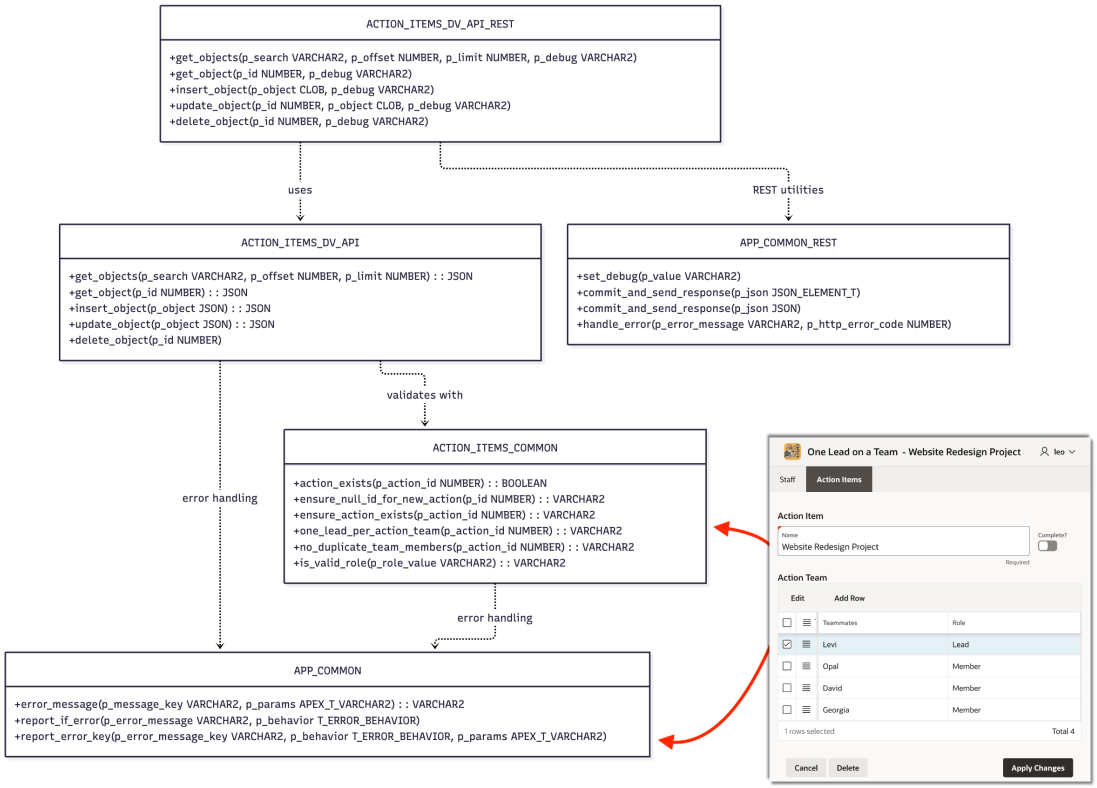
The figure illustrates a package dependency diagram showing:

- `ACTION_ITEMS_DV_API_REST` uses
  - `ACTION_ITEMS_API` for logic
  - `APP_COMMON_REST` as a REST services helper
- `ACTION_ITEMS_DV_API` uses
  - `ACTION_ITEMS_COMMON` for validation
  - `APP_COMMON` for error handling
- `ACTION_ITEMS_COMMON` also uses `APP_COMMON` for error handling

In addition, the diagram shows that the *Action Items* edit page uses the same packages:

- `ACTION_ITEMS_COMMON` for validation
- `APP_COMMON` for error handling

Figure 14-30 Layered Package Design for Sharing Business Logic with a Page



14.9.4 Seeing How Duality View Simplifies Logic

Compare the 26ai duality view API with the 19c version to see how much code it removes.

Using `JSON` type and a Duality View simplifies many aspects of the REST API implementation with custom business logic. For this action items REST API example, the 26ai version requires 51% less code to implement the same functionality:

- 158 lines in `ACTION_ITEMS_DV_API`, versus

Copyright © 2026, Oracle and/or its affiliates.

- 325 lines in Oracle 19*c*-compatible `ACTION_ITEMS_API`.
- Saving JSON Using SQL Without Parsing
  Save JSON directly through a duality view without parsing it into PL/SQL records.
- Transforming JSON Declaratively in SQL
  Use `JSON_TRANSFORM` to add, change, or remove JSON elements declaratively in SQL.
- Processing and Accessing JSON Easily
  Process nested JSON values directly with `JSON_TABLE` and `JSON_VALUE`.
- Querying Predefined JSON Structure
  Query the duality view's `DATA` column to return JSON in the required structure.
- Inspecting the Core 26ai Action Item API
  Inspect the 26ai API package that saves JSON through a duality view.

14.9.4.1 Saving JSON Using SQL Without Parsing

Save JSON directly through a duality view without parsing it into PL/SQL records.

By using the `JSON` type and the `ACTION_ITEMS_DV` duality view, your `ACTION_ITEMS_DV_API` package no longer needs to parse JSON to populate PL/SQL record structure or manage foreign keys. It also lets you insert, update, and delete the JSON with a single SQL statement. The `SAVE` procedure becomes simple and elegant as shown below.

```
-- in package actions_items_dv_api
procedure save(
    p_action_item in out json,
    p_operation   in     t_operation)
is
begin
    case p_operation
        when c_operation_insert then
            insert into action_items_dv(data)
            values (p_action_item)
            returning data into p_action_item;
        when c_operation_update then
            update action_items_dv
               set data = p_action_item
             where json_value(data,'$."_id"')
                   = json_value(p_action_item,'$."_id"');
        when c_operation_delete then
            delete from action_items_dv
             where json_value(data,'$."_id"')
                   = json_value(p_action_item,'$."_id"');
    end case;
end save;
```

14.9.4.2 Transforming JSON Declaratively in SQL

Use `JSON_TRANSFORM` to add, change, or remove JSON elements declaratively in SQL.

Using `JSON_TRANSFORM`, in the `BEFORE_SAVE` procedure you can default the `role` property to the value `MEMBER` on any `team` array object where `role` is `null` or is missing using the following

Copyright © 2026, Oracle and/or its affiliates.

declarative operation. When needed, the function lets you perform multiple transformations in a single command to add, change, or remove JSON document elements.

```
-- Default role to member if not supplied
select json_transform(p_action_item,
            set '$.team[*]?(@.role == null ||
                            !exists(@.role)).role' = 'MEMBER')
into p_action_item;
```

14.9.4.3 Processing and Accessing JSON Easily

Process nested JSON values directly with `JSON_TABLE` and `JSON_VALUE`.

Using a cursor for loop, you can process nested team members using a combination of `JSON_TABLE` and `JSON_VALUE` on the `JSON`-type payload without reparsing the document each time.

```
-- Ensure role values are legal
for k in (select team_member
            from json_table(p_action_item,'$.team[*]'
                 columns (team_member json path '$')))
loop
    app_common.report_if_error(
        action_items_common.is_valid_role(
            json_value(k.team_member,'$.role')));
end loop;
```

14.9.4.4 Querying Predefined JSON Structure

Query the duality view's `DATA` column to return JSON in the required structure.

The `GET_OBJECT` function can simply select the `DATA` column of the duality view to get the JSON document for an action item in exactly the format needed.

```
function get_object(
    p_id in number)
return json
is
    l_ret json;
begin
    app_common.report_if_error(
        action_items_common.ensure_action_exists(p_id));
    select data
      into l_ret
      from action_items_dv
      where json_value(data,'$."_id"') = p_id;
    return l_ret;
end get_object;
```

14.9.4.5 Inspecting the Core 26ai Action Item API

Inspect the 26ai API package that saves JSON through a duality view.

Copyright © 2026, Oracle and/or its affiliates.

The full code of the `ACTION_ITEMS_DV_API` package body appears below.

```
package body action_items_dv_api as

    subtype t_operation is pls_integer range 1..3;
    c_operation_insert      constant t_operation := 1;
    c_operation_update      constant t_operation := 2;
    c_operation_delete      constant t_operation := 3;

    -----------------------------------
    procedure before_save(
        p_action_item in out json,
        p_operation   in     t_operation)
    is
    begin
        if p_operation = c_operation_insert then
            -- Providing an ID with insert not allowed
            app_common.report_if_error(
                action_items_common.ensure_null_id_for_new_action(
                    json_value(p_action_item,'$."_id"')));
        end if;
        if p_operation in (c_operation_insert,c_operation_update) then
            -- Ensure role values are legal
            for k in (select team_member
                        from json_table(p_action_item,'$.team[*]'
                             columns (team_member json path '$')))
            loop
                app_common.report_if_error(
                    action_items_common.is_valid_role(
                        json_value(k.team_member,'$.role')));
            end loop;
            -- Default role to member if not supplied
            select json_transform(p_action_item,
                        set '$.team[*]?(@.role == null ||
                                        !exists(@.role)).role' = 'MEMBER')
            into p_action_item;
        end if;
    end before_save;
    -----------------------------------
    procedure save(
        p_action_item in out json,
        p_operation   in     t_operation)
    is
    begin
        case p_operation
            when c_operation_insert then
                insert into action_items_dv(data)
                values (p_action_item)
                returning data into p_action_item;
            when c_operation_update then
                update action_items_dv
                   set data = p_action_item
                 where json_value(data,'$."_id"')
                       = json_value(p_action_item,'$."_id"');
            when c_operation_delete then
                delete from action_items_dv
```


Copyright © 2026, Oracle and/or its affiliates.

```
            where json_value(data,'$."_id"')
                  = json_value(p_action_item,'$."_id"');
    end case;
end save;
-----------------------------------
procedure after_save(
    p_action_item in json,
    p_operation   in t_operation)
is
    l_error varchar2(4000);
    l_id number := json_value(p_action_item,'$."_id"');
begin
    if p_operation in (c_operation_insert,c_operation_update) then
        app_common.report_if_error(
            action_items_common.no_duplicate_team_members(l_id));
        app_common.report_if_error(
            action_items_common.one_lead_per_action_team(l_id));
    end if;
end after_save;
-----------------------------------
function get_object(
    p_id in number)
return json
is
    l_ret json;
begin
    app_common.report_if_error(
        action_items_common.ensure_action_exists(p_id));
    select data
      into l_ret
      from action_items_dv
      where json_value(data,'$."_id"') = p_id;
    return l_ret;
end get_object;
-----------------------------------
function get_objects(
    p_search in varchar2,
    p_offset in number default null,
    p_limit  in number default null)
return json
is
    l_offset pls_integer := nvl(p_offset,0);
    l_limit  pls_integer := nvl(p_limit,25);
    l_ret json_object_t := json_object_t();
    l_items json_array_t := json_array_t();
    l_row_count  pls_integer := 0;
    l_total_rows pls_integer := 0;
begin
    for j in (select json_value(data,'$."_id"') id,
                     count(*) over() as total_count
                from action_items_dv
               where upper(json_value(data,'$.name'))
                     like upper('%'||p_search||'%')
               order by upper(json_value(data,'$.name'))
               offset l_offset rows
               fetch first l_limit rows only)
```


Copyright © 2026, Oracle and/or its affiliates.

```
            loop
                if l_total_rows = 0 then
                    l_total_rows := j.total_count;
                end if;
                l_row_count := l_row_count + 1;
                l_items.append(json_object_t(get_object(j.id)));
            end loop;
            l_ret.put('items',l_items);
            l_ret.put('hasMore', l_offset + l_row_count < l_total_rows);
            l_ret.put('totalRows',l_total_rows);
            return l_ret.to_json;
        end get_objects;
        ----------------------------------
        function insert_object(
            p_object in json)
        return json
        is
            l_object json := p_object;
        begin
            before_save(l_object,c_operation_insert);
            save(l_object,c_operation_insert);
            after_save(l_object,c_operation_insert);
            return get_object(json_value(l_object,'$."_id"'));
        end insert_object;
        ----------------------------------
        function update_object(
            p_object in json)
        return json
        is
            l_object json := p_object;
        begin
            before_save(l_object,c_operation_update);
            save(l_object,c_operation_update);
            after_save(l_object,c_operation_update);
            return get_object(json_value(l_object,'$."_id"'));
        end update_object;
        ----------------------------------
        procedure delete_object(
            p_id in number)
        is
            l_id_only_object json_object_t := json_object_t();
            l_object json;
        begin
            l_id_only_object.put('_id',p_id);
            l_object := l_id_only_object.to_json;
            save(l_object,c_operation_delete);
        end delete_object;
    end action_items_dv_api;
```

14.10 Securing APIs with Role-Based Access Control

Secure REST APIs with roles and OAuth 2.0 clients for authorized partner access.

Copyright © 2026, Oracle and/or its affiliates.

For production application integration use, secure any REST APIs you create with role-based access control. Then, issue any partner application a secure OAuth 2.0 client for authenticated access to the APIs you authorize them to use.

- Defining ORDS Role and Privileges
  Define ORDS roles and privileges to control access to protected REST API modules.
- Issuing OAuth Client for Authentication
  Issue an OAuth client so an external system can authenticate and access authorized APIs.
- Using REST API with an OAuth Client
  Use an OAuth client to get a bearer token and call authorized REST APIs.

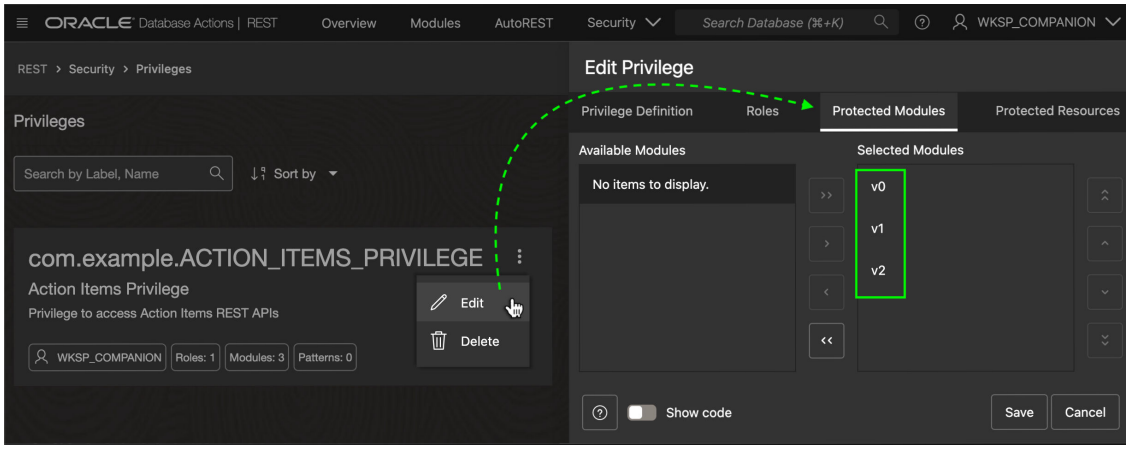
14.10.1 Defining ORDS Role and Privileges

Define ORDS roles and privileges to control access to protected REST API modules.

Define an ORDS Role in SQL Developer Web to name to a group of access privileges. Use the *Security* > *Roles* menu option from the SQL Developer Web REST Designer. For example, create a role named *External Application Integration* to which you can associate the set of privileges required for an external system to integrate with your Action Items application REST API endpoints.

Next, define a privilege to manage access to a specific set of modules or URL patterns. For example, you can create a privilege named `com.example.ACTION_ITEMS_PRIVILEGE` with display label *Action Items Privilege*. As shown below, you configure the privilege to protect the modules `v0`, `v1`, and `v2` created in this section.

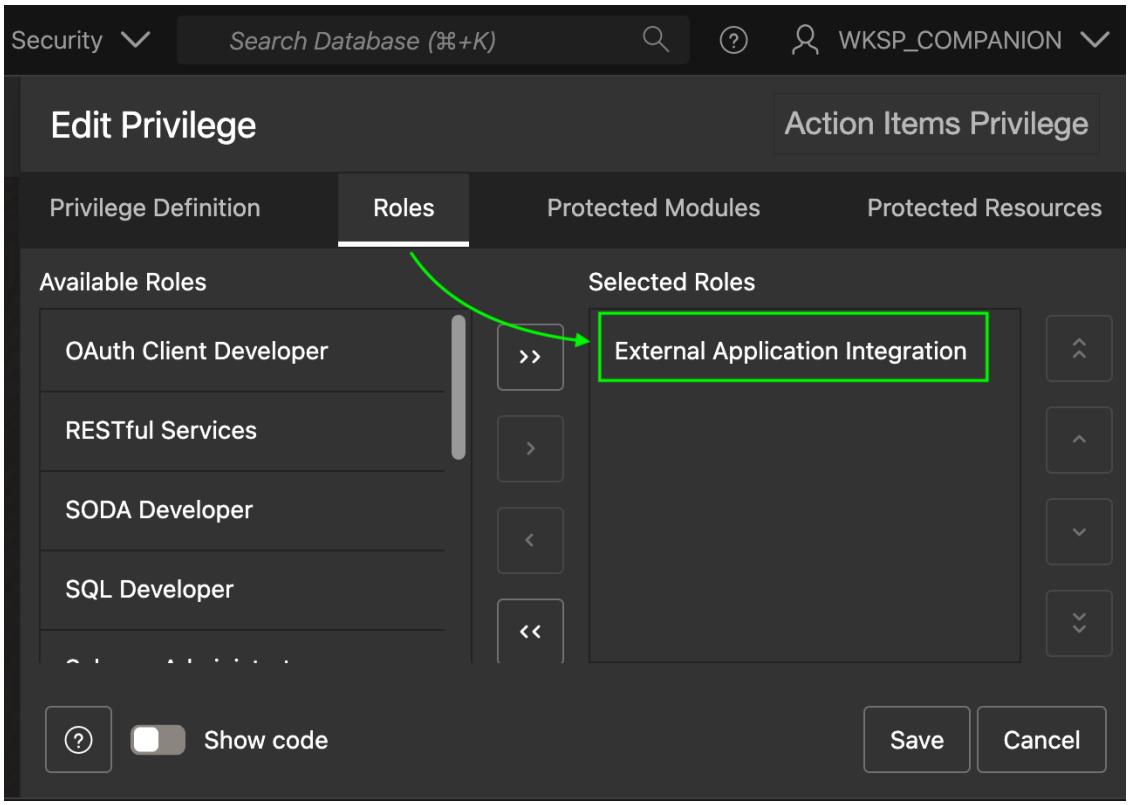
Figure 14-31 Defining a Privilege in SQL Developer Web REST Designer



To grant this privilege to the *External Application Integration* role created above, switch to the *Roles* tab and shuttle it into the *Selected Roles* list as shown below. If you create many different privileges to protect different modules or URL patterns, you can grant all the appropriate privileges in this same way to *External Application Integration* role.

Copyright © 2026, Oracle and/or its affiliates.

Figure 14-32 Granting the Action Items Privilege to External Application Integration Role



14.10.2 Issuing OAuth Client for Authentication

Issue an OAuth client so an external system can authenticate and access authorized APIs.

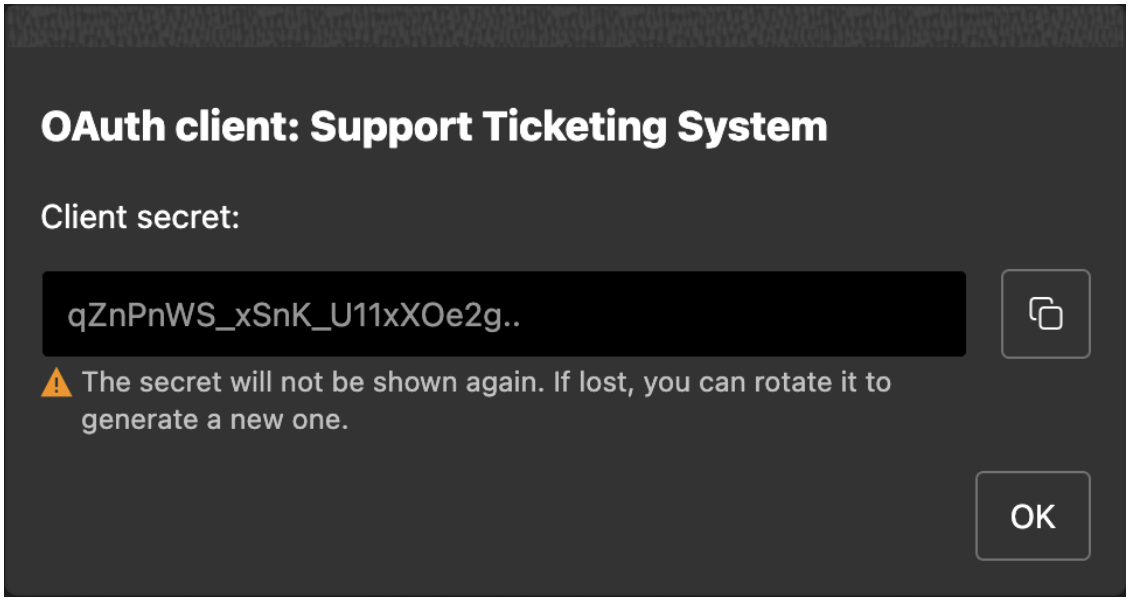
When an external app like the Support Ticketing System needs to access your REST APIs, issue it an OAuth 2.0 client. This acts as its "ticket" for secure access. ORDS generates a client id and secret, which you share securely. The third-party client uses these credentials to get a bearer token and includes it in the `Authorization` header of every REST API call. The role you assign to the OAuth client controls which APIs they can access, based on the privileges granted to that role. Create a *distinct* OAuth client for each external system that needs access.

Use the *Security* > *OAuth Clients* menu in the SQL Developer Web REST Designer to access the list of existing clients. Then, click *(Create OAuth Client)* to add a new one. Choose the *Grant Type* of client credentials (`CLIENT_CRED`), give it a name and description like *Support Ticketing System*, and enter a support email. On the *Roles* tab, associate the new client to *External Application Integration* role to grant it all privileges the role provides.

As shown below, when you finish the process by clicking *(Create)*, ORDS assigns a random client id and client secret. You get ***one chance*** to see the client credential, so store it in your password vault for safe keeping. As shown below, the new OAuth client appears in the list.

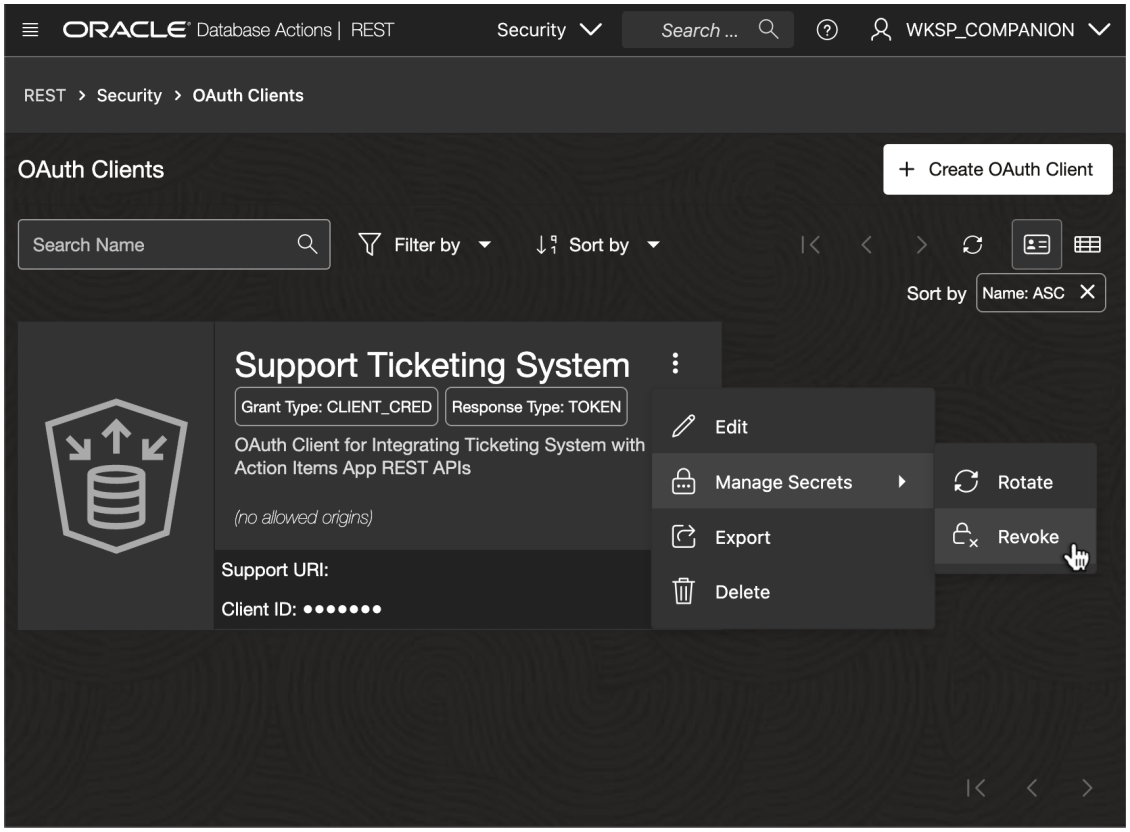
Copyright © 2026, Oracle and/or its affiliates.

Figure 14-33 Store Client Secret Securely, then Share Securely with External App Team



Your new client appears in the OAuth Clients list in SQL Developer Web's REST Designer as shown below. Options on the three-vertical-dots menu let you revoke the secret if needed, or rotate it should it get lost or you want to change it.

Figure 14-34 ORDS OAuth Client Used to Authenticate External System Integration



Copyright © 2026, Oracle and/or its affiliates.

14.10.3 Using REST API with an OAuth Client

Use an OAuth client to get a bearer token and call authorized REST APIs.

To use an OAuth client, the third party system starts by acquiring a bearer token using the client id and client secret provisioned for their integration. You can test the scenario on the command line using `curl`. Notice that the URL shares the same base as your REST APIs up to an including the schema alias (e.g. `cloudcompanion`). Then it ends with the `/oauth/token` path segments. The `--data` flag makes `curl` send a `POST` request with `Content-Type` header set to `application/x-www-form-urlencoded` and a request body containing `grant_type=client_credentials`. If the client id and secret are valid, the token endpoint returns a JSON response containing the bearer-type access token. It also includes the number of seconds that the token is valid. In the example below, the token expires in 3600 seconds, or 1 hour.

```
$ curl --user "QvUVx6Gp8P02dTb0C-II1A..:szUul--kEJ1Te-kS454_tQ.." \
--data "grant_type=client_credentials" \
https://example.com/ords/cloudcompanion/oauth/token

{"access_token":"SNVDe2Tjough44I_X6tmXg","token_type":"bearer","expires_in":36
00}
```

Once the system gets its token, for an hour it can send that token in the `Authorization` header of any REST API request it makes to the APIs their OAuth client's role authorizes them to use. For example, using `curl` this access looks like this. Notice the word `Bearer`, followed by a space, *precedes* the token value in the header.

```
$ curl -H "Authorization: Bearer SNVDe2Tjough44I_X6tmXg"  \
https://example.com/ords/cloudcompanion/v1/actionitems/14
```

As long as the token is still valid, your REST API returns the expected response. After an hour, the external system can contact the token URL again to get a "fresh" token. If they fail to get a new token before it expires, their attempt to use an expired token will response with an HTTP `401` Unauthorized error, including the payload below:

```
{
    "code": "Unauthorized",
    "message": "Unauthorized",
    "type": "tag:oracle.com,2020:error/Unauthorized",
    "instance": "tag:oracle.com,2020:ecid/636fea296204b162106f0cdb5d7e22bc"
}
```


Copyright © 2026, Oracle and/or its affiliates.

15

Geocoding Addresses

Geocoding an address identifies its geographic coordinates.

Having longitude and latitude information for an address makes it easy to:

- Visualize it on a map
- Compute its distance from another address
- Find it when it lies within a certain radius of a point of interest

The *Geocoded Address* page item type lets you do geocoding in any page in the browser. If your application runs on Autonomous Database, you can also do server-side geocoding using a dedicated page process or workflow activity.

- Geocoding with User Confirmation
  Use a *Geocoded Address* page item to show a location and get coordinates for a user-entered address.
- Geocoding in the Background

15.1 Geocoding with User Confirmation

Use a *Geocoded Address* page item to show a location and get coordinates for a user-entered address.

It appears as a map on the page, and works with one or more address-related text items for data entry. If the user changes the address, they must confirm it from a list of likely matches when they submit the page. Under the covers, the address they select includes the latitude and longitude coordinates. The *Geocoded Address* item automatically matches addresses and coordinates using an Oracle-managed service.

> **Tip**
>
> Your application's use of the Oracle eLocation service requires no additional setup or cost.

- Producing GeoJSON from Coordinates
  The *Geocoded Address* page item works with GeoJSON, the web standard format for mapping information.
- Avoiding SQL Expression Update Error
  When you include a SQL Expression column like `GEO_JSON_POINT`, remember to also enable the *Query Only* switch on it in the Property Editor.
- Identifying Address Text Fields
  Associate a *Geocoded Address* page item with one or more related fields that display the address as text.
- Confirming an Address Interactively
  Follow the user experience for entering or editing a geocoded address.

Copyright © 2026, Oracle and/or its affiliates.

- Extracting Coordinates for Storage
  To extract the geocoded address's longitude and latitude into page items, define a dynamic action event handler on the *Result Selection [Geocoded Address]* event.
- Unprotecting Client-Set Hidden Items
  Disable **Value Protected** when client-side logic sets a hidden page item's value.
- Exploring Additional Page Features
  Explore several page techniques used by the Geocoded Address example.

15.1.1 Producing GeoJSON from Coordinates

The *Geocoded Address* page item works with GeoJSON, the web standard format for mapping information.

If you store address coordinates in an `SDO_GEOMETRY` column, or as GeoJSON in a `VARCHAR2` or `CLOB`, your *Geocoded Address* item can read and write address coordinates *directly*. Sometimes, however, you store `LONGITUDE` and `LATITUDE` in separate number columns.

Consider an `EMP_ADDRESSES` table that stores each employee's address, and the following `EMP_AND_ADDRESS` view that joins `EMP` and `EMP_ADDRESS` tables:

```
create or replace view emp_and_address as
select   empno,
       e.ename,
       a.address,
       a.latitude,
       a.longitude
  from emp e
  join emp_addresses a
  using (empno);
```

> **Tip**
>
> When the join column in both tables has the *same* name like `EMPNO` here, then the `USING (EMPNO)` clause is an alternative to `ON e.EMPNO = a.EMPNO` to avoid repeating the column name. Notice in this case that the `SELECT` list *omits* the table alias before the join column name `EMPNO`.

Luckily, the GeoJSON format for a point is simple:

```
{"type":"Point","coordinates":[longitude,latitude]}
```

Starting with a *Form* region based on the `EMP_AND_ADDRESS` view, you can switch it to *SQL Query* type instead. Then, add the SQL Expression below as one additional `SELECT` list column. After doing this, your *Geocoded Address* page item can reference the `GEO_JSON_POINT` column. If the latitude and longitude values are both provided, then the expression returns the GeoJSON document representing a point with those coordinates. Otherwise, the `CASE` statement returns `null`.

```
select EMPNO,
       ENAME,
       ADDRESS,
```


Copyright © 2026, Oracle and/or its affiliates.

```
      LATITUDE,
      LONGITUDE,
      case
        when longitude is not null and latitude is not null then
          json_object(
            'type'        value 'Point',
            'coordinates' value json_array(longitude, latitude))
      end as geo_json_point
from EMP_AND_ADDRESS
```

After adding this column to the Form region's data source query, it will appear in the region as a `P60_GEO_JSON_POINT` page item whose type you can set to *Geocoded Address*.

Tip

Using your helper functions from Simplifying Points and Distance, you could alternatively use the following expression for the `GEO_JSON_POINT` column:

```
sdo_util.to_geojson(make_point(longitude, latitude)) as geo_json_point
```

Caution

Use `JSON_OBJECT` and `JSON_ARRAY` to create GeoJSON. They format decimal coordinates as JSON-compatible decimal values regardless of the end user's browser language. Using a string-based approach like the following would work fine in English and other locales that use the "." as their decimal separator. However, it would fail when running in Italian, for example, where a decimal is formatted like `33,12345` instead.

```
-- NOTE: This is not locale-safe
apex_string.format(
   '{"type":"Point","coordinates":[%s,%s]}',
   longitude,latitude)
```

A string approach can work, but would require formatting the numbers using `TO_CHAR()` by explicitly specifying the `'NLS_NUMERIC_CHARACTERS=.,'` in its optional third parameter.

15.1.2 Avoiding SQL Expression Update Error

When you include a SQL Expression column like `GEO_JSON_POINT`, remember to also enable the *Query Only* switch on it in the Property Editor.

This avoids the error:

```
ORA-01733: virtual column not allowed here
```

that results when the user submits a form page that attempts to update a computed column.

Copyright © 2026, Oracle and/or its affiliates.

15.1.3 Identifying Address Text Fields

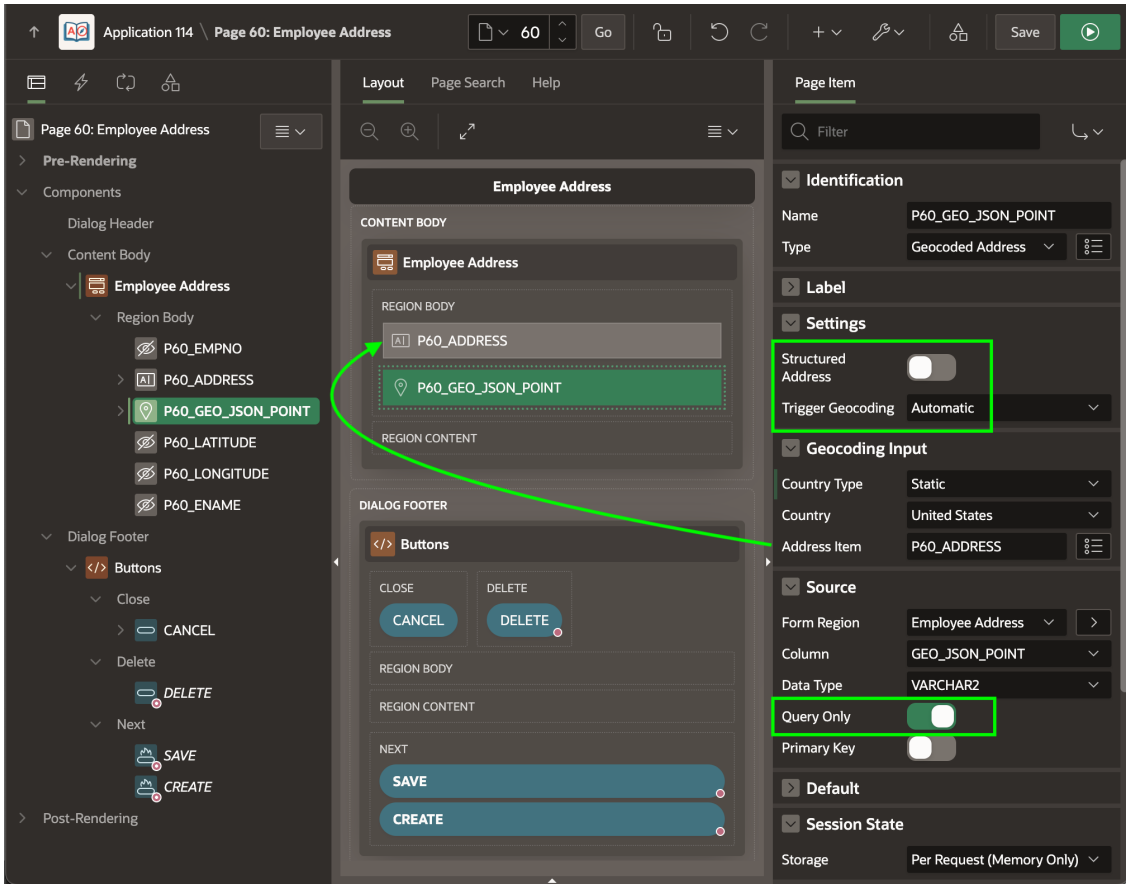
Associate a *Geocoded Address* page item with one or more related fields that display the address as text.

When the Geocoded Address works with an unstructured address, then a single additional *Address Item* like `P60_ADDRESS` shown below does the job. It's where the user can type a value like:

```
3130 Pacific Ave, San Francisco, CA
```

Alternatively, use *separate* page items to handle each part of a *structured* address. By default, the *Trigger Geocoding* is set to *Automatic*. This means the item preempts the page submit to get user confirmation of a modified address. The Oracle-managed geolocation service requires the context of the country name in which to locate the address. You can select a static country value from a list, or configure a *Country* page item to let the end user decide.

Figure 15-1 Configuring Geocoded Address Page Item



15.1.4 Confirming an Address Interactively

Follow the user experience for entering or editing a geocoded address.

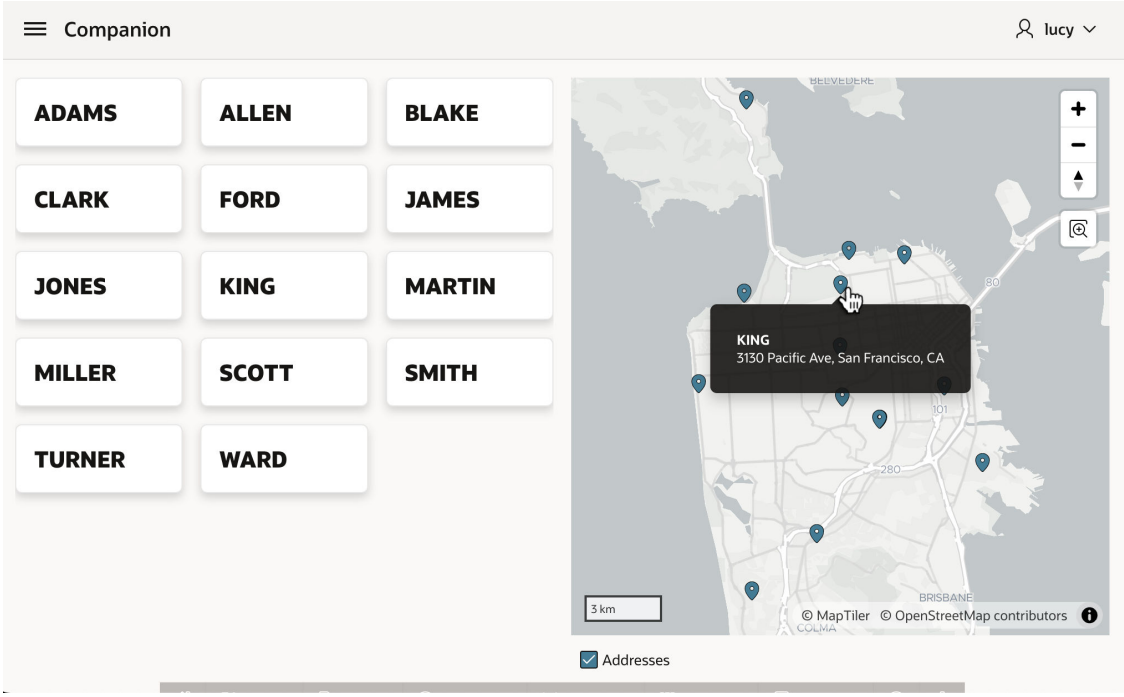
Copyright © 2026, Oracle and/or its affiliates.

Assume employee KING has moved recently, and a user needs to change her address. As shown below, your page can pair a *Cards* region on the `EMP_WITH_ADDRESS` view with a map region showing all their addresses using the technique explained in Using Filtered Data Primary Keys.The map tooltip uses the HTML expression below to include the address information if available.

```
<strong>&ENAME.</strong>{if ADDRESS/}<br>&ADDRESS.{endif/}
```

The figure shows a cards region on the left half of the page displaying employees `ENAME` on small tiles. On the right side a map displays employee's addresses as map markers. The user's mouse hovers over the map marker for employee KING, to see that she currently lives at 3130 Pacific Avenue, San Francisco, California.

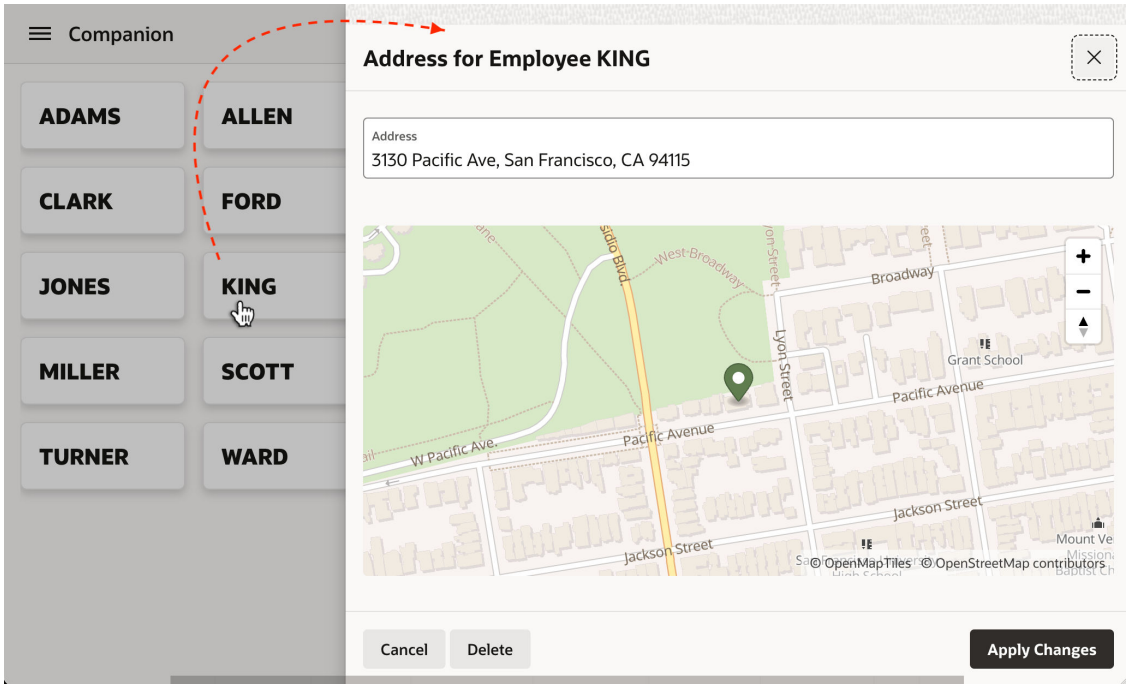
Figure 15-2 Viewing KING's Existing Address in a Map Tooltip



You create a *Full Card* action to open the modal drawer page shown below, passing in the value of the current employee's `EMPNO` as the primary key page item `P60_EMPNO`. So, as shown below, clicking on KING's card opens the modal drawer dialog. The `P60_ADDRESS` text field display the unstructured address. The *Geocoded Address* item `P60_GEO_JSON_POINT` shows the address coordinates your GeoJSON SQL expression produced for the point using the `LONGITUDE` and `LATITUDE` values in KING's row in the `EMP_AND_ADDRESS` view.

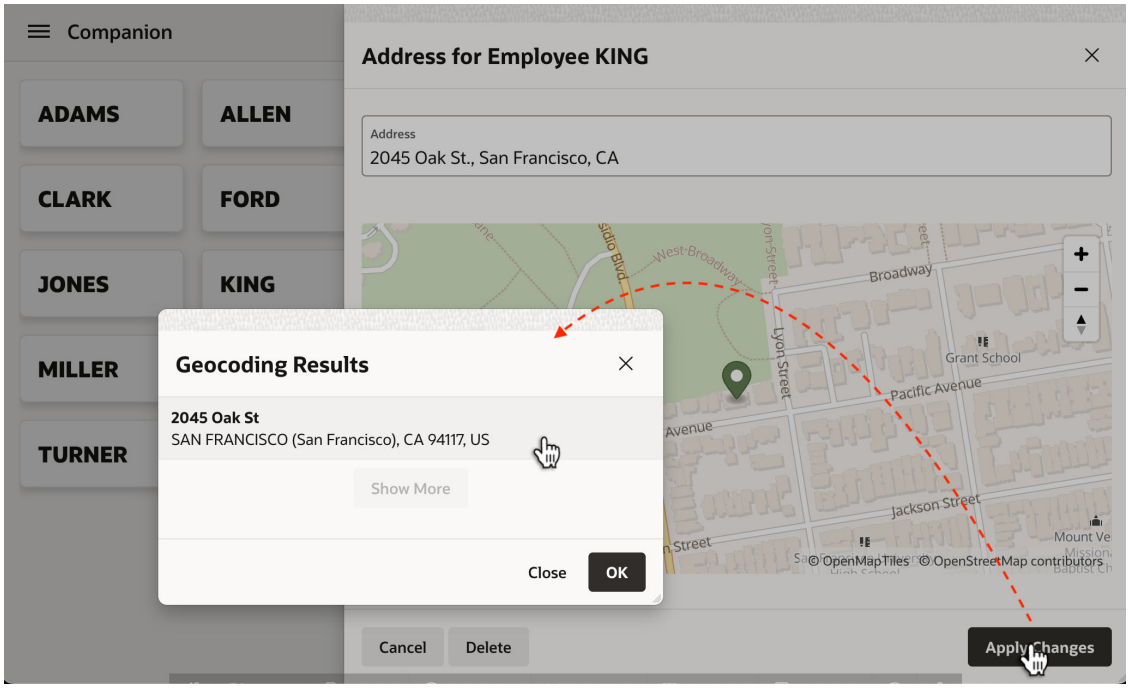
Copyright © 2026, Oracle and/or its affiliates.

Figure 15-3 Geocoded Address Displays Existing Address On Page Load



The user enters a new address and clicks the *(Apply Changes)* button to save. The *Automatic* setting of *Trigger Geocoding* on the `P60_GEO_JSON_POINT` *Geocoded Address* item causes it to preempt the page submit and validate the changed address. As shown below, matching addresses appear in a modal *Geocoding Results* dialog.

Figure 15-4 User Confirms Correct Address

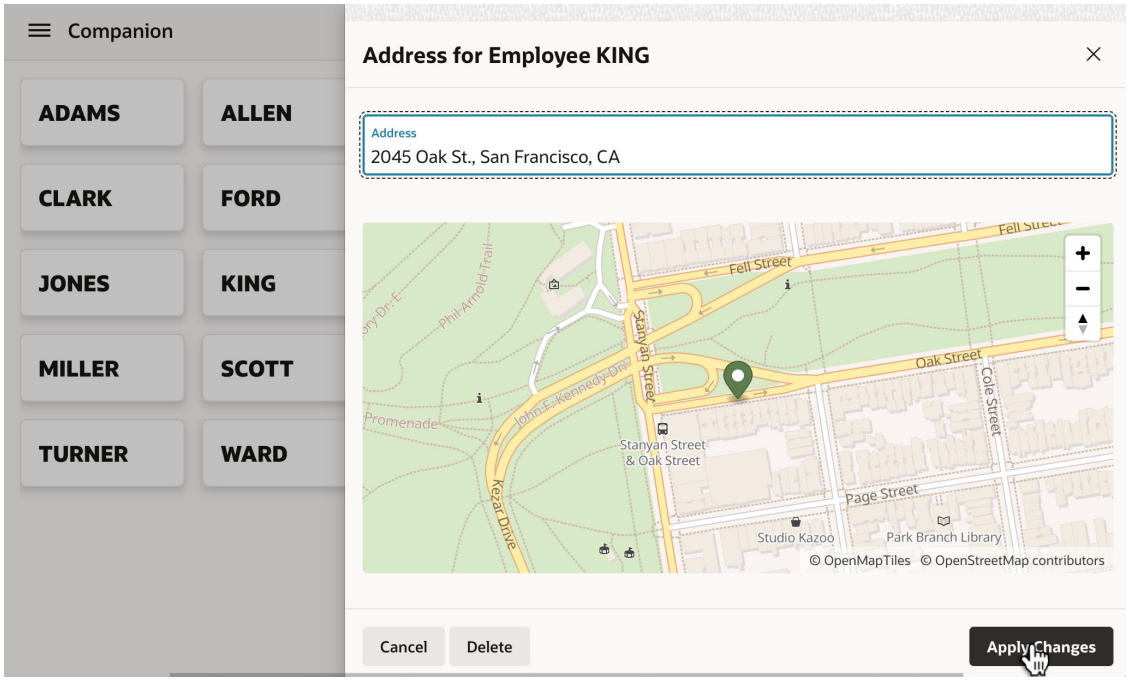


Copyright © 2026, Oracle and/or its affiliates.

> **Tip**
>
> You can customize the title of the *Geocoding Results* dialog by creating a shared component *Text Message* with name `APEX.ITEM.GEOCODE.DIALOG_TITLE` that enables the *Used in JavaScript* option.

Once the user confirms the address, as shown below, she can click *(Apply Changes)* again to save the change.

Figure 15-5 Geocoded Address Map Updates to Show New Address



15.1.5 Extracting Coordinates for Storage

To extract the geocoded address's longitude and latitude into page items, define a dynamic action event handler on the *Result Selection [Geocoded Address]* event.

An *Execute JavaScript Code* dynamic action step with the following two lines handles the task. The first line uses the *shortcut* syntax to set page item `P60_LONGITUDE` to the result of evaluating the `this.data.longitude` expression. The second line shows the *explicit* syntax for setting the value of the `P60_LATITUDE` to the equivalent expression for *that* coordinate value.

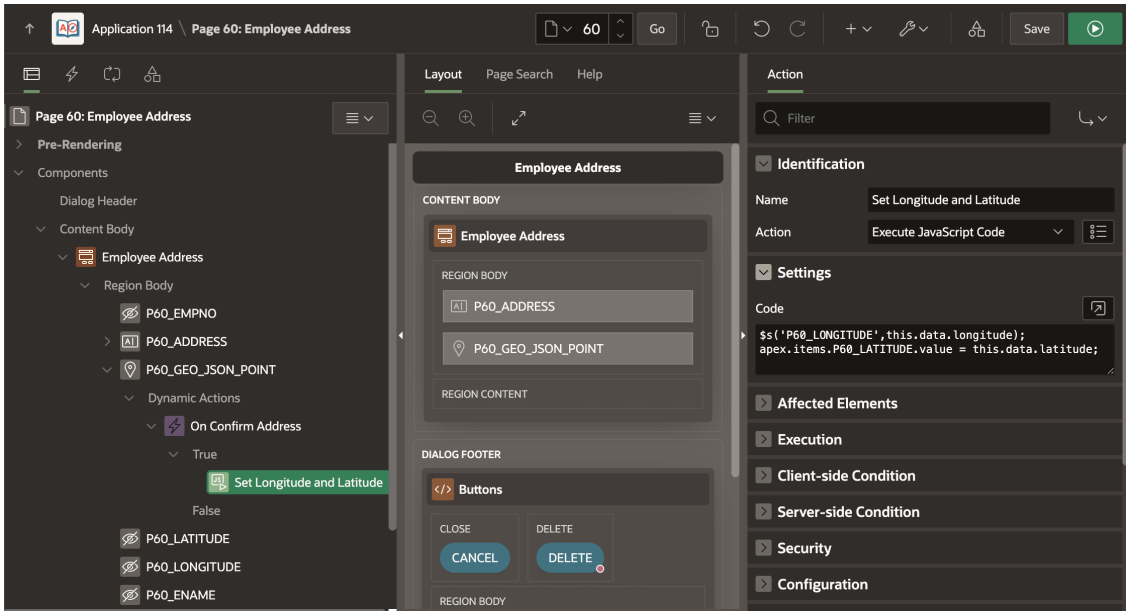
Both assignment approaches accomplish the same result. `$s('item',value)` is less to type, but offers no code completion. Using the *explicit* syntax, as you type each dot in the expression, the Code Editor lets you pick from a list of available choices. Just remember to include the `.value` at the end of the expression after the page item name.

```
$s('P60_LONGITUDE', this.data.longitude);
apex.items.P60_LATITUDE.value = this.data.latitude;
```


Copyright © 2026, Oracle and/or its affiliates.

The figure shows the *Set Longitude and Latitude* dynamic action step in Page Designer that fires when the user selects a result in the Geocoded Address item's confirmation dialog.

Figure 15-6 Extracting Longitude and Latitude from Geocoded Address



The `this.data` object available to a *Result Selection* event's actions also contains other properties. The JSON below shows all the names your `this.data.`*`propName`* expressions can reference.

```
{
  "sequence": 0,
  "latitude": 37.77112,
  "longitude": -122.45304,
  "matchCode": 1,
  "matchVector": "??01010101??004?",
  "matchVectorScore": 100,
  "houseNumber": "2045",
  "street": "Oak St",
  "settlement": "San Francisco",
  "municipality": "SAN FRANCISCO",
  "region": "CA",
  "country": "US",
  "language": "ENG",
  "postalCode": "94117",
  "side": "R",
  "percent": 0.24,
  "edgeId": 120887751
}
```


Copyright © 2026, Oracle and/or its affiliates.

> **Tip**
>
> If the `P60_GEO_JSON_POINT` *Geocoded Address* item were based directly on an `SDO_GEOMETRY` column, or a `VARCHAR2` or `CLOB` containing GeoJSON, then the column could be directly updatable (i.e. *Query Only* = OFF). Saving the coordinates would happen automatically, and no step would be necessary to *extract* the coordinates into separate page items. In addition, no SQL Expression would be necessary to create the point GeoJSON.

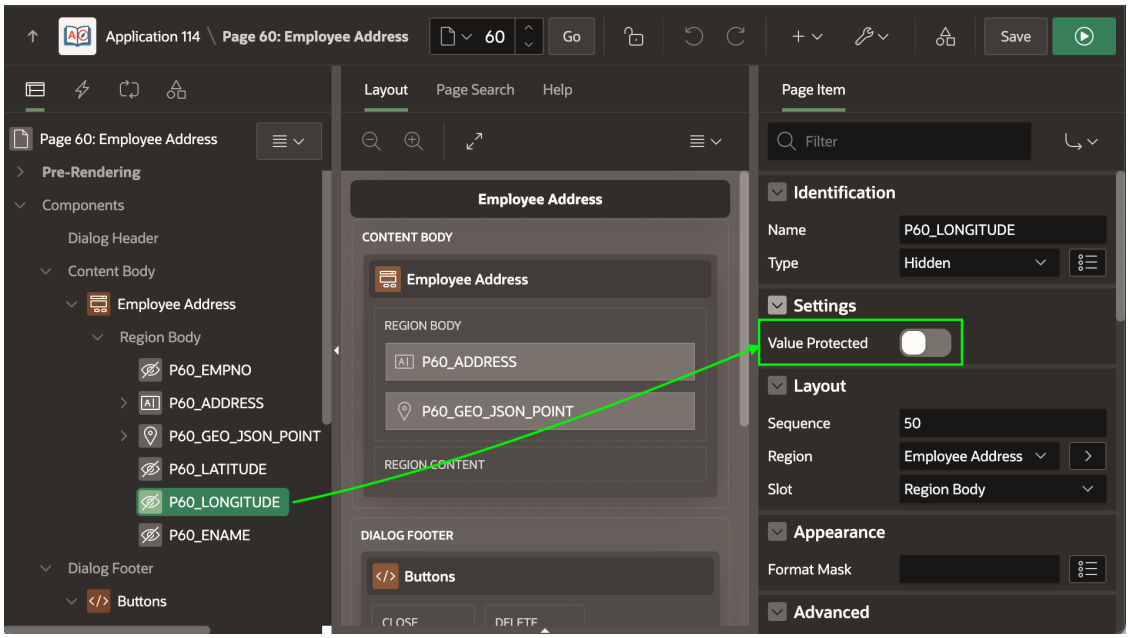
15.1.6 Unprotecting Client-Set Hidden Items

Disable **Value Protected** when client-side logic sets a hidden page item's value.

Since `P60_LONGITUDE` and `P60_LATITUDE` are *Hidden* page items whose value gets set by a dynamic action in the page, remember to *disable* their *Value Protected* switch as shown below. If you *forget* this detail, your end users will see the following error when submitting the page:

```
Session state protection violation:
This may be caused by manual alteration of protected page item P60_LATITUDE.
```

Figure 15-7 Disabling Value Protected on Hidden Item Set by Dynamic Actions



15.1.7 Exploring Additional Page Features

Explore several page techniques used by the Geocoded Address example.

The page in this *Geocoded Address* example uses several additional features worth exploring:

- Clicking an employee address "pin" on the map opens the edit dialog,
- The *Map* on the *Cards* page refreshes when the dialog closes to show the updated address,

Copyright © 2026, Oracle and/or its affiliates.

- Both maps stretch vertically to use the available space,
- The dialog prevents the *[Enter]* key from submitting the page so it can trigger geocoding instead,
- An app-wide JavaScript helper function sets the dialog title, and
- The title references a translatable text message with value placeholders.

- Clicking Map Point to Open Edit Dialog
  Configure a drill down on a map layer item by setting up its link target.
- Refreshing Region on Dialog Close
  Use a dynamic action event handler to refresh a calling page's region when the user closes an edit dialog.
- Stretching to Fill Vertical Space
  The 12-column grid system simplifies filling the available horizontal space. However, sometimes you want a component to stretch *vertically* to use available height.
- Preventing Enter Key from Submitting
  Prevent the *Enter* key from submitting a single-field dialog by adding a visually hidden text field.
- Triggering Geocoding with Enter Key
  Use a native *Trigger Geocoding* dynamic action step to do address confirmation on-demand.
- Including Context Info in Dialog Title
  Use JavaScript to set a modal dialog's title to include context information from the calling page.
- Providing App-Wide Helper Functions
  Put JavaScript helper functions used on multiple pages in an application-wide library.
- Using Translatable Text in JavaScript
  Consider using a translatable Text Message when setting a dynamic dialog title from JavaScript.

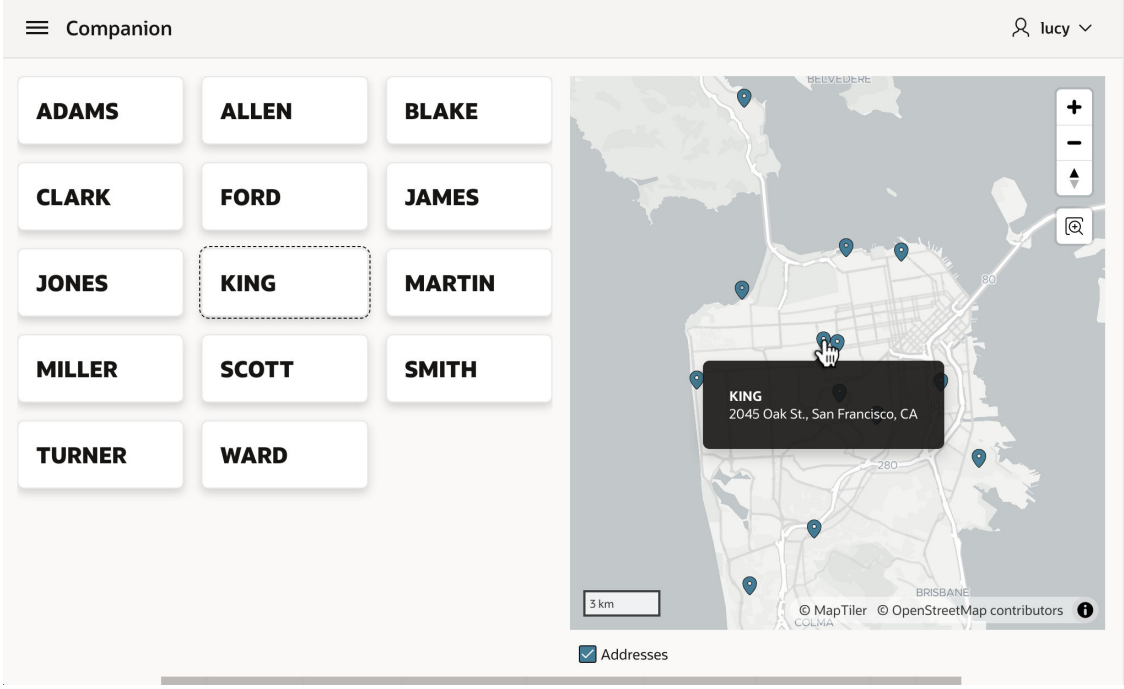
15.1.7.1 Clicking Map Point to Open Edit Dialog

Configure a drill down on a map layer item by setting up its link target.

In the example shown below, notice the map region shows employee addresses on the calling page. While exploring the map by hovering over its points, a user *might* click a point to edit that employee's address. Rather than expecting them to click on a card, you can *anticipate* this user instinct to improve their experience.

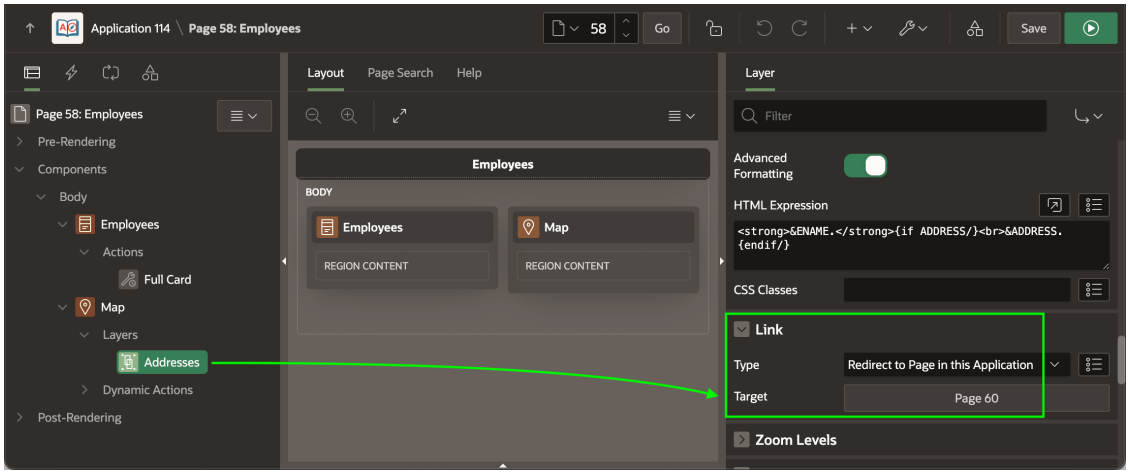
Copyright © 2026, Oracle and/or its affiliates.

Figure 15-8 Clicking Refreshes on Return from Dialog to Show KING's Updated Address



As shown below, after selecting the *Addresses* map layer, you can directly configure the *Link* > *Target* using the *Link Builder* dialog you're familiar with for configuring buttons and links. By choosing page 60 and configuring the value of `P60_EMPNO` to come from the `EMPNO` column, you let the user open the modal *Employee Address* edit dialog using whichever way feels right to them.

Figure 15-9 Linking to Another Page from a Map Point



15.1.7.2 Refreshing Region on Dialog Close

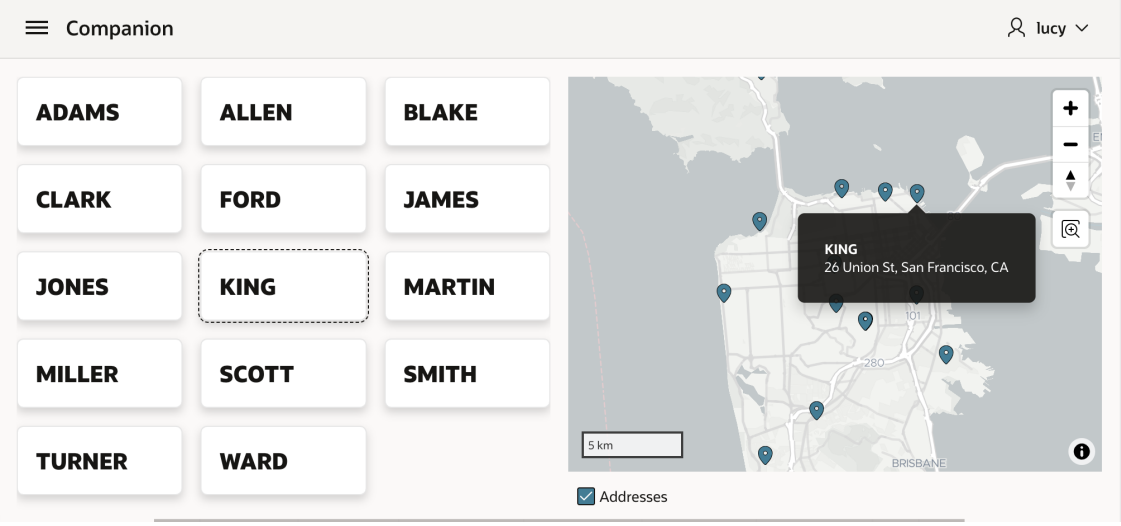
Use a dynamic action event handler to refresh a calling page's region when the user closes an edit dialog.

Copyright © 2026, Oracle and/or its affiliates.

The user clicks a card or a map point. Page 60 opens as a modal drawer to edit that employee's address. When the user *cancels* the dialog by clicking the "x" icon or pressing the *[Esc]* key, the address remains unchanged. However, if they *close* the dialog by clicking the *(Apply Changes)* button, they might have updated the employee's address.

End users expect their change to immediately reflect on the map in the calling page. Just use a *Refresh* action for the map region in a *Dialog Closed* event handler. This ensures that if you update KING's address again, as shown below, the map instantly shows her new location.

Figure 15-10 Map Refreshes on Return from Dialog to Show KING's Updated Address



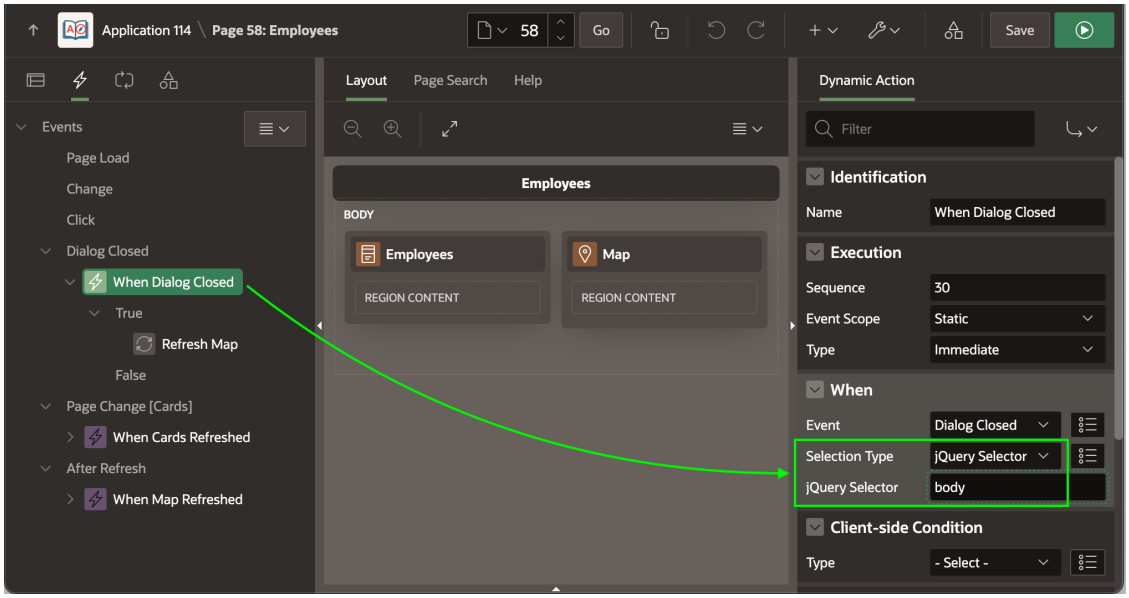
To define a Dynamic Action event handler on the *Dialog Closed* event, select the *Dynamic Actions* tab in Page Designer, right-click on the *Dialog Closed* event heading in the tree, and choose *Create Dynamic Action*.

In the *When* section of the Property Editor, specify *which* dialog's closing should trigger this action using *Selection Type*. For example, you can target a dialog opened by a particular button, by a component in a particular region, or by a particular element using a *jQuery Selector*.

To react to the closing of a dialog opened by *any* element in the page, target the page's `<body>` element that contains all other elements. As shown below, choose *Selection Type* of *jQuery Selector* with a *Selector* of `body` to do this. This ensures the map refresh occurs whether the user opened the dialog by clicking on a card or a map point.

Copyright © 2026, Oracle and/or its affiliates.

Figure 15-11 Configuring a Dialog Close Dynamic Action Event Handler



15.1.7.3 Stretching to Fill Vertical Space

The 12-column grid system simplifies filling the available horizontal space. However, sometimes you want a component to stretch *vertically* to use available height.

- Stretching Vertically Using CSS
  In Cascading Style Sheets (CSS), the *viewport* is the visible browser area. One vertical stretching strategy sizes a component to the remaining vertical space after accounting for other page elements.
- Inspecting Potential Stretch Targets
  Vertical stretching requires inspecting the web page to determine an appropriate container element to target in your CSS rules.
- Calculating Height in Inline CSS
  CSS rules can use calculated expressions.
- Applying Vertical Stretch in a Dialog
  Use CSS to stretch a component vertically in a modal dialog.

15.1.7.3.1 Stretching Vertically Using CSS

In Cascading Style Sheets (CSS), the *viewport* is the visible browser area. One vertical stretching strategy sizes a component to the remaining vertical space after accounting for other page elements.

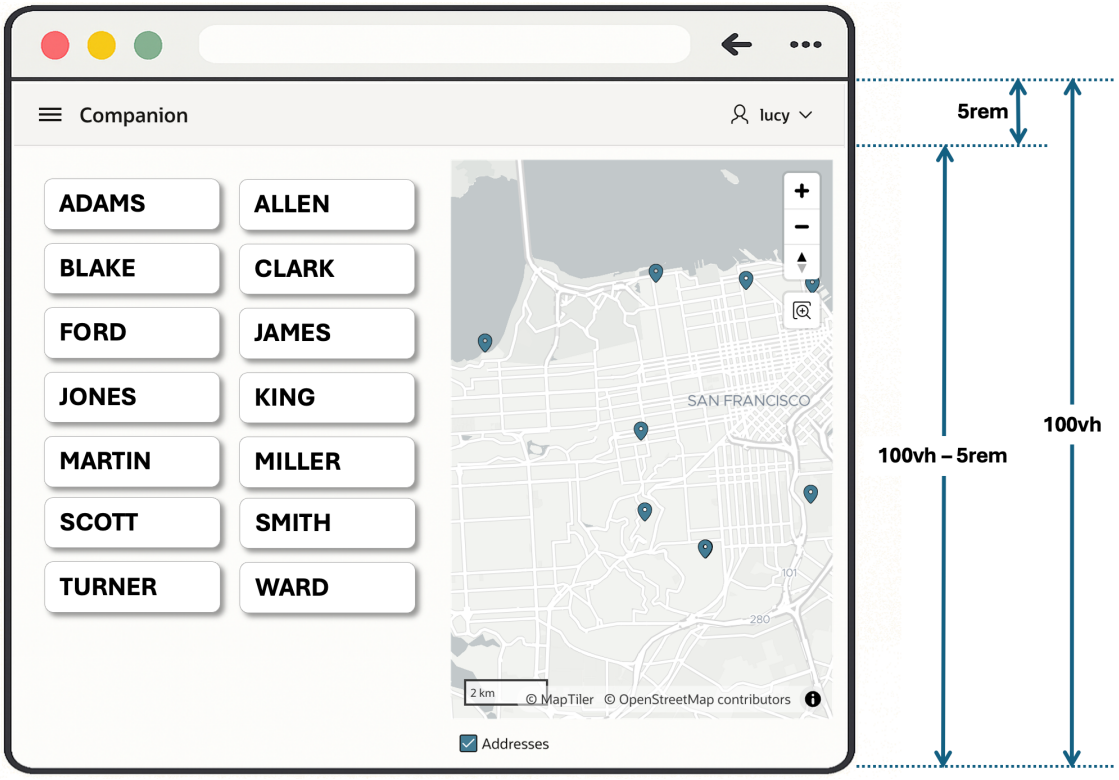
As shown below, the CSS expression `100vh` represents 100% of the **v**iewport **h**eight. One `rem` unit is the height of a character in the current page's *root* font. It's a *relative* unit so it works correctly even when the user zooms or shrinks the font size at runtime. For example, you might estimate the height of the navigation bar to be about five (5) lines of text in the root font size (`5rem`). So the expression below represents the vertical height remaining after accounting for the fixed navigation bar.

```
100vh - 5rem
```


Copyright © 2026, Oracle and/or its affiliates.

The figure illustrates the height of the viewport, the height of the navigation bar, and the difference between the two representing the remaining vertical space.

Figure 15-12 Subtracting Height of Fixed Page Elements from Viewport Height



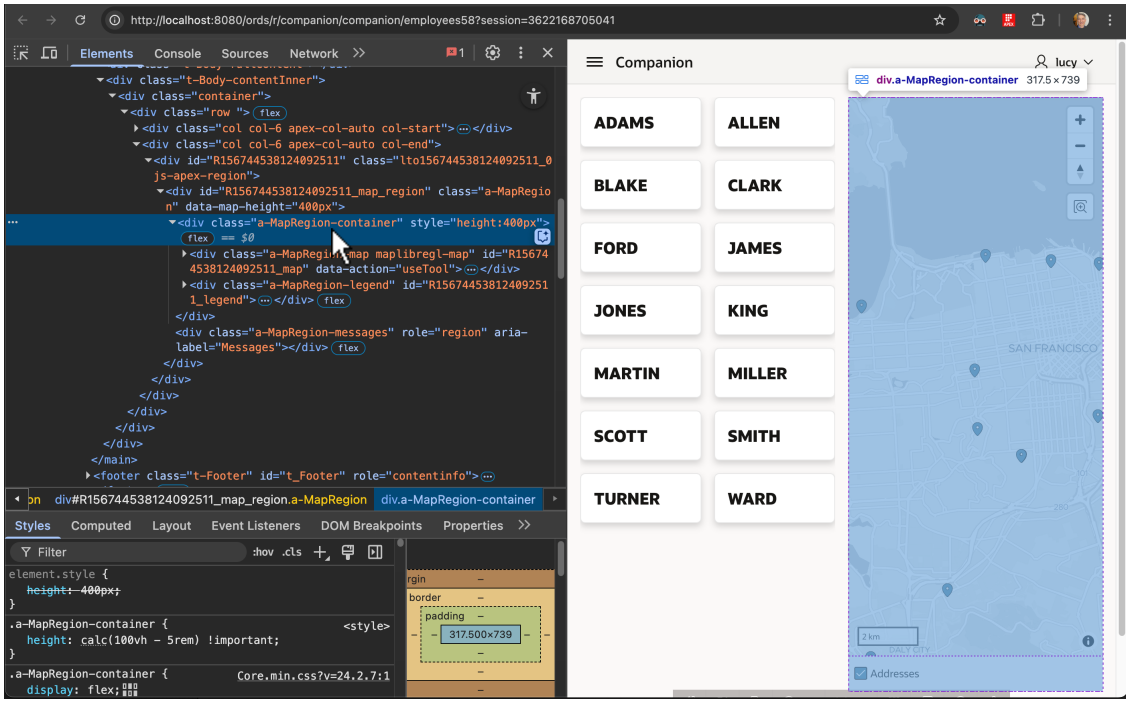
15.1.7.3.2 Inspecting Potential Stretch Targets

Vertical stretching requires inspecting the web page to determine an appropriate container element to target in your CSS rules.

Right click the page and choose the *Inspect* context menu option to activate your browser's *Developer Tools*. As shown below, use the *Elements* tab to identify which element sets the component's height. For example, here the map region's `<div>` element with class `a-MapRegion-container` explicitly sets the height to 400 pixels (`400px`).

Copyright © 2026, Oracle and/or its affiliates.

Figure 15-13 Determining the Element or Class to Target with a CSS Rule



15.1.7.3.3 Calculating Height in Inline CSS

CSS rules can use calculated expressions.

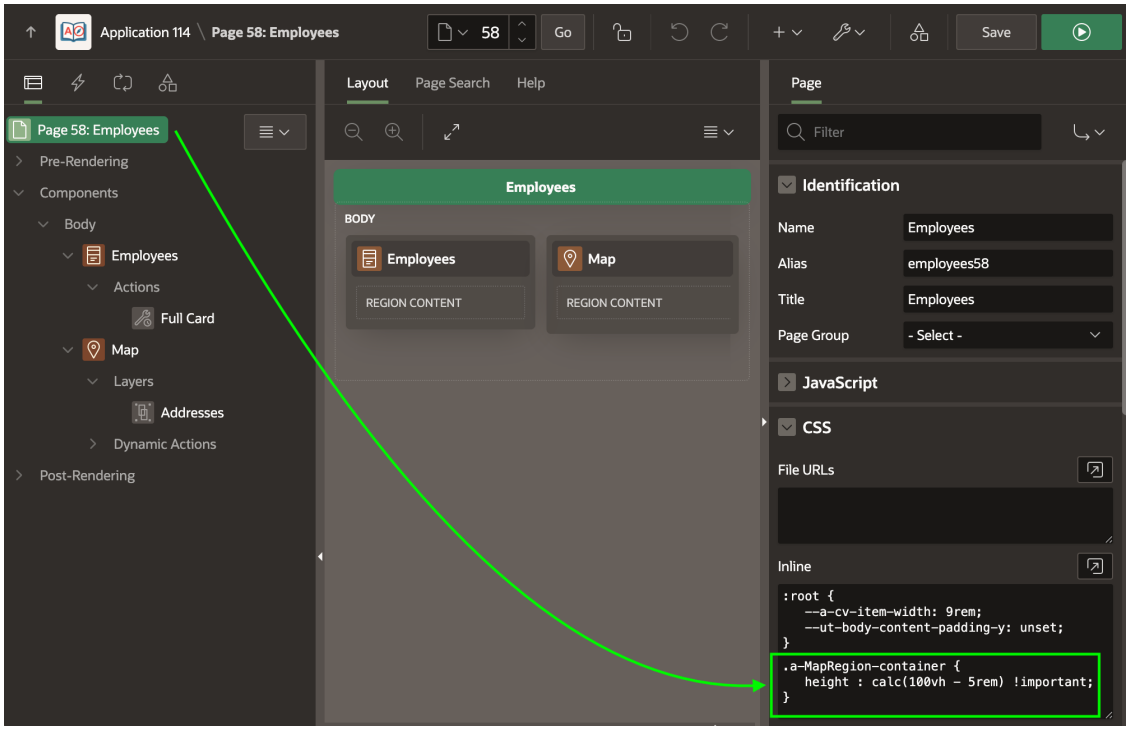
The following CSS rule targets the `a-MapRegion-container` class and uses the `calc()` function to set the `height` of elements with that class to expression `100vh - 5rem`. Notice the class name has a dot before it in the rule. The additional `!important` flag causes this rule to take *precedence* over any default height specification in the page. Like a spreadsheet formula, as the user resizes the browser the CSS `height` updates instantly based on the new viewport size.

```
.a-MapRegion-container {
   height : calc(100vh - 5rem) !important;
}
```

After selecting the page in the component tree, you can add this rule to the page-level *CSS* > *Inline* section as shown below.

Copyright © 2026, Oracle and/or its affiliates.

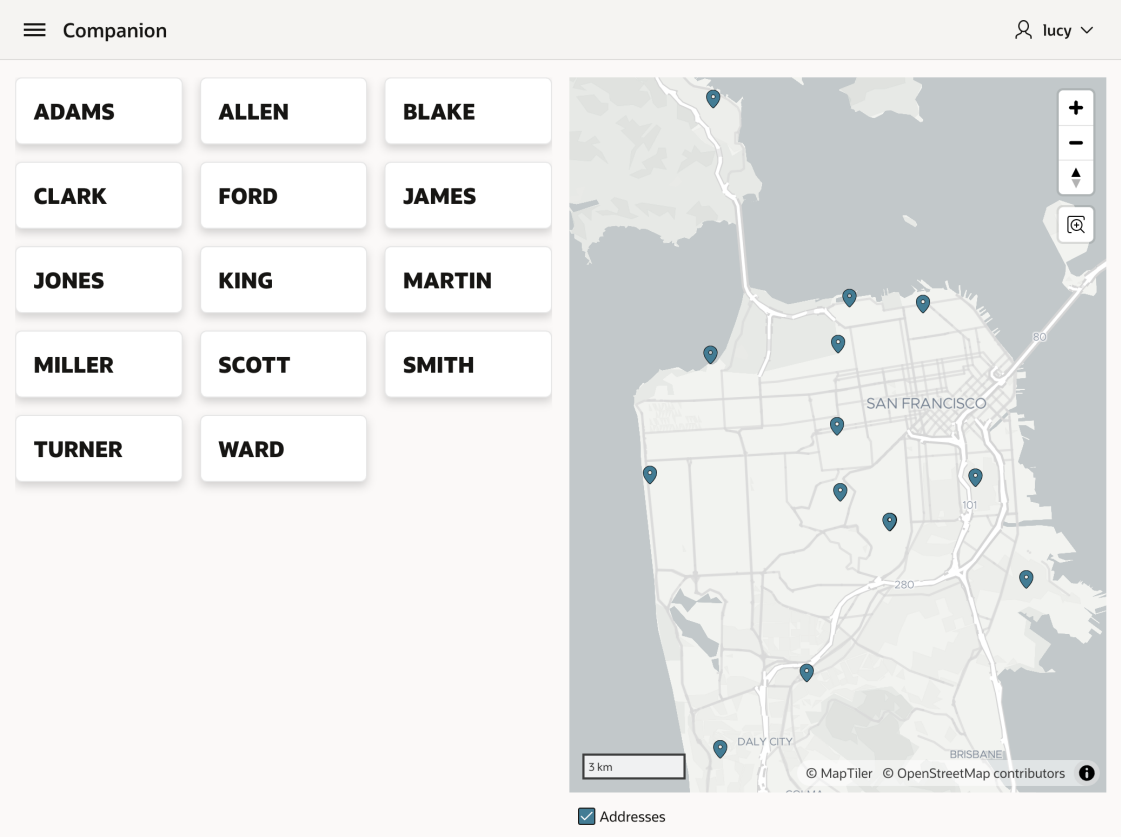
Figure 15-14 Adding Inline CSS Rule to a Page



This produces the outcome below. Notice in the figure that map uses all the remaining vertical height available to give users a better view.

Copyright © 2026, Oracle and/or its affiliates.

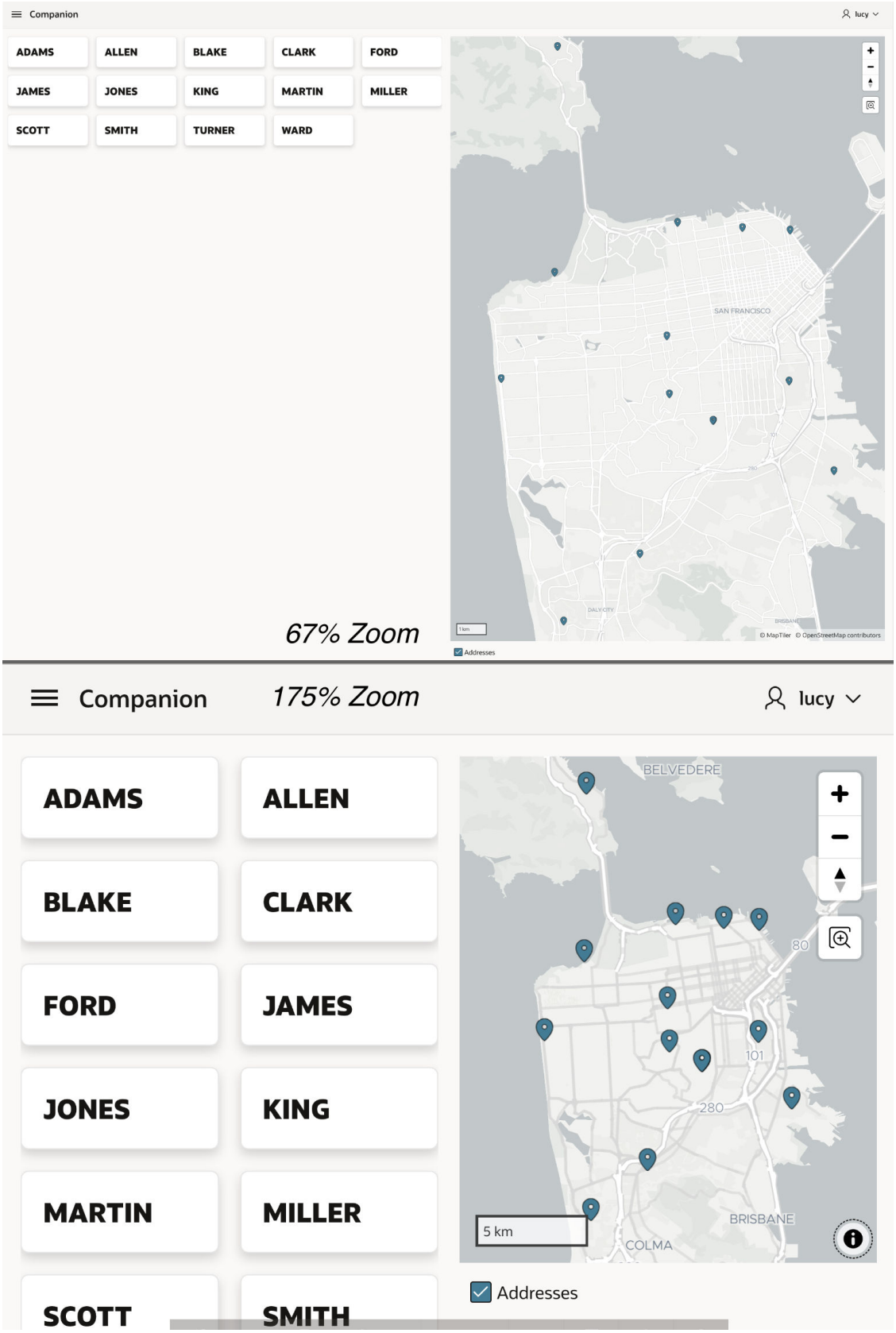
Figure 15-15 Vertical Stretch in Action on a Map Region in a Normal Page



Since you use the relative CSS `rem` units, as shown below the calculated expression works correctly whether the user zooms the font smaller or larger. The figure shows two versions of the same cards and map page of employees. The first is zoomed to 67% and the map uses all the remaining vertical space as expected. The second is zoomed to 175% and the map has adjusted appropriately in that case as well.

Copyright © 2026, Oracle and/or its affiliates.

Figure 15-16 Relative CSS rem Unit Works Correctly at Any Zoom Percent



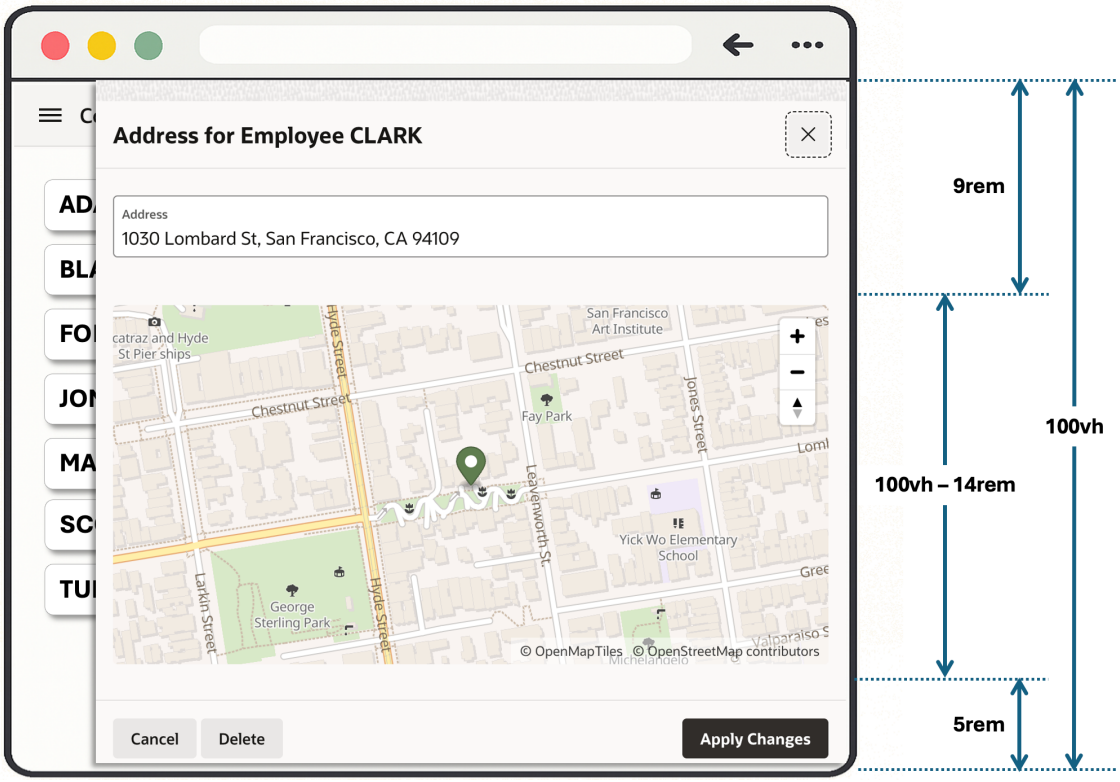
Copyright © 2026, Oracle and/or its affiliates.

15.1.7.3.4 Applying Vertical Stretch in a Dialog

Use CSS to stretch a component vertically in a modal dialog.

As shown below, to stretch the *Geocoded Address* map in the modal drawer page, the fixed amount of page contents includes the title bar and *Address* field at the top as well as the dialog footer containing the buttons at the bottom. After some experimentation you settle on `100vh - 14rem` as the expression that best approximates the remaining vertical space. This example highlights the *approach*: the actual amount to subtract for a specific dialog you build depends on your use case. The figure illustrates the empirical heights in CSS `rem` units of the dialog footer (`5rem`), the viewport (`100vh`), and the header and other contents above the map (`9rem`), along with the calculated height of the map component to stretch vertically.

Figure 15-17 Vertical Stretching May Need to Account for Fixed Content Above and Below



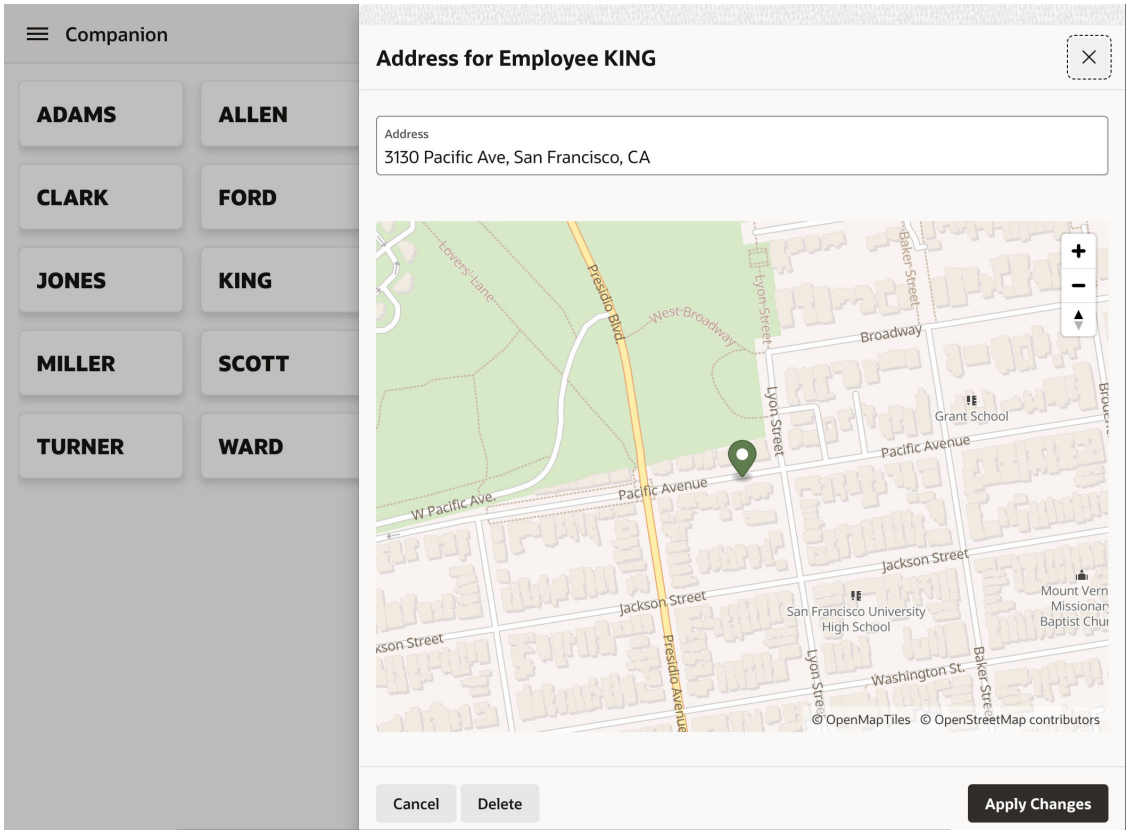
Using the browser Developer Tools, you identify the `a-GeoCoder-map` class as the one to target, so you add the following rule to the modal drawer page's *CSS* > *Inline* section:

```
.a-GeoCoder-map  {
   height : calc(100vh - 14rem) !important;
}
```

In no time, as shown below, your modal dialog's *Geocoded Address* map is now also efficiently using the remaining vertical space. The figure shows the Geocoded Address map occupying the entire space below the **Address** text field and above the dialog footer.

Copyright © 2026, Oracle and/or its affiliates.

Figure 15-18 Vertical Stretch in Action in a Model Drawer for a Geocoded Address Map



15.1.7.4 Preventing Enter Key from Submitting

Prevent the *Enter* key from submitting a single-field dialog by adding a visually hidden text field.

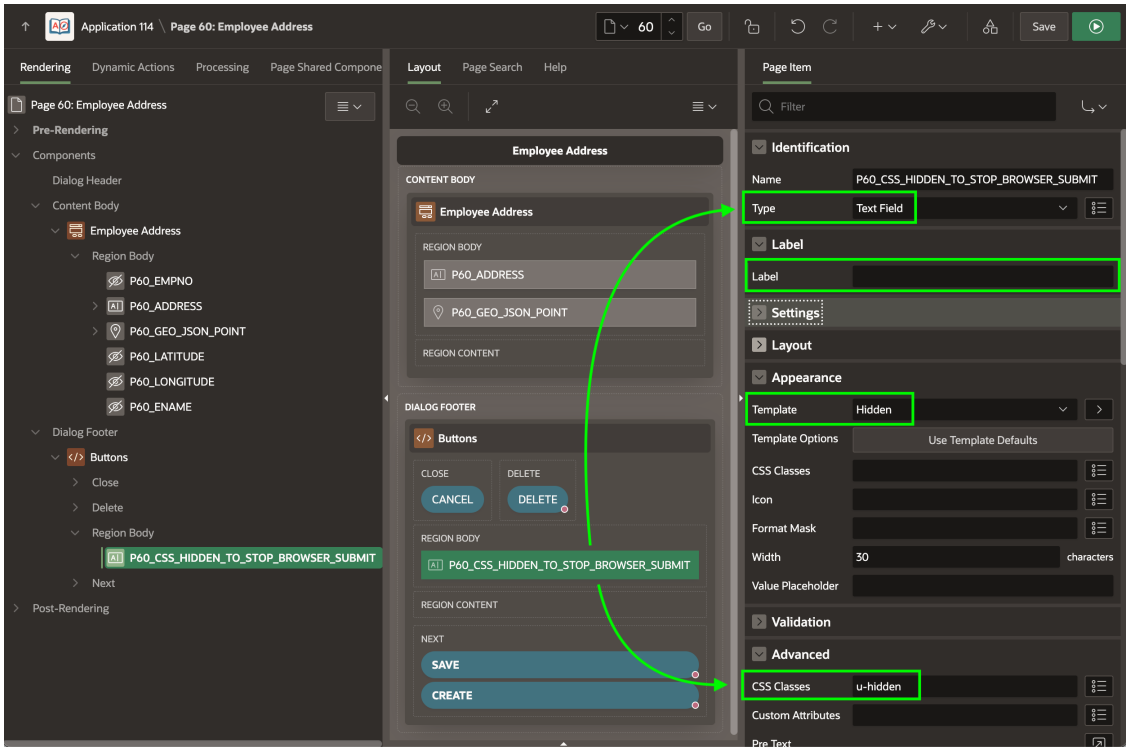
When a user presses the *[Enter]* key in a web page, for historical reasons, all browsers submit the page when it contains just a *single* input field. In contrast, if the page has two or more input fields, then the *[Enter]* key does ***not*** do this.

The *Employee Address* modal drawer is a "single input field" page. Technically, its only input field is `P60_ADDRESS` since the *Geocoded Address* page item `P60_GEO_JSON_POINT` renders as a map. In this situation, the simplest solution to avoid *[Enter]*'s submitting the page is to add a *second* input field. By *hiding* this second field as described, the end user does not even know it's there.

When adding this hidden field, give it a name that clarifies its special purpose. As shown below, leave its page item type as *Text Field* instead of *Hidden*. Then add the Universal Theme CSS class `u-hidden` to its *Advanced* > *CSS Classes* property to hide it instead. Notice the label is blank and the *Appearance* > *Template* is set to *Hidden* as well. Despite its not being *visible* to the user, the browser counts this second field which avoids the *[Enter]* key from submitting the page.

Copyright © 2026, Oracle and/or its affiliates.

Figure 15-19 Adding CSS-Hidden Field to Stop Enter Key from Submitting Page



15.1.7.5 Triggering Geocoding with Enter Key

Use a native *Trigger Geocoding* dynamic action step to do address confirmation on-demand.

For example, as shown below, you can use it from a dynamic action event handler on the *Key Press* event of the `P60_ADDRESS` page item. Use the following JavaScript expression in the event handler's *Client-side Condition* to target the pressing of the *[Enter]* key:

```
this.browserEvent.key === "Enter"
```

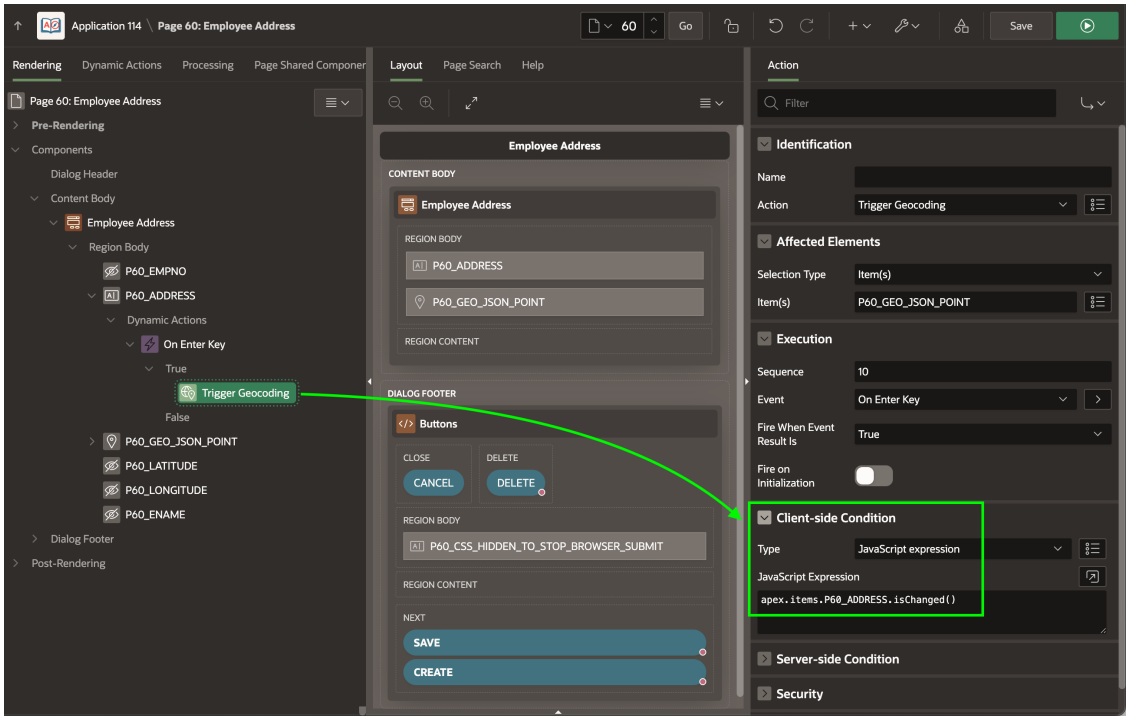
In the *Client-side Condition* of the *Trigger Geocoding* action step, use this JavaScript expression to ensure it only happens if the user has changed the address before pressing *[Enter]*.

```
apex.items.P60_ADDRESS.isChanged()
```

With these configurations in place, now the user gets immediate geocoding feedback if they press this *[Enter]*. When they later click *(Apply Changes)*, the *Automatic* geocode trigger is smart enough to avoid *repeating* the process if it was already done on-demand.

Copyright © 2026, Oracle and/or its affiliates.

Figure 15-20 Triggering Address Geocoding Using the Enter Key

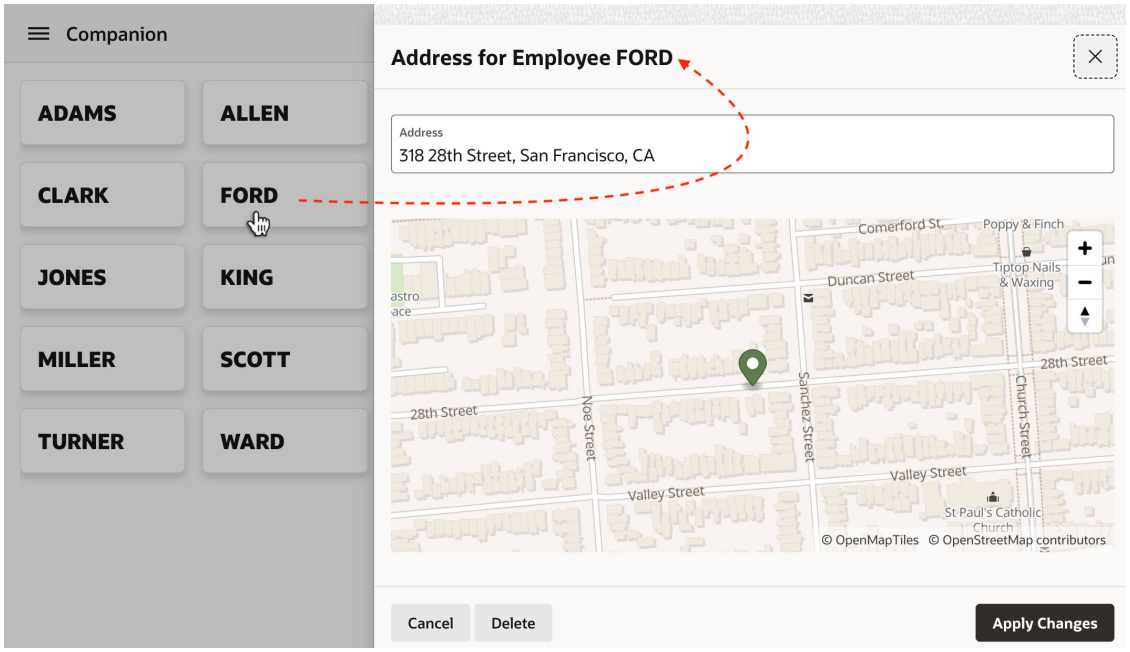


15.1.7.6 Including Context Info in Dialog Title

Use JavaScript to set a modal dialog's title to include context information from the calling page.

As shown below, when opening a modal dialog, it can be useful to include some context information in the dialog title like the name of the employee whose address the user is editing.

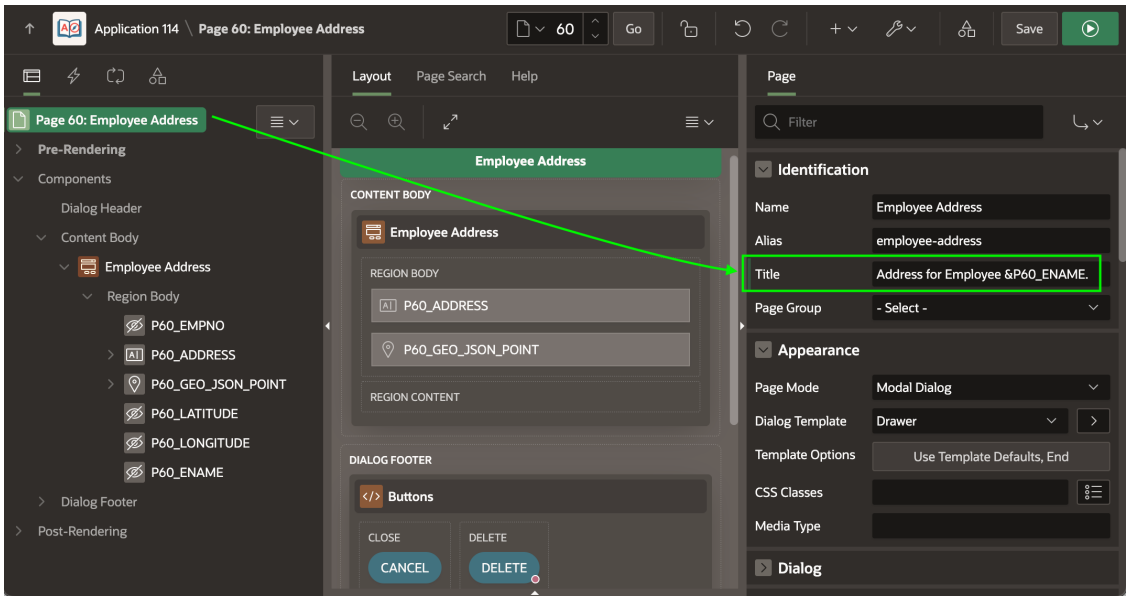
Figure 15-21 Showing ENAME in Modal Dialog Provides Useful Context



Copyright © 2026, Oracle and/or its affiliates.

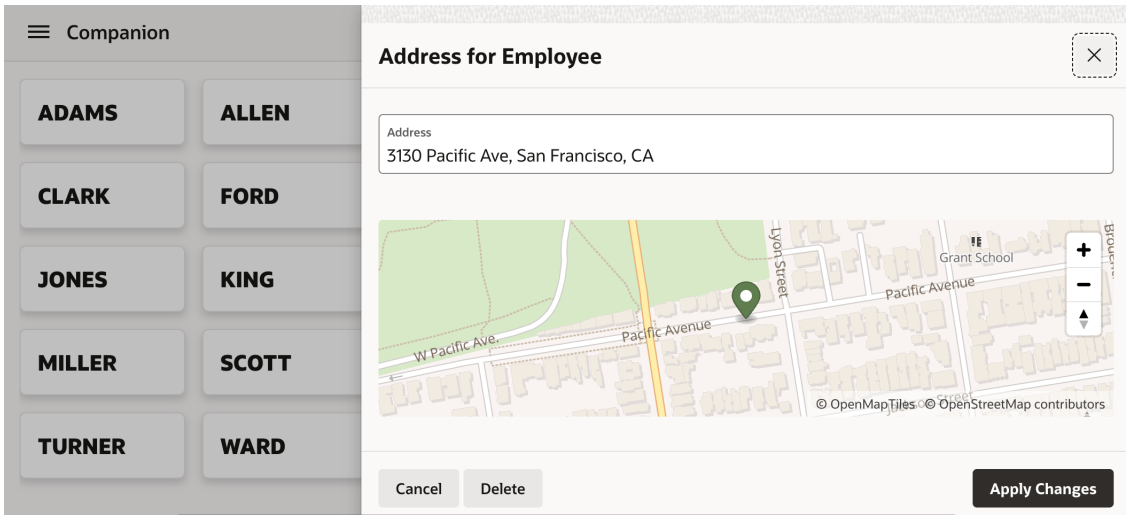
The Form region in the modal drawer page 60 includes the `P60_ENAME` page item, and the *Form Initialization* page process automatically retrieves the `EMP_AND_ADDRESS` view row based on the `P60_EMPNO` primary key value you pass in. So you already have the `ENAME` value on hand. Your instinctive first attempt shown below is to include `&P60_ENAME.` in the *Title*.

Figure 15-22 First Unsuccessful Attempt to Include Context Info in Dialog Title



However, when you run the page, the `&P60_ENAME.` comes up puzzlingly empty as shown below.

Figure 15-23 The &P60_ENAME. in Dialog Title Comes Up Empty



Copyright © 2026, Oracle and/or its affiliates.

In short, due to the timing of when the dialog title is determined at runtime, to include context information you need to set the dialog title dynamically in the *Page Load* event using code like the following:

```
// JavaScript in Page Load event action to set dialog title
apex.util.getTopApex()
         .jQuery(".ui-dialog-content")
         .dialog("option", "title",
                 "Address for Employee " + $v('P60_ENAME'));
```

> **Tip**
>
> To avoid having a static title display before your *Page Load* event sets the *dynamic* title, reference a non-existent substitution string like `&INITIALLY_BLANK.` in the page *Title* in Page Designer.

15.1.7.7 Providing App-Wide Helper Functions

Put JavaScript helper functions used on multiple pages in an application-wide library.

If you foresee setting a dynamic dialog title in many pages in your app, then it's a great candidate to include in an application-wide helper function library. For maintenance purposes, this idea beats the alternative of copying and pasting the same code over and over. In each usage, only the title text would be different. Creating an application-wide JavaScript function library involves two steps:

- Adding a *Static Application File* to contain the helper code
- Referencing the file in your application's *User Interface* JavaScript File URLs list.

- Adding Static Application JavaScript File
  Use a *Static Application File* shared component to include files of any kind with your app.
- Including Helper Functions in All Pages
  Once you add JavaScript file to Static Application Files, cite the script's reference URL in application settings to load it on all pages.

15.1.7.7.1 Adding Static Application JavaScript File

Use a *Static Application File* shared component to include files of any kind with your app.

From the *Shared Components* > *Static Application Files* page in the App Builder, add a file named `app.js` to include application-wide helper functions.

> **Tip**
>
> The suggested file name `app.js` can be any name you like, but keep the `.js` extension.

- Using JavaScript Namespace Template
  Use an `app.js` file to define a single global namespace for reusable JavaScript helper functions.

Copyright © 2026, Oracle and/or its affiliates.

- Comparing Namespace to PL/SQL Package
  Use an *immediately invoked function expression* (IIFE) to create an app namespace for reusable JavaScript helper functions.

15.1.7.7.1.1 Using JavaScript Namespace Template

Use an `app.js` file to define a single global namespace for reusable JavaScript helper functions.

Use the following `app.js` code as a template for a JavaScript helper function library. This one includes two "public" functions named `setDialogTitle()` and `areDifferent()`. This block of JavaScript code defines a reusable helper function "namespace" called `app`. You can think of it like a PL/SQL package:

- It has *public* functions (returned in the `return {…}` block).
- It may also have *private* helper functions defined inside the block but not returned.
- Only the returned functions are visible from outside.

This approach creates a single global symbol `app` instead of adding many separate function names into the global JavaScript scope. That keeps things clean and avoids naming conflicts.

```
// Export single global symbol "app" to keep code clean
const app = (function($) {

    // --- Private Functions ---

    // function myPrivateFunction(itemName) {
    //   ▒
    // }

    // --- Public Functions (returned below!) ---

    //------------------------------------------------------
    // Set the title of the current dialog window
    //------------------------------------------------------
    function setDialogTitle(title) {
        apex.util.getTopApex()
                 .jQuery(".ui-dialog-content")
                 .dialog("option", "title", title);
    }

    //------------------------------------------------------
    // Return true if x and y are different, treating null
    // and empty string as equivalent in the comparison
    //------------------------------------------------------
    function areDifferent(x, y) {
        return (x || '') !== (y || '');
    }

    // Only these functions are exposed to JavaScript code
    // in APEX pages; everything else stays private.
    return {
        setDialogTitle,
        areDifferent
    };
```


Copyright © 2026, Oracle and/or its affiliates.

```
    // Ensure $ in app namespace resolves to correct jQuery
})(apex.jQuery);
```

15.1.7.7.1.2 Comparing Namespace to PL/SQL Package

Use an *immediately invoked function expression* (IIFE) to create an app namespace for reusable JavaScript helper functions.

JavaScript doesn't have packages like PL/SQL, so developers use the pattern in `app.js` to simulate a modular structure. The high-level pattern looks like this:

```
// Define an "app" namespace with exported functions
const app = (function($) {

    // --- Private Functions ---
    function myPrivateFunction(itemName) {…}

    // --- Public Functions (returned below!) ---
    function setDialogTitle(title) {…}
    function areDifferent(x, y) {…}

    return {
        setDialogTitle,
        areDifferent
    };
})(apex.jQuery);
```

Known as an *immediately invoked function expression* (IIFE) the pattern breaks down as follows:

- The function `(function($) {…})(apex.jQuery);` is defined and then immediately called.
- Its single parameter named `$` is passed the correct version of *jQuery* APEX uses.
- It returns any functions to expose publicly as properties of an object.
- The `{setDialogTitle, areDifferent}` in the `return` statement is legal shorthand for:

  ```
  {
     setDialogTitle: setDialogTitle,
     areDifferent: areDifferent
  }
  ```

- That object becomes the value of the `app` variable.

So when you later write the following line of code in an APEX page, it calls the `setDialogTitle` function inside the `app` function library.

```
app.setDialogTitle("Hello");
```

Since `app` defines a unique name to unambiguously qualify its exported function names, JavaScript developers call it a *namespace*.

By attaching your helper functions to a single namespace object (`app`), you:

- Keep your global "footprint" of additional top-level names small.
- Group related logic in one place.

Copyright © 2026, Oracle and/or its affiliates.

- Make your code easier to organize and maintain.
- Define a public *specification* of exported functions.
- Maintain your ability to change private functions in the namespace *body*.

This is very similar to how you use packages in PL/SQL.

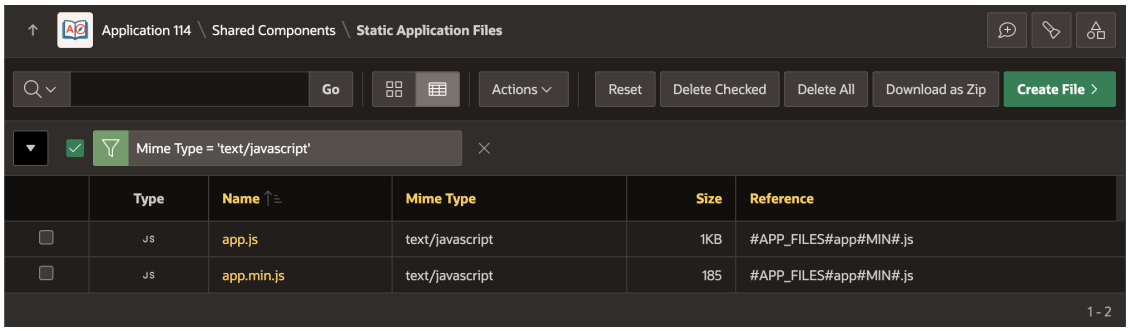
> **⚠ Caution**
>
> Unlike PL/SQL, JavaScript ***does not support*** defining multiple versions of the same function name with different parameter lists. If you inadvertently do this, the function defined last in the file "wins". To achieve a similar effect, define your function with all *possible* parameters, then test whether a particular parameter was ***not supplied*** by checking an expression like the following:
>
> ```
> if (typeof yourParam === "undefined") {
>   // yourParam was not passed
> }
> ```

15.1.7.7.2 Including Helper Functions in All Pages

Once you add JavaScript file to Static Application Files, cite the script's reference URL in application settings to load it on all pages.

In the *Shared Components* > *Static Application Files* list page, as shown below, you can find the *Reference* URL for any file. When useful, since the list is an Interactive Report, use its filtering features to show only JavaScript files. Notice the *Reference* column value for JavaScript files includes both the `#APP_FILES#` and `#MIN#` substitutions. Copy the *Reference* value for the `app.js` file to the clipboard to use in the next step.

Figure 15-24 List Page Shows Reference Path for a Static Application Files

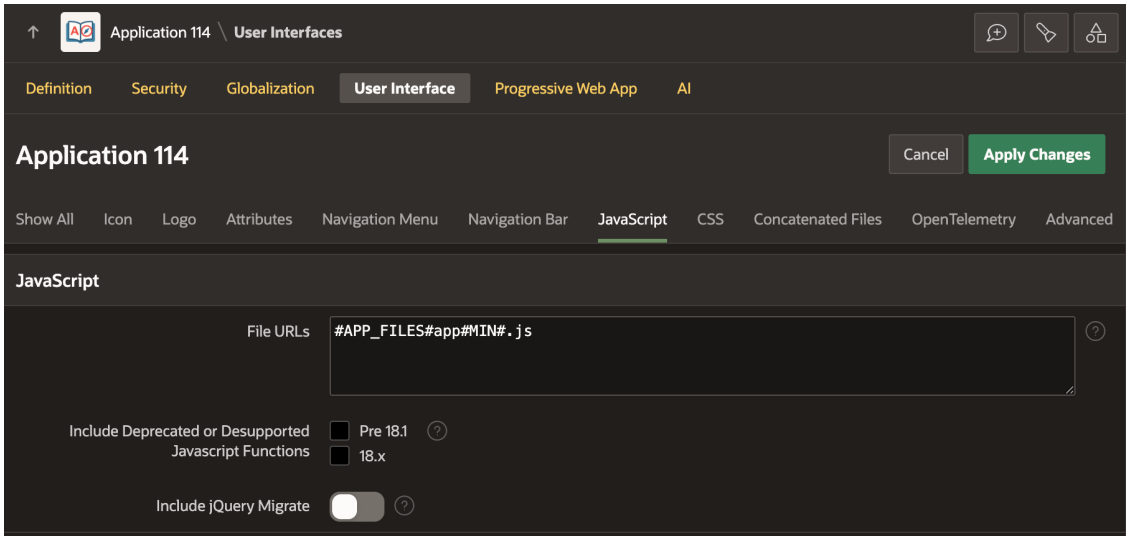


Copyright © 2026, Oracle and/or its affiliates.

> **Tip**
>
> When running with debug tracing, the original JavaScript source file is used instead.When you save a JavaScript file like `app.js`, App Builder automatically creates a minified version named `app.min.js`. Minification removes whitespace and shortens names without changing the code's logic, making the file smaller. APEX uses the minified file at runtime to improve page load performance, except when Debug tracing is enabled. With Debug tracing, APEX uses the original JavaScript source file.

To include a static application JavaScript file in every page, start by copying its *Reference* path from the page above. Then, as shown below, paste that value into the *File URLs* list field on the *JavaScript* tab of the *Shared Component* > *User Interface Attributes* page. If you include multiple files, list one per line.

Figure 15-25 Including Static App File Reference in Application JavaScript File URLs



15.1.7.8 Using Translatable Text in JavaScript

Consider using a translatable Text Message when setting a dynamic dialog title from JavaScript.

Once your `app` helper function "namespace" is available application-wide, your *Employee Address* modal drawer page's *Page Load* dynamic action can now set its dynamic title using the one-liner below:

```
app.setDialogTitle('Address for Employee ' + $v('P60_ENAME'));
```

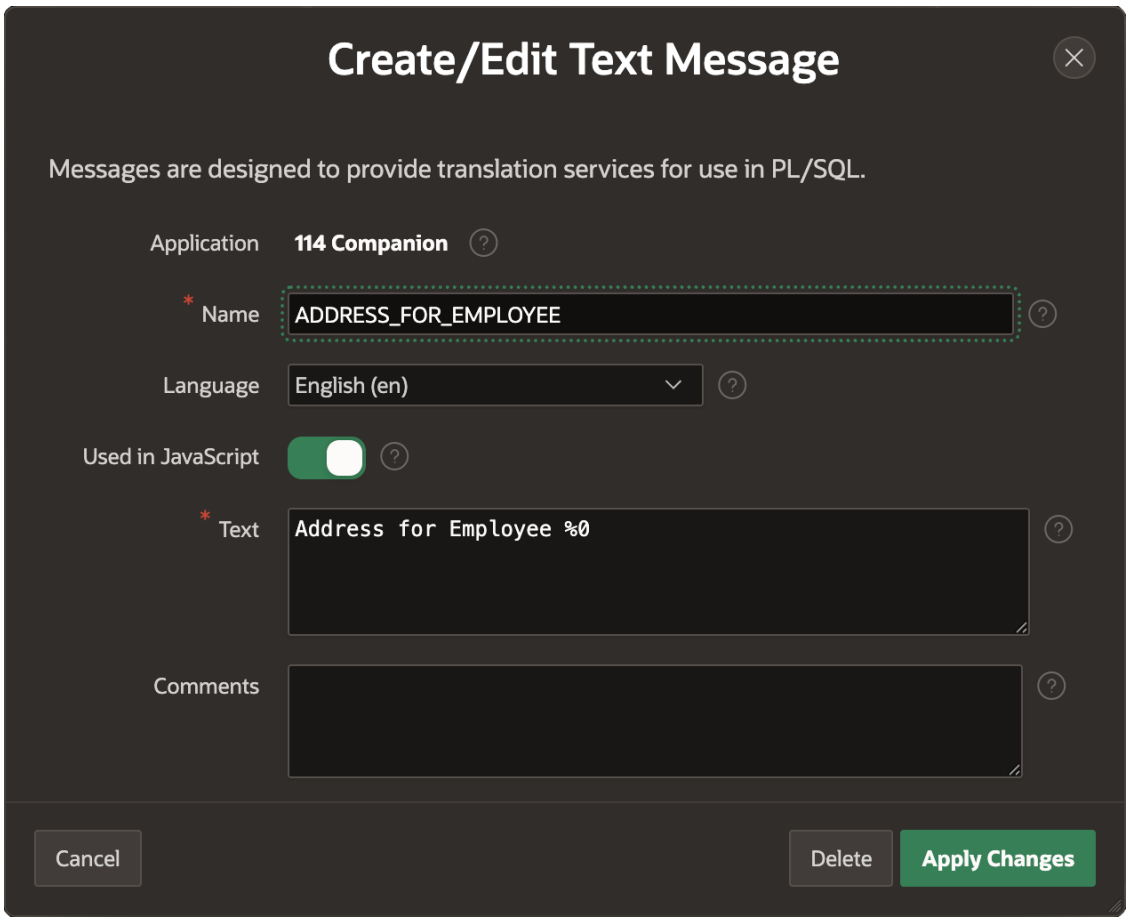
However, if your application may ever need to be translated, rather than hard-coding the dialog title text, use a shared component *Text Message* instead. For the dialog title in this example, navigate to **Shared Components** > **Text Messages** to create a translatable text message as shown below with name `ADDRESS_FOR_EMPLOYEE`. Notice its corresponding *Text* for the *English (en)* language is the following. It includes the numbered parameter `%0` as a placeholder.

```
Address for Employee %0
```


Copyright © 2026, Oracle and/or its affiliates.

Importantly, you enable the *Used in JavaScript* switch since your call to set the dynamic dialog title happens in JavaScript. The figure shows the **Create/Edit Text Message** dialog for the `ADDRESS_FOR_EMPLOYEE` text message with this switch enabled.

Figure 15-26 Configuring a Translatable Text Message for Use in JavaScript



After defining this text message, you can reference it in your *Page Load* dynamic action's call to set the dynamic dialog title as follows:

```
app.setDialogTitle(
    apex.lang.formatMessage(
        'ADDRESS_FOR_EMPLOYEE',
        apex.items.P60_ENAME.value));
```

15.2 Geocoding in the Background

When your application runs on Autonomous Database, you can geocode addresses in the background in:

- *Workflows* – using a native activity,
- *Execution chain processes* – using a native page process, or

Copyright © 2026, Oracle and/or its affiliates.

- *Automations* – using a native *Server-side Geocoding* action type.

You configure the address to match as a:

- single, unstructured address field, or
- set of structured address fields, each providing one part of the address.

After configuring the one or more address items, specify either or both of these options:

- *Coordinates Item* to receive a GeoJSON point with coordinates of the *best* match, or
- *Collection Name* to process the possibly multiple matching addresses.

> **Tip**
>
> When using the unstructured address, remember to include the ISO 2-letter abbreviation of the country at the end of the address value to geocode.

- Selecting Items for Server-side Geocoding
- Referencing Coordinates Item Values
- Processing Collection of Matches
- Configuring Match Mode
  When doing server-side geocoding with a structure address, your choice of match mode influences how strict to be.

15.2.1 Selecting Items for Server-side Geocoding

When doing server-side geocoding, select item types appropriate to the context. For example, with the:

- *native workflow activity* – use *Additional Data* columns, parameters, or workflow variables,
- *native page process* – use page items or application items, and
- *native automation action type* – use query columns or application items.

> **Tip**
>
> Automation actions use bind variables to reference column values of the current query row they are processing. You can *assign* values to the bind variables, too. These column items can not only supply address values to geocode but also be the *Coordinates Item* to receive the result. Subsequent actions can "see" updated row values, but they are not permanently *stored* anywhere.

15.2.2 Referencing Coordinates Item Values

If your *Coordinates Item* is named `COORDINATES`, then you can access the longitude and latitude values in the following ways.

Copyright © 2026, Oracle and/or its affiliates.

Using an SDO_GEOMETRY type, access them like this in PL/SQL:

```
declare
   l_coords    sdo_geometry := sdo_util.from_geojson(:COORDINATES);
   l_longitude number       := l_coords.sdo_point.x;
   l_latitude  number       := l_coords.sdo_point.y;
begin
   -- Do something with l_longitude & l_latitude here
end;
```

To access them using SQL instead, use:

```
for j in (select coords.sdo_point.x as longitude,
                 coords.sdo_point.y as latitude
            from (
               select sdo_util.from_geojson(:COORDINATES) as coords
                 from dual))
loop
   -- Do something with j.longitude & j.latitude here
end loop;
```

If you prefer to work with the GeoJSON document, use the following in PL/SQL:

```
declare
   l_longitude number;
   l_latitude  number;
begin
   l_longitude := json_value(:COORDINATES,'$.coordinates[0]');
   l_latitude  := json_value(:COORDINATES,'$.coordinates[1]');
   -- Do something with l_longitude & l_latitude here
end;
```

Alternatively, to access them via the GeoJSON document in SQL, use:

```
for j in (select longitude, latitude
            from json_table(:COORDINATES,
                    '$.coordinates'
                    columns (
                       longitude number path '$[0]',
                       latitude  number path '$[1]')))
loop
   -- Do something with j.longitude & j.latitude here
end loop;
```

15.2.3 Processing Collection of Matches

If your *Collection Name* is GEOCODE_RESULTS then you can access the address matches using the following query:

```
select seq_id,
       c001 as street,
       c002 as house_number,
```


Copyright © 2026, Oracle and/or its affiliates.

```
       c003 as postalcode,
       c004 as municipality,
       c005 as settlement,
       c006 as region,
       c007 as country,
       c011 as matchvector,
       n001 as longitude,
       n002 as latitude,
       d001 as geocoding_timestamp
from apex_collections
where collection_name = 'GEOCODE_RESULTS'
order by seq_id
```

Tip

The results are added to the collection in the order "best match first" order the geocoding service provides them, so ordering by the collection's `SEQ_ID` column retains that order for your processing.

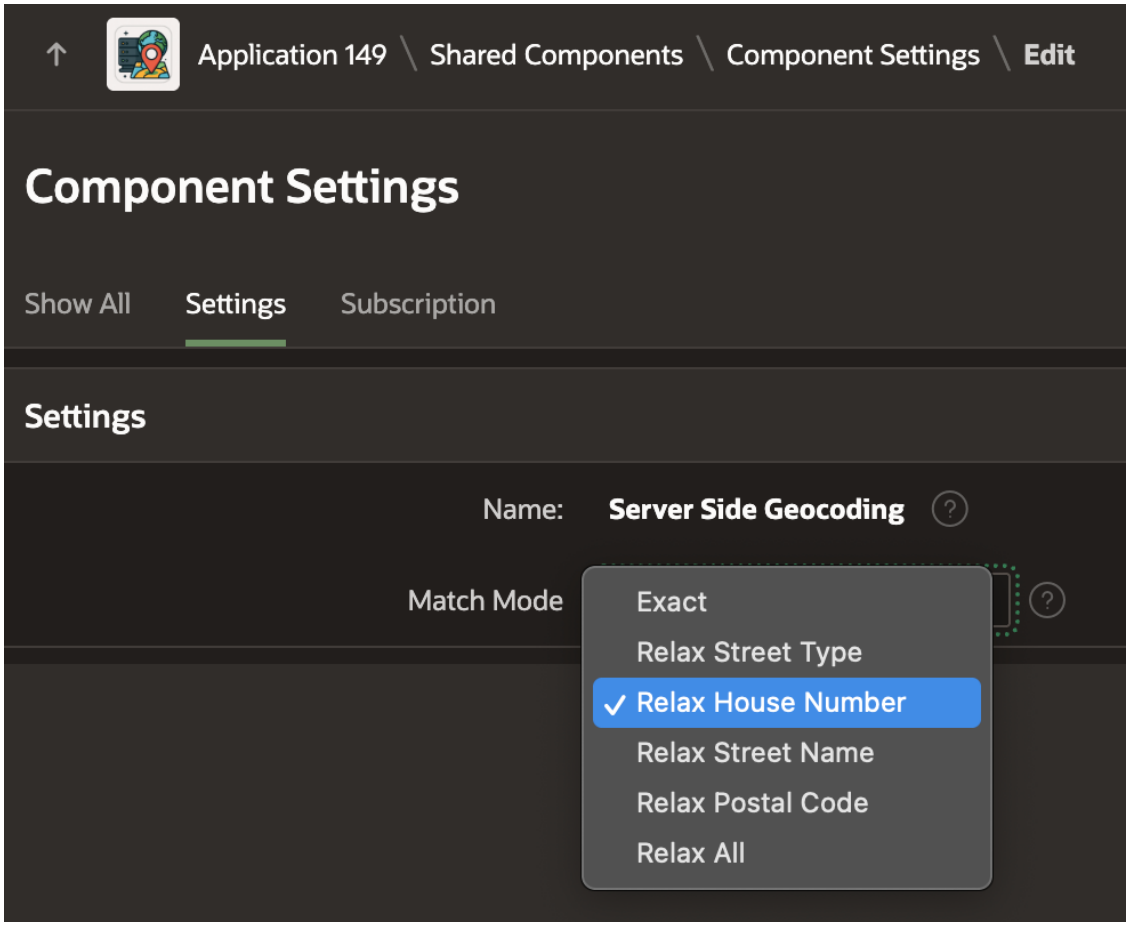
15.2.4 Configuring Match Mode

When doing server-side geocoding with a structure address, your choice of match mode influences how strict to be.

As shown below, you configure it in the *Shared Components* > *Component Settings* page for the *Server Side Geocoding* component. This setting has no effect when geocoding an unstructured address.

Copyright © 2026, Oracle and/or its affiliates.

Figure 15-27 Configuring Match Mode for Server-side Geocoding



Copyright © 2026, Oracle and/or its affiliates.

16

Uploading, Viewing, and Downloading Files

Use *Image Upload* for images, *File Upload* for other file types, and native download features to serve stored files.

Use an *Image Upload* page item to let end users submit one or more images to store in a BLOB column or an Oracle Cloud Infrastructure (OCI) object bucket. This component provides image-specific options like cropping and client-side scaling to avoid multi-megapixel images from modern smartphones if the task doesn't *require* that resolution

To handle other file types – for example PDF – use the more general *File Upload* page item. It offers similar single and multiple file abilities for any kind of file.

Your pages can include a native *Download* dynamic action or page process to declaratively download images or other files, and a page can serve inline images or files using the `DOWNLOAD` procedure in the `APEX_HTTP` package.

- Reviewing Break Room and Referrals Pages
  Explore two Woods HR use cases: sharing employee photos and submitting candidate CVs for open position referrals.
- Storing Employee ID in an Application Item
  Populate an application item at login to reuse a session-specific value throughout the app.
- Uploading an Image to a BLOB Column
  Use an *Image Upload* page item to let users provide a photo or image file to your app.
- Displaying BLOB Column Images
  Display images from a URL or BLOB column in common APEX regions, with a special approach for Interactive Grid.
- Uploading a File to a BLOB Column
  Use a *File Upload* page item to let users provide a file to your app. When you use it as part of a *Form* region, the combination automates creating, editing, and deleting the row with the file BLOB column.
- Previewing BLOB Column PDF Files Inline
  To preview a PDF file stored in a BLOB column, use an `iframe`.
- Downloading a File for Offline Use
  Use the native *Download* page process or dynamic action to let the user download a file for offline use.
- Uploading Multiple Images or Files
  Your apps can use native temporary storage of uploaded files to support any business requirement.
- Using an OCI Object Bucket for Files
  Use an OCI object bucket to store application files outside your transactional database.

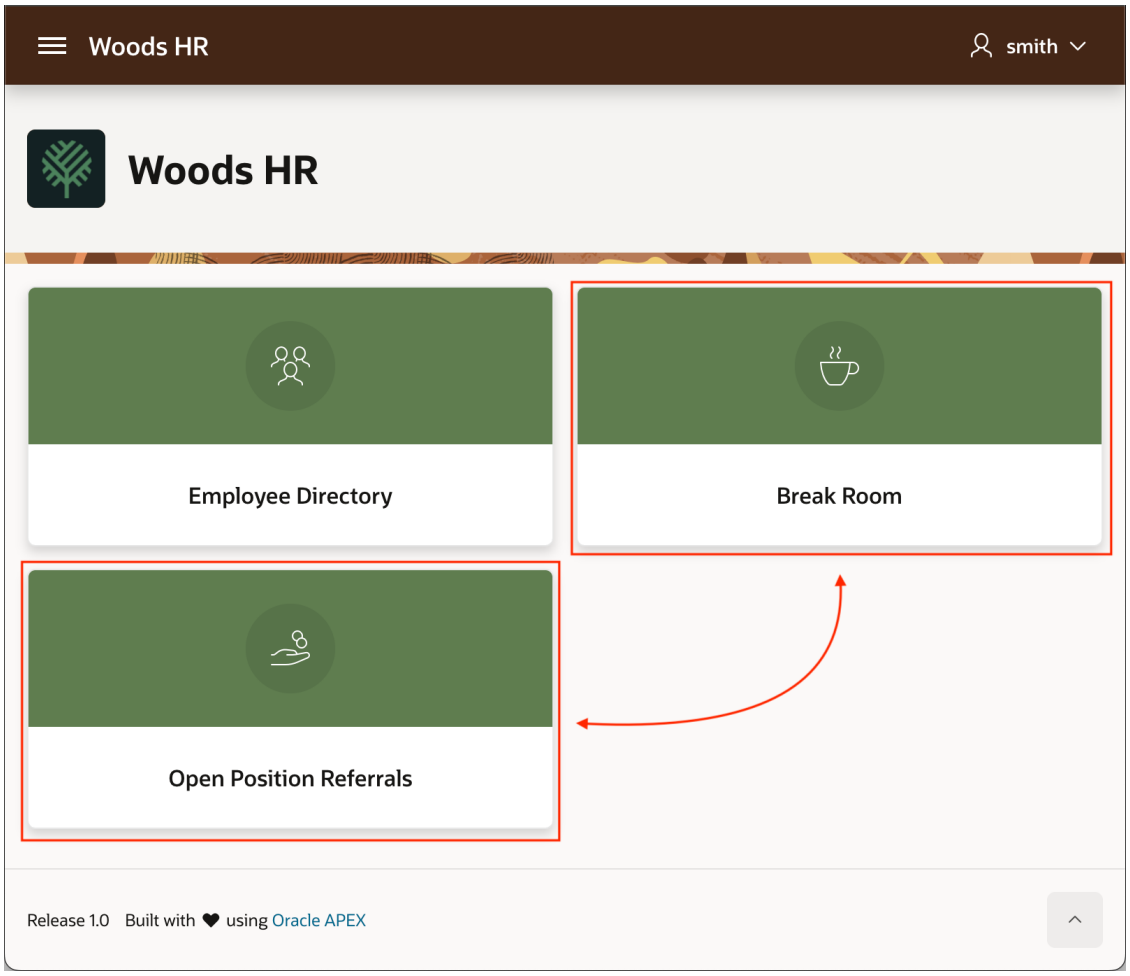
16.1 Reviewing Break Room and Referrals Pages

Explore two Woods HR use cases: sharing employee photos and submitting candidate CVs for open position referrals.

Copyright © 2026, Oracle and/or its affiliates.

Your Woods HR app lets employees share photos with colleagues on the *Break Room* page. It also rewards staff who submit a candidate's PDF curriculum vitae (CV) on the *Open Position Referrals* page if their referral gets hired. The figure shows the home page of the Woods HR application where employees access an employee directory, a virtual break room, and a candidate referral function.

Figure 16-1 Break Room and Open Positions Referrals Pages in Woods HR App



- Sharing Photos in the Break Room
  Review the Break Room page, where employees browse shared photos and add one or more images.
- Submitting a Referral to Earn a Bonus
  Preview the Open Position Referrals page, where employees review, submit, and download candidate CVs for open jobs..
- Studying Supporting Tables and Views
  Learn how the `EBA_DEMO` tables and views support Woods HR photo sharing and job referrals.

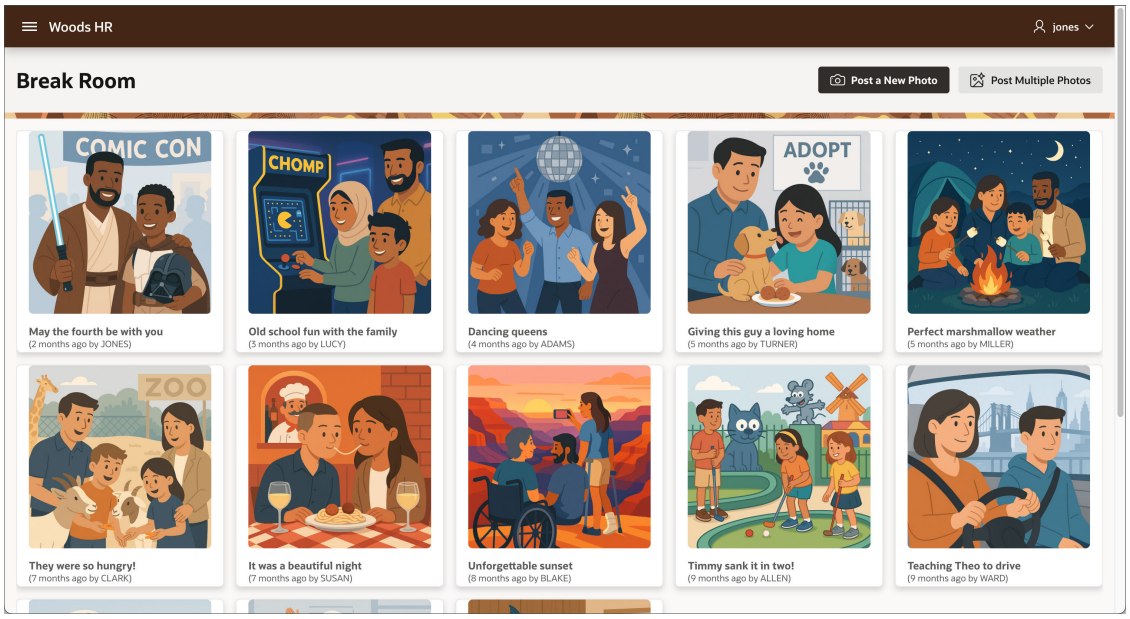
16.1.1 Sharing Photos in the Break Room

Review the Break Room page, where employees browse shared photos and add one or more images.

Copyright © 2026, Oracle and/or its affiliates.

On the *Break Room* page, users browse photos shared by others, and buttons let them post a single new photo, or multiple new photos. The images appear in a *Cards* region using its *Media* settings. By applying an *All Employees* authorization scheme, any employee can access this page. The figure shows the virtual break room with recently posted photos from employees across the company.

Figure 16-2 Break Room Page Lets Employees Share Photos with Colleagues



16.1.2 Submitting a Referral to Earn a Bonus

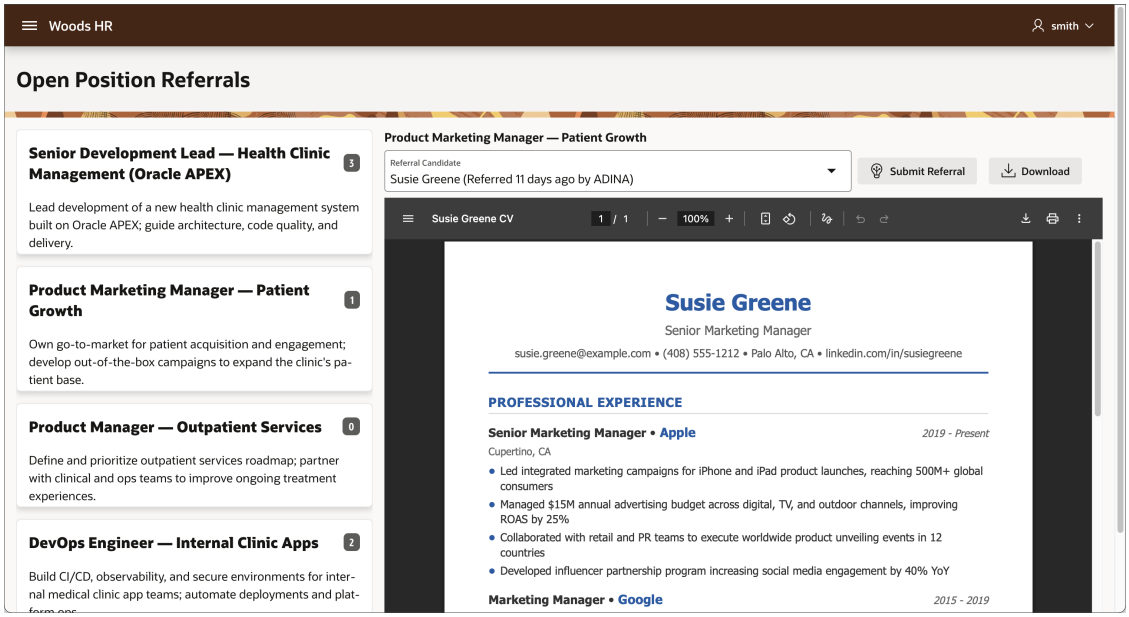
Preview the Open Position Referrals page, where employees review, submit, and download candidate CVs for open jobs..

On the *Open Position Referrals* page, users select an open position to review submitted candidate CVs for that job. Buttons let them submit the CV of a *new* referral or download one of the existing CVs to review offline. By applying an *All Employees* authorization scheme, any employee can access this page.

The figure shows the page employees use to refer candidates for open positions. Tiles representing open positions appear on the left in a Cards region, a select list of existing candidates lets users review existing CVs inline, and a **Submit Referral** button lets an employee provide a new candidate's CV for a position.

Copyright © 2026, Oracle and/or its affiliates.

Figure 16-3 Employees Submit Candidate CVs for Open Job Positions



16.1.3 Studying Supporting Tables and Views

Learn how the `EBA_DEMO` tables and views support Woods HR photo sharing and job referrals.

The tables and views supporting the Woods HR app all use the `EBA_DEMO` prefix. The `DEPT` and `EMP` tables shown below mimic the EMP/DEPT sample dataset. Three related tables support the break room and job referrals pages:

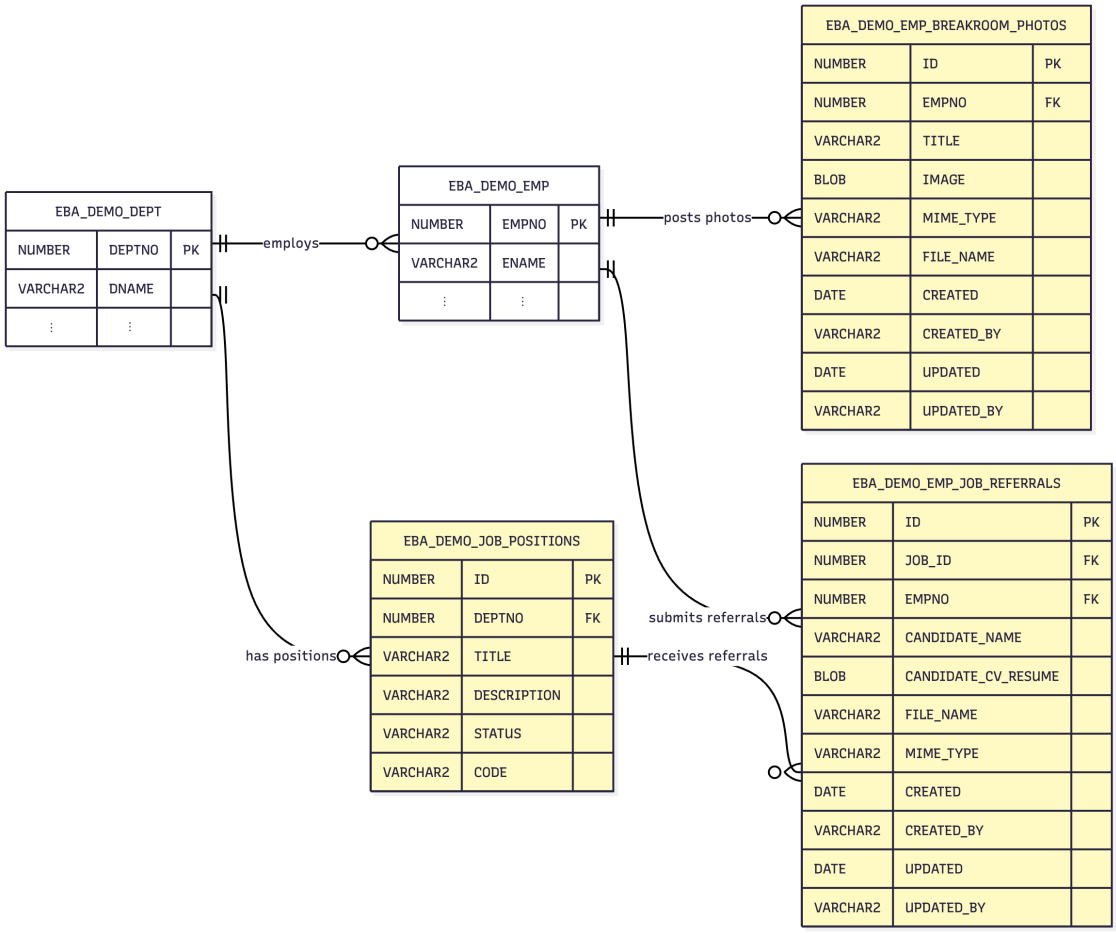
- `EMP_BREAKROOM_PHOTOS` – stores uploaded pictures employees post.
- `JOB_POSITIONS` – tracks open job positions
- `EMP_JOB_REFERRALS` – contains employee-referred job *candidates* and respective CVs.

While all employees use *Break Room* and *Open Job Referrals*, only HR reps manage job positions.

The figure shows the structure of and relationships between the five tables involved.

Copyright © 2026, Oracle and/or its affiliates.

Figure 16-4 Additional Tables to Support Woods HR Photos and CVs



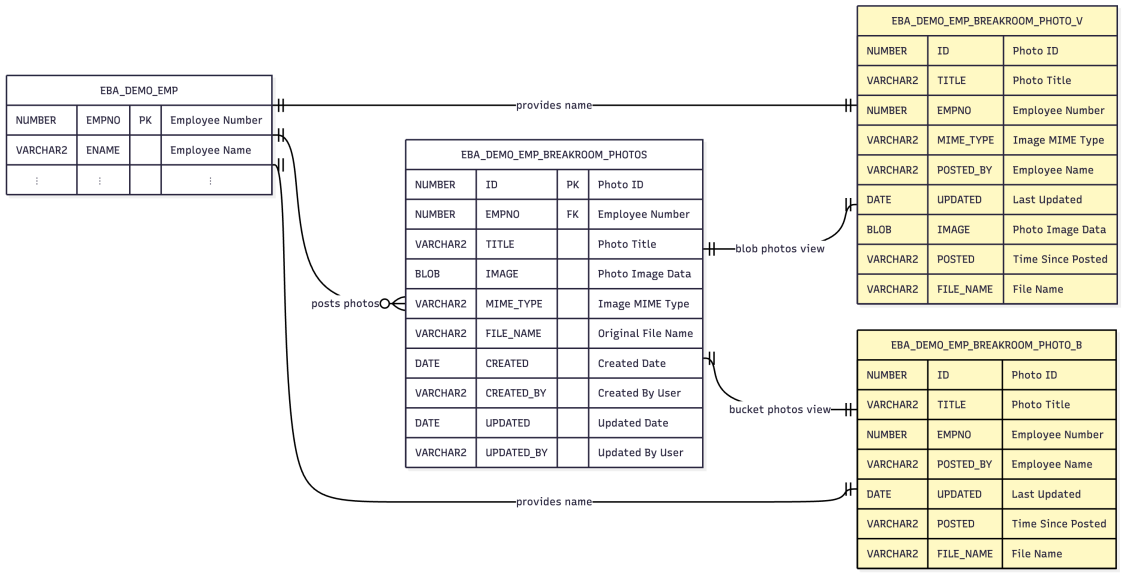
Since you explore two approaches for storing photos in this section, the views shown below provide two different "interfaces" on the photo storage table. Both join the `EMP` table to use its `ENAME` columns as the `POSTED_BY` employee. Both also use the `APEX_UTIL.GET_SINCE` function to format the underlying `CREATED` date in a more useful way for UI display like "3 days ago". The views differ in the following way:

- `EMP_BREAKROOM_PHOTO_V` – Has `IMAGE` BLOB and `MIME_TYPE` for local photo storage
- `EMP_BREAKROOM_PHOTO_B` – Has only `FILE_NAME` for OCI object bucket image.

The figure depicts the structure of these two breakroom photo views and how they related to the two underlying tables `EBA_DEMO_EMP` and `EBA_DEMO_EMP_BREAKROOM_PHOTOS`.

Copyright © 2026, Oracle and/or its affiliates.

Figure 16-5 Views Present Two "Interfaces" on Breakroom Photos Table



The `EMP_BREAKROOM_PHOTO_V` view includes the `WHERE` clause below to only return rows from the underlying table that have a valid `IMAGE` BLOB stored.

```
-- Used for photos stored locally in a BLOB
create view eba_demo_emp_breakroom_photo_v as
  select p.id,
         p.title,
         p.empno,
         p.mime_type,
         e.ename                        as posted_by,
         p.updated,
         p.image,
         apex_util.get_since(created) as posted,
         file_name
from eba_demo_emp_breakroom_photos p
join eba_demo_emp e on ( p.empno = e.empno )
where      image is not null
       and dbms_lob.getlength(image) > 0
```

In contrast, the `EMP_BREAKROOM_PHOTO_B` view's `WHERE` clause returns only rows with *no* `IMAGE` BLOB stored. These rows represent images stored in an Oracle Cloud Infrastructure (OCI) object bucket.

```
-- Used for photos stored in an OCI object bucket
create view eba_demo_emp_breakroom_photo_b as
  select p.id,
         p.title,
         p.empno,
         e.ename                        as posted_by,
         p.updated,
         apex_util.get_since(created) as posted,
         file_name
    from eba_demo_emp_breakroom_photos p
```


Copyright © 2026, Oracle and/or its affiliates.

```
join eba_demo_emp e on ( p.empno = e.empno )
where    image is null
      or dbms_lob.getlength(image) = 0
```

16.2 Storing Employee ID in an Application Item

Populate an application item at login to reuse a session-specific value throughout the app.

Break Room photos and Open Position Referrals both record the logged-in user's `EMPNO`. Populate an application item at login to avoid repeated lookups during the session.

- Defining an Application Item
  Define an *Application Item* to hold a session-scoped value, accessible from any page.
- Computing an Application Item at Login
  Use an *Application Computation* with *After Authentication* execution point to set an application item's value after the user logs in.
- Referencing the App Item Value
  Reference an application item from any page or part of your app during the user session.

16.2.1 Defining an Application Item

Define an *Application Item* to hold a session-scoped value, accessible from any page.

Since it acts like a global variable for the current user, consider using a prefix like `G_` to distinguish it from native substitution strings like `APP_USER`. For example, use the *Shared Components* > *Application Items* page shown below to create one named `G_APP_USER_EMPNO` to hold the numeric employee ID of the logged-in app user.

> **Tip**
>
> While an application item may contain a value that *looks* like a number (e.g. "1234"), it is stored as a `VARCHAR2` value. Use the built-in `NV('YOUR_APP_ITEM_NAME')` to get the item value as a `NUMBER` in PL/SQL or `APEX_SESSION_STATE.GET_TIMESTAMP('YOUR_APP_ITEM_NAME')` to get the item as a date value.

Compute its value declaratively using an *Application Computation*. You can also assign a value from an *Application Process* using its name as a bind variable or calling `SET_VALUE` in the `APEX_SESSION_STATE` package as shown below.

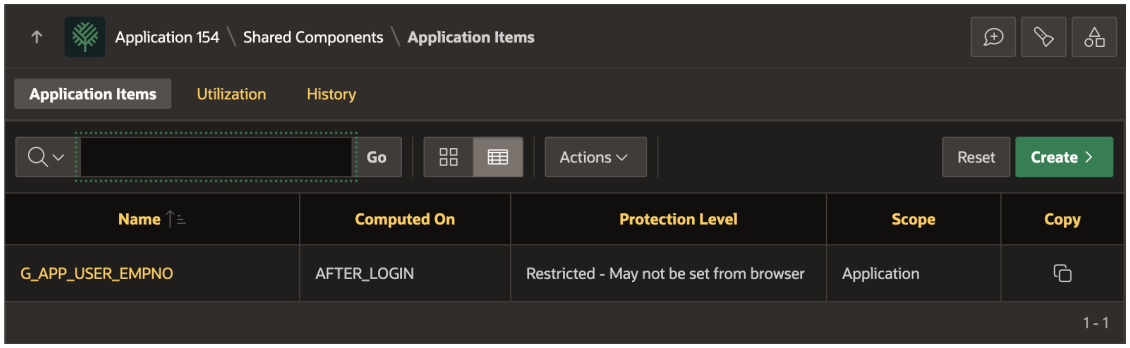
Both the application computation and application process have a *Processing Point*. Set it to *After Authentication* to run immediately after login. You can also assign a new value to an application item from anywhere in your application using the same two approaches shown below.

```
-- Assigning an app item as a bind variable
:G_APP_USER_EMPNO := 1234;
-- Setting an app item value with a procedure
apex_session_state.set_value('G_APP_USER_EMPNO', 1234);
```

The figure shows the Application Items list page in **Shared Components** after defining the `G_APP_USER_EMPNO` item.

Copyright © 2026, Oracle and/or its affiliates.

Figure 16-6 Shared Component Application Items List



16.2.2 Computing an Application Item at Login

Use an *Application Computation* with *After Authentication* execution point to set an application item's value after the user logs in.

The one shown below computes the value of `G_APP_USER_EMPNO` using the SQL statement below that returns a single value.

```
select empno
  from eba_demo_emp
 where ename = :APP_USER
```

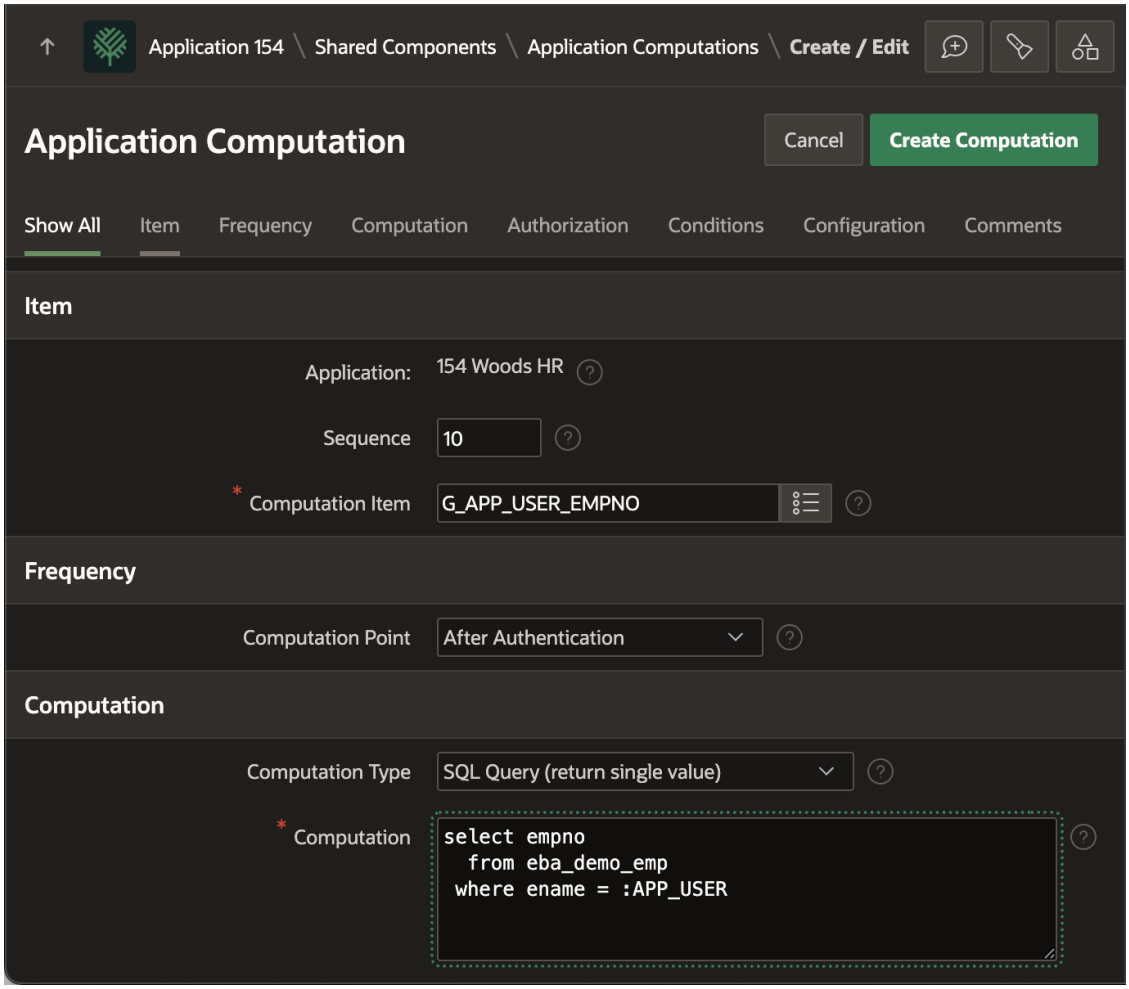
You can also use a PL/SQL expression, function body returning a result, or other methods.

> **Tip**
>
> You can also assign the application item value in an *Application Process* with an *After Authentication* processing point.

The figure shows the **Application Computation** edit page in App Builder where the value of the `G_APP_USER_EMPNO` item is computed after user authentication using a one-column `SELECT` statement that returns one row.

Copyright © 2026, Oracle and/or its affiliates.

Figure 16-7 Computing an Application Item's Value After Login



16.2.3 Referencing the App Item Value

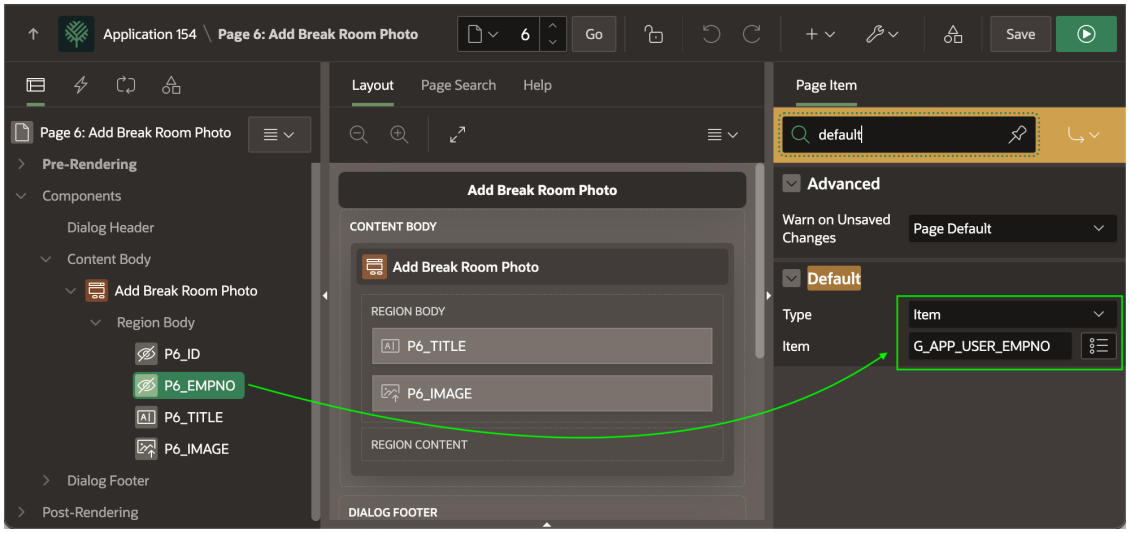
Reference an application item from any page or part of your app during the user session.

Reference the value of an application item like `G_APP_USER_EMPNO` anywhere you need it. Depending on the context, use `:G_APP_USER_EMPNO`, `&G_APP_USER_EMPNO.`, or just the item name if declaratively specifying the name of an item whose value to use.

For example, the *Add Break Room Photo* page shown below configures the default value of its hidden `P6_EMPNO` page item to come from the `G_APP_USER_EMPNO` application item. This ensures the photo is correctly attributed to the employee who submits it using this page.

Copyright © 2026, Oracle and/or its affiliates.

Figure 16-8 Using an Application Item to Default the Posting Employee ID



16.3 Uploading an Image to a BLOB Column

Use an *Image Upload* page item to let users provide a photo or image file to your app.

When you use it as part of a *Form* region, the combination automates creating, editing, and deleting the row with the image BLOB column. Configure the *Form* region source with the table or view name, then set the *Image Upload* page item's source to the BLOB column. Other declarative settings control the features end users see while uploading the image.

- Configuring Upload Image Item Source
  After creating a form region on your table containing the BLOB image column, set the item type of the BLOB column to *Image Upload*.
- Controlling Image Upload User Experience
  Use the *Image Upload* page item's properties in the *Display*, *Cropping*, and *Resize To* sections to control your end users' experience when uploading an image.
- Enlarging Image Upload Icon Preview
  Use CSS to enlarge the Image Upload icon preview if needed.
- Configuring Application-Wide Style Rules
  Use a page's *CSS* > *Inline* settings to define style rules that target a *specific* page item or ones that should affect only that page. In contrast, put rules that affect *all* pages in a *Shared Components* > *Static Application Files* CSS style sheet.
- Posting a Break Room Photo
  Walk through a photo upload flow, from image selection to the refreshed Cards region on the calling page.

16.3.1 Configuring Upload Image Item Source

After creating a form region on your table containing the BLOB image column, set the item type of the BLOB column to *Image Upload*.

Set the *Storage* > *Type* to:

- *BLOB column specified in Item Source attribute*.

Copyright © 2026, Oracle and/or its affiliates.

Check the *Source* section in the *Property Editor* to confirm the expected BLOB column is selected. Finally, as shown below, configure the column names for the:

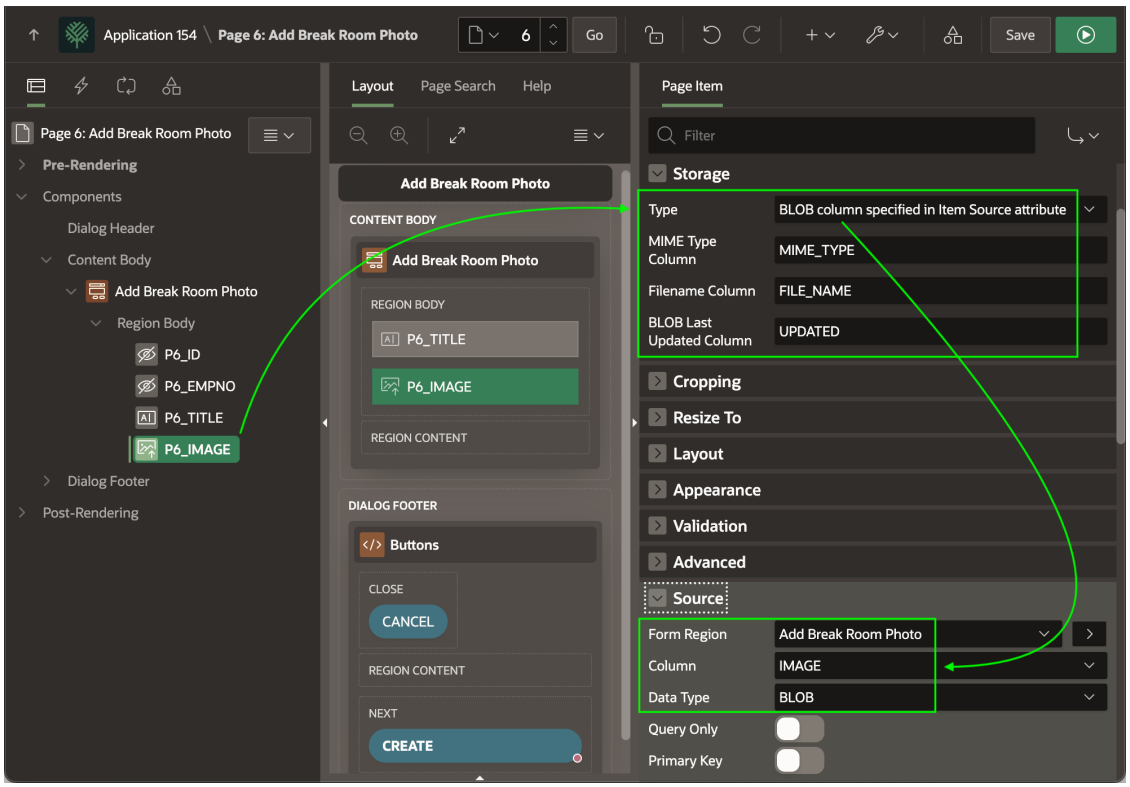
- *MIME Type Column* – the type of image (e.g. `image/jpeg`, `image/png`)
- *Filename Column* – the uploaded file name
- *BLOB Last Updated Column* – date the image was last modified.

Setting up these additional column names is important. It ensures the browser handles the image correctly when viewed or downloaded.

> **Tip**
>
> Enable the *Primary Key* switch for form items tied to primary key columns. (e.g. `P6_ID` here)

The figure shows the P6_IMAGE page item selected in the Page Designer component tree. It highlights the **Storage** and **Source** group properties related to automatic BLOB file handling.

Figure 16-9 Configuring Source Properties for an Upload Image Item



Then, in the *Processing* tab of the Page Designer tree, ensure you have:

- a page process of type *Form - Automatic Row Processing (DML)*
  - with its *Form Region* property set to the form region,
  - sequenced *before* the *Close Dialog* process if the page is a modal dialog or drawer, and

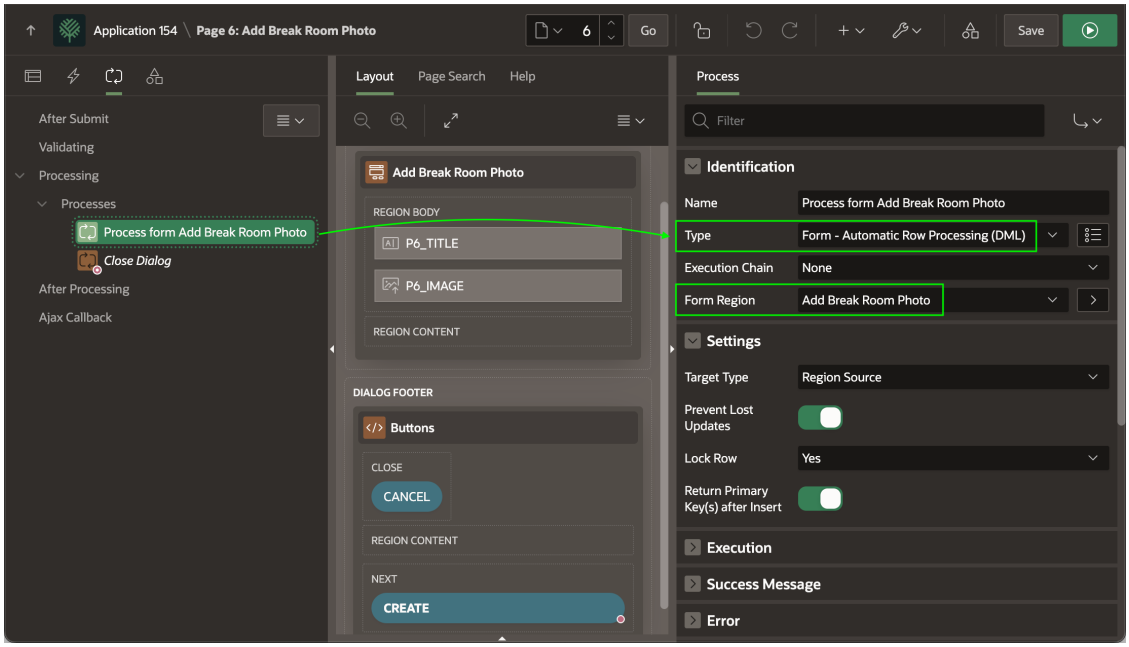
Copyright © 2026, Oracle and/or its affiliates.

- a button like the CREATE one here with its *Database Action* property set to *SQL INSERT action*.

> **Tip**
>
> The Create Page Wizard creates the DML page process on a new *Form* page, and sets the *Database Action* property to *SQL INSERT action* on the CREATE button. However, when you add a Form region to an existing page you need to do these two steps yourself.

The figure shows the **Process form Add Break Room Photo** page process selected in the component tree. It highlights the process type is *Form - Automatic Row Processing (DML)* and the associated Form region is *Add Break Room Photo*.

Figure 16-10 Configuring Form DML Page Process



16.3.2 Controlling Image Upload User Experience

Use the *Image Upload* page item's properties in the *Display*, *Cropping*, and *Resize To* sections to control your end users' experience when uploading an image.

Allow Cropping lets the end user adjust the selected portion of the image to upload. When enabled, an additional *Aspect Ratio* list appears to impose a standard format if appropriate.

Use the *Maximum File Size*, *Maximum Width*, and *Maximum Height* to constrain the size of the image. If needed, the page item resizes the file on the client before uploading. This helps avoid storing multi-megapixel images from modern smartphones when that resolution adds no value.

To let the user take a photo, set *Capture Using* to open a device's main or selfie camera. Users can still select an *existing* image file instead.

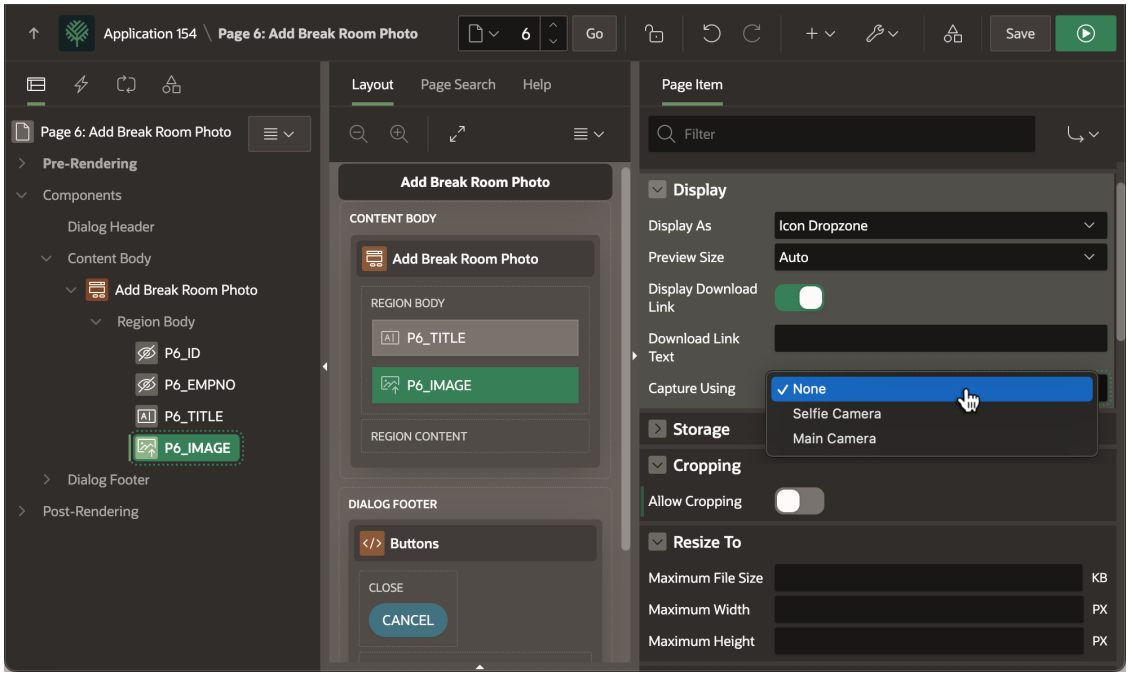
Copyright © 2026, Oracle and/or its affiliates.

The *Display As* and *Preview Size* control how the user sees the image upload item on the page. By default, it shows as an *Icon Dropzone* with a preview of the selected image.

The *Display Download Link* controls whether the user can download an existing image when editing a row. When enabled, you can also set the *Download Link Text*.

The figure shows the P6_IMAGE Image Upload page item selected in the component tree and values of its properties in the **Display**, **Cropping**, and **Resize To** sections in the Property Editor. The mouse is showing the different values of the **Capture Using** property: **Selfie Camera**, **Main Camera**, or **None**.

Figure 16-11 Configuring Properties That Control Image Upload Behavior



16.3.3 Enlarging Image Upload Icon Preview

Use CSS to enlarge the Image Upload icon preview if needed.

If none of the *Preview Size* options meets your needs, set the file drop icon size using the `--a-filedrop-icon-size` CSS variable. You can specify it on the item using the item name in the rule like this:

```
#P6_IMAGE {
  --a-filedrop-icon-size: 25rem;
}
```

Alternatively, specify it on the root of the page using the `:root` pseudo-class selector like this. CSS defines this *pseudo-class* to match the page's root element.

```
:root {
  --a-filedrop-icon-size: 25rem;
}
```


Copyright © 2026, Oracle and/or its affiliates.

This rule makes the preview icon size the width of 25 characters in the size of page's base font.

> **Tip**
>
> CSS `rem` units are tied to the page's base font size. If the user zooms in or out in their browser, units specified in `rem` scale automatically with the new font size.

16.3.4 Configuring Application-Wide Style Rules

Use a page's *CSS* > *Inline* settings to define style rules that target a *specific* page item or ones that should affect only that page. In contrast, put rules that affect *all* pages in a *Shared Components* > *Static Application Files* CSS style sheet.

Rules that target the `:root` selector are good candidates if they should affect all pages. CSS defines this *pseudo-class* to match any page's root element.

For example, as shown below, the following static application `app.css` file defines one rule using the `:root` selector to specify values for three CSS variables:

- `--ut-body-content-padding-y` – vertical padding for Universal Theme body content
- `--a-cv-item-width` – width of each card in a Cards region
- `--a-cv-body-padding-y` – vertical padding for Cards region card body section

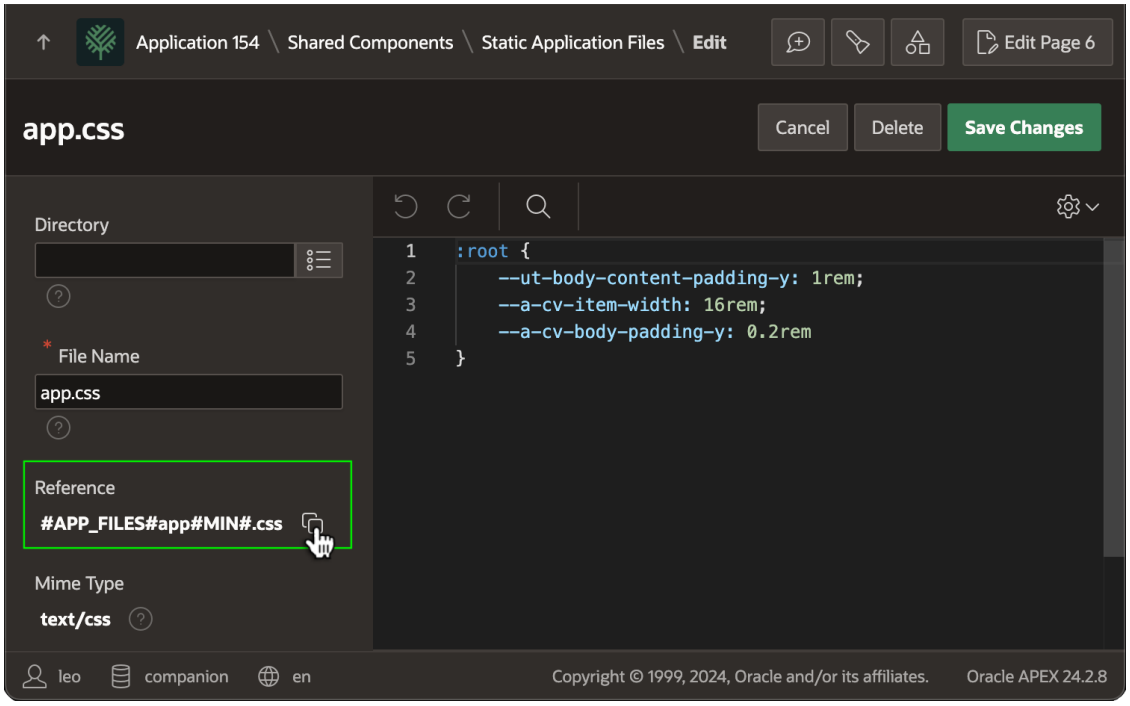
> **Tip**
>
> See CSS Variables in the Universal Theme reference app to learn about others.

```
:root {
    --ut-body-content-padding-y: 1rem;
    --a-cv-item-width: 16rem;
    --a-cv-body-padding-y: 0.2rem
}
```

The figure below shows the edit page for the `app.css` static application file. Notice its *Reference* field. It's the URL for this file. The `#APP_FILES#` represents base URL for your app's static files.

Copyright © 2026, Oracle and/or its affiliates.

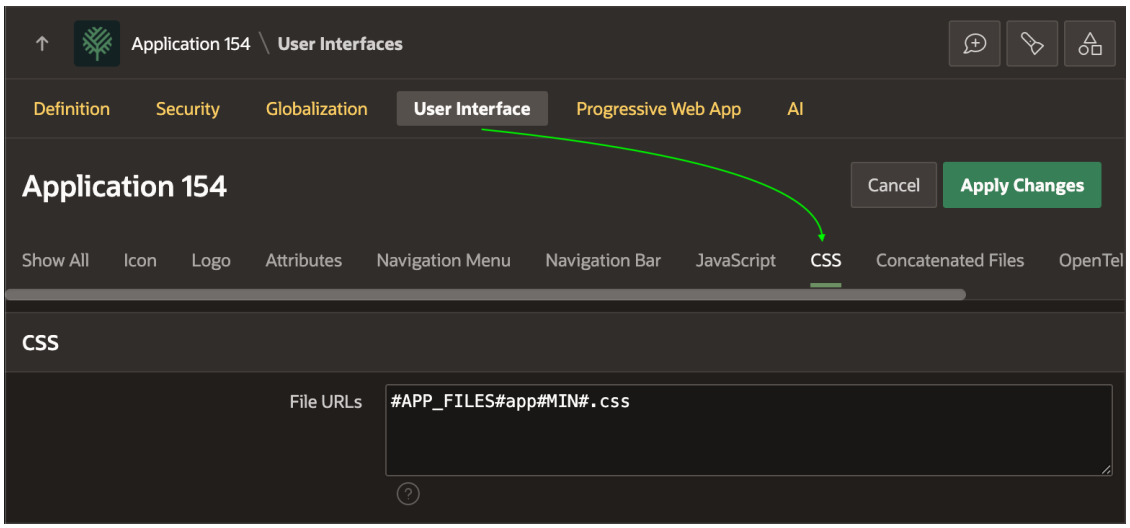
Figure 16-12 Defining a CSS Stylesheet in Static Application Files



To include this CSS style sheet in all pages of your application, start by clicking the *Copy* icon next to the style sheet's *Reference* URL to copy it to the clipboard. Next, go to the *User Interface Attributes* section of *Shared Components*. There, select the *CSS* tab as shown below. Paste the *Reference* value from the clipboard into the *File URLs* field. The result: CSS rules in your `app.css` now apply to all pages.

> **Tip**
>
> If you need to reference multiple URLs, put each on its own line in *File URLs* section.

Figure 16-13 Including a CSS Style Sheet in Every Page of Your App



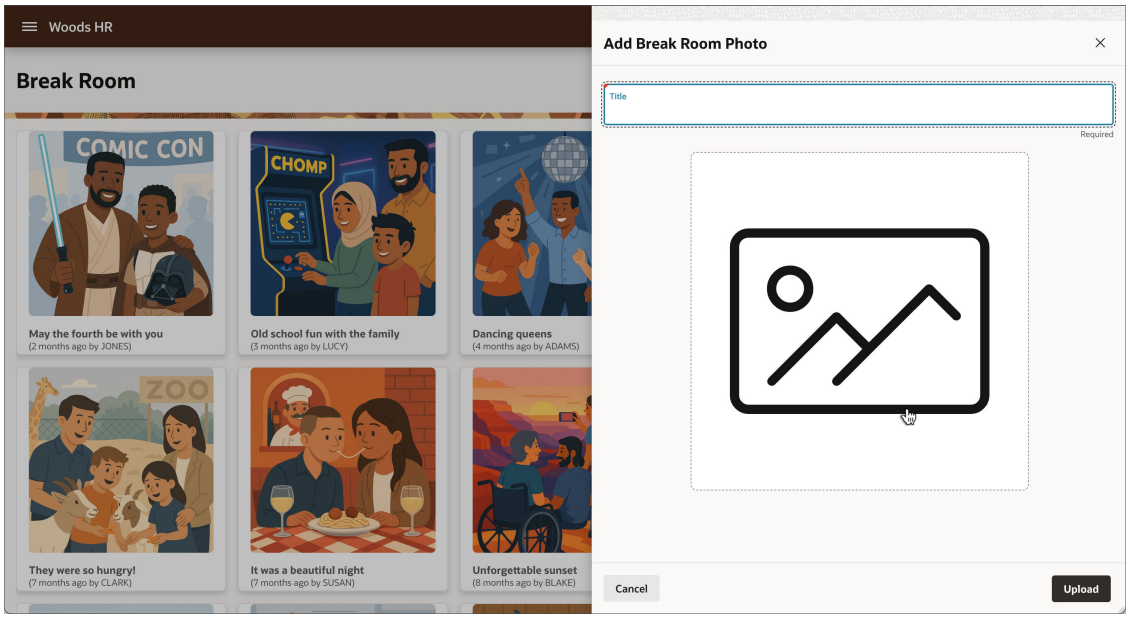
Copyright © 2026, Oracle and/or its affiliates.

16.3.5 Posting a Break Room Photo

Walk through a photo upload flow, from image selection to the refreshed Cards region on the calling page.

Now, when user *SMITH* adds a break room photo, she sees the modal drawer dialog shown below with a large upload image item and a field to enter a title.

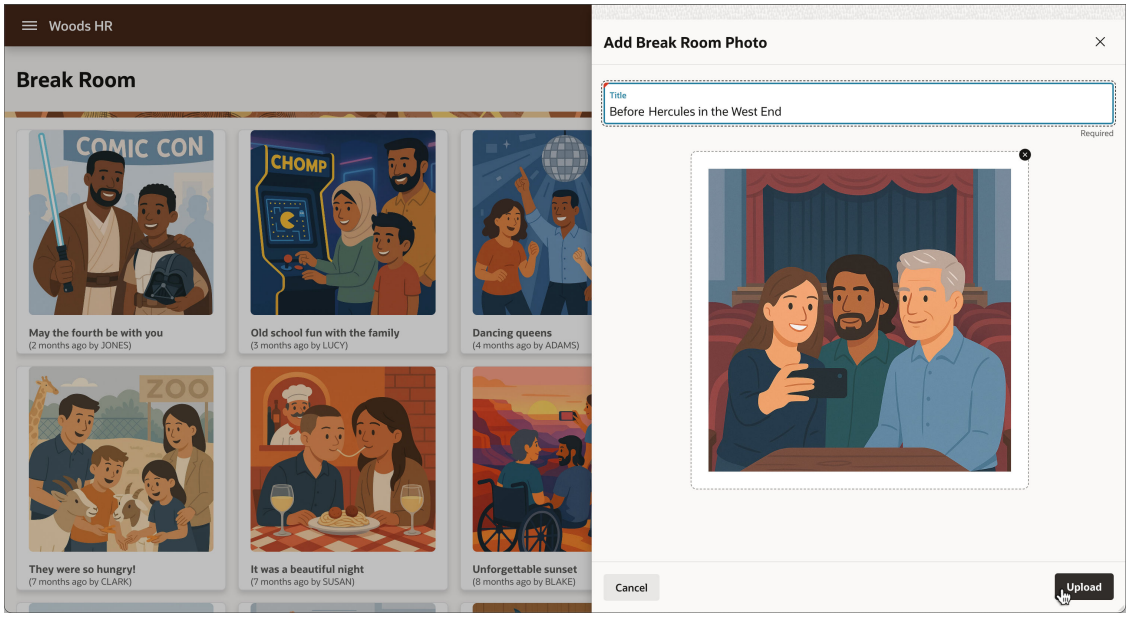
Figure 16-14 Upload Image Page Item with Custom Preview Icon Size



After clicking on the empty image icon a file dialog appears. If *Capture Using* had been set, the device's camera would have turned on to snap a new photo. After choosing – or taking – a photo, the preview of the selected photo appears as shown below. Clicking *(Upload)* submits the form. The *Form - Automatic DML* page process inserts the new row into the form's data source: the `EBA_DEMO_EMP_BREAKROOM_PHOTO_V` view. It sets the title, image, and hidden `P6_EMPNO` value defaulted from the `G_APP_USER_EMPNO` application item. This results in a new row in the underlying `EBA_DEMO_EMP_BREAKROOM_PHOTOS` table with the `IMAGE` BLOB column containing the uploaded photo.

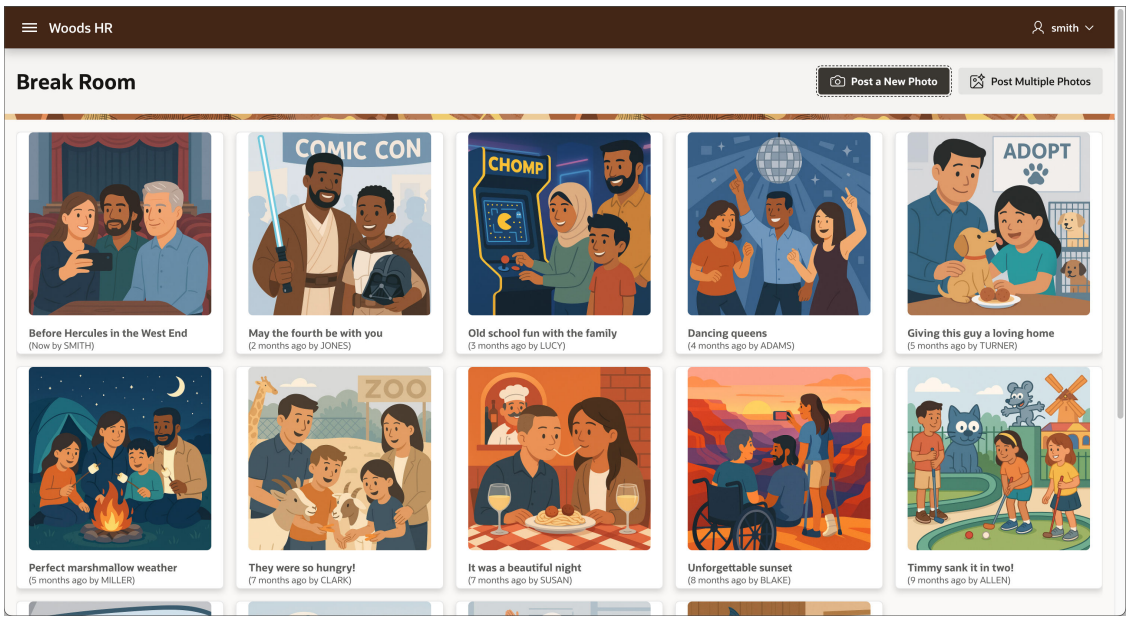
Copyright © 2026, Oracle and/or its affiliates.

Figure 16-15 User Sees Large Image Preview Before Clicking Upload



A dynamic action on the *Dialog Closed* event refreshes the Cards region, so *SMITH* immediately sees her posted photo.

Figure 16-16 Cards Page Refreshes on Dialog Close to Show Latest Photo



16.4 Displaying BLOB Column Images

Display images from a URL or BLOB column in common APEX regions, with a special approach for Interactive Grid.

Copyright © 2026, Oracle and/or its affiliates.

All regions can display images using a URL, but *many* can display images directly from a BLOB column including: *Form*, *Cards*, *Classic Report*, *Interactive Report*, *Template Component*, and *Search*. Showing a graphic in an *Interactive Grid* requires an `<img>` HTML tag whose `src` attribute references the URL of the graphic to display.

- Showing BLOB Column Images on Cards
  Configure a Cards region to display uploaded photos from a BLOB column with captions and preview links.
- Previewing Image in a Form Region
  To preview an image from a BLOB image column, use a *Form* region based on the table or view that includes it, or use a SQL query that selects it.
- Displaying BLOB Images in Classic Reports
  In a *Classic Report* region, include the *size* of the BLOB in your query instead of the BLOB column itself.
- Using BLOB Images in Interactive Reports
  In an *Interactive Report* region, include the *size* of the BLOB in your query instead of the BLOB column itself.
- Including BLOB Images in Content Row
  In a *Content Row* Template Component region, choose a table or view that includes the BLOB image column.
- Viewing BLOB Images in Search Results
  When creating a *Search* page, base your related *Search Configuration* on table or view including the BLOB image column.
- Showing BLOB Images in Interactive Grid
  An Interactive Grid has no native *Display Image* column type. To show an image in a row, use an HTML `<img>` tag with a `src` URL the browser can fetch and render inline.
- Using BLOB Column Image URLs Anywhere
  While the Interactive Grid *requires* it, use your preferred image URL generation approach anywhere that supports an image URL.

16.4.1 Showing BLOB Column Images on Cards

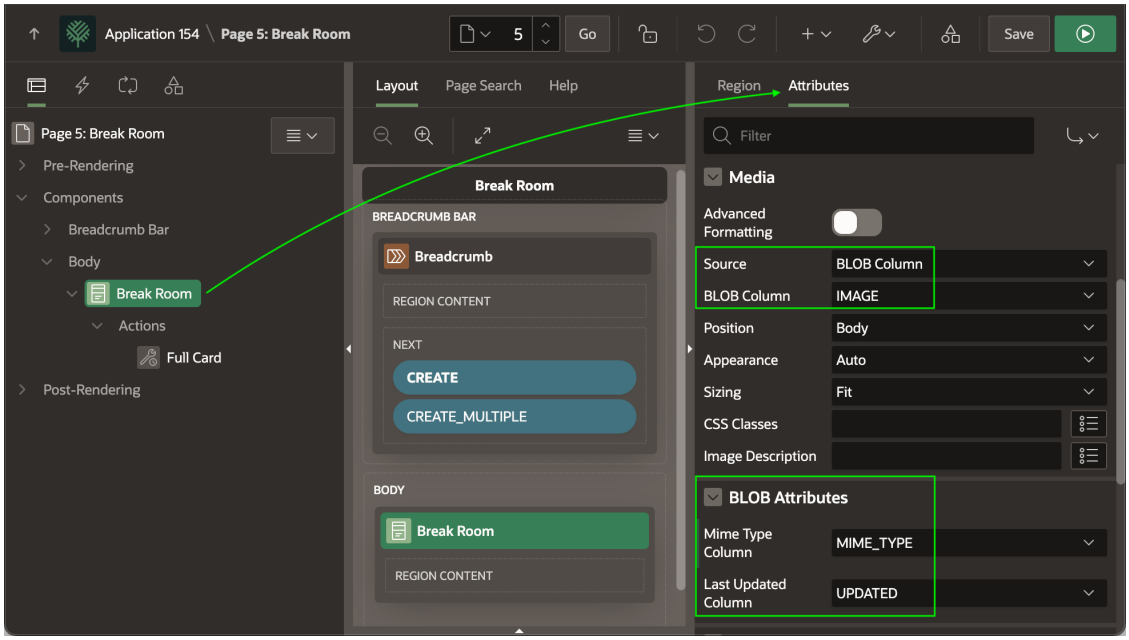
Configure a Cards region to display uploaded photos from a BLOB column with captions and preview links.

The *Break Room* page has a *Cards* region based on the `EBA_DEMO_EMP_BREAKROOM_PHOTO_V` view. Set the region's *Media* attributes to display the `IMAGE` BLOB column. As shown below, in the *BLOB Attributes* section it also specifies the:

- *Mime Type Column* – to help the browser correctly handle the image, and
- *Last Updated Column* – to let it know when to reload a cached image.

Copyright © 2026, Oracle and/or its affiliates.

Figure 16-17 Configuring Card Media Attributes to Display BLOB Column Image



To customize the presentation of the card's information, enable *Advanced Formatting* in the region's *Secondary Body* section, and use the following HTML with substitutions. This formats the `TITLE` value in bold and – after a line break (`<br>`) – info on who posted the photo and when in an extra small text body font.

```
<strong>&TITLE.</strong><br>
<span class="u-text-body-xs">(&POSTED. by &POSTED_BY.)</span>
```

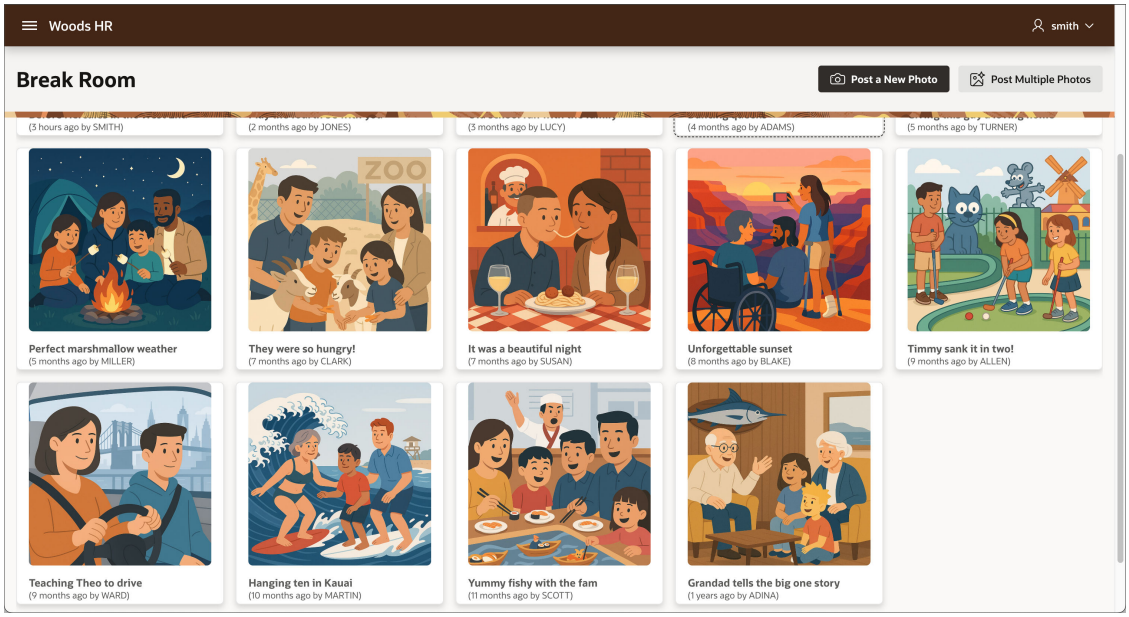
The region's cards are 16 characters wide, based on the page's base font size. The app-wide CSS rule that includes the following "Card View Item Width" variable assignment establishes this behavior for all *Cards* regions in the application.

```
    --a-cv-item-width: 16rem;
```

To let users see any image in a larger size, a *Full Card* action on the region redirects the user to a *Preview Break Room Photo* modal dialog page, passing the selected image `ID`. These few settings make the Break Room photo browsing experience below.

Copyright © 2026, Oracle and/or its affiliates.

Figure 16-18 Configuring Card Item Media from a CLOB Image Column



16.4.2 Previewing Image in a Form Region

To preview an image from a BLOB image column, use a *Form* region based on the table or view that includes it, or use a SQL query that selects it.

IF you pick a SQL query approach, ensure you indicate the primary key column. You can hide all columns other than the image and any title you want to display. For example, mark the `ID` column as *Primary Key*, and set the type of all columns *except* `P18_TITLE` and `P18_IMAGE` to *Hidden*.

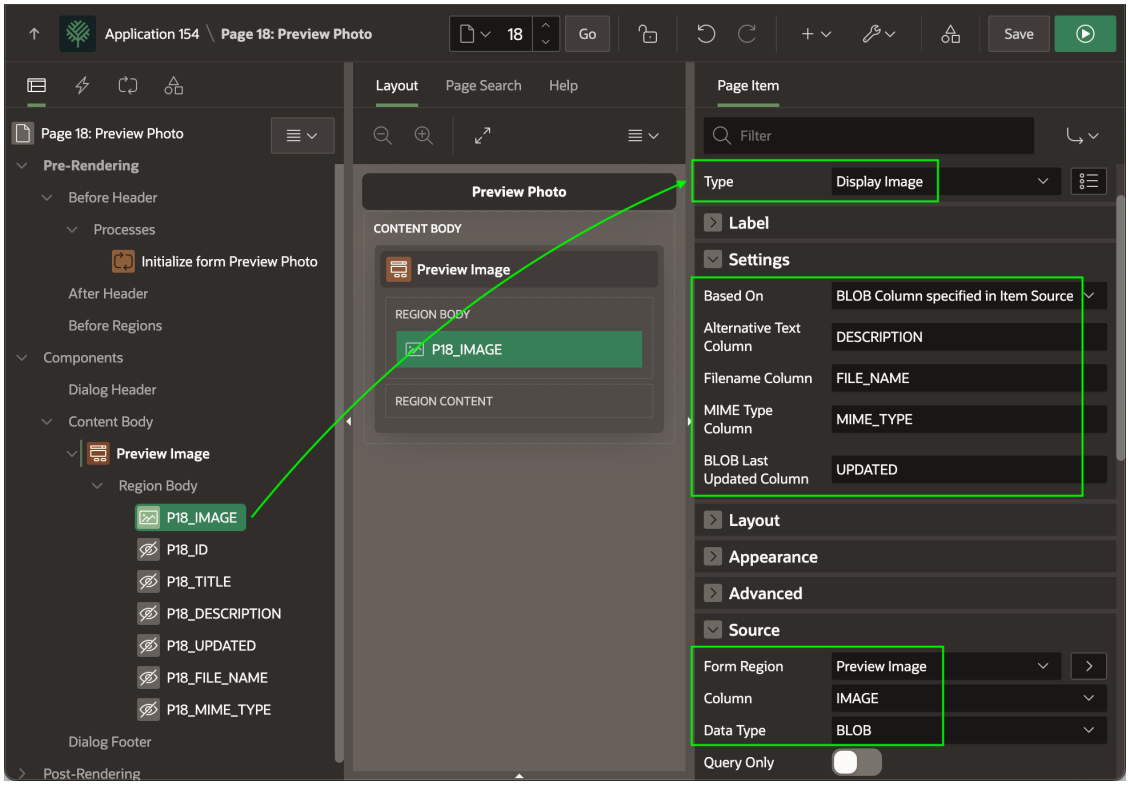
```
select id,
       title,
       image, /* BLOB image */
       updated,
       file_name,
       mime_type,
       apex_string.format('%s (%s by %s)',
            title,
            posted,
            posted_by) as description
  from eba_demo_emp_breakroom_photo_v
```

Then, as shown below, set `P18_IMAGE` to have type *Display Image*, to be based on a *BLOB Column*, and configure additional column names for *Alternative Text*, *Filename*, *MIME Type*, and *Last Updated*.

> **Tip**
>
> In addition to providing improved accessibility for screen-readers, the *Alternative Text* value provides the hover-over "tooltip" as well.

Copyright © 2026, Oracle and/or its affiliates.

Figure 16-19 Configuring Display Image Form Page Item for Image BLOB



To *omit* a label for the `P18_IMAGE` item:

- Blank out its *Label* property
- Set *Appearance* > *Template* to *Hidden*, and
- Set *Label Column Span* to zero (`0`).

To center the image and vertically stretch it to fill the dialog, set its *Advanced* > *CSS Classes* to `vertical_stretch` and add the following two CSS rules to the page's *CSS* > *Inline* setting.

```
.vertical_stretch {
   height: calc(100vh - 3em);
}

#P18_IMAGE {
   display : block;
   margin  : auto;
}
```

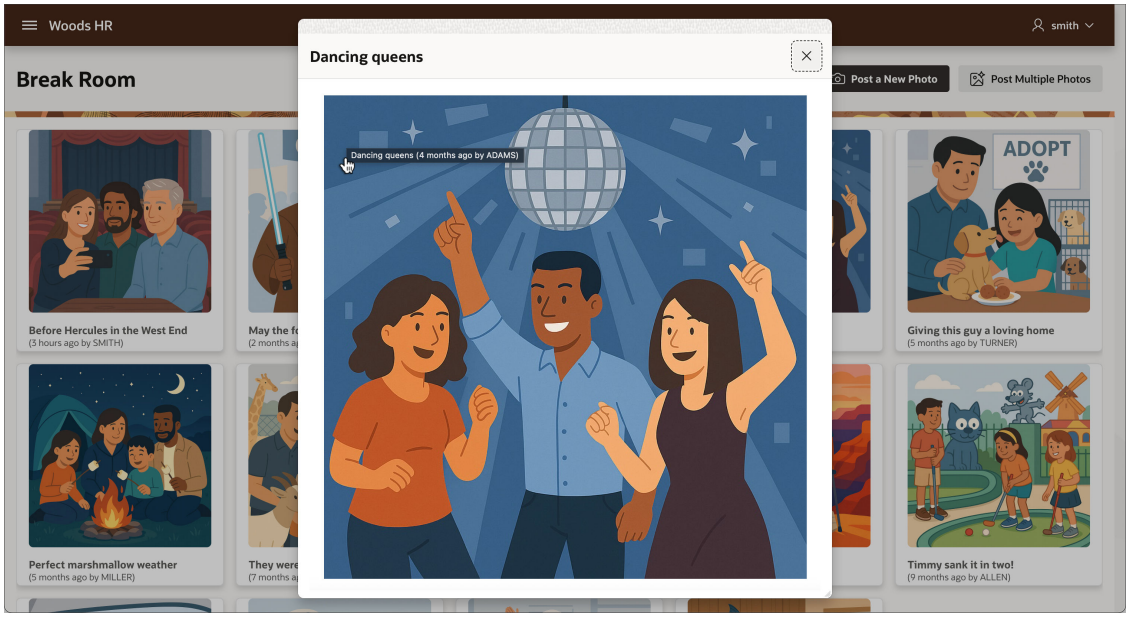
Tip

The `calc(100vh - 3em)` computes the remaining vertical height after accounting for the height of three (3) characters in the page's base font size used by the dialog title area. For more information, see Applying Vertical Stretch in a Dialog.

Copyright © 2026, Oracle and/or its affiliates.

If you use a modal dialog page, you can set a dynamic title on page load to show the image's title. In a *Page Load* dynamic action, call the `setDialogTitle()` JavaScript function from Providing App-Wide Helper Functions to set the dialog title dynamically to the title of the break room photo. After doing these few steps, you end up with the result shown below. A *Full Card* action on the *Break Room* Cards region in the calling page redirects to your new modal image preview dialog page, passing in the value of the `P18_ID` image primary key. This lets users get a better look at any photo. Hovering their mouse over the image, they can also see the image description in the tooltip.

Figure 16-20 Previewing an Image from a BLOB Column in a Form Region



16.4.3 Displaying BLOB Images in Classic Reports

In a *Classic Report* region, include the *size* of the BLOB in your query instead of the BLOB column itself.

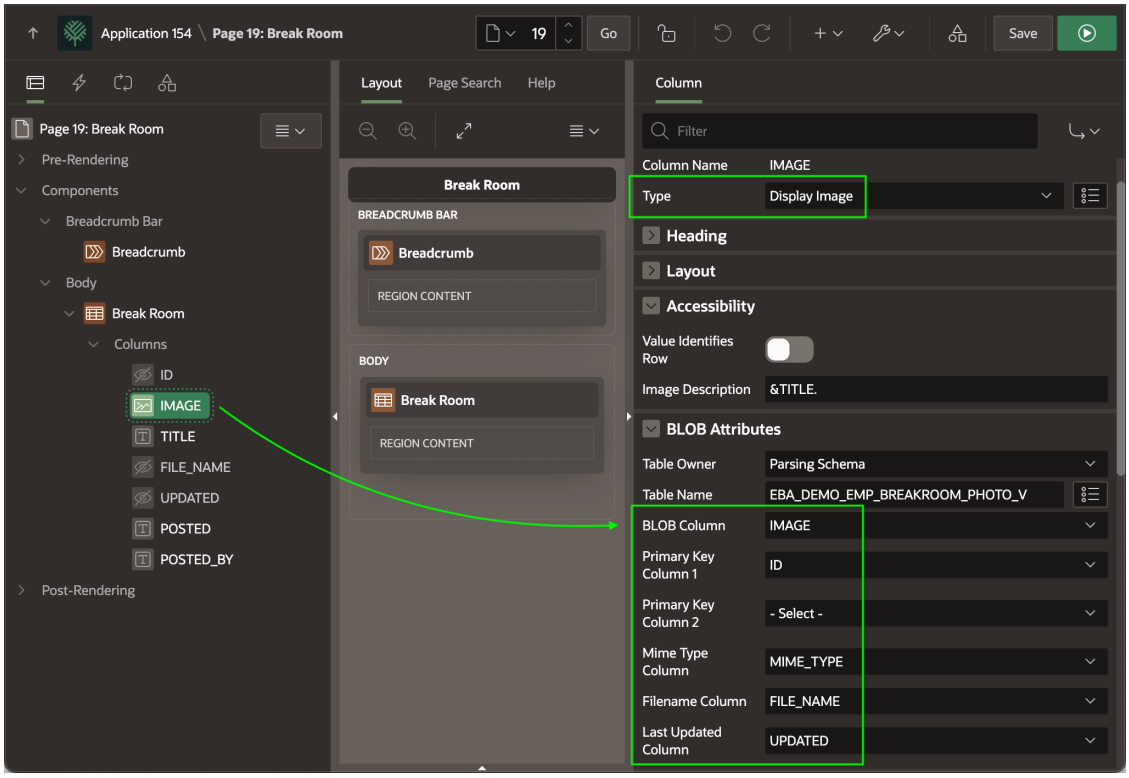
Use the `GETLENGTH` function in the `DBMS_LOB` package to compute it.

```
select  id,
        title,
        sys.dbms_lob.getlength(image) image, /* Size of BLOB Image */
        file_name,
        posted,
        posted_by,
        updated
from eba_demo_emp_breakroom_photo_v
order by updated desc
```

As shown below, set the *Type* of the `IMAGE` (size) column to *Display Image* and provide the additional image-related column names in the *BLOB Attributes* section. These include the BLOB image, Primary Key, Mime Type, Filename, and Last Updated date.

Copyright © 2026, Oracle and/or its affiliates.

Figure 16-21 Configuring Display Image BLOB Column in Classic Report



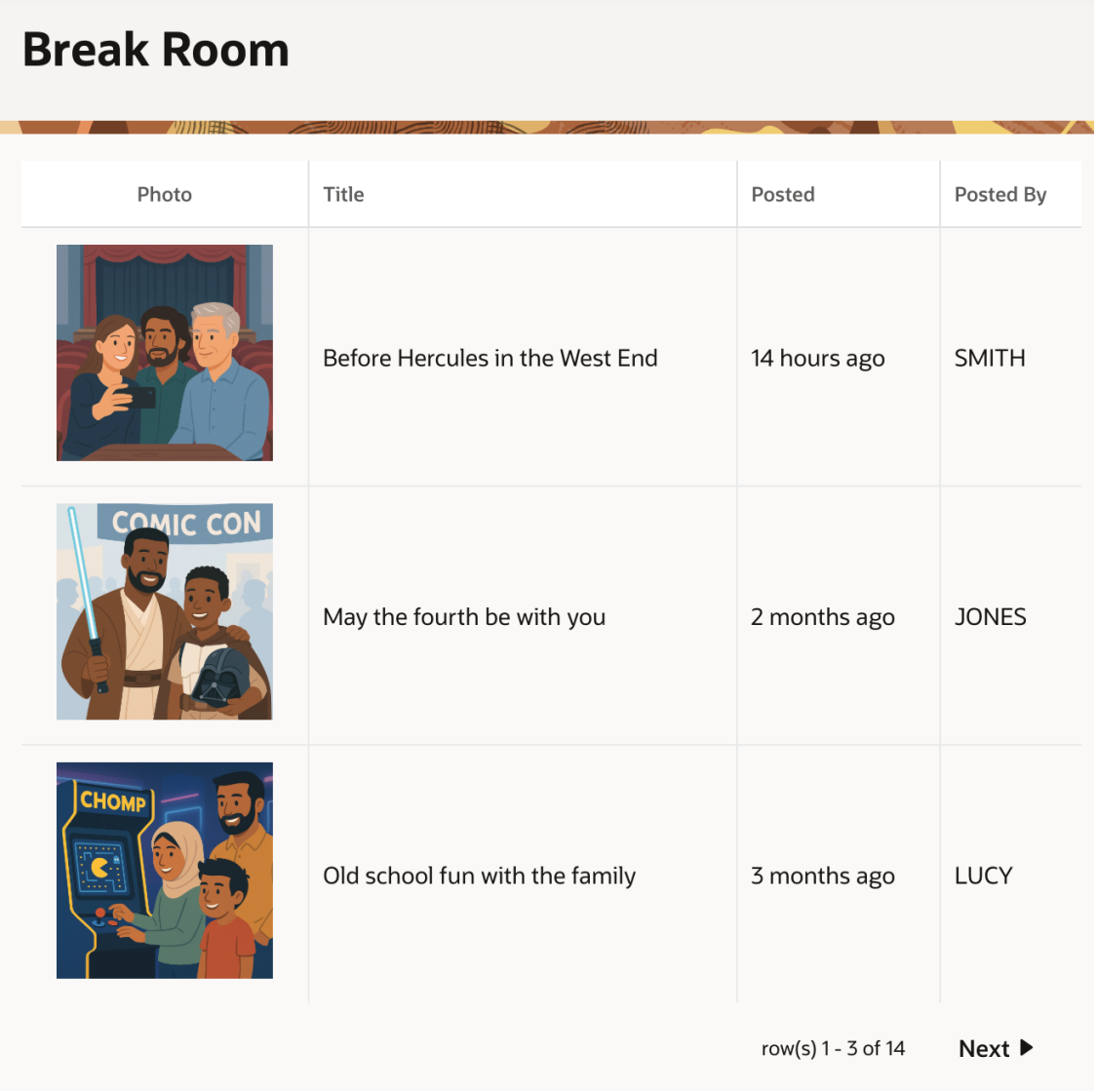
To control the size of the image preview, add the following rule to the *CSS* > *Inline* section of your page. It constrains the height and keeps the image's aspect ratio. You can also just constrain the width by specifying that property instead of `height`.

```
.t-Report-report img {
    /* constrain width or height + keep aspect ratio
     *
     * width: 130px;
     */
    height: 130px;
    object-fit: contain;
}
```

This combination produces an image in each row as shown below.

Copyright © 2026, Oracle and/or its affiliates.

Figure 16-22 Custom-Sized Images in a Classic Report



16.4.4 Using BLOB Images in Interactive Reports

In an *Interactive Report* region, include the *size* of the BLOB in your query instead of the BLOB column itself.

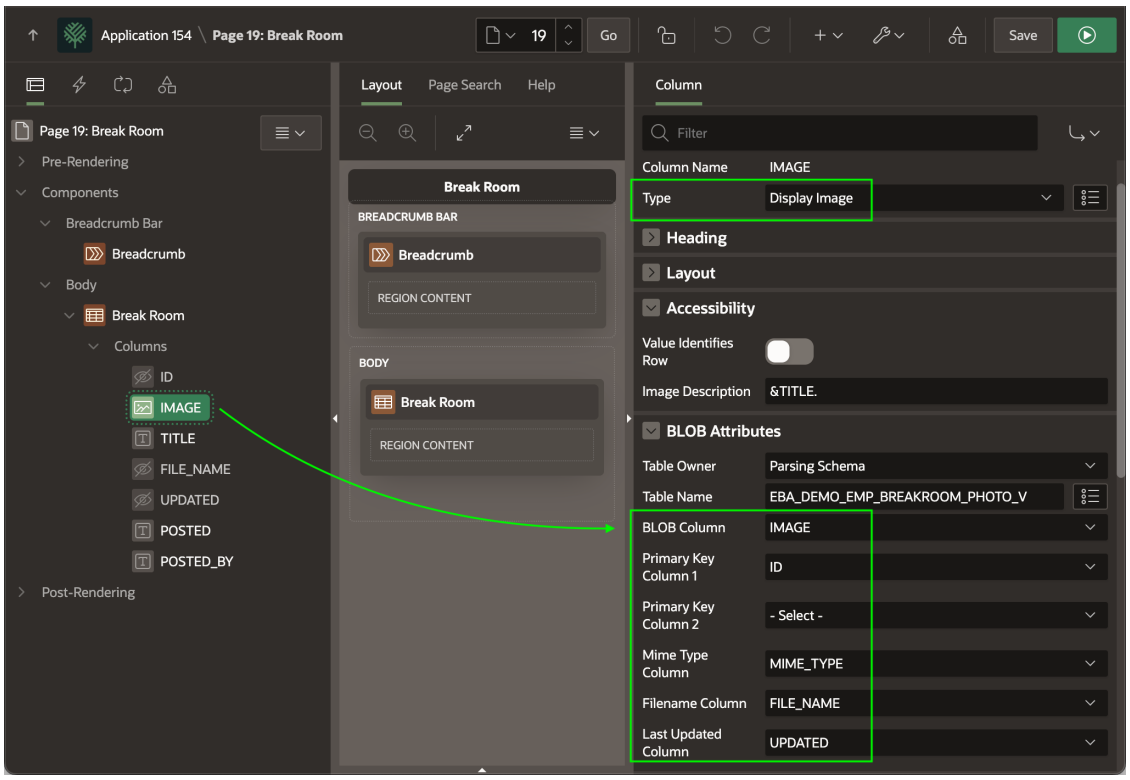
Use the `GETLENGTH` function in the `DBMS_LOB` package to compute it.

```
select id,
       title,
       sys.dbms_lob.getlength(image) image, /* Size of BLOB Image */
       file_name,
       posted,
       posted_by,
       updated
from eba_demo_emp_breakroom_photo_v
order by updated desc
```


Copyright © 2026, Oracle and/or its affiliates.

As shown below, set the *Type* of the `IMAGE` (size) column to *Display Image* and provide the additional image-related column names in the *BLOB Attributes* section. This includes column names for the BLOB image, Primary Key, Mime Type, Filename, and Last Updated date.

Figure 16-23 Configuring BLOB Image Column in an Interactive Report



To control the size of the image preview in each row, add the following rule to the page's *CSS > Inline* section. It constrains the height and keeps the image's aspect ratio. You can also just constrain the `width` by specifying that property instead of `height`.

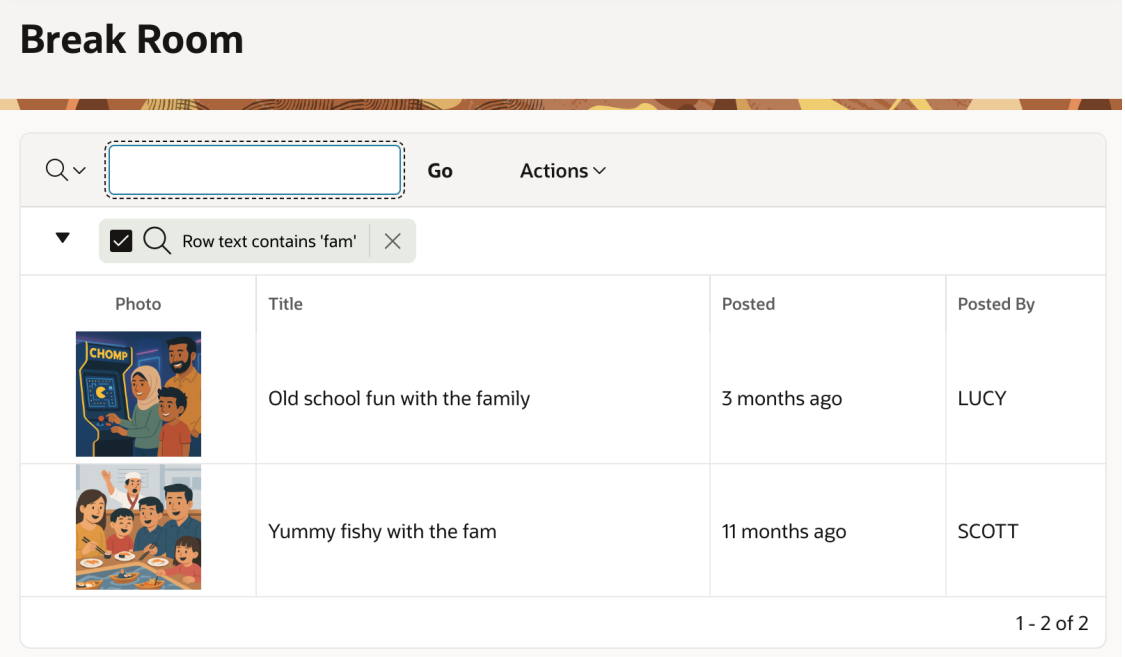
```
.a-IRR-table img {
    height: 90px;
    object-fit: contain;
}

.a-IRR {
   --a-gv-cell-padding-y: 0px;
}
```

This combination produces the interactive report with image preview shown below.

Copyright © 2026, Oracle and/or its affiliates.

Figure 16-24 BLOB Images in an Interactive Report



16.4.5 Including BLOB Images in Content Row

In a *Content Row* Template Component region, choose a table or view that includes the BLOB image column.

Alternatively, select the BLOB image column in the query as shown below.

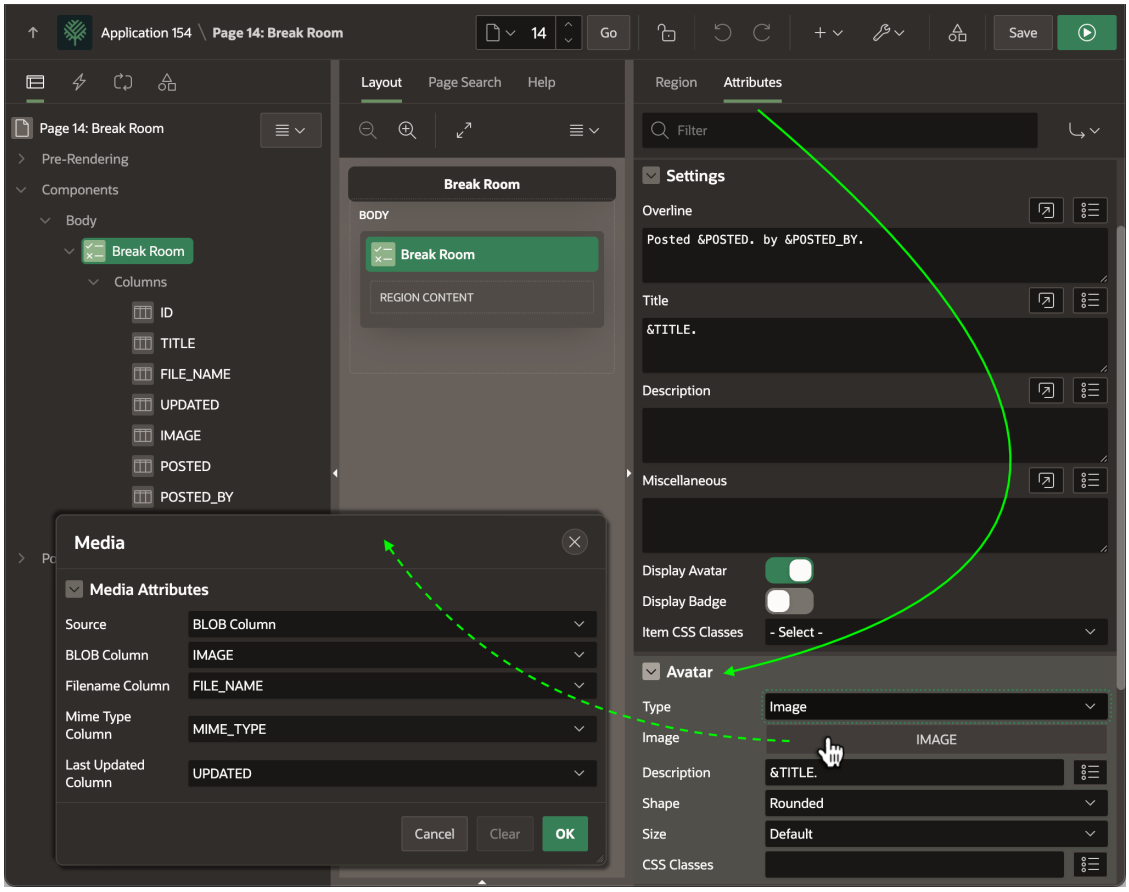
```
select id,
       title,
       image, /* BLOB image */
       file_name,
       posted,
       posted_by,
       mime_type,
       updated
from eba_demo_emp_breakroom_photo_v
```

Start by verifying that the primary key column has its *Primary Key* switch enabled.

On the region's *Attributes* tab, set the *Avatar* > *Type* to *Image*. Then, click the *Image* display button to open the *Media* dialog where you configure the BLOB image related column names.

Copyright © 2026, Oracle and/or its affiliates.

Figure 16-25 Configuring BLOB Image in a Content Row



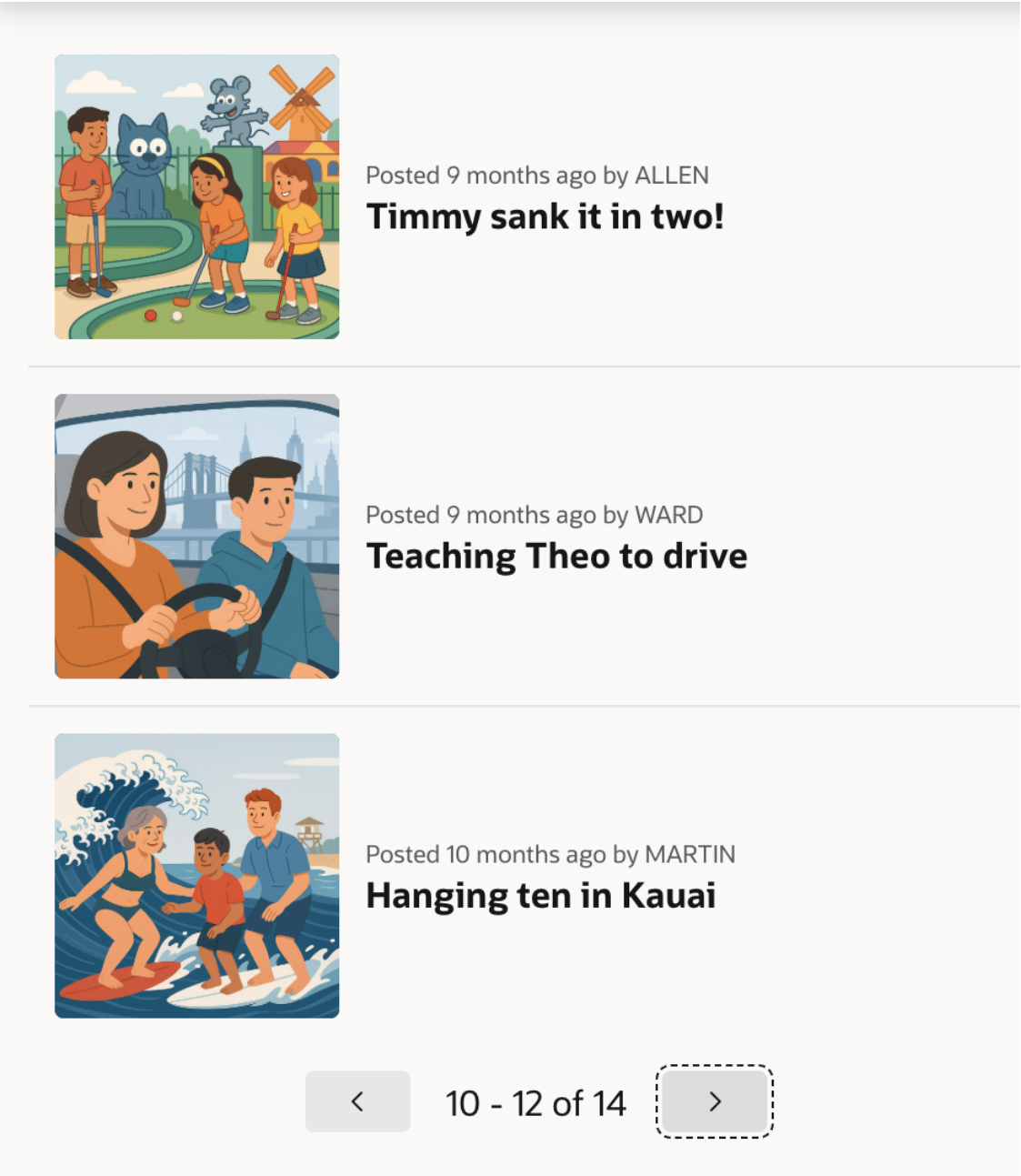
To control the size of the avatar image, add the following rule to the *CSS* > *Inline* section of the page:

```
.t-Avatar--image {
   --ut-avatar-size: 10rem;
}
```

This combination produces the result shown below. The figure shows a Content Row region showing three break room photos per page.

Copyright © 2026, Oracle and/or its affiliates.

Figure 16-26 Content Row with BLOB Image in the Avatar Image Slot



16.4.6 Viewing BLOB Images in Search Results

When creating a *Search* page, base your related *Search Configuration* on table or view including the BLOB image column.

It can also use a query that includes the image column like:

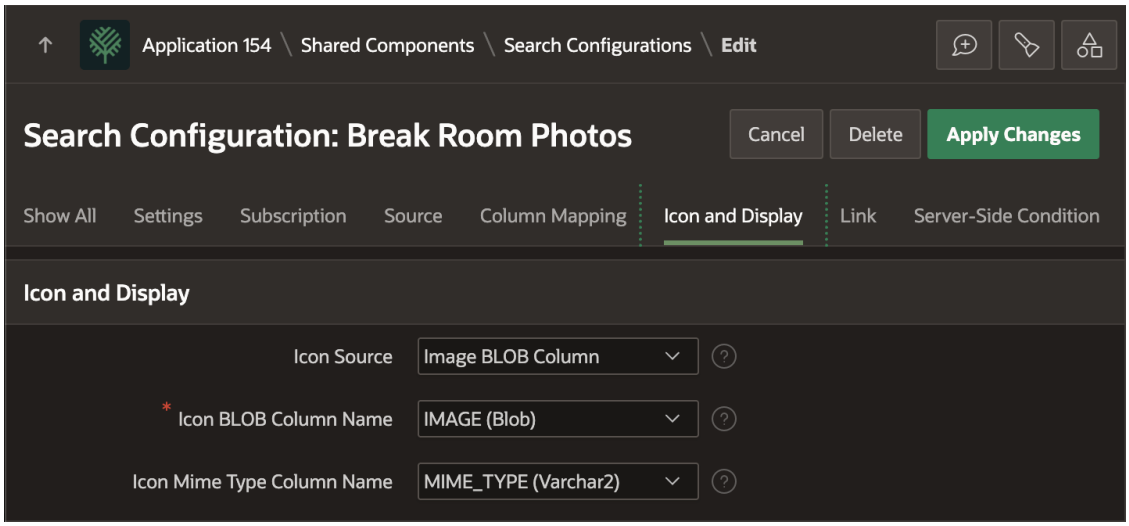
```
select id,
       title,
       image, /* BLOB image */
       file_name,
```


Copyright © 2026, Oracle and/or its affiliates.

```
       posted,
       posted_by,
       mime_type,
       updated
from eba_demo_emp_breakroom_photo_v
```

Set the *Icon Source* to *Image BLOB Column* in the *Icon and Display* section. Then choose the image BLOB column and Mime Type Column as shown below.

Figure 16-27 Setting Up Search Configuration to Use a BLOB Image Column



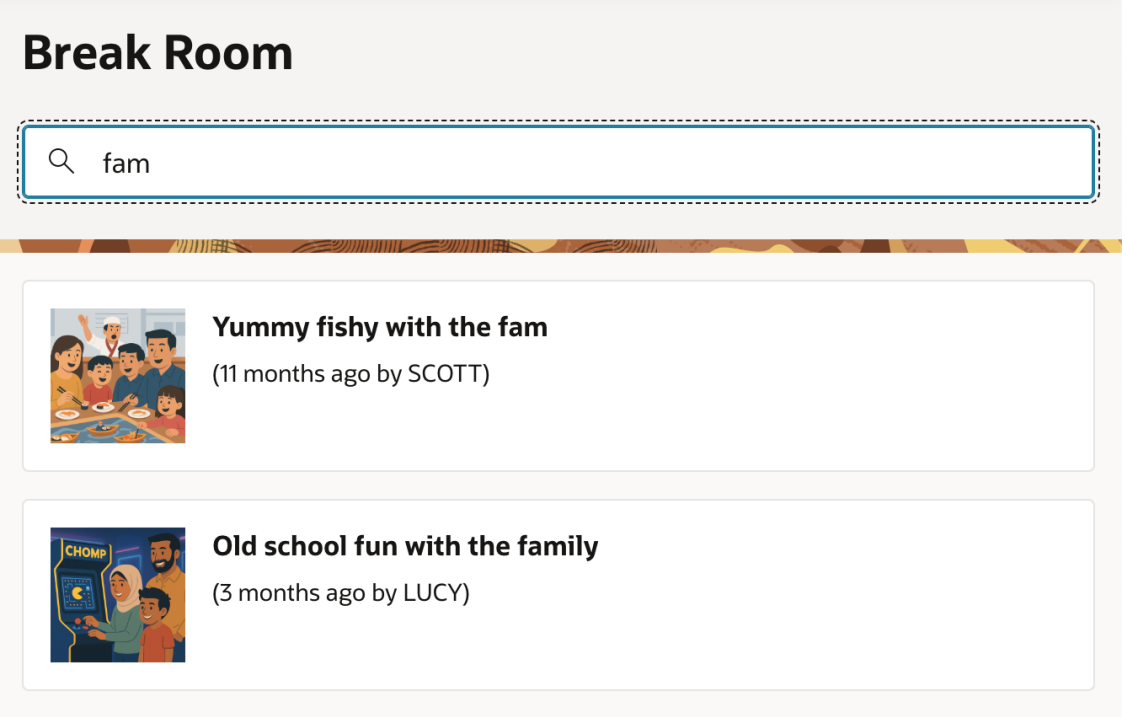
In the *Search* page, size the image preview by adding the following rule to the page's *CSS* > *Inline* settings. It constrains the width and keeps the image's aspect ratio. You can also just constrain the height by specifying that property instead of `width`.

```
.a-ResultsItem-image img {
    /* constrain width or height + keep aspect ratio
     *
     * height: 50px;
     */
   width: 50px;
   object-fit: contain;
}
```

With that in place, your search results show the BLOB Image exactly to your specifications.

Copyright © 2026, Oracle and/or its affiliates.

Figure 16-28 Including BLOB Column Image in Search Page Results



16.4.7 Showing BLOB Images in Interactive Grid

An Interactive Grid has no native *Display Image* column type. To show an image in a row, use an HTML `<img>` tag with a `src` URL the browser can fetch and render inline.

The `GET_BLOB_FILE_SRC` function in the `APEX_UTIL` package generates the URL you need. Alternatively, you can serve the inline BLOB images from a custom page.

- Using GET_BLOB_FILE_SRC for Image URL
  Build an image URL with `GET_BLOB_FILE_SRC` and display it in an Interactive Grid using a generated `img` tag.
- Configuring Reference Image Page Item
  Use a BLOB-backed form item to let `GET_BLOB_FILE_SRC` infer the file metadata for an image URL.
- Serving Inline BLOB Images with a Page
  Serve an inline image from a page process by calling `APEX_HTTP.DOWNLOAD` with the image `ID`.
- Using an Inline Image Serving Page
  Generate an image URL with `APEX_PAGE.GET_URL` and pass the image ID to your image-serving page.
- Weighing X01 Against a Secure Page Item
  The `X01` parameter is *convenient*, but not checksum-protected. Understand the potential risk of using this parameter to decide if a secure page item is more appropriate for your use case.

Copyright © 2026, Oracle and/or its affiliates.

16.4.7.1 Using GET_BLOB_FILE_SRC for Image URL

Build an image URL with `GET_BLOB_FILE_SRC` and display it in an Interactive Grid using a generated `img` tag.

The grid in the page below uses the following query. Its `CASE` expression for the `IMAGE` column checks that the BLOB contains data. If so, it calls `APEX_STRING.FORMAT` to build an `<img>` tag. The `alt` attribute aids accessibility, `title` is the tooltip text, and `src` contains the URL to fetch the image. `FORMAT` replaces each `%s` with the expression values that follow, in order. The first and second placeholders for the `alt` and `title` attributes both get the image `TITLE` "escaped" as needed, while the `GET_BLOB_FILE_SRC` function generates the image URL for the `src` attribute using the third `%s` placeholder. For details on the arguments to this function and the required `P12_IMAGE BLOB` page item it references by name, see Configuring Reference Image Page Item.

```
select id,
       title,
       case
           when     image is not null
                and dbms_lob.getlength(image) > 0
           then apex_string.format('<img alt="%s" title="%s" src="%s">',
                apex_escape.html_attribute(title),
                apex_escape.html_attribute(title),
                apex_util.get_blob_file_src('P12_IMAGE', ID))
       end as image,
       file_name,
       posted,
       posted_by,
       updated
from eba_demo_emp_breakroom_photo_v
```

Tip

Some characters have special meaning in the Hypertext Markup Language (HTML). The `HTML_ATTRIBUTE` function in the `APEX_ESCAPE` package replaces them with equivalent character references using the Unicode *numeric* representation. This is known as "escaping" the value. For example, a break room photo title like:

```
Team's "Best" Food & Drink
```

turns into the following when you escape it for use in an HTML attribute:

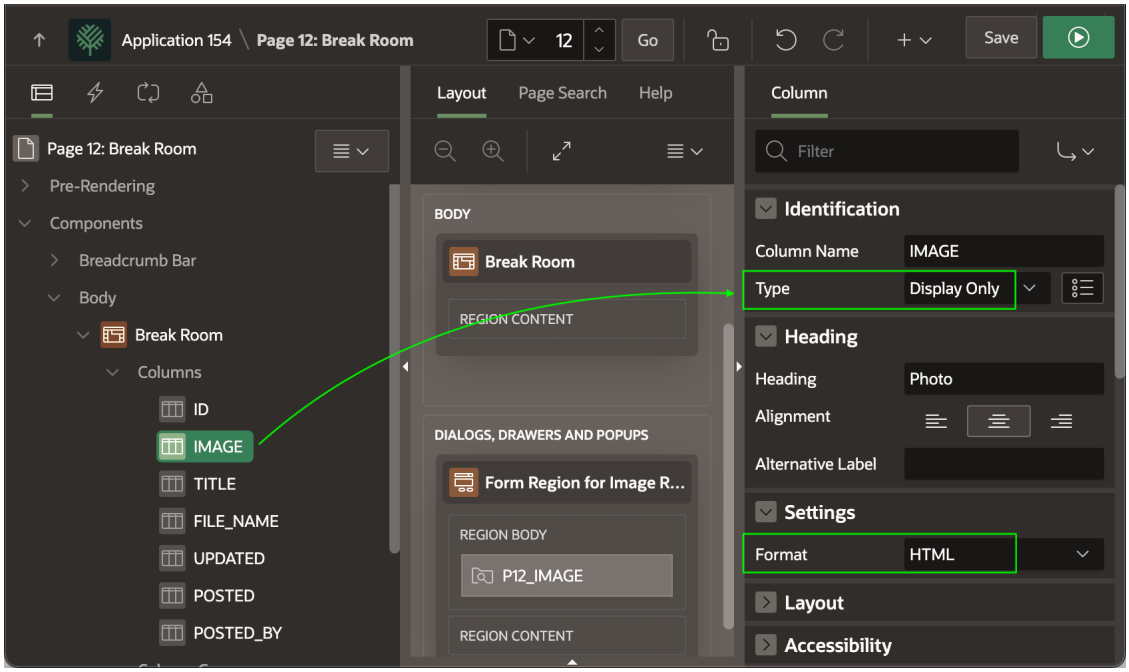
```
Team&#x27;s&#x20;&#x22;Best&#x22;&#x20;Food&#x20;&#x26;&#x20;Drink
```

Forgetting to escape values can lead to unexpected formatting or incorrect behavior.

The query produces the `<img>` tag as the `IMAGE` column value. To configure it to display as an image, set its column *Type* to *Display Only*, and its *Format* to *HTML*, as shown below.

Copyright © 2026, Oracle and/or its affiliates.

Figure 16-29 Configuring HTML IMAGE tag to Display Only



Size the image by adding the rule below to the page's *CSS* > *Inline* section. It constrains the height and keeps the image's aspect ratio. You can also just constrain the width by specifying that property instead of `height`.

```
.a-IG img {
    /* constrain width or height + keep aspect ratio
     * width: 90px;
     */
    height: 90px;
    object-fit: contain;
}
```

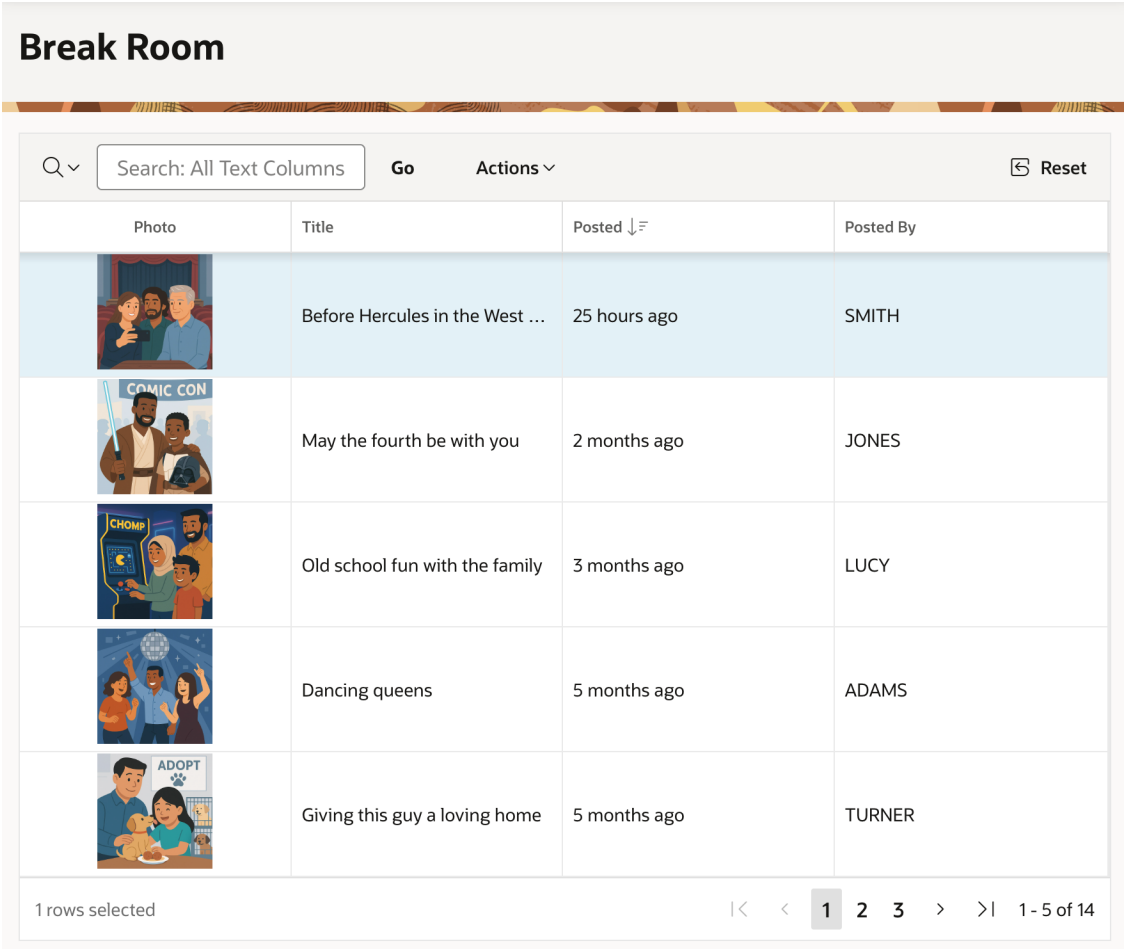
This combination produces the Interactive Grid shown below.

> **Tip**
>
> If your Interactive Grid is *editable*, make sure to enable *Query Only* on the `IMAGE` column.

The figure displays breakroom photos page-by-page in a read-only Interactive Grid, five rows at a time.

Copyright © 2026, Oracle and/or its affiliates.

Figure 16-30 Displaying BLOB Image Using Image URL via GET_BLOB_FILE_SRC



16.4.7.2 Configuring Reference Image Page Item

Use a BLOB-backed form item to let `GET_BLOB_FILE_SRC` infer the file metadata for an image URL.

Recall the expression used in the query to generate the image URL:

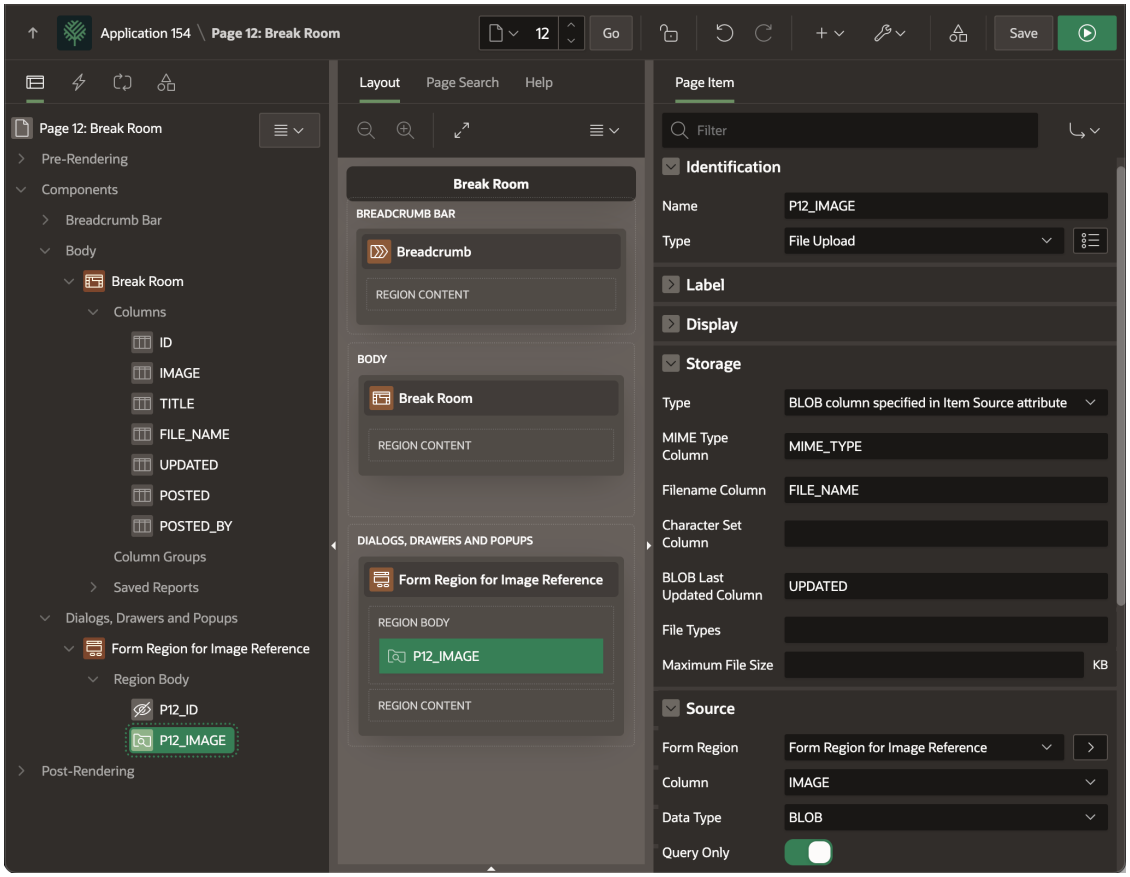
```
select
       ▒
         apex_util.get_blob_file_src('P12_IMAGE', ID)
       ▒
  from eba_demo_emp_breakroom_photo_v
```

`ID` is the primary key column in the employee break room photos view, and `P12_IMAGE` names a form page item whose *source* is the image BLOB. `GET_BLOB_FILE_SRC` infers the column names for the file BLOB, MIME Type, and Last Updated date from this page item. It gets the *table* name from the source of the item's related *Form* region.

The `P12_IMAGE` item does not need to be *visible* on the page. As shown below, it can reside in the *Dialogs, Drawers, and Popups* slot on the page or be visually hidden by applying the `u-hidden` CSS class.

Copyright © 2026, Oracle and/or its affiliates.

Figure 16-31 Form Image Item GET_BLOB_FILE_SRC References for BLOB Metadata



16.4.7.3 Serving Inline BLOB Images with a Page

Serve an inline image from a page process by calling `APEX_HTTP.DOWNLOAD` with the image `ID`.

Instead of generating a URL for APEX's native inline image handler with `APEX_UTIL.GET_BLOB_FILE_SRC`, you can create your own page to serve the image. The page calls `APEX_HTTP.DOWNLOAD` in a *Pre-Rendering* process in the *Before Header* section. For simplicity, it can use the native `X01` parameter APEX defines. The following `DOWNLOAD_BREAKROOM_PHOTO` procedure retrieves the image data and MIME type from the break room photos table using the image ID. Then it downloads the image data. Notice it passes `true` to the `p_is_inline` parameter so the image is sent to the client in the way the browser can use as an inline image.

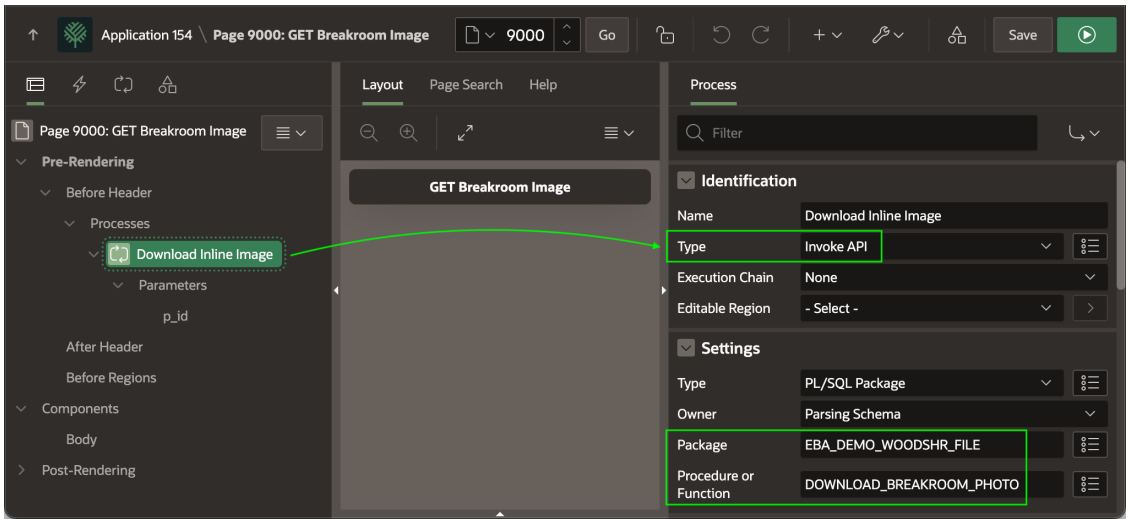
```
-- In package eba_demo_woodshr_file
procedure download_breakroom_photo(
    p_id in number)
is
    l_file_blob eba_demo_emp_breakroom_photos.image%type;
    l_mime_type eba_demo_emp_breakroom_photos.mime_type%type;
begin
    select image, mime_type
      into l_file_blob, l_mime_type
      from eba_demo_emp_breakroom_photos
     where id = p_id;
```


Copyright © 2026, Oracle and/or its affiliates.

```
    apex_http.download(
        p_blob         => l_file_blob,
        p_content_type => l_mime_type,
        p_is_inline    => true );
end download_breakroom_photo;
```

Call this procedure with an Invoke API process in the *Before Header* section. The figure shows the Download Inline Image page process selected in the component tree in Page Designer, highlighting its **Type** of **Invoke API** and its **Package** name and **Procedure** name to invoke the `DOWNLOAD_BREAKROOM_PHOTO` routine in the `EBA_DEMO_WOODSHR_FILE` package.

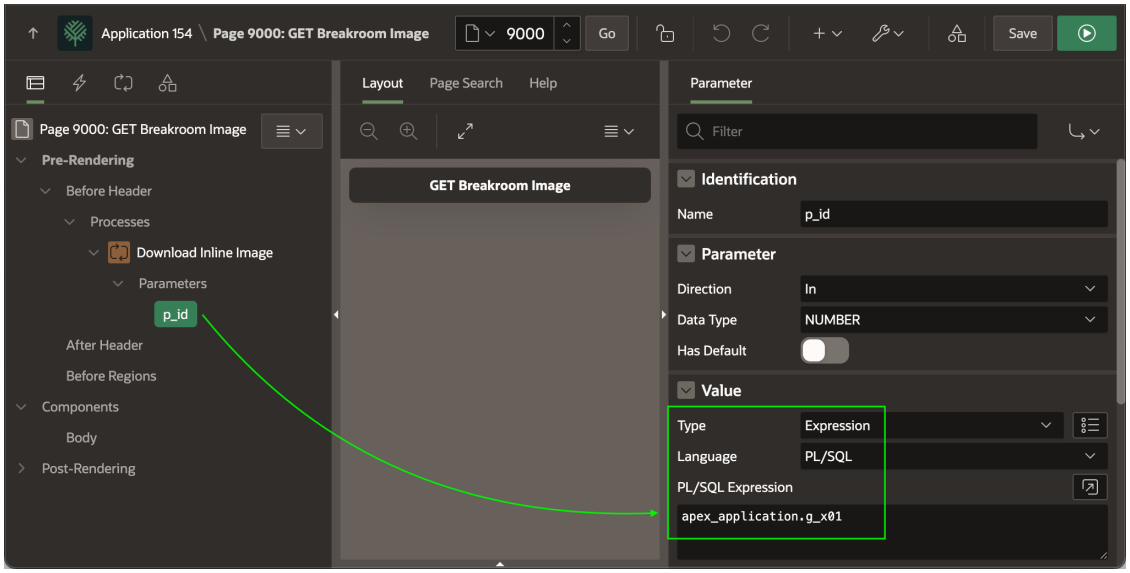
Figure 16-32 Pre-Rendering Invoke API Process to Download an Image



Configure the value of its `p_id` parameter using the PL/SQL expression `apex_application.g_x01`. The figure shows the `p_id` parameter of the `DOWNLOAD_BREAKROOM_PHOTO` routine selected in the component tree in Page Designer, and highlights the value configured is a PL/SQL expression to return the value of the predefined PL/SQL global variable `G_X01` in the `APEX_APPLICATION` package. This global variable contains the `X01` query string parameter value from the URL that fetches the image by ID.

Copyright © 2026, Oracle and/or its affiliates.

Figure 16-33 Configuring Break Room Image ID Parameter Value



16.4.7.4 Using an Inline Image Serving Page

Generate an image URL with `APEX_PAGE.GET_URL` and pass the image ID to your image-serving page.

With the image-serving page 9000 in place, use `APEX_PAGE.GET_URL` to generate an image URL. Pass the image ID in the `x01` parameter as shown below in the modified source query for an Interactive Grid:

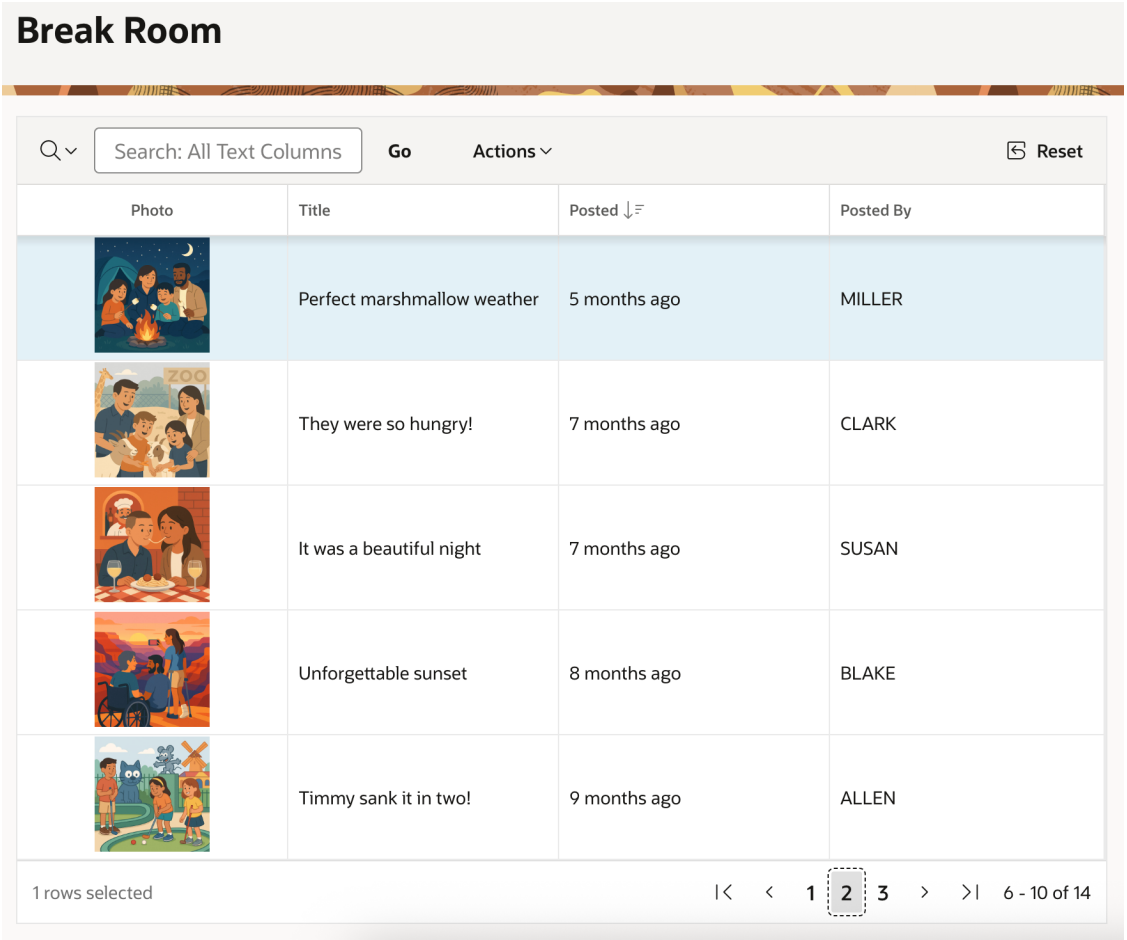
```
select id,
       title,
       case
           when     image is not null
                and dbms_lob.getlength(image) > 0
           then apex_string.format('<img alt="%s" title="%s" src="%s">',
                apex_escape.html_attribute(title),
                apex_escape.html_attribute(title),
                apex_page.get_url(
                    p_page => 9000,
                    p_x01  => ID))
       end as image,
       file_name,
       posted,
       posted_by,
       updated
from eba_demo_emp_breakroom_photo_v
```


Copyright © 2026, Oracle and/or its affiliates.

> **Tip**
>
> If using `APEX_PAGE.GET_URL` in a modal dialog or drawer page to reference an inline image, pass `true` to the optional `p_plain_url` parameter to ***avoid*** including JavaScript code to close the dialog before navigating to the supplied URL.

As shown below, this produces the same image-in-a-grid result as before without requiring the hidden form page item the `GET_BLOB_FILE_SRC` function needs.

Figure 16-34 Inline BLOB Images in an Interactive Grid Using Image-Serving Page



16.4.7.5 Weighing X01 Against a Secure Page Item

The `X01` parameter is *convenient*, but not checksum-protected. Understand the potential risk of using this parameter to decide if a secure page item is more appropriate for your use case.

- Understanding Implications of Using X01
  Understand how an authenticated user can change the value of an image ID in the image URL using an image-serving page.
- Downloading Image with Secure Page Item
  When images are user-specific or sensitive, use a hidden page item instead of `X01`.

Copyright © 2026, Oracle and/or its affiliates.

- Generating Image URL with Secure Item
  Generate checksum-protected image URLs by passing the image ID using the tamper-resistant page item name to `get_url` using its `p_items` and `p_values` parameters.
- Experiencing URL Checksum Protection
  Witness the URL tampering protection for image serving pages using a page item for the image ID.

16.4.7.5.1 Understanding Implications of Using X01

Understand how an authenticated user can change the value of an image ID in the image URL using an image-serving page.

When viewing a page containing image URLs using the `x01` parameter, if an *authenticated* user knows the ID of a *different* image, they can view it by:

- Selecting *Copy Image Address* from the context menu on an existing image in the page,
- Creating a new browser tab and pasting the copied URL into the address bar,
- Manually modifying the `x01=123456` to `x01=98765`, and pressing *[Enter]*.

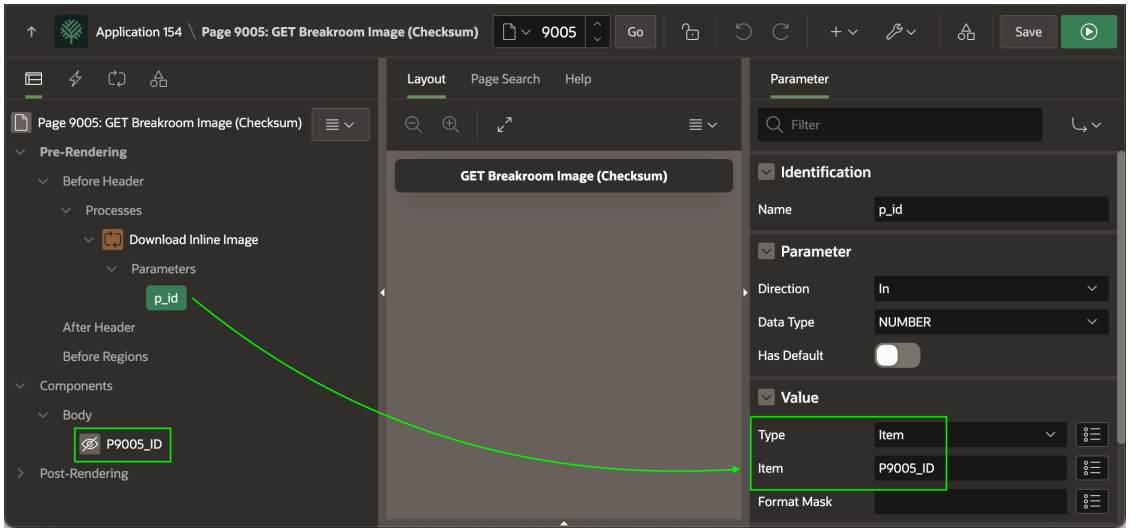
In *Woods HR*, where any *authenticated* user can view *any* break room photo, it's no problem.

16.4.7.5.2 Downloading Image with Secure Page Item

When images are user-specific or sensitive, use a hidden page item instead of `X01`.

For example, page 9005 is a *copy* of the image-serving page 9000, but it uses the hidden `P9005_ID` page item value for the `P_ID` parameter in its `DOWNLOAD_BREAKROOM_IMAGE` call.

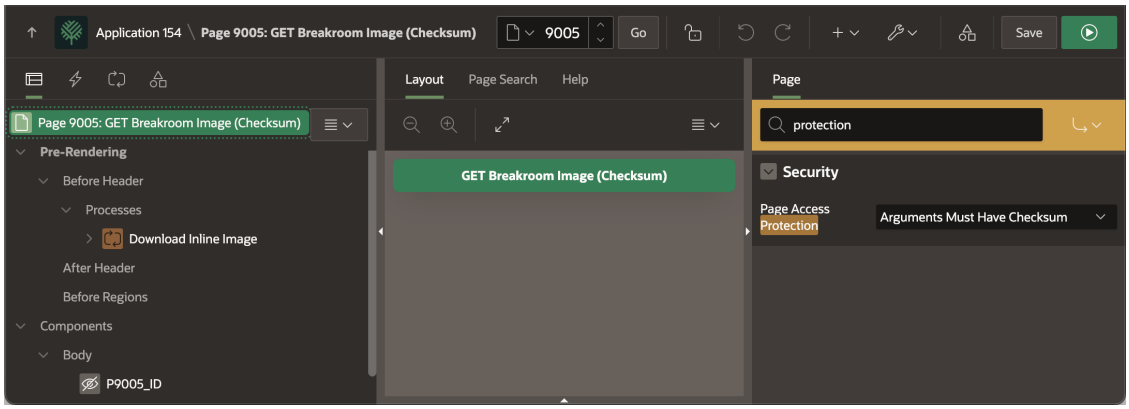
Figure 16-35 Passing Checksum-Protected Hidden Page Item for Image ID



By default, pages use *Arguments Must Have Checksum* as the *Page Access Protection* setting as shown below. This causes `APEX_PAGE.GET_URL` to generate an *additional* checksum parameter in every URL. This extra token protects parameter values from manual manipulation.

Copyright © 2026, Oracle and/or its affiliates.

Figure 16-36 Default Page Access Protection Setting Requires Checksum for URL Arguments



16.4.7.5.3 Generating Image URL with Secure Item

Generate checksum-protected image URLs by passing the image ID using the tamper-resistant page item name to `get_url` using its `p_items` and `p_values` parameters.

When using page 9005 to generate image URLs, the `get_url` call appears below in the modified Interactive Grid query. The `p_items` and `p_values` parameters replace the previous `p_x01`:

```
select id,
       title,
       case
           when     image is not null
                and dbms_lob.getlength(image) > 0
           then apex_string.format('<img alt="%s" title="%s" src="%s">',
                apex_escape.html_attribute(title),
                apex_escape.html_attribute(title),
                apex_page.get_url(
                    p_page   => 9005,
                    p_items  => 'P9005_ID',
                    p_values => ID))
       end as image,
       file_name,
       posted,
       posted_by,
       updated
from eba_demo_emp_breakroom_photo_v
```


Copyright © 2026, Oracle and/or its affiliates.

Tip

If the image to download has a *multi-column* primary key, then pass multiple items and values in the `get_url` call by separating values using commas:

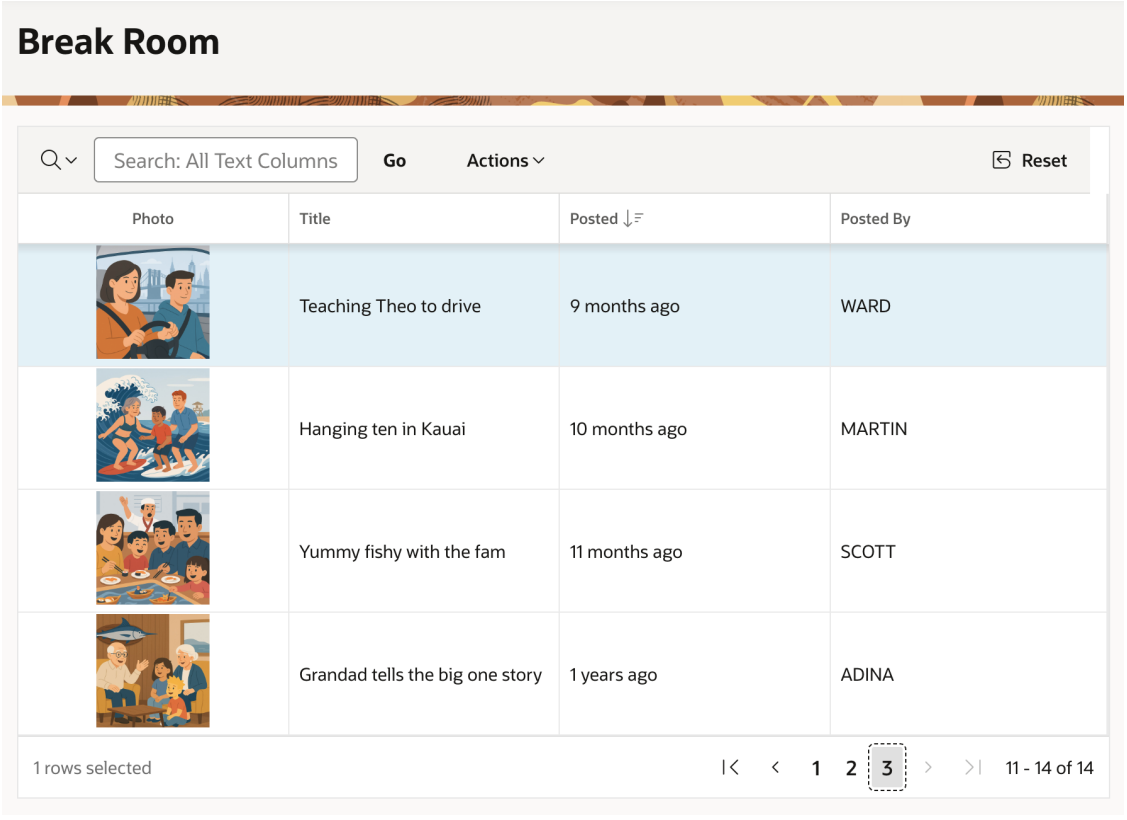
```
apex_page.get_url(
    p_page   => 9005,
    p_items  => 'P9005_ID1,P9005_ID2',
    p_values => ID1||','||ID2)
```

16.4.7.5.4 Experiencing URL Checksum Protection

Witness the URL tampering protection for image serving pages using a page item for the image ID.

As shown below, the break room Interactive Grid using the checksum-protected URLs is identical for the end user:

Figure 16-37 End Users Notice No Changes Using Checksum-Protected Image URLs



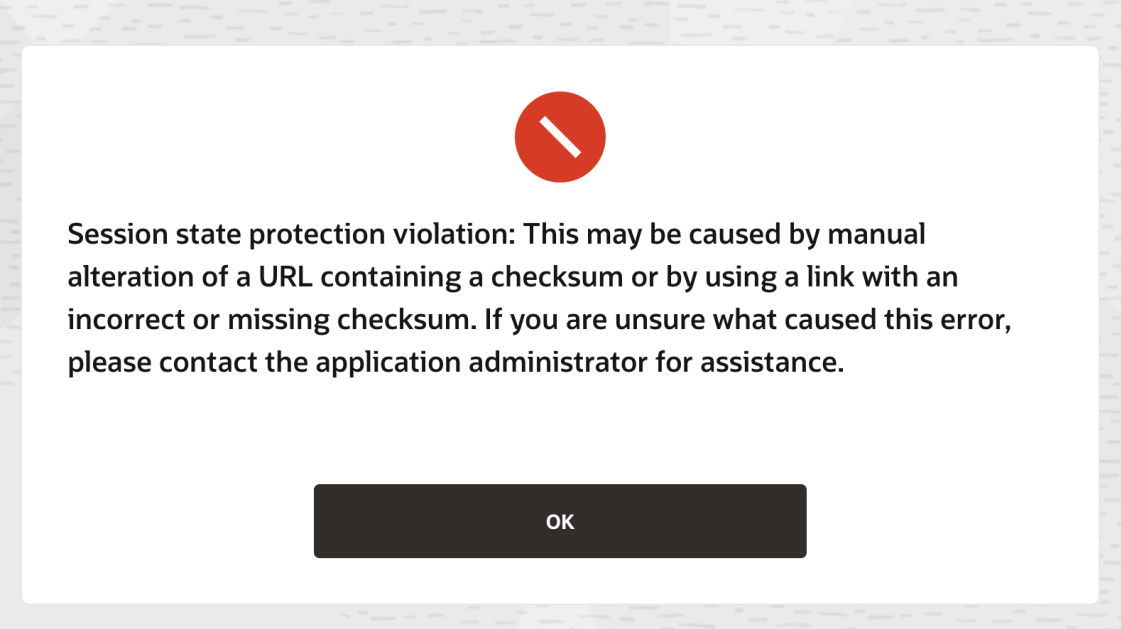
However, now every image URL ends with the a long `cs` checksum parameter whose impossible-to-guess value differs for each image URL produced:

```
/woods-hr/get-breakroom-image?p9005_id=13&session=304573&cs=1kSRFyrUoqRt
```


Copyright © 2026, Oracle and/or its affiliates.

A user who copies the URL for image ID 13 and tries to reuse it for a different image ID value receives the following error page. The figure shows the error dialog they see instead of the image they are not permitted to view.

Figure 16-38 Clever Users See Error When Manually Modifying an Image URL



16.4.8 Using BLOB Column Image URLs Anywhere

While the Interactive Grid *requires* it, use your preferred image URL generation approach anywhere that supports an image URL.

For example, the *Static Content* region has a convenient *Image* template with numerous declarative options for image aspect ratio, scale, shape, and filter. In a *Pre-Rendering* computation, compute the value of the image URL with a SQL query like the following.

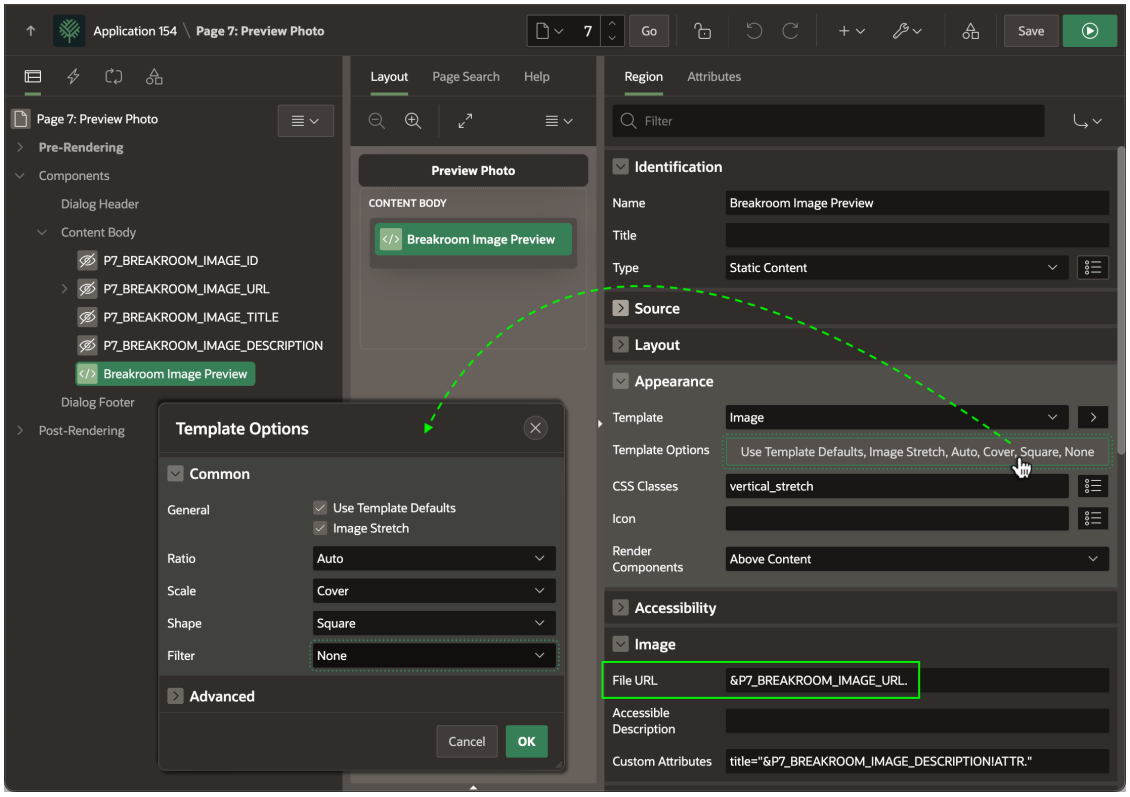
```
select apex_page.get_url(
            p_page      => 9000,
            p_x01       => id,
            p_plain_url => true)
       end as image_url
from eba_demo_emp_breakroom_photo_v
where id = :P7_BREAKROOM_IMAGE_ID
```

> **Tip**
>
> Pass `true` for `p_plain_url` when using `GET_URL` to generate *image* URLs for a modal dialog or drawer. It avoids including JavaScript code to close the dialog.

Then, you can reference the URL using `&P7_BREAKROOM_IMAGE_URL.` in the region's *File URL* property as shown below. Click the *Template Options* button to configure the declarative image formatting settings.

Copyright © 2026, Oracle and/or its affiliates.

Figure 16-39 Using Computed Image URL in a Static Content Image Template



16.5 Uploading a File to a BLOB Column

Use a *File Upload* page item to let users provide a file to your app. When you use it as part of a *Form* region, the combination automates creating, editing, and deleting the row with the file BLOB column.

Configure the *Form* region source with the table or view name, then set the *File Upload* page item's source to the BLOB column. Other declarative settings control what end users see when uploading the file.

- Selecting an Open Job to Add Referral
  Upload a candidate's CV from a modal form linked to the selected open job.
- Passing Client-Side Data to Dialogs
  Construct appropriate dialog links to pass *dynamic* client-side values to a dialog, with or without server-generated checksums.
- Configuring File Upload to a BLOB Column
  Configure a File Upload item to store an uploaded file in a BLOB column with related metadata.
- Ensuring the Automatic DML Process
  Set up the automatic DML process and submit button with database action to insert the uploaded file row.
- Controlling File Upload Item Behavior
  Use the properties in the *Display* section to control your end users' experience when uploading a file.

Copyright © 2026, Oracle and/or its affiliates.

- Submitting a Job Referral CV
  Walk through the referral CV upload flow, from selecting a PDF to refreshing the calling page.

16.5.1 Selecting an Open Job to Add Referral

Upload a candidate's CV from a modal form linked to the selected open job.

Each uploaded PDF file in the *Woods HR* app links to a *particular* open job position. In the *Open Positions Referrals* page, the user first selects the job to which to their referral pertains. Then they click a *(Submit First Referral)* or *(Submit Referral)* button to open a modal form page. They use this modal dialog or drawer page to add a new referral and upload the PDF curriculum vitae (CV). The calling page passes in the dynamic selected job ID so the new referral links to the right job when saved.

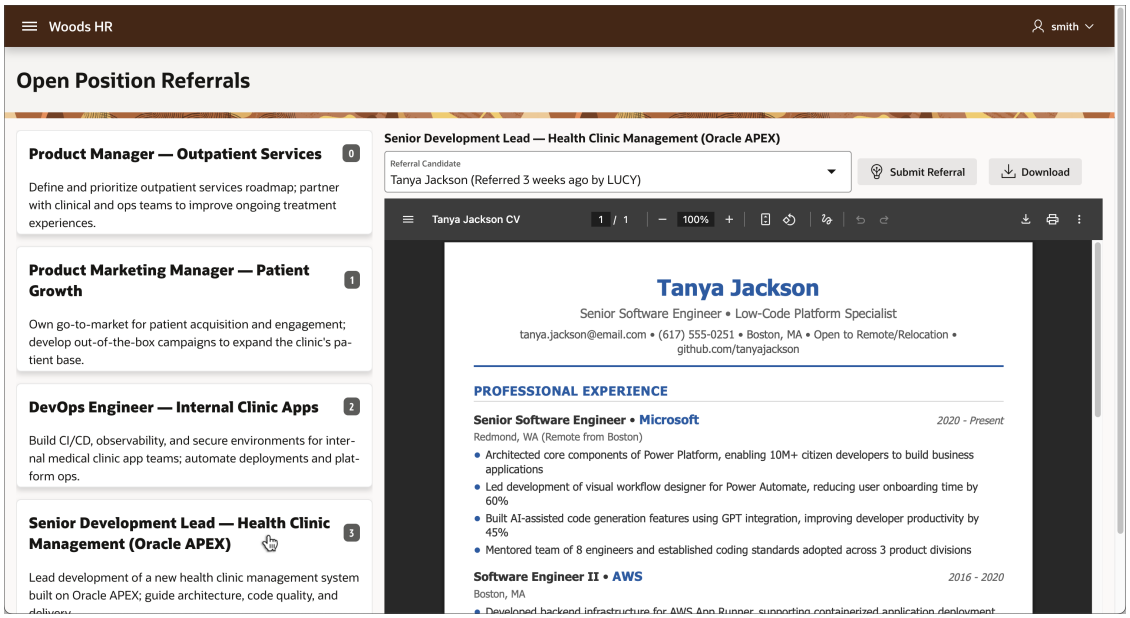
- Using Trigger Actions to Set Selected Job Info
  Use triggered actions to set the selected primary key and refresh other regions when the user clicks a card.
- Adjusting UI When Selected Job Changes
  React to a value change using a dynamic action event handler.

16.5.1.1 Using Trigger Actions to Set Selected Job Info

Use triggered actions to set the selected primary key and refresh other regions when the user clicks a card.

The open jobs appear on cards along the left side of the page as shown below. The small "badge" on each card reflects the count of existing referrals employees have submitted for that job. Clicking on a card sets the current job ID and updates the right side of the page based on how many referrals the selected job has. For example, the *Senior Development Lead* job has 3 referrals, so the right side updates to show the selected job title, a *Referral Candidate* Select List, a *(Submit Referral)* button, and a *(Download)* button. An inline preview of the selected referral candidate's CV also appears.

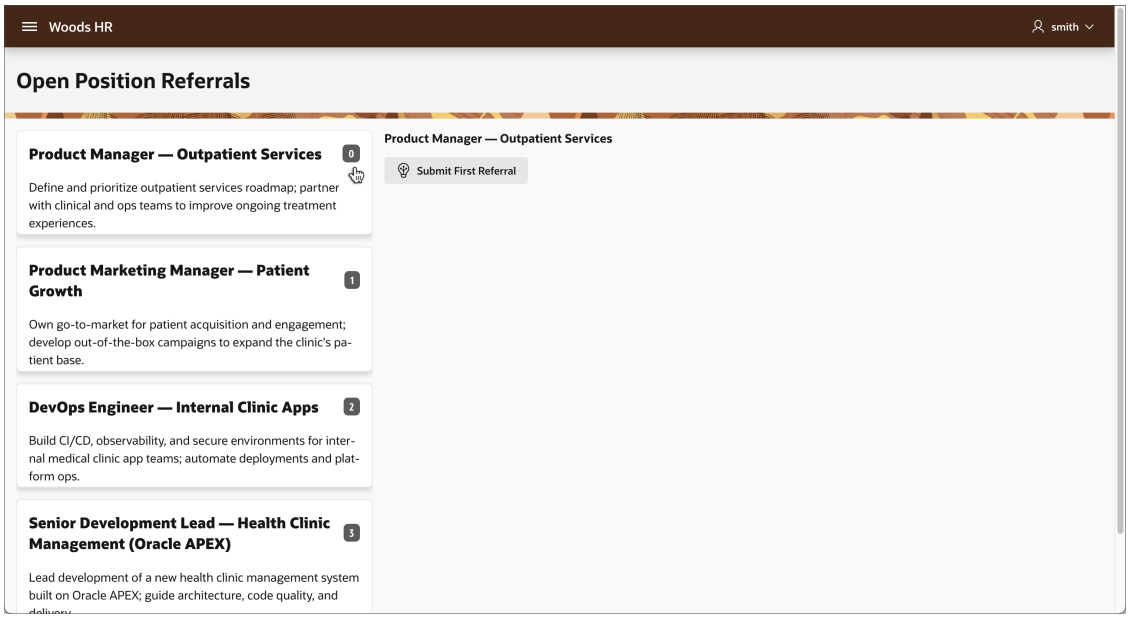
Figure 16-40 Selecting an Open Job Card with Existing Referrals



Copyright © 2026, Oracle and/or its affiliates.

When clicking on a job card like *Product Manager - Outpatient Services* with no referrals yet, the right side updates as seen below to show the selected job title and a *(Submit First Referral)* button.

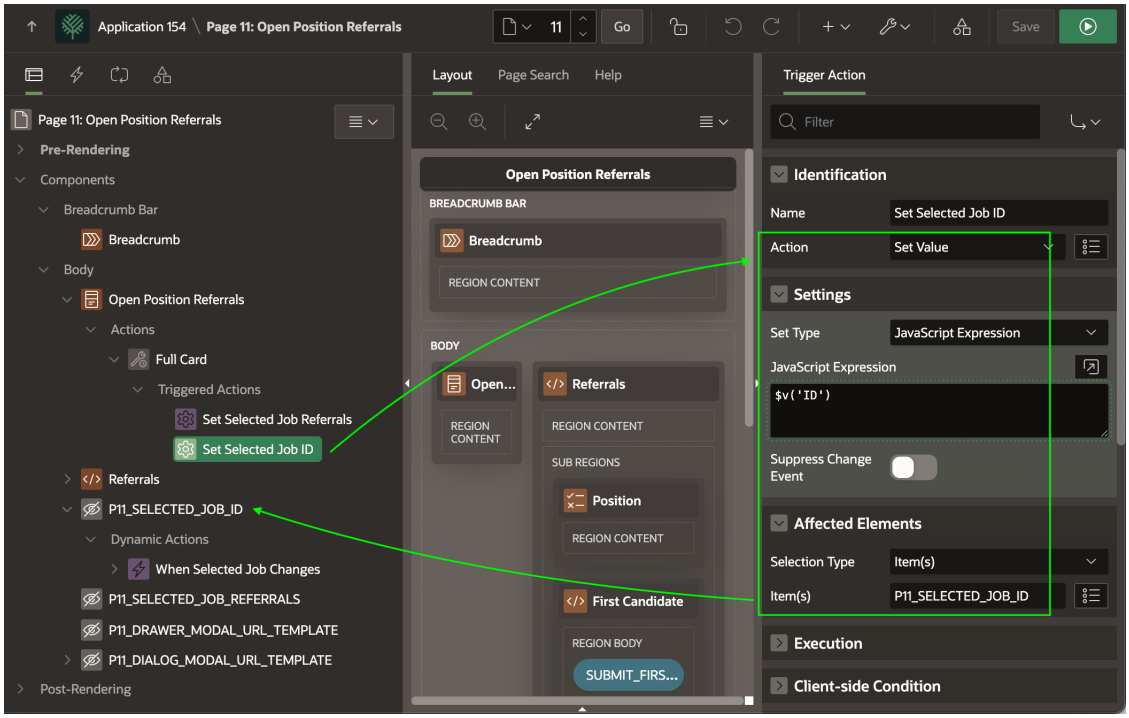
Figure 16-41 Selecting an Open Job with No Referrals Yet



Declarative *Trigger Actions* set the value of the selected job's referral count and ID. As shown below, the *Set Selected Job ID* action step of type *Set Value* assigns the hidden page item `P11_SELECTED_JOB_ID` to the ID of the selected job using the JavaScript expression `$v('ID')`. The companion *Set Selected Job Referrals* action step does the same to set the hidden page item `P11_SELECTED_JOB_REFERRALS` page item to `$v('REFERRALS')` column value.

Copyright © 2026, Oracle and/or its affiliates.

Figure 16-42 Full Card Action Sets Select Job Referral Count and ID



16.5.1.2 Adjusting UI When Selected Job Changes

React to a value change using a dynamic action event handler.

When the hidden `P11_SELECTED_JOB_ID` changes value, a *When Selected Job Changes* dynamic action handles that event. It evaluates its *Client-side Condition* highlighted below:

- *Type*: `Item is not null`
- *Item*: `P11_SELECTED_JOB_ID`

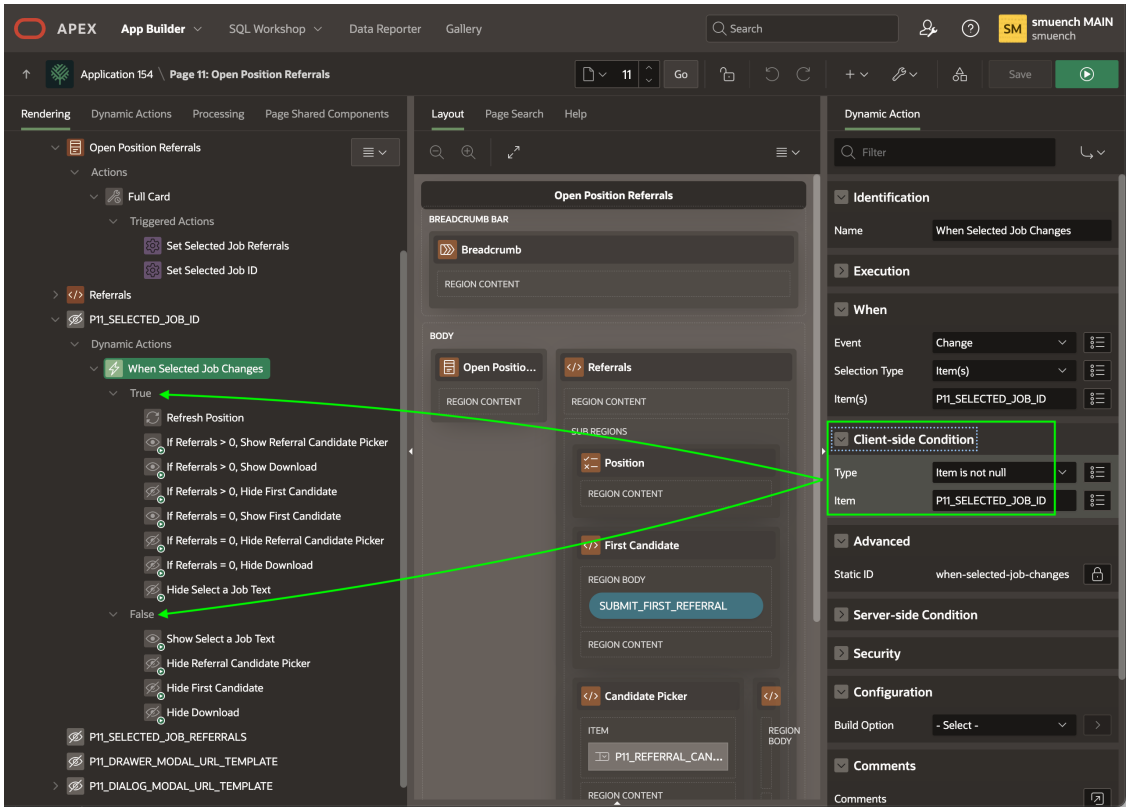
If `P11_SELECTED_JOB_ID` is not null, then the *True* list of action steps executes. If it *is* null, then the *False* list of action steps runs instead. These actions combine declarative *Refresh*, *Hide*, and *Show* action steps with appropriate client-side conditions based on the `P11_SELECTED_JOB_REFERRALS` to decide what to refresh, hide, and show on the right side of the page.

> **Tip**
>
> Assigning a descriptive name to each action step makes the logic easier to follow when you or a teammates needs to enhance this page in the future.

The figure shows the When Selected Job Changes dynamic action event handler selected in the component tree. It highlights how the true or false result of its configured **Client-side Condition** affects which set of action steps execute.

Copyright © 2026, Oracle and/or its affiliates.

Figure 16-43 Using True or False Dynamic Action Steps to Adjust the User Experience



16.5.2 Passing Client-Side Data to Dialogs

Construct appropriate dialog links to pass *dynamic* client-side values to a dialog, with or without server-generated checksums.

Pages can link *declaratively* to others when the passed parameter values don't change after rendering. But here the selected job ID is *dynamic*. The user changes the hidden selected job id page item by clicking on job cards.

Only the *server* can compute checksums, so the two options for generating a URL that passes dynamic *client* values to a dialog page are:

- Use `APEX_PAGE.GET_URL` in a trigger action to generate it on the server, or
- Update placeholder values in a dialog URL *template*, without the additional server round trip.

Avoiding server-side URL generation, however, requires disabling session state protection on the receiving dialog page items. In both cases, JavaScript routines in an app-wide `app.js` can simplify the job.

- Opening Server-Generated Dialog URL
  Open a dialog page with client-side values by passing a server-generated URL to a helper function.
- Passing Client Data Directly to Dialogs
  Open a dialog with dynamic client-side values by substituting page item placeholders in a URL template.

Copyright © 2026, Oracle and/or its affiliates.

16.5.2.1 Opening Server-Generated Dialog URL

Open a dialog page with client-side values by passing a server-generated URL to a helper function.

To open a modal dialog or drawer page passing in dynamic client data, use triggered actions to:

- Call `APEX_PAGE.GET_URL` to generate the URL on the server, then
- Pass it to an `app.openDialogURL()` or `app.openDrawerURL()` helper function.

- Generating Dialog URL on the Server
  Generate a checksum-protected dialog URL on the server before opening a modal page.
- Opening Dialog URL Passing Client Data
  Add JavaScript helper functions that simplify opening modal dialog and drawer pages from generated URLs.

16.5.2.1.1 Generating Dialog URL on the Server

Generate a checksum-protected dialog URL on the server before opening a modal page.

In the *Open Position Referrals* page, the user submits a new job referral by clicking either:

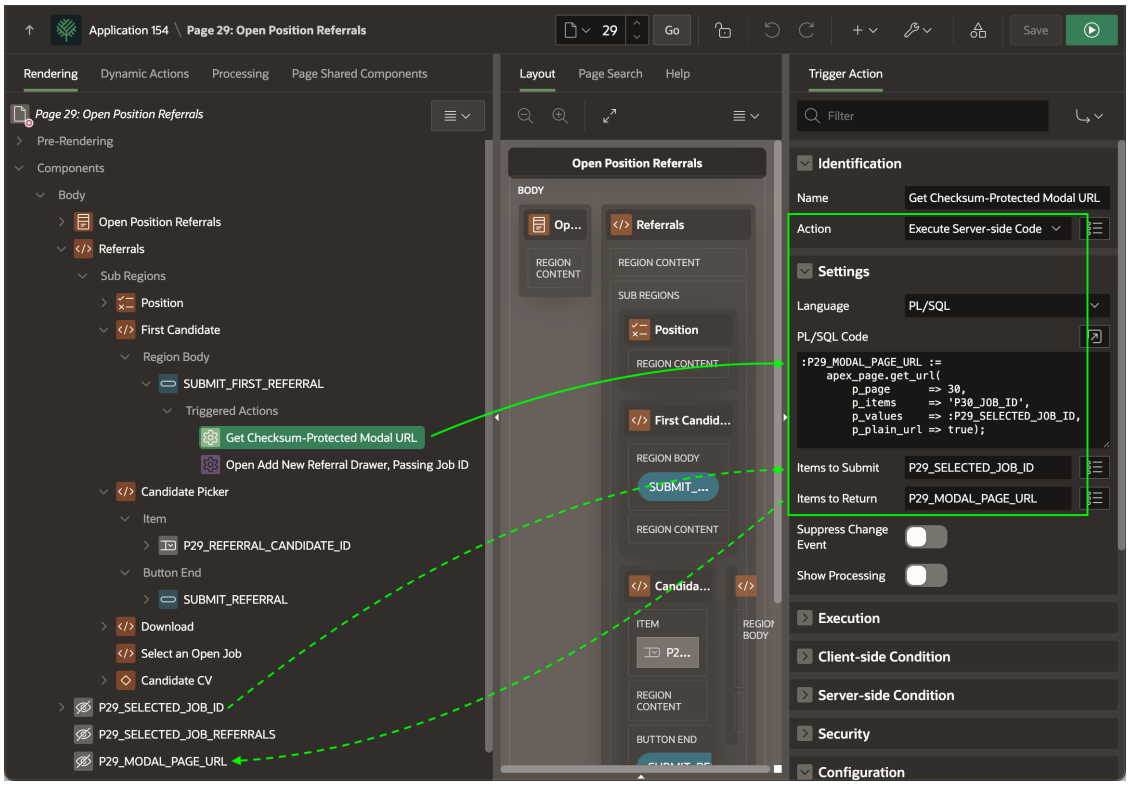
- *(Submit First Referral)* that opens a modal drawer, or
- *(Submit Referral)* that opens a modal dialog.

Both of these dialog pages accept the selected job id as a parameter into a hidden page item so the new job referral gets linked to the appropriate job.

Use a triggered action to generate the modal drawer URL on the server and store it in a hidden page item as shown below. Using *Execute Server-side Code*, the *Get Checksum-Protected Modal URL* triggered action on the `SUBMIT_FIRST_REFERRAL` button sets the value of `P29_MODAL_PAGE_URL` by calling `APEX_PAGE.GET_URL`. That generates the URL to the modal drawer page 30, passing in the value of `P29_SELECTED_JOB_ID` to the `P30_JOB_ID` hidden page item in the target. Notice the *Items to Submit* mentions the selected job ID page item name the PL/SQL references as a bind variable. The *Items to Return* lists the modal page URL item name.

Copyright © 2026, Oracle and/or its affiliates.

Figure 16-44 Triggered Action Generates Dialog URL Before Opening It Using JavaScript



A similar triggered action on the `SUBMIT_REFERRAL` button runs the code below to generate the modal dialog URL to open page 31, passing in the value of `P29_SELECTED_JOB_ID` to the `P31_JOB_ID` hidden page item in the target. Its *Items to Submit* and *Items to Return* properties are configured the same as above.

```
:P29_MODAL_PAGE_URL :=
    apex_page.get_url(
        p_page      => 31,
        p_items     => 'P31_JOB_ID',
        p_values    => :P29_SELECTED_JOB_ID,
        p_plain_url => true);
```

16.5.2.1.2 Opening Dialog URL Passing Client Data

Add JavaScript helper functions that simplify opening modal dialog and drawer pages from generated URLs.

Expanding on the technique explained in Providing App-Wide Helper Functions, you can add additional ones to the application's `app.js` JavaScript file in Static Application Files to simplify opening modal drawer and dialog pages by URL.

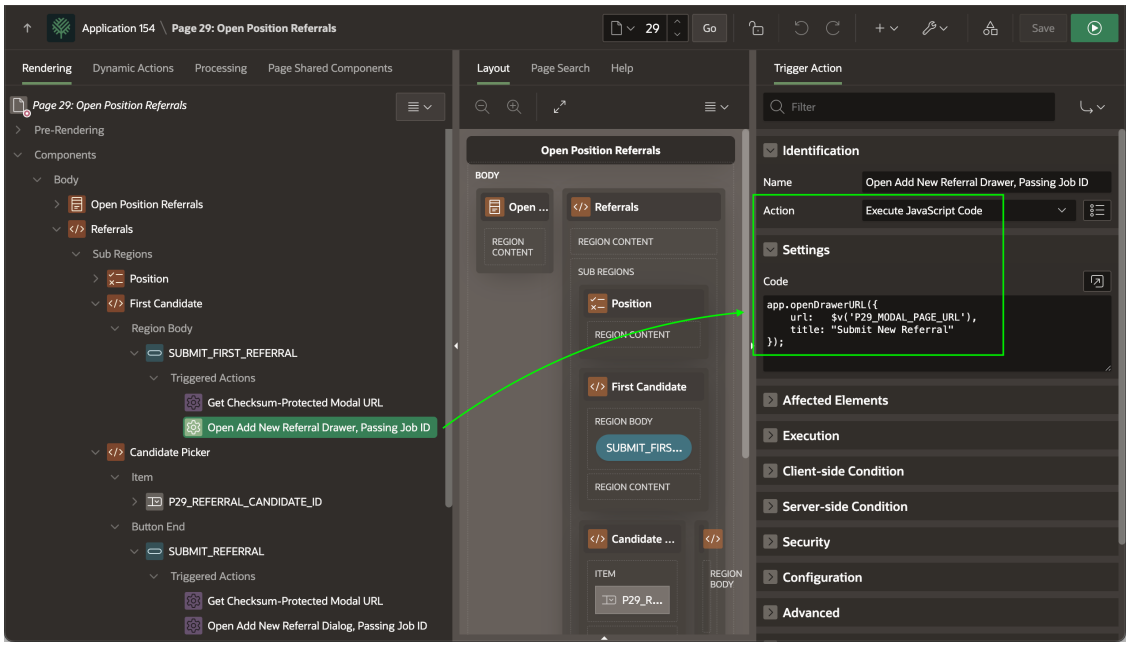
With the server-generated modal page URL in the hidden item `P29_MODAL_PAGE_URL`, as shown below a subsequent *Execute JavaScript Code* triggered action uses that item's value to open

Copyright © 2026, Oracle and/or its affiliates.

the drawer. It calls the `app.open`***`Drawer`***`URL()` function, passing in values for the `url` and `title` parameters by name:

```
app.openDrawerURL({
    url:   $v('P29_MODAL_PAGE_URL'),
    title: "Submit New Referral"
});
```

Figure 16-45 Calling JavaScript Helper Function to Open Dialog



Likewise, a triggered action on the `SUBMIT_REFERRAL` button uses a similar helper function to open a modal dialog. It calls the `app.open`***`Dialog`***`URL()` function, passing in values for the `url` and `title` parameters by name:

```
app.openDialogURL({
    url:   $v('P29_MODAL_PAGE_URL'),
    title: "Submit New Referral"
});
```

The source of these two helper functions appears below. They both call the standard APEX JavaScript API `apex.navigation.dialog` function, but let some common arguments like `width`, `maxWidth`, and `triggeringElement` get defaulted. This simplifies their use throughout your application.

Copyright © 2026, Oracle and/or its affiliates.

> **Tip**
>
> The `openDialogURL` and `openDrawerURL` functions use a handy JavaScript feature to accept "destructured" arguments. This lets you specify default values for different optional parameters. It also lets a caller pass parameters using a named notation. Default values assigned in the function get automatically used if the caller omits those optional parameters.

```
// In app.js static application file
//-----------------------------------------------------
// Open modal dialog
//-----------------------------------------------------
function openDialogURL({
    url,
    title,
    width = 400,
    maxWidth = 1000,
    triggeringElement = $('body')[0]
}) {
    apex.navigation.dialog(
    url,
    {
        title: title,
        height: "auto",
        width: width,
        maxWidth: maxWidth,
        modal: true,
        resizable: true
    },
    "t-Dialog-page--standard",
    triggeringElement
    );
}

//-----------------------------------------------------
// Open Drawer URL
//-----------------------------------------------------
function openDrawerURL({
    url,
    title,
    width = 400,
    triggeringElement = $('body')[0]
}) {
    apex.navigation.dialog(
    url,
    {
        title: title,
        width: width,
        modal: true
    },
    "t-Drawer-page--standard t-Drawer--pullOutEnd",
    triggeringElement
```


Copyright © 2026, Oracle and/or its affiliates.

```
    );
}
```

16.5.2.2 Passing Client Data Directly to Dialogs

Open a dialog with dynamic client-side values by substituting page item placeholders in a URL template.

To pass dynamic client data to a dialog *directly*, thus avoiding a server round trip:

- Use `APEX_PAGE.GET_URL` to default a dialog URL *template* with page item value *placeholders*
- Use a JavaScript helper function to substitute the page item placeholders and open the dialog
- Call the helper function in a triggered action to open the dialog, passing in the *template* URL
- Disable session state protection on the target dialog page item receiving the dynamic value.

- Defaulting URL Template Page Items
  Use `APEX_PAGE.GET_URL` to compute the *template* URL containing page item placeholders you replace with client values when opening the dialog.
- Substituting Client-Side Data in Dialog URL
- Opening Dialog, Passing Client-Side Data
  Open modal drawer and dialog pages by passing a URL template to JavaScript helper functions
- Unprotecting Dynamic Client Arguments
  Relax session state protection for dialog page items that receive dynamic client-modifiable values.

16.5.2.2.1 Defaulting URL Template Page Items

Use `APEX_PAGE.GET_URL` to compute the *template* URL containing page item placeholders you replace with client values when opening the dialog.

This variant of the *Open Position Referrals* page contains two hidden page items to store URL templates:

- `P11_DRAWER_MODAL_URL_TEMPLATE` – to open modal drawer page 23
- `P11_DIALOG_MODAL_URL_TEMPLATE` – to open modal dialog page 27

The expression to compute the value of the drawer modal URL is:

```
apex_page.get_url(
    p_page      => 23,
    p_items     => 'P23_JOB_ID',
    p_values    => '$P11_SELECTED_JOB_ID$',
    p_plain_url => true)
```

Notice, the value of the `P23_JOB_ID` parameter is set to the name of the `P11_SELECTED_JOB_ID` page item, surrounded by dollar signs. The JavaScript helper code looks for names of this pattern and replaces them in the template with the current value of the named page item.

Copyright © 2026, Oracle and/or its affiliates.

The expression for the dialog modal URL is the same, just using a different page number:

```
apex_page.get_url(
    p_page      => 27,
    p_items     => 'P27_JOB_ID',
    p_values    => '$P11_SELECTED_JOB_ID$',
    p_plain_url => true)
```

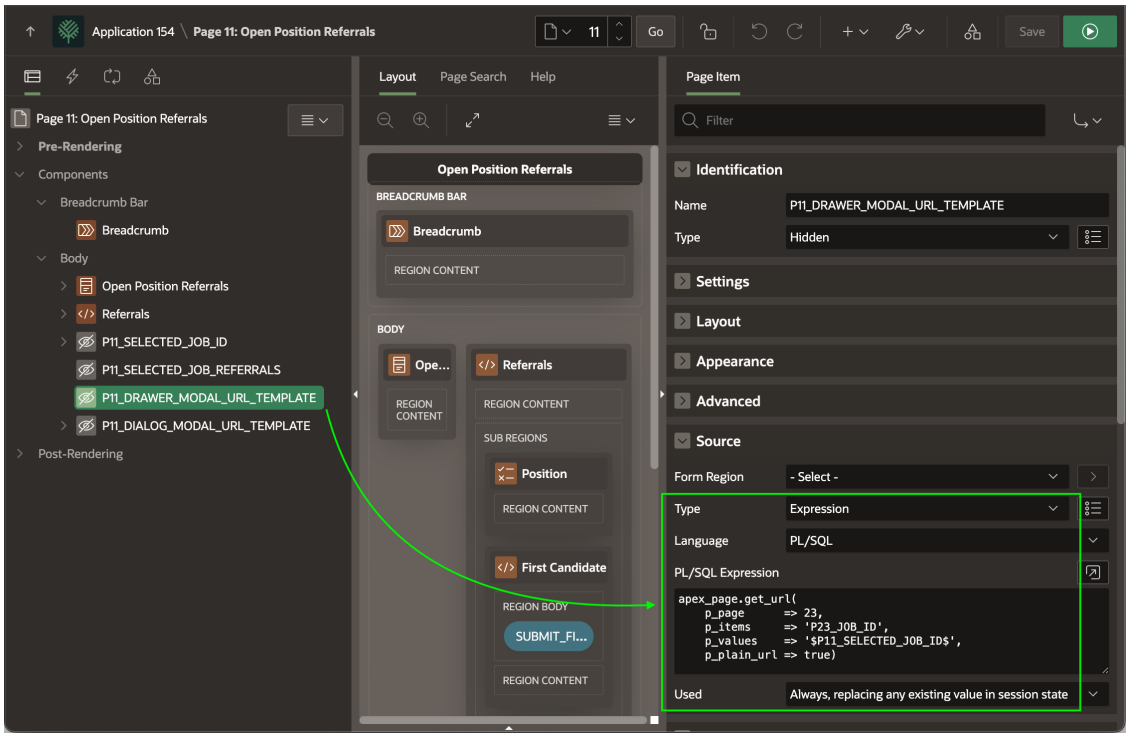
There are three ways to default the value of page item:

Table 16-1 Three Approaches to Defaulting a Page Item Value

Approach	Use When You Need
Default setting	An *initial* value if the page item is otherwise null.
Source setting	A computed value to *always* be used, when not associated to a form region.
Computation or page process assignment in the *Pre-Rendering* section	An optionally-conditional assignment visible at the "top" of the page rendering tree.

The *Open Position Referrals* page shows two of these three approaches. As shown below, the `P11_DRAWER_MODAL_URL_TEMPLATE`'s source value uses a PL/SQL Expression set to always be used. The figure shows the `P11_DRAWER_MODAL_URL_TEMPLATE` hidden page item selected in the component tree, and highlights the `APEX_PAGE.GET_URL` expression described above that generates its URL template value.

Figure 16-46 "Always" Source Defaulting the Drawer Model URL Template

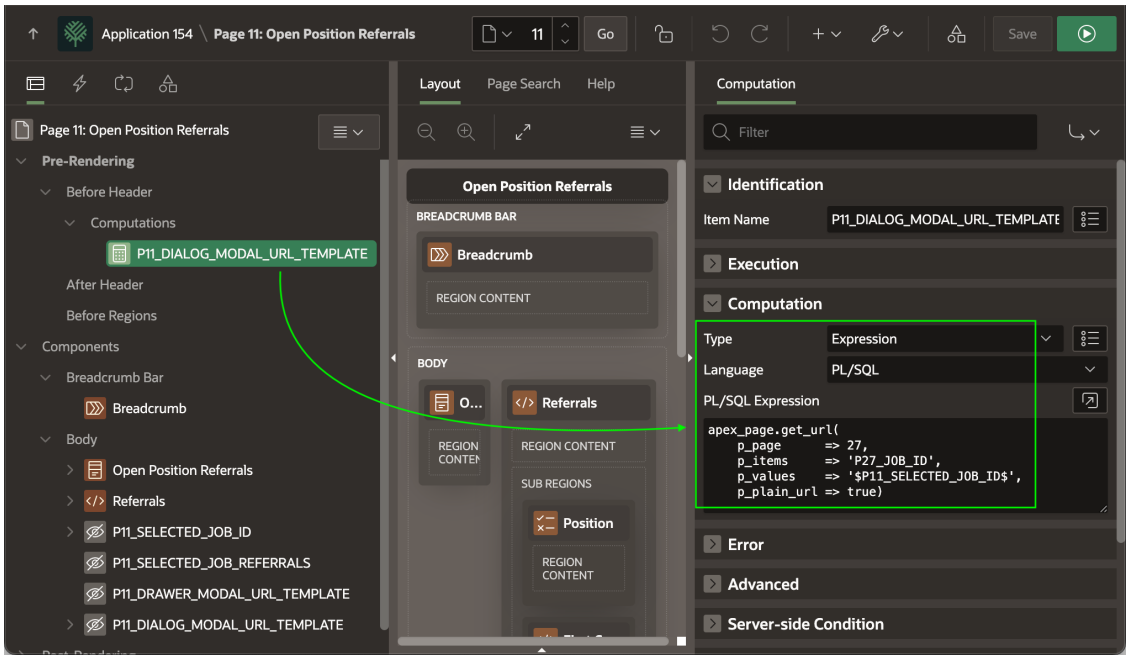


As an alternative, the `P11_DIALOG_MODAL_URL_TEMPLATE` uses a *Pre-Rendering* computation. The figure shows the `P11_DIALOG_MODEL_URL_TEMPLATE` computation selected in the

Copyright © 2026, Oracle and/or its affiliates.

component tree, and highlights the `APEX_PAGE.GET_URL` expression described above that generates its URL template value.

Figure 16-47 Computation Defaulting the Drawer Model URL Template



16.5.2.2.2 Substituting Client-Side Data in Dialog URL

Following the technique covered in Providing App-Wide Helper Functions, you can add additional ones to your application's `app.js` JavaScript file in Static Application Files to simplify substituting client-side data in dialog URLs.

The `makeDialogURL` function below replaces Unicode strings like `\u00e9` by their corresponding character and all occurrences of `$Pnn_SOMENAME$` with the value of the `Pnn_SOMENAME` page item. Since the page defaults the URL template string with `APEX_PAGE.GET_URL`, the dollar sign will appear in its "URL-encoded" form as `%24`. So, the regular expression that looks for the page item name placeholders uses this variant of the name delimiter:

```
function makeDialogURL(url) {
    // replace unicode escapes like \u00e9
    url = url.replace(/\\u([0-9a-fA-F]{4})/g, (_, h) =>
        String.fromCharCode(parseInt(h, 16))
    );

    // Replace %24Pnn_SOMENAME%24 with encodeURIComponent($v('Pnn_SOMENAME'))
    // Regex finds %24<identifier>%24 where identifier = letters, digits,
underscore
    url = url.replace(/%24([A-Za-z0-9_]+)%24/g, function (_, p1) {
        const val = $v(p1);
        return encodeURIComponent(val != null ? val : "");
    });
```


Copyright © 2026, Oracle and/or its affiliates.

```
    return url;
}
```

The `openDialogURLWithClientPageItemPlaceholders` function calls the private `makeDialogURL` function above, then opens the dialog using the `openDialogURL` function explained in Opening Dialog URL Passing Client Data.

> **Tip**
>
> The `openDialogURLWithClientPageItemPlaceholders` function uses a handy JavaScript feature to accept "destructured" arguments. This lets you specify default values for different optional parameters. It also lets a caller pass parameters using a named notation. Default values assigned in the function get automatically used if the caller omits those optional parameters.

```
// In app.js static application file
//------------------------------------------------------
// Substitute $PNN_NAME$ placeholders and open modal dialog
//------------------------------------------------------
function openDialogURLWithClientPageItemPlaceholders({
    urlTemplate,
    title,
    width = 400,
    maxWidth = 1000,
    triggeringElement = $('body')[0]
}) {
    openDialogURL({
        url : makeDialogURL(urlTemplate),
        title: title,
        height: height,
        width: width,
        maxWidth: maxWidth,
        triggeringElement : triggeringElement
    });
}
```

The `openDrawerURLWithClientPageItemPlaceholders` function is similar but calls `openDrawerURL` instead passing fewer parameters:

```
// In app.js static application file
//------------------------------------------------------
// Substitute $PNN_NAME$ placeholders and open drawer
//------------------------------------------------------
function openDrawerURLWithClientPageItemPlaceholders({
    urlTemplate,
    title,
    width = 400,
    triggeringElement = $('body')[0]
}) {
    openDrawerURL({
        url : makeDialogURL(urlTemplate);
        title: title,
```


Copyright © 2026, Oracle and/or its affiliates.

```
        width: width,
        triggeringElement : triggeringElement
    });
}
```

The full source of the *Woods HR* application's `app.js` file appears below. It's included in the *File URLs* setting on the *JavaScript* tab of the *Shared Components > User Interface Attributes* using its reference string `#APP_FILES#app#MIN#.js` from the *Static Application Files* list page.

```
// Export single global symbol "app" to keep code clean
const app = (function ($) {

    // --- Private Functions ---

    function makeDialogURL(url) {
        // replace unicode escapes like \u00e9
        url = url.replace(/\\u([0-9a-fA-F]{4})/g, (_, h) =>
            String.fromCharCode(parseInt(h, 16))
        );

        // Replace %24Pnn_SOMENAME%24 with
encodeURIComponent($v('Pnn_SOMENAME'))
        // Regex finds %24<identifier>%24 where identifier = letters, digits,
underscore
        url = url.replace(/%24([A-Za-z0-9_]+)%24/g, function (_, p1) {
            const val = $v(p1);
            return encodeURIComponent(val != null ? val : "");
        });

        return url;
    }

    // --- Public Functions (returned below!) ---

    //------------------------------------------------------
    // Set the title of the current dialog window
    //------------------------------------------------------
    function setDialogTitle(title) {
        apex.util.getTopApex()
        .jQuery(".ui-dialog-content")
        .dialog("option", "title", title);
    }

    //------------------------------------------------------
    // Return true if x and y are different, treating null
    // and empty string as equivalent in the comparison
    //------------------------------------------------------
    function areDifferent(x, y) {
        return (x || '') !== (y || '');
    }

    //------------------------------------------------------
    // Open modal dialog
    //------------------------------------------------------
    function openDialogURL({
```


Copyright © 2026, Oracle and/or its affiliates.

```
    url,
    title,
    width = 400,
    maxWidth = 1000,
    triggeringElement = $('body')[0]
}) {
    apex.navigation.dialog(
    url,
    {
        title: title,
        height: "auto",
        width: width,
        maxWidth: maxWidth,
        modal: true,
        resizable: true
    },
    "t-Dialog-page--standard",
    triggeringElement
    );
}

//-----------------------------------------------------
// Open Drawer URL
//-----------------------------------------------------
function openDrawerURL({
    url,
    title,
    width = 400,
    triggeringElement = $('body')[0]
}) {
    apex.navigation.dialog(
    url,
    {
        title: title,
        width: width,
        modal: true
    },
    "t-Drawer-page--standard t-Drawer--pullOutEnd",
    triggeringElement
    );
}

//-----------------------------------------------------
// Substitute $PNN_NAME$ placeholders and open modal dialog
//-----------------------------------------------------
function openDialogURLWithClientPageItemPlaceholders({
    urlTemplate,
    title,
    width = 400,
    maxWidth = 1000,
    triggeringElement = $('body')[0]
}) {
    openDialogURL({
        url : makeDialogURL(urlTemplate),
        title: title,
        height: height,
```


Copyright © 2026, Oracle and/or its affiliates.

```
            width: width,
            maxWidth: maxWidth,
            triggeringElement : triggeringElement
        });
    }

    //-----------------------------------------------------
    // Substitute $PNN_NAME$ placeholders and open drawer
    //-----------------------------------------------------
    function openDrawerURLWithClientPageItemPlaceholders({
        urlTemplate,
        title,
        width = 400,
        triggeringElement = $('body')[0]
    }) {
         openDrawerURL({
            url : makeDialogURL(urlTemplate),
            title: title,
            width: width,
            triggeringElement : triggeringElement
        });
    }

    // Only these functions are exposed to JavaScript code
    // in APEX pages; everything else stays private.
    return {
        setDialogTitle,
        areDifferent,
        openDialogURL,
        openDrawerURL,
        openDrawerURLWithClientPageItemPlaceholders,
        openDialogURLWithClientPageItemPlaceholders
    };

  // Ensure $ in app namespace resolves to correct jQuery
})(apex.jQuery);
```

16.5.2.2.3 Opening Dialog, Passing Client-Side Data

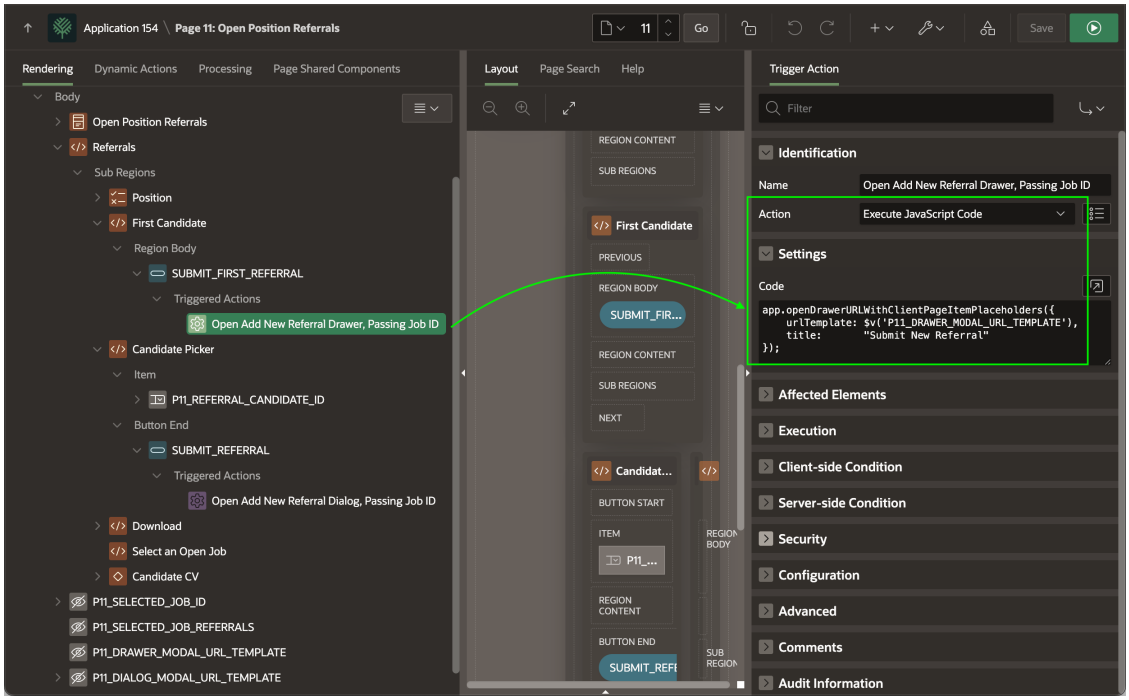
Open modal drawer and dialog pages by passing a URL template to JavaScript helper functions

As shown below, the *(Submit First Referral)* button opens the modal drawer page using the appropriate helper function from the `app` namespace defined in the application-wide `app.js` file. It passes in the value of the `urlTemplate` and `title` arguments by name, letting any other named arguments use their default valued specified in the function definition.

```
app.openDrawerURLWithClientPageItemPlaceholders({
    urlTemplate: $v('P11_DRAWER_MODAL_URL_TEMPLATE'),
    title:       "Submit New Referral"
});
```


Copyright © 2026, Oracle and/or its affiliates.

Figure 16-48 Opening Modal Drawer Passing Dynamic Values in URL Template



The *(Submit Referral)* button calls the other helper method for dialogs with the same two arguments:

```
app.openDialogURLWithClientPageItemPlaceholders({
    urlTemplate: $v('P11_DIALOG_MODAL_URL_TEMPLATE'),
    title:       "Submit New Referral"
});
```

16.5.2.2.4 Unprotecting Dynamic Client Arguments

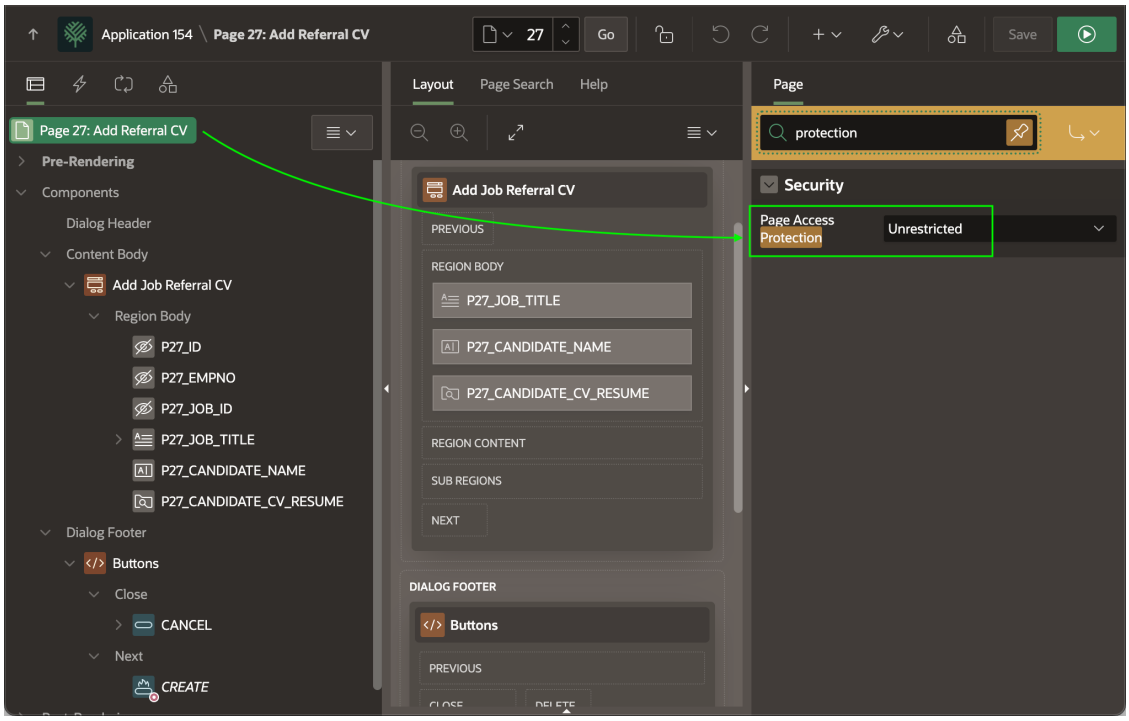
Relax session state protection for dialog page items that receive dynamic client-modifiable values.

Page Access Protection defaults to *Arguments Must Have Checksum*. This prevents URL-passed page item values from being altered. The checksum secures the values of any page items with *Session State Protection* enabled. Any attempt to modify URL parameter values results in a runtime error .However, client-modifiable values can't be checksum-protected without a server round trip. So, to open a modal dialog *directly* from the client, exclude dynamic arguments from the checksum.

Start by setting the page-level *Page Access Protection* to *Unrestricted* as shown below.

Copyright © 2026, Oracle and/or its affiliates.

Figure 16-49 Adjusting Page Access Protection to Unrestricted to Accept Dynamic Parameters

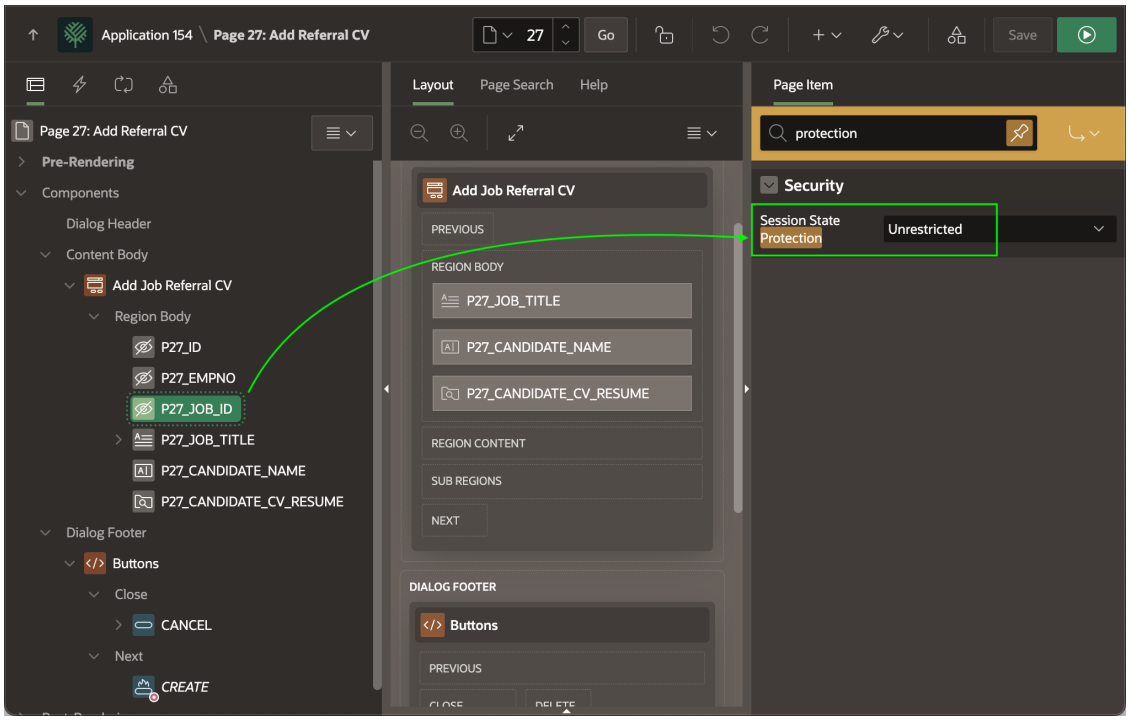


Then, select the page items receiving dynamic client-modifiable values as parameters and, as shown below, set their *Session State Protection* property to *Unrestricted* as well.

> **Tip**
>
> Other page items in the dialog page can retain their *Session State Protection* setting to require a checksum. Just not the ones receiving a dynamic, client-modifiable parameter from the calling page.

Copyright © 2026, Oracle and/or its affiliates.

Figure 16-50 Adjusting Session State Protection to Unrestricted to Accept Dynamic Parameters



16.5.3 Configuring File Upload to a BLOB Column

Configure a File Upload item to store an uploaded file in a BLOB column with related metadata.

After creating a form region on your table containing the BLOB file column, set the item type of the BLOB column to *File Upload*. Set the *Storage* > *Type* to *BLOB column specified in Item Source attribute*. Check the *Source* section in the *Property Editor* to ensure the expected BLOB column is selected. Finally, as shown below, configure the column names for the:

- *MIME Type Column* – the type of file (e.g. `application/pdf`)
- *Filename Column* – the uploaded file name
- *BLOB Last Updated Column* – date the file was last modified.

Setting up these additional column names is important. It ensures the browser handles the file correctly when viewed or downloaded.

> **Tip**
>
> Enable the *Primary Key* switch for form items tied to primary key columns. (e.g. `P23_ID` here)

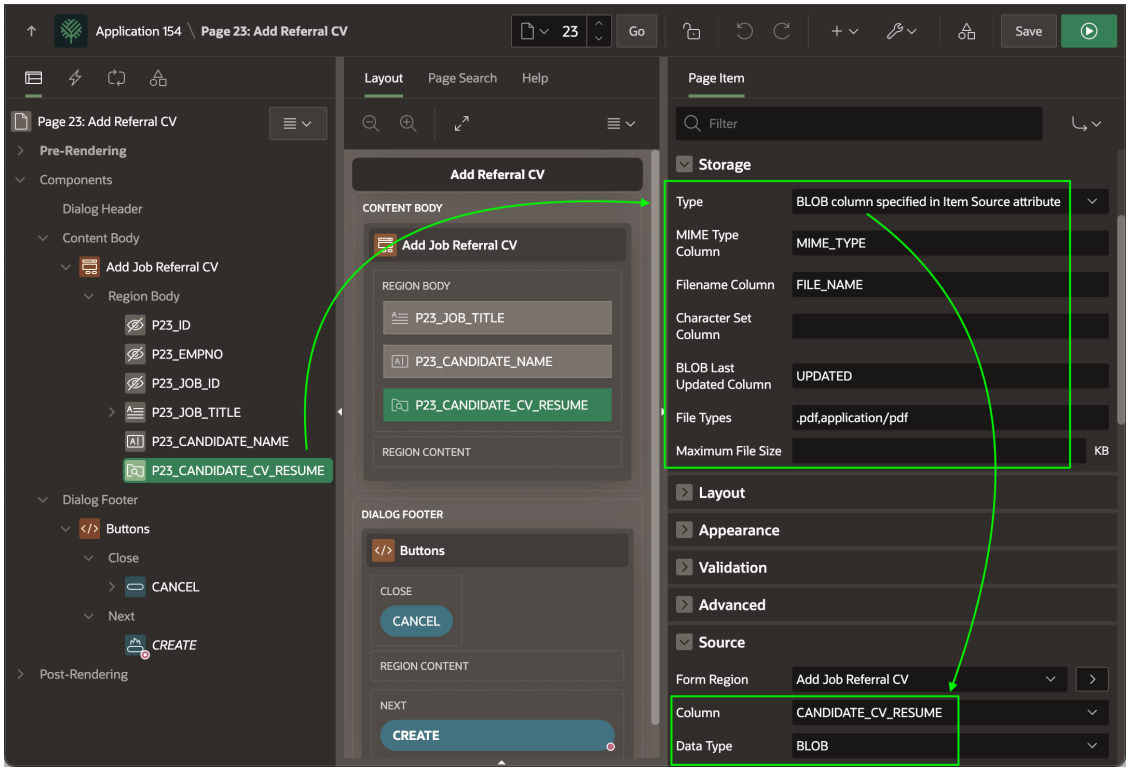
To allow only a subset of file types – for example, only PDF files – set the *File Types* field to a comma-separated list of file extensions and MIME type strings. For PDF files, the extension is `.pdf` and the MIME Type is `application/pdf`. These values inform the system file picker to help users to pick the kind or kinds of file your application needs.

Copyright © 2026, Oracle and/or its affiliates.

If you set a *Maximum File Size*, then the user sees a helpful validation error like the following. Then they can pick another file and try the upload again:

- *File is too large. Maximum file size is 3MB.*

Figure 16-51 Configuring File Upload Page Item



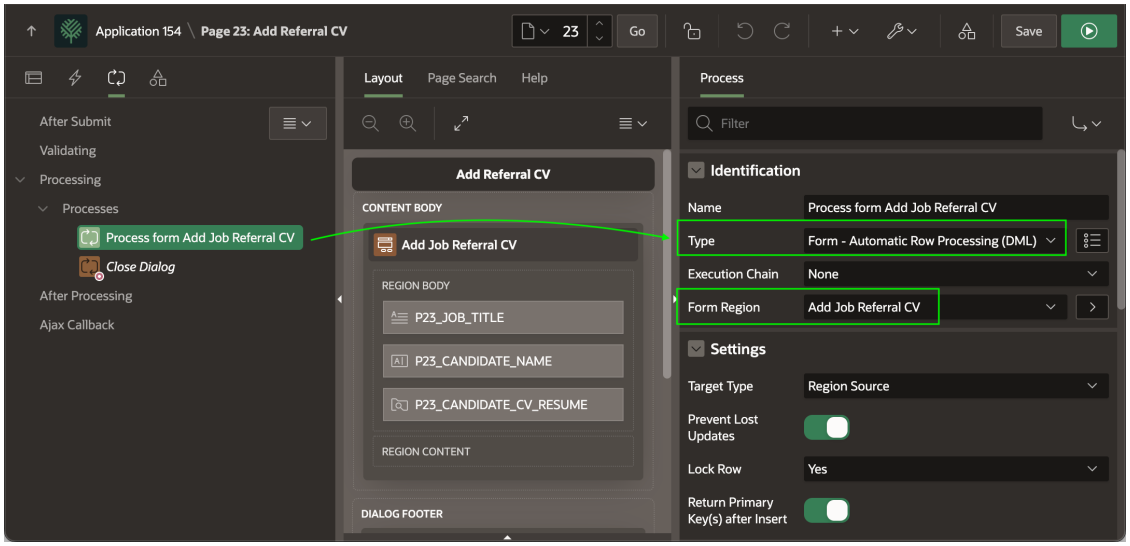
Then, in the *Processing* tab of the Page Designer tree, ensure you have:

- a page process of type *Form - Automatic Row Processing (DML)*
  - with its *Form Region* property set to the form region,
  - sequenced *before* the *Close Dialog* process if the page is a modal dialog or drawer, and
- a button like the `CREATE` one here with its *Database Action* property set to *SQL INSERT action*.

> **Tip**
>
> The Create Page Wizard creates the DML page process on a new *Form* page, and sets the *Database Action* property to *SQL INSERT action* on the `CREATE` button. However, when you add a Form region to an existing page you need to do these two steps yourself.

Copyright © 2026, Oracle and/or its affiliates.

Figure 16-52 Ensure Automatic DML Page Process Targets the Form



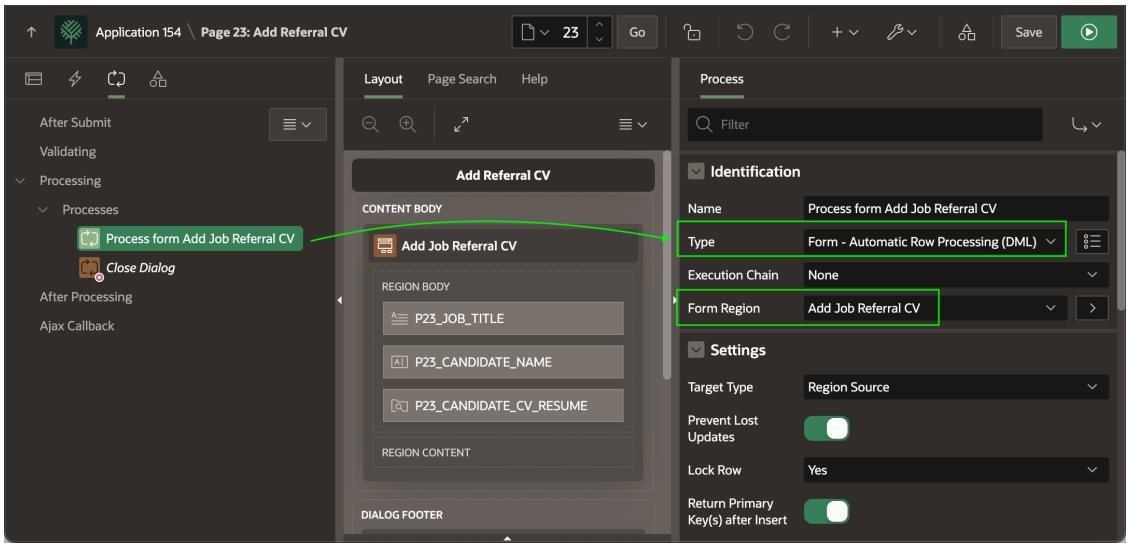
16.5.4 Ensuring the Automatic DML Process

Set up the automatic DML process and submit button with database action to insert the uploaded file row.

In the *Processing* tab of the Page Designer tree, ensure you have:

- a page process of type *Form - Automatic Row Processing (DML)*
  - with its *Form Region* property set to the form region, and
  - sequenced *before* the *Close Dialog* process if the page is a modal dialog or drawer

Figure 16-53 Ensure Automatic DML Page Process Targets the Form

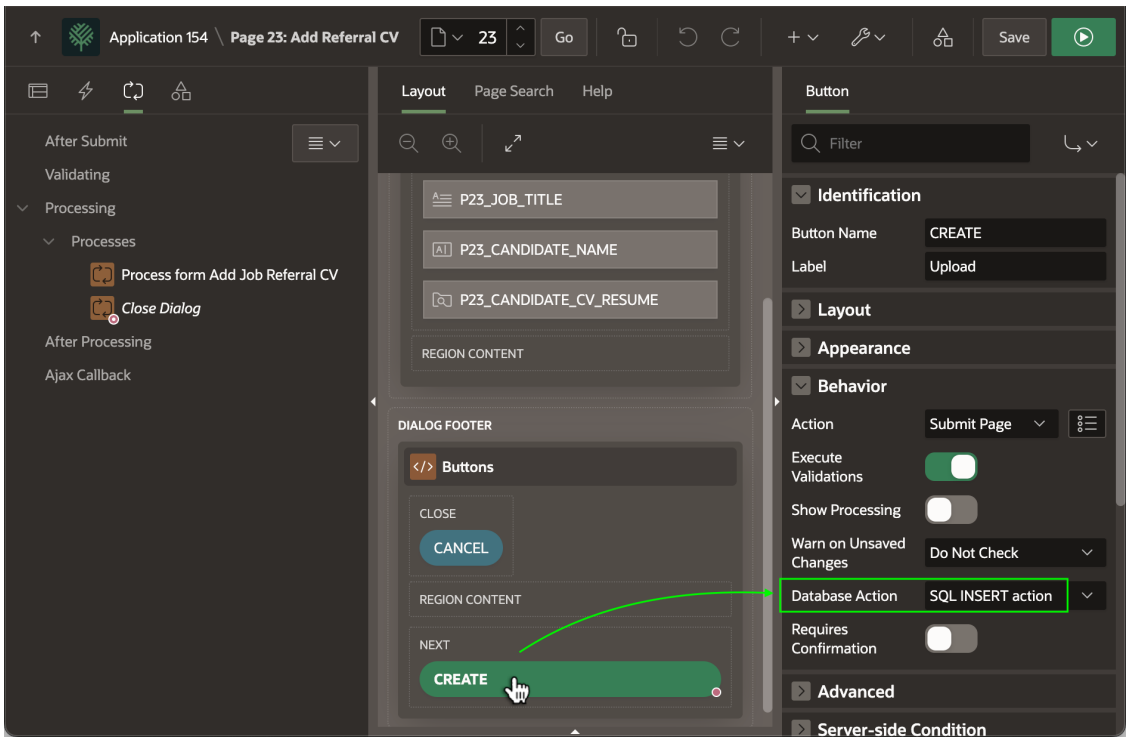


Copyright © 2026, Oracle and/or its affiliates.

Importantly, also check that the button that submits the page – like `CREATE` below – has its *Database Action* property set to *SQL INSERT action*. This informs the *Automatic Row Processing (DML)* page process to insert the submitted row.

> **Tip**
>
> The Create Page Wizard creates the DML page process on a new *Form* page, and sets the *Database Action* property to *SQL INSERT action* on the `CREATE` button. However, when you add a Form region to an existing page you need to do these two steps yourself.

Figure 16-54 Database Action on Submitting Button Informs Automatic DML Process



16.5.5 Controlling File Upload Item Behavior

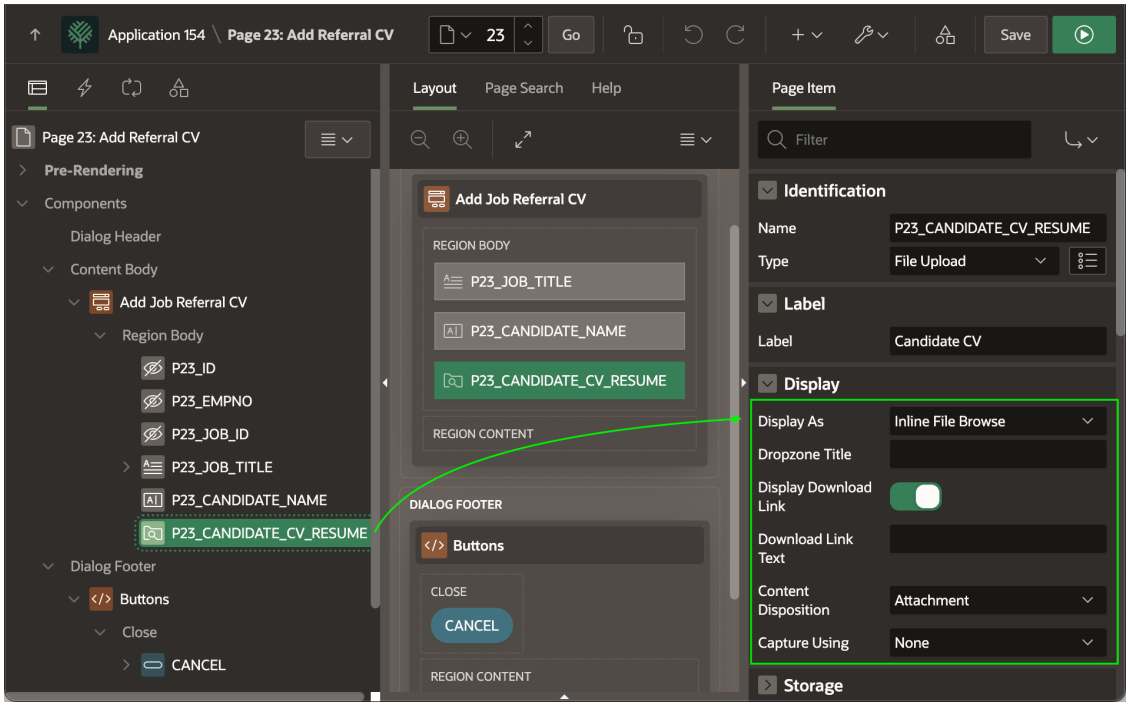
Use the properties in the *Display* section to control your end users' experience when uploading a file.

The *Display As* controls how the user sees the file upload item on the page. By default, it shows as an *Inline File Browse* but other styles featuring a drop zone are also available. The *Display Download Link*, *Download Link Text*, and *Content Disposition* settings define the download link the user can click to download the file when editing an existing row.

To let the user upload a "live" photo, set *Capture Using* to open a device's main or selfie camera. Users can still select an *existing* file instead. The figure shows the File Upload page item `P23_CANDIDATE_CV_RESUME` selected in the component tree, and highlights the **Display** group properties that control how user's see and interact with it.

Copyright © 2026, Oracle and/or its affiliates.

Figure 16-55 Configuring File Upload Page Item Display Behavior

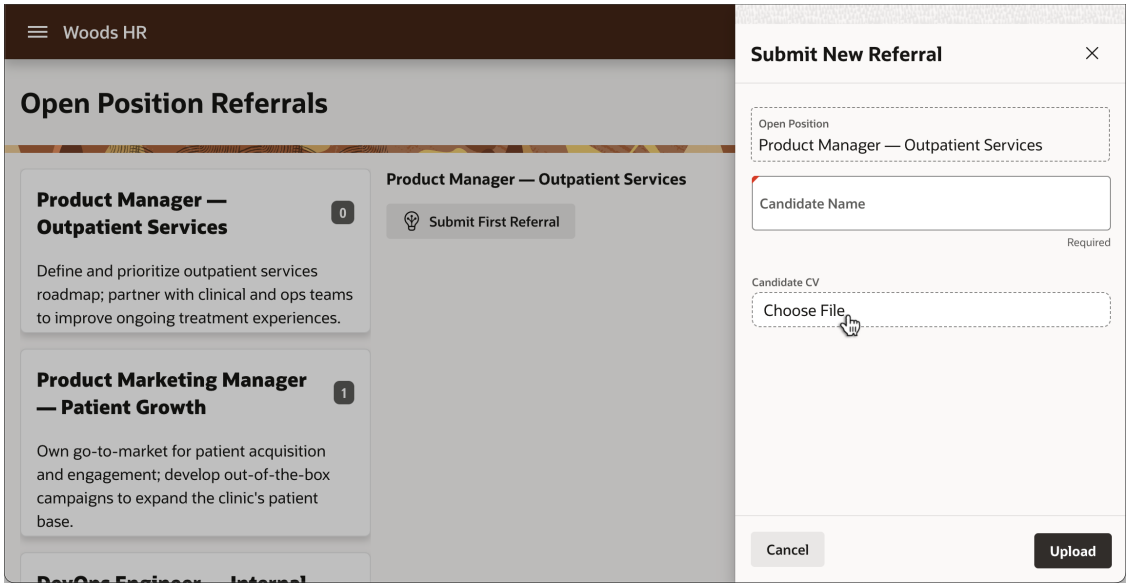


16.5.6 Submitting a Job Referral CV

Walk through the referral CV upload flow, from selecting a PDF to refreshing the calling page.

Now, when user *SMITH* selects the *Product Manager – Outpatient Services* job card and clicks *(Submit First Referral)* to enter a new one, she sees the modal dialog shown below to choose the PDF curriculum vitae (CV) and supply a candidate name.

Figure 16-56 Submitting a New Job Referral Candidate's CV

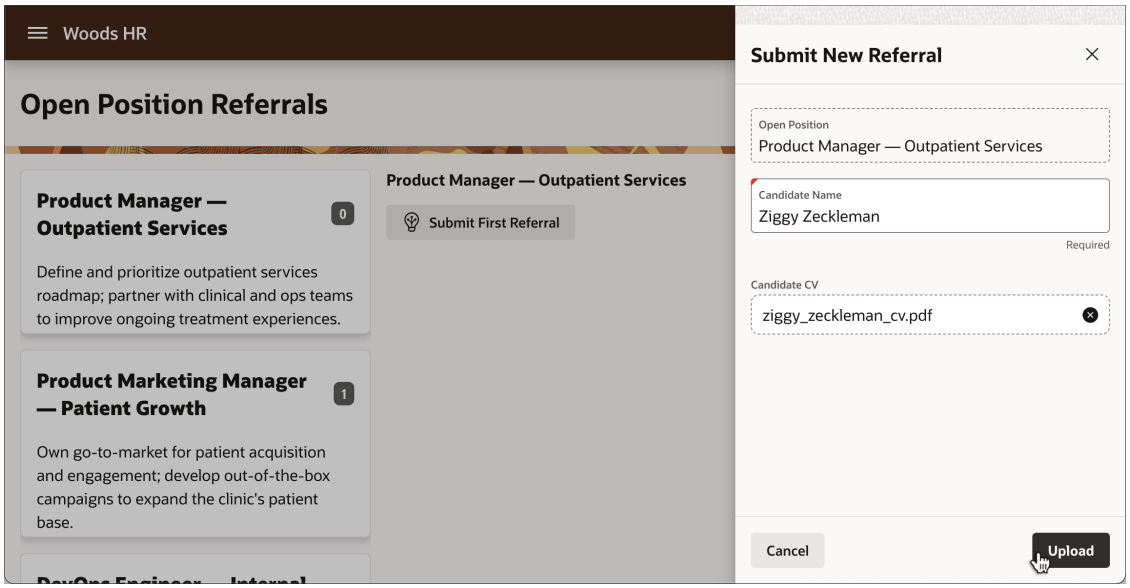


Copyright © 2026, Oracle and/or its affiliates.

When she clicks on the inline file browse area, the system file picker dialog appears, with only PDF files enabled for selection. After she picks a PDF file containing the candidate's CV and enters the candidate's name, the dialog reflects the file name to be uploaded as shown below.

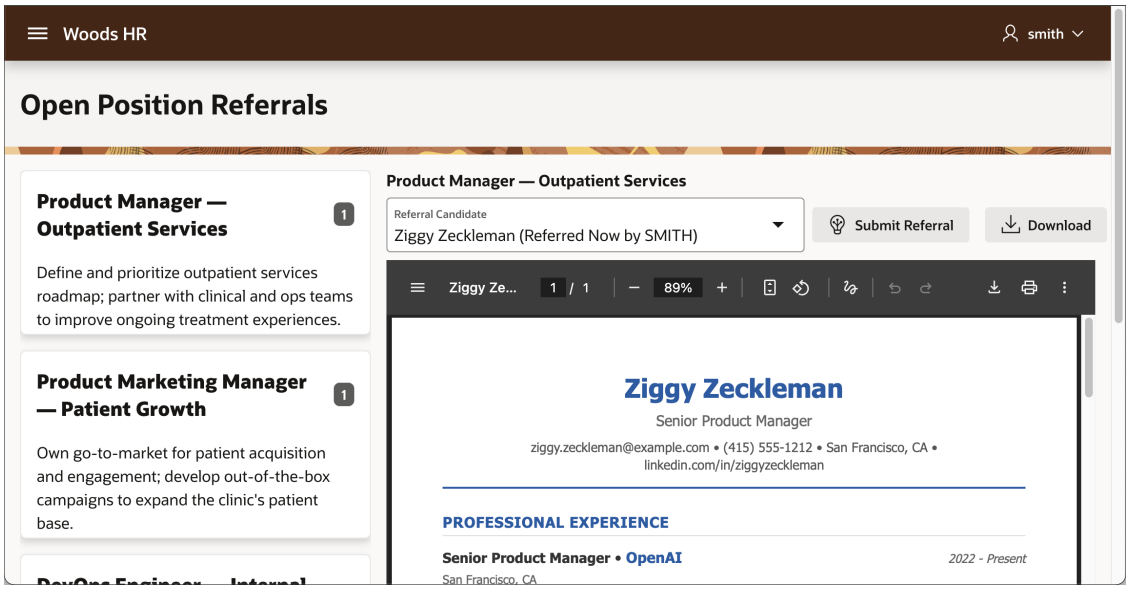
Finally, after clicking on the *(Upload)* button, the *Form – Automatic Row Processing (DML)* page process inserts the new row into the `EBA_DEMO_EMP_JOB_REFERRALS` table, including the PDF file into the `CANDIDATE_CV_RESUME` BLOB column and the *Close Dialog* process does its job.

Figure 16-57 Uploading a File to a BLOB Column



The dialog's closing triggers a *Dialog Closed* dynamic action that refreshes the *Referral Candidates* list and the *Candidate CV* PDF Preview region to immediately reflect the newly added candidate and his CV.

Figure 16-58 Previewing Newly Uploaded Job Referral Candidate's CV



Copyright © 2026, Oracle and/or its affiliates.

16.6 Previewing BLOB Column PDF Files Inline

To preview a PDF file stored in a BLOB column, use an `iframe`.

This HTML tag lets you include one page's contents as part of another page. To simplify setup, create a template component that lets developers just set the URL of the file to preview. Combine this with a page that downloads the PDF file inline. To finish the job, adjust your application's *Embed in Frames* browser security setting to allow embedding a frame from the same domain.

- Creating PDF Viewer Template Component
  Use a Template Component to simplify viewing PDF files inline.
- Serving an Inline PDF from a BLOB Column
  Use a page to serve inline PDF data.
- Using PDF Viewer in Job Referrals Page
  Use a PDF Viewer template component to display a PDF file inline in a page.
- Allowing Embedding from Same Domain
  Allow same-origin frames so the PDF Viewer can preview inline PDFs served by your app.

16.6.1 Creating PDF Viewer Template Component

Use a Template Component to simplify viewing PDF files inline.

As shown below, the *PDF Viewer* component consists of simple HTML markup. It uses the `{if/}` Template Directive to conditionally include an `<iframe>` tag if the *PDF URL* custom attribute is set. The `iframe`'s `src` attribute gets its value from this *PDF URL* parameter, using the substitution `#PDF_URL#`.

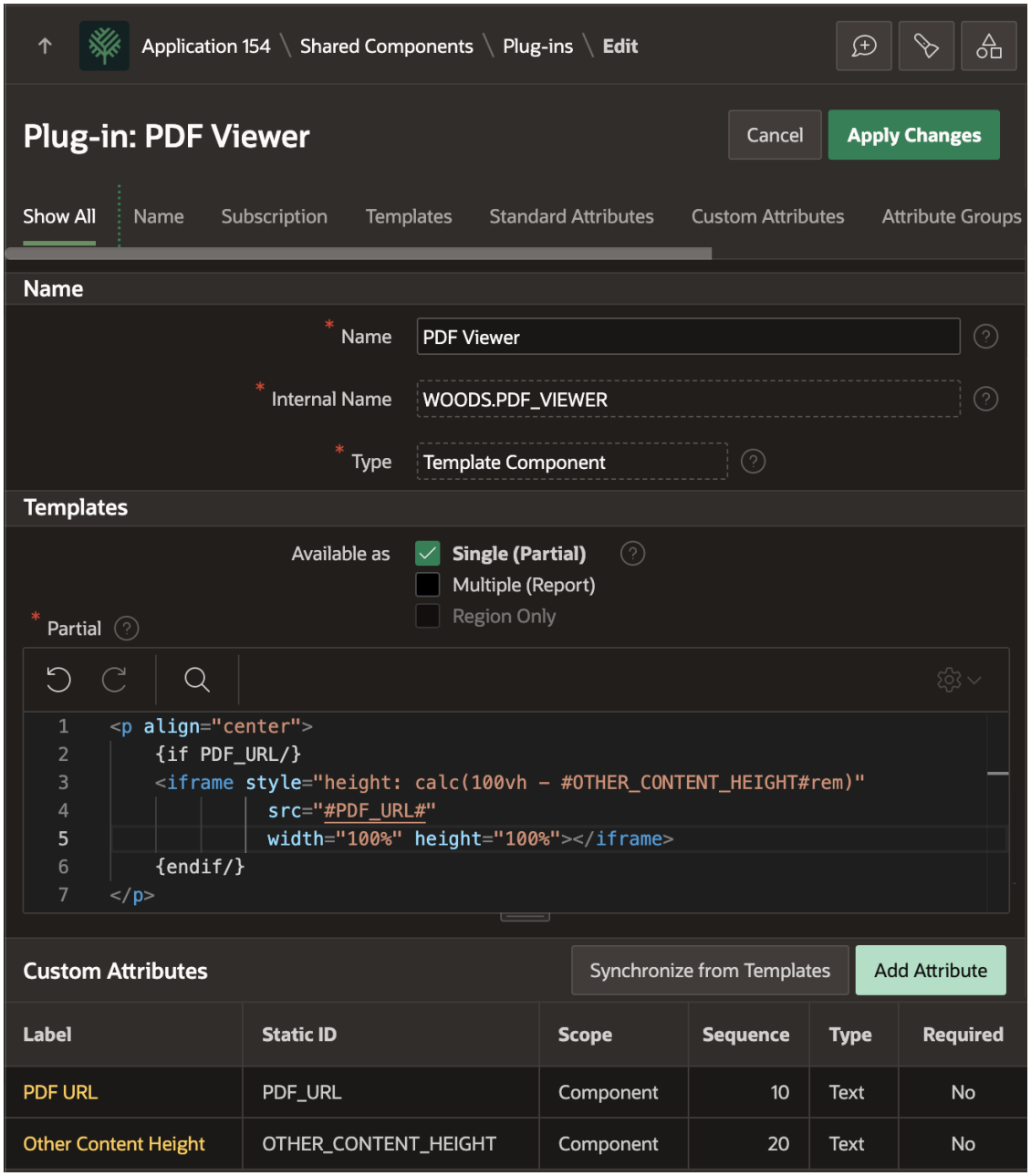
The developer can also set the *Other Content Height* attribute to vertically stretch the PDF-viewing frame. Its height will be the full vertical height of the viewport (`100vh`) minus the other content height (in `rem` units).

```
<p align="center">
    {if PDF_URL/}
    <iframe style="height: calc(100vh - #OTHER_CONTENT_HEIGHT#rem)"
              src="#PDF_URL#"
              width="100%" height="100%"></iframe>
    {endif/}
</p>
```

The *Available As* setting of *Single (Partial)* means the component expects to be used with a data source that retrieves a *single* row of data.

Copyright © 2026, Oracle and/or its affiliates.

Figure 16-59 Simple Template Component to More Easily View PDFs Inline



16.6.2 Serving an Inline PDF from a BLOB Column

Use a page to serve inline PDF data.

It calls `APEX_HTTP.DOWNLOAD` in a *Pre-Rendering* process in the *Before Header* section. For convenience, it can use the native `X01` parameter APEX defines. The following `DOWNLOAD_REFERRAL_CV_PDF` procedure retrieves the PDF file data and MIME type from the employee job referrals table using the referral ID. Then it downloads the file data. Notice it

Copyright © 2026, Oracle and/or its affiliates.

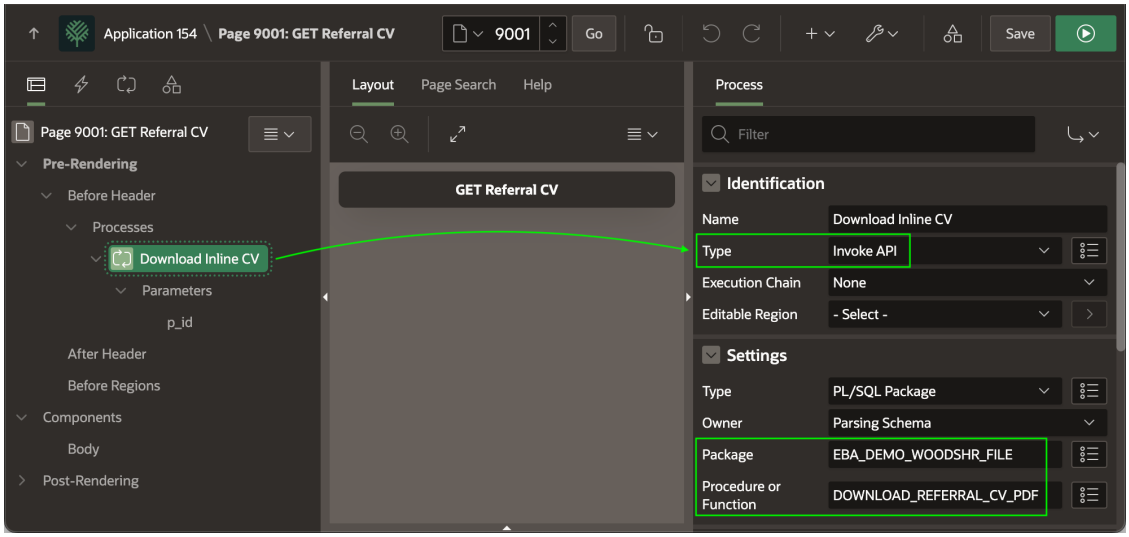
passes `true` to the `p_is_inline` parameter so the PDF is sent to the client in the way the browser can use as an inline PDF.

```
-- In package eba_demo_woodshr_file
procedure download_referral_cv_pdf(
    p_id in number)
is
    l_file_blob eba_demo_emp_job_referrals.candidate_cv_resume%type;
    l_mime_type eba_demo_emp_job_referrals.mime_type%type;
begin
    select candidate_cv_resume, mime_type
      into l_file_blob, l_mime_type
      from eba_demo_emp_job_referrals
     where id = p_id;

    apex_http.download(
        p_blob         => l_file_blob,
        p_content_type => l_mime_type,
        p_is_inline    => true );
end download_referral_cv_pdf;
```

Call this procedure with an Invoke API process in the *Before Header* section as shown below. The figure shows the Download Inline CV page process and highlights its **Type** of **Invoke API**, **Package** name of `EBA_DEMO_WOODSHR_FILE`, and procedure name of `DOWNLOAD_REFERRAL_CV_PDF`.

Figure 16-60 Pre-Rendering Invoke API Process to Download a PDF

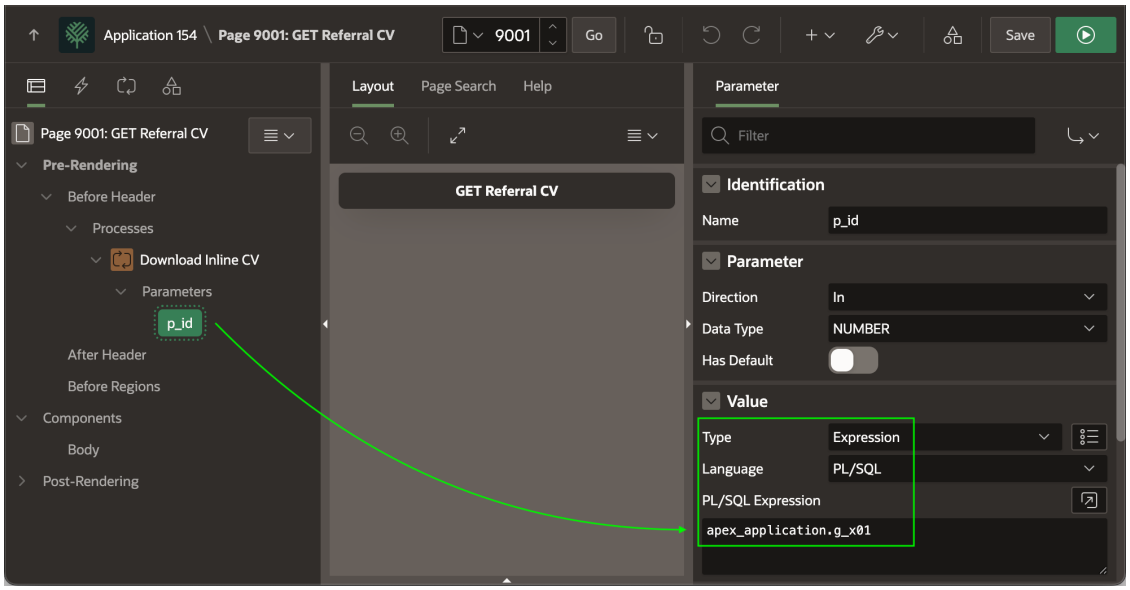


Configure the value of its `p_id` parameter using the PL/SQL expression `apex_application.g_x01` as shown below.

Copyright © 2026, Oracle and/or its affiliates.

> **⚠ Caution**
>
> If any authenticated user can view any PDF, as in this Woods HR example, using the convenient `X01` parameter is fine. However, to guarantee a generated URL referencing a particular PDF BLOB file cannot be manipulated to view a different one, use a checksum-protected hidden page item instead to pass the file ID to download.

Figure 16-61 Configuring Break Room Referral ID Parameter Value



16.6.3 Using PDF Viewer in Job Referrals Page

Use a PDF Viewer template component to display a PDF file inline in a page.

The *Open Position Referrals* page shown in Page Designer below includes a *PDF Viewer* template component. It's available only as a *Single (Partial)*, so it shows a single row of data. Accordingly, the *Source* query below returns a single row with a single `PDF_PAGE_URL` column.

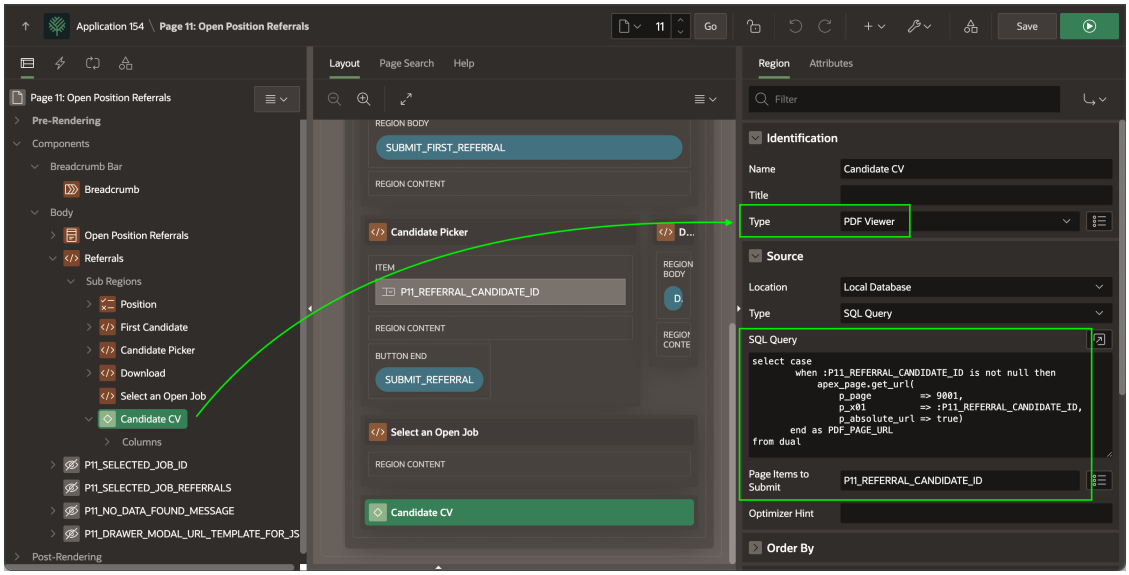
The `CASE` statement uses `APEX_PAGE.GET_URL` to generate the URL to page 9001. It serves an inline PDF from the BLOB column in the employee job referrals table, based on the referral ID passed in the `X01` parameter. Since it references the `P11_REFERRAL_CANDIDATE_ID` hidden page item as a bind variable, the *Page Items to Submit* property reflects that. This ensures the latest value of that item is always sent to the server when the region gets refreshed.

```
select case
        when :P11_REFERRAL_CANDIDATE_ID is not null then
            apex_page.get_url(
                p_page         => 9001,
                p_x01          => :P11_REFERRAL_CANDIDATE_ID,
                p_absolute_url => true)
       end as PDF_PAGE_URL
from dual
```


Copyright © 2026, Oracle and/or its affiliates.

> **Tip**
>
> An `iframe` is separate document inside another page that includes it. Any *relative* URLs it contains resolve against its *own* location, rather than the containing parent page's. This explains why the `PDF_PAGE_URL` value here must be an *absolute* URL to work correctly. You request one by passing `true` for the `p_absolute_url` parameter to `APEX_PAGE.GET_URL`.

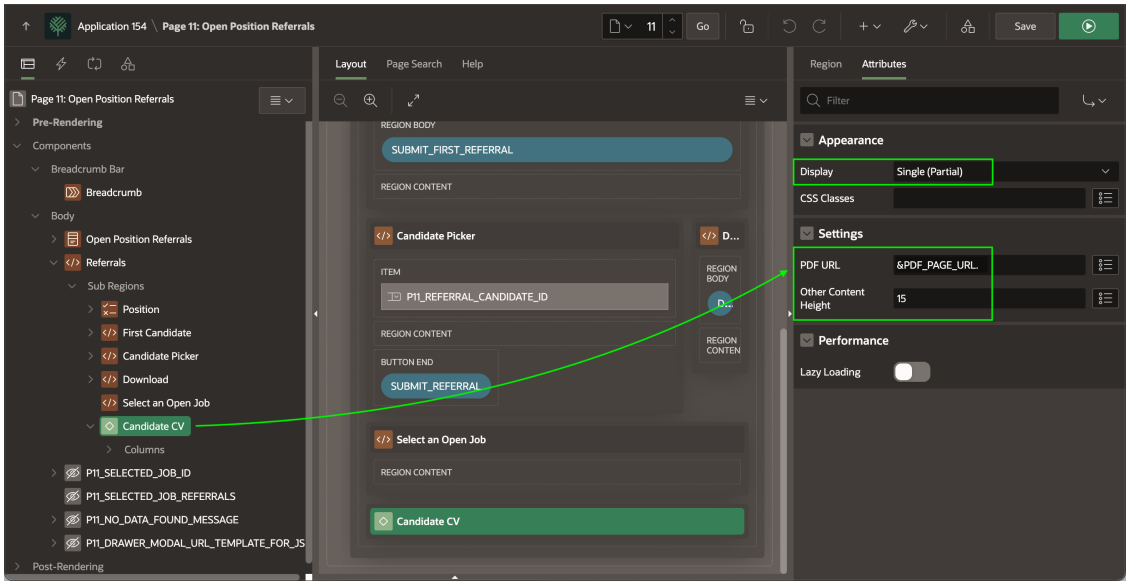
Figure 16-62 Sourcing the PDF_URL in the Region's Query



On the *Attributes* tab in the Property Editor, notice *Appearance* > *Display* is set to *Single (Partial)*. The *PDF URL* property gets its value from the `PDF_PAGE_URL` column in the regions single-row query result using the `&PDF_PAGE_URL.` substitution. The *Other Content Height* is set to 15 to indicate that the other content on the page takes up about 15 lines of height in the base font size.

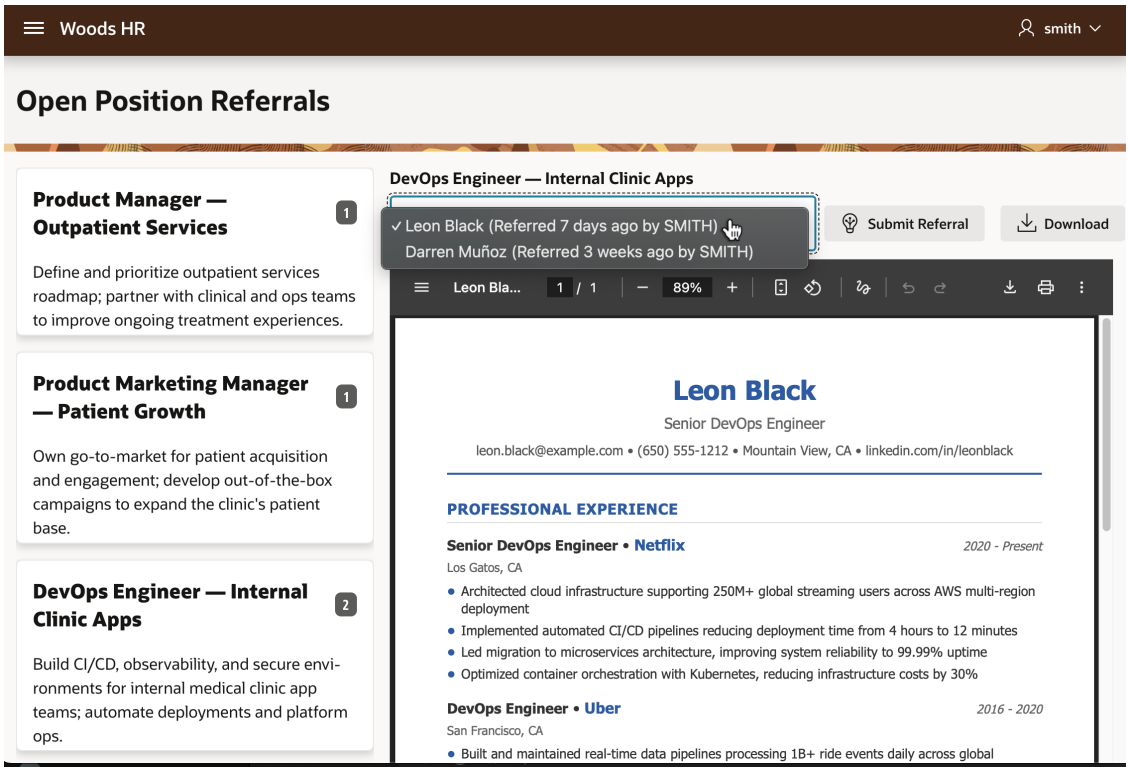
Copyright © 2026, Oracle and/or its affiliates.

Figure 16-63 Configuring PDF URL



Finally, a dynamic action on the value *Change* event of the `P11_REFERRAL_CANDIDATE_ID` candidate picker Select List page item refreshes the *Candidate CV* PDF Viewer region. This combination produces an interactive inline PDF viewer that updates to show the CV of the selected candidate as shown below.

Figure 16-64 Candidate Picker Select List Refreshes PDF Viewer Template Component



Copyright © 2026, Oracle and/or its affiliates.

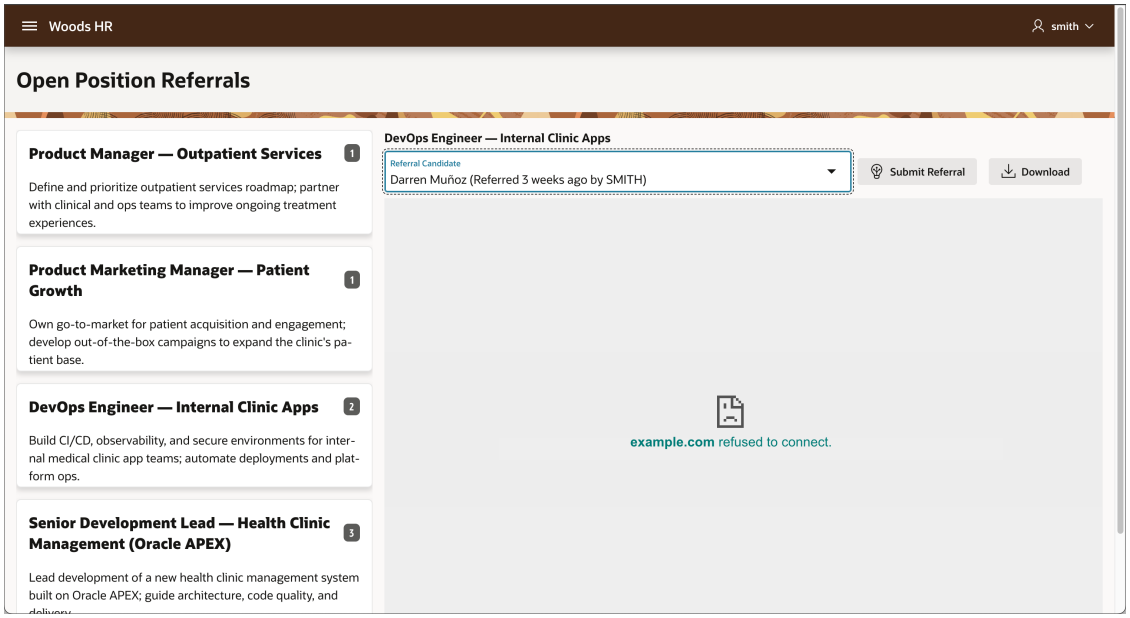
16.6.4 Allowing Embedding from Same Domain

Allow same-origin frames so the PDF Viewer can preview inline PDFs served by your app.

By default, your app disallows `iframe` content. This ensures you are always aware of any content your pages embed after ensuring it's from a trusted source. The Shared Components *Security* > *Browser Security* > *Embed in Frames* property lets you control this behavior.

Leaving this setting at its default of *Deny*, when the *PDF Viewer* tries to preview an inline PDF it encounters an error as shown below.

Figure 16-65 Encountering an Error Due to "Embed in Frames" = Deny



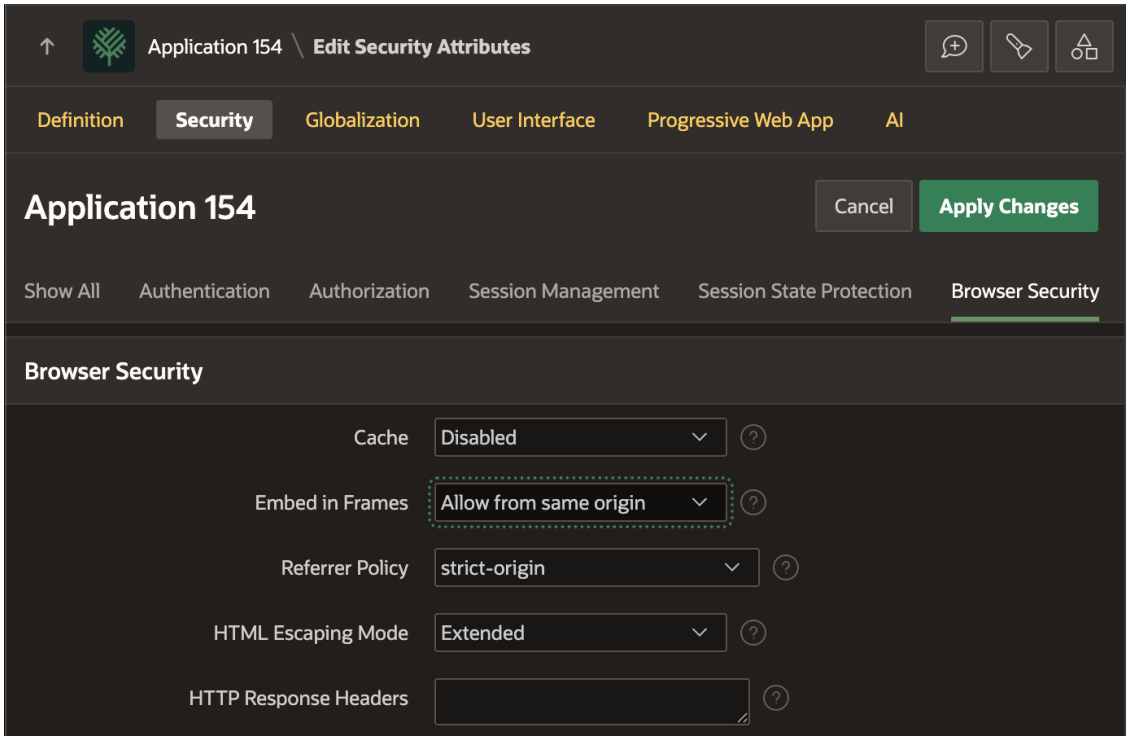
You can see additional information in your browser's developer tools console:

```
Refused to display 'https://example.com/' in a frame
because it set 'X-Frame-Options' to 'deny'.
```

The *PDF Viewer* template component includes the inline PDF BLOB content from a well-known table, and is served by a page that's part of your application. Therefore, it's safe to change the *Embed in Frames* property to *Allow from same origin* as shown below. This change makes the PDF preview work as expected.

Copyright © 2026, Oracle and/or its affiliates.

Figure 16-66 Allowing Embedded Frame Only from the Same Domain



16.7 Downloading a File for Offline Use

Use the native *Download* page process or dynamic action to let the user download a file for offline use.

Both perform the same functionality, and you configure them in a similar way. Both variants offer a single-file and a multiple-file option.

- Configuring Single-File Download Action
  Download a single file with a SQL query that returns the BLOB, file name, and MIME type.
- Downloading ZIP Archive of Multiple Files
  Download multiple files as a zip archive by returning one BLOB and file name per row.

16.7.1 Configuring Single-File Download Action

Download a single file with a SQL query that returns the BLOB, file name, and MIME type.

When downloading a single file using the *Download* dynamic action, provide a three-column SQL query as shown below that retrieves the BLOB file data, the file name, and the MIME Type of a single row. If the query references page item values as bind variables, make sure to include their names in the comma-separated *Items to Submit* list. The query retrieves the PDF BLOB containing the CV of the candidate whose ID is stored in the hidden page item `P11_REFERRAL_CANDIDATE_ID`. This is the Select List page item that picks the current candidate to review for the selected job.

```
select candidate_cv_resume,
       file_name,
       mime_type
```

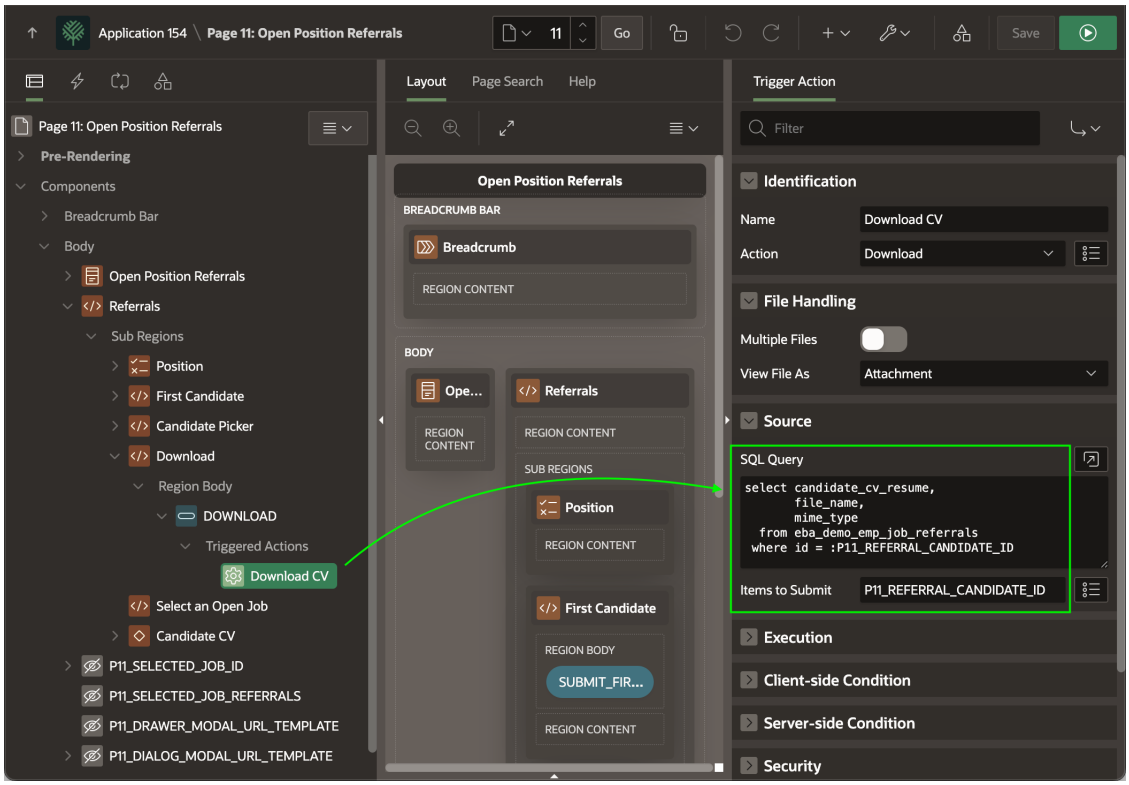

Copyright © 2026, Oracle and/or its affiliates.

```
  from eba_demo_emp_job_referrals
 where id = :P11_REFERRAL_CANDIDATE_ID
```

The figure shows the **Download CV** dynamic action selected in the component tree and highlights the SQL query that retrieves the BLOB content to download, its file name, and MIME type.

> **Tip**
>
> When using the *Download* page process, the SQL query is the same but there are no **Items to Submit** to configure since it happens on the server.

Figure 16-67 Downloading a Single File with the Native Dynamic Action



16.7.2 Downloading ZIP Archive of Multiple Files

Download multiple files as a zip archive by returning one BLOB and file name per row.

Multiple files get downloaded automatically in a zip archive. With a couple of small changes, you can change the *(Download)* button to instead download all CVs for the selected job instead. To do this, enable the *Multiple Files* switch, configure a name for the zip file, and adjust the query to remove the MIME Type column and retrieve multiple rows as shown below. The file name field can include substitutions like `&P11_SELECTED_JOB_ID.` as part of the file

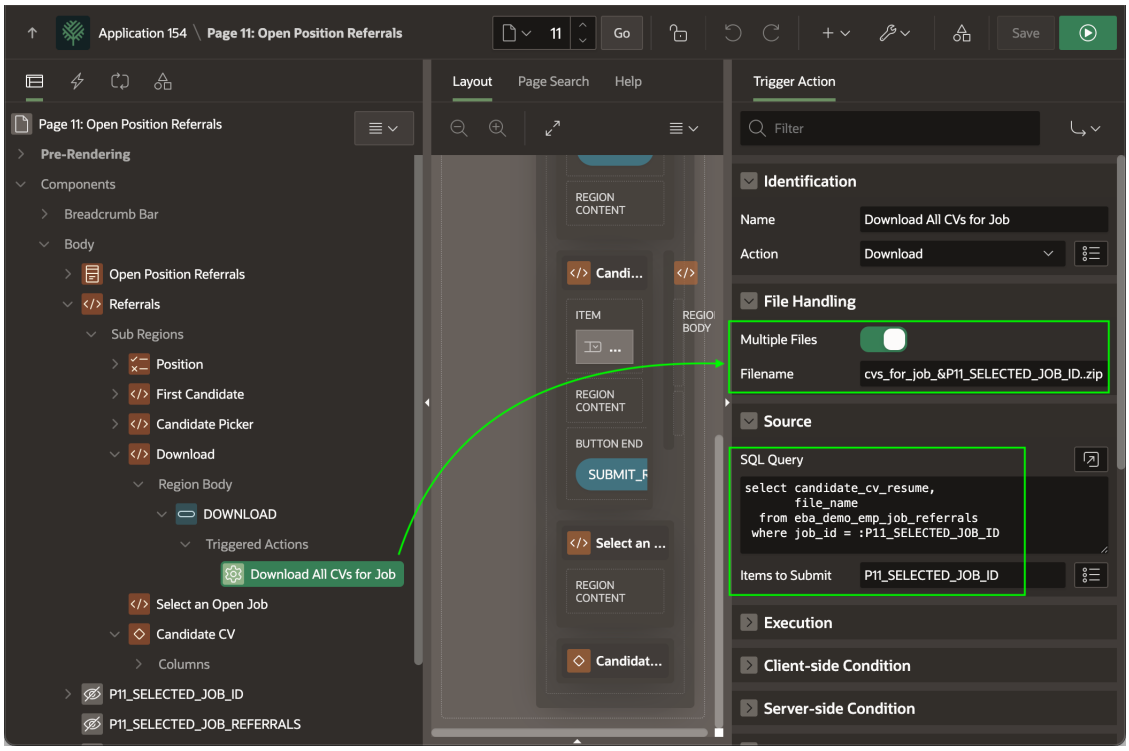
Copyright © 2026, Oracle and/or its affiliates.

name. Since the multi-CV query uses a different `P11_SELECTED_JOB_ID` page item as a bind variable, adjust the *Items to Submit* accordingly.

```
select candidate_cv_resume,
       file_name
  from eba_demo_emp_job_referrals
 where job_id = :P11_SELECTED_JOB_ID
```

> **Tip**
>
> When using the *Download* page process, the SQL query is the same but there are no **Items to Submit** to configure since it happens on the server.

Figure 16-68 Downloading Multiple Files in a ZIP Archive with the Native Dynamic Action



16.8 Uploading Multiple Images or Files

Your apps can use native temporary storage of uploaded files to support any business requirement.

- Handling Temporary Uploaded Files
  Use `APEX_APPLICATION_TEMP_FILES` to stage uploaded images or files before saving them.
- Posting Multiple Break Room Photos
  Post multiple Break Room photos by staging uploads, tracking titles, previewing them, and saving them together.

Copyright © 2026, Oracle and/or its affiliates.

16.8.1 Handling Temporary Uploaded Files

Use APEX_APPLICATION_TEMP_FILES to stage uploaded images or files before saving them.

To support uploading multiple images or files in a logical flow that can span one or more pages, change the *Storage* > *Type* of your *Image Upload* or *File Upload* page item to *Table APEX_APPLICATION_TEMP_FILES*.

- Controlling Retention of Temporary Files
  Control how long uploaded files remain in APEX_APPLICATION_TEMP_FILES before you process them or APEX purges them.
- Referencing Temp Files by Unique Name
  Understand how upload item values identify one or more files in APEX_APPLICATION_TEMP_FILES.
- Displaying Temp File Names and Types
  Show temporary upload file names and readable file types in a report region.
- Processing Temporary Uploaded Files
  Process multiple uploaded files by splitting the upload item's colon-delimited file name list.
- Deleting Temporary Uploaded Files
  Delete temporary upload files from APEX_APPLICATION_TEMP_FILES when you no longer need them.

16.8.1.1 Controlling Retention of Temporary Files

Control how long uploaded files remain in APEX_APPLICATION_TEMP_FILES before you process them or APEX purges them.

Setting their storage type to *Table APEX_APPLICATION_TEMP_FILES* makes *Image Upload* and *File Upload* items automatically save uploaded files in this table. Its columns appear in the table below.

Table 16-2 APEX_APPLICATION_TEMP_FILES Table Structure

Column Name	Data Type	Description
NAME	VARCHAR2	Unique name of the uploaded file
BLOB_CONTENT	BLOB	Binary content of the file
FILENAME	VARCHAR2	Original file name
MIME_TYPE	VARCHAR2	MIME type of the file
APPLICATION_ID	NUMBER	APEX Application ID
CREATED_ON	DATE	Date and time when file was created

The related *Purge Files at* setting determines the retention period. Set it to *End of Request* if you process the uploaded files to permanent storage during the *same* request that submits them. Set it to *End of Session* instead to process them later, for example, when the user completes a multi-step process.

16.8.1.2 Referencing Temp Files by Unique Name

Understand how upload item values identify one or more files in APEX_APPLICATION_TEMP_FILES.

Copyright © 2026, Oracle and/or its affiliates.

The value of the *Image Upload* or *File Upload* page item contains:

- *for single-file upload* – the unique name of the uploaded file
- *for multi-file upload* – the colon-delimited list of unique names of the uploaded files.

16.8.1.3 Displaying Temp File Names and Types

Show temporary upload file names and readable file types in a report region.

Show the end-user the file names and user-friendly file types in temporary storage using a report region with a query like:

```
select filename,
       case mime_type
          when 'application/pdf' then 'PDF Document'
          when 'image/jpeg'      then 'JPEG Image'
          when 'image/png'       then 'PNG Image'
          when 'image/gif'       then 'GIF Image'
          when 'text/plain'      then 'Text File'
          when 'application/zip' then 'ZIP Archive'
          when 'text/csv'        then 'CSV File'
          when 'application/vnd.openxmlformats-
officedocument.wordprocessingml.document'
          then 'Word Document'
          when 'application/vnd.openxmlformats-
officedocument.spreadsheetml.sheet'
          then 'Excel Spreadsheet'
          when 'application/vnd.openxmlformats-
officedocument.presentationml.presentation'
          then 'PowerPoint Presentation'
          else 'Other Type'
       end as file_type_description
 from apex_application_temp_files
where application_id = :APP_ID
```

16.8.1.4 Processing Temporary Uploaded Files

Process multiple uploaded files by splitting the upload item's colon-delimited file name list.

Process all uploaded files for the current application using a cursor `for` loop like the following. Notice it's passing the `':'` as the delimiter character to `APEX_STRING.SPLIT` to process the distinct unique uploaded file names.

```
for j in (select blob_content,
                 filename,
                 mime_type
            from apex_application_temp_files
           where application_id = :APP_ID
             and name in (select column_value
                            from apex_string.split(:P17_IMAGE),':')))
loop
    -- Reference j.mime_type, j.filename, j.blob_content in here
end loop;
```


Copyright © 2026, Oracle and/or its affiliates.

16.8.1.5 Deleting Temporary Uploaded Files

Delete temporary upload files from APEX_APPLICATION_TEMP_FILES when you no longer need them.

When the *Purge Files at* retention period ends, temporary files are automatically deleted. However, to *eagerly* delete them, use any form of SQL `DELETE` command against the `APEX_APPLICATION_TEMP_FILES` table. For example, to delete:

- All temp files in the session:

```
delete from apex_application_temp_files;
```

- Just files for current application page's upload item:

```
delete from apex_application_temp_files
 where application_id = :APP_ID
             and name in (select column_value
                            from apex_string.split(:P17_IMAGE),':'));
```

16.8.2 Posting Multiple Break Room Photos

Post multiple Break Room photos by staging uploads, tracking titles, previewing them, and saving them together.

The *Break Room* page in the *Woods HR* app lets users post multiple photos at a time and assign titles to the set of pictures they choose. The solution combines an *Image Upload* page item with:

- Temporary image storage in `APEX_APPLICATION_TEMP_FILES` with *End of Session* retention
- "Sidecar" collection to store uploaded image titles, keyed by unique uploaded file name
- `EBA_DEMO_WOODSHR_TEMP_PHOTOS` view joining temporary uploads and titles together
- Interactive Grid, based on the view, to preview uploaded images and add associated titles
- `EBA_DEMO_WOODSHR_FILE` package containing related application logic.
- Configuring Multiple File Image Upload
  Configure an Image Upload item to stage multiple photos in `APEX_APPLICATION_TEMP_FILES` until the session ends.
- Experiencing Post Multiple Photos Flow
  Walk through the multi-photo posting flow, from selecting images to adding titles and posting them.
- Using a Collection to Capture Image Titles
  Use a session-scoped collection to store titles for uploaded images by their unique temporary file names.
- Joining Collection with Temp Files Table
  Join `APEX_APPLICATION_TEMP_FILES` to a collection on unique file name to process uploaded images with additional information related to them.
- Preparing and Saving Images with Titles
  Preview the multi-file upload flow that stages images in a collection before saving them.

Copyright © 2026, Oracle and/or its affiliates.

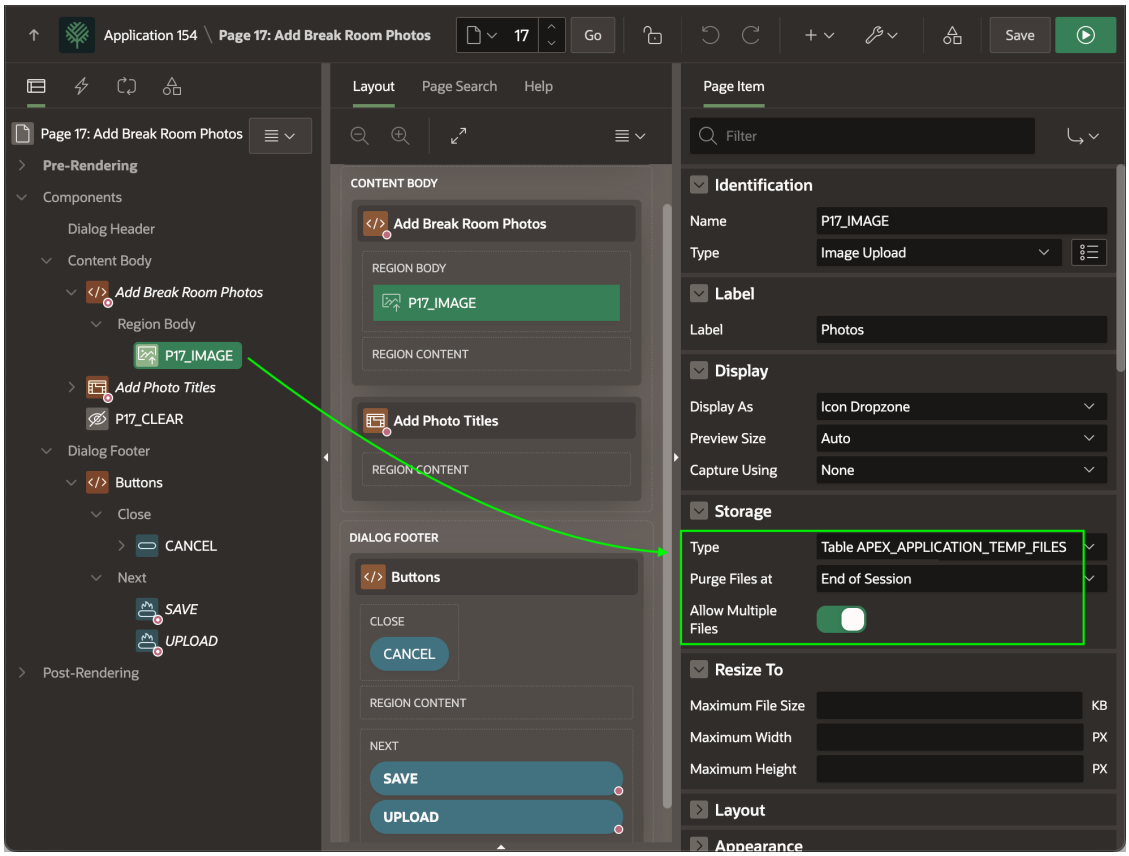
16.8.2.1 Configuring Multiple File Image Upload

Configure an Image Upload item to stage multiple photos in `APEX_APPLICATION_TEMP_FILES` until the session ends.

The *Break Room* page in the *Woods HR* app has a *(Post Multiple Photos)* button that lets users share many pictures at once. It redirects to the *Add Break Room Photos* modal drawer page shown below whose:

- *Image Upload* item uses *Table APEX_APPLICATION_TEMP_FILES*,
- Purges files at the *End of Session*, and
- Enables the *Allow Multiple Files* switch.

Figure 16-69 Add Break Room Photos Page with Multiple Files Image Upload



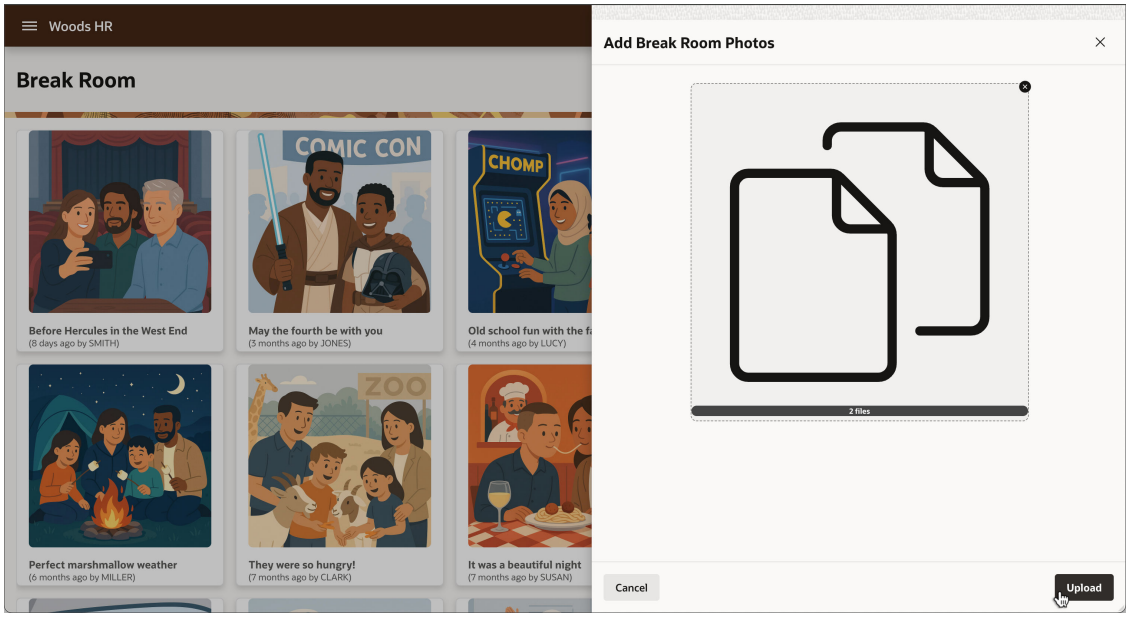
16.8.2.2 Experiencing Post Multiple Photos Flow

Walk through the multi-photo posting flow, from selecting images to adding titles and posting them.

At runtime, when the user clicks on the *Image Upload* preview drop zone, it opens the system file dialog. The user can select multiple images now. For example, after *SMITH* selects two photos to share, the preview item looks like the figure below. The *Add Break Room Photos* modal drawer page shows multiple file icons in the Image Upload dropzone.

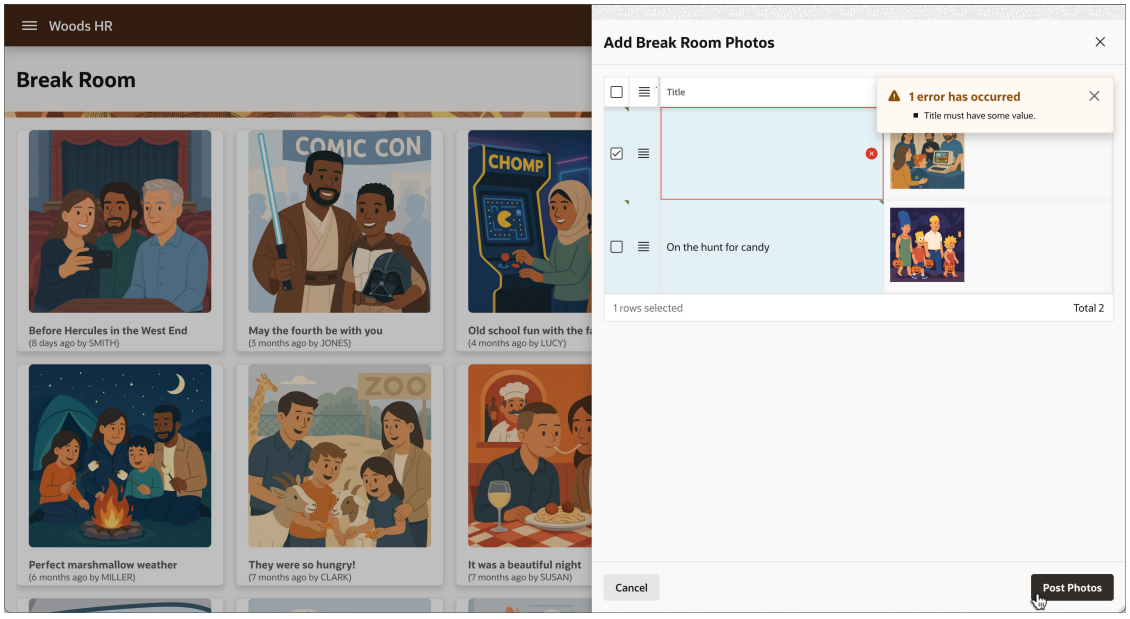
Copyright © 2026, Oracle and/or its affiliates.

Figure 16-70 Preview of Multiple Images Ready to Upload



When the user clicks *(Upload)*, they see an Interactive Grid with image previews and a place to assign a title to each uploaded image. Validation ensures they enter a title for each one.

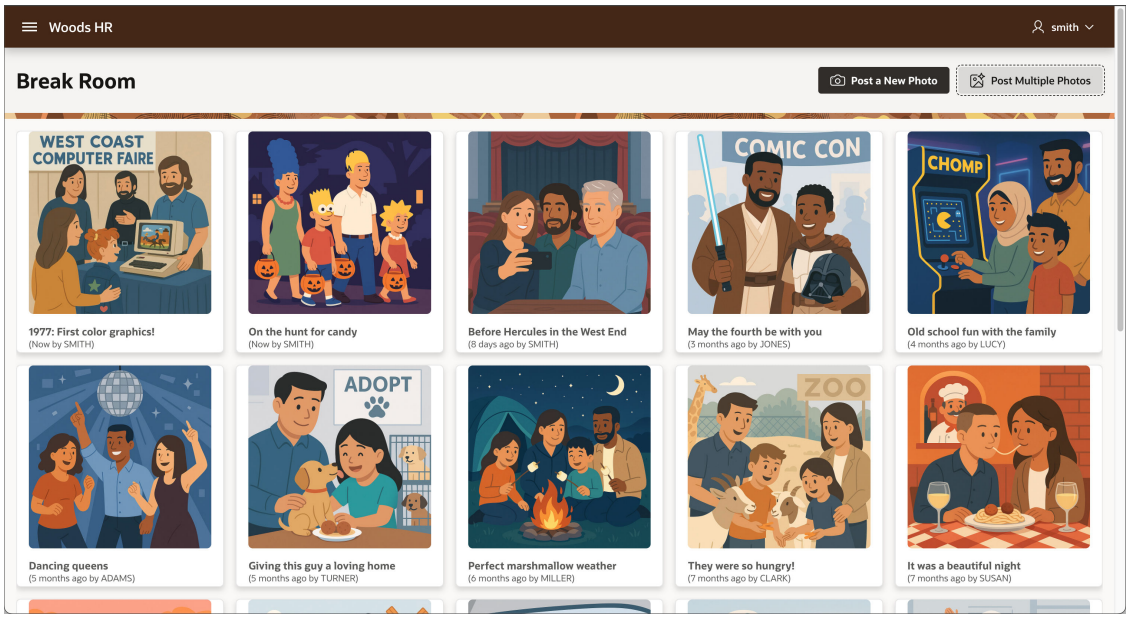
Figure 16-71 Entering Titles for Multiple Posted Photos with Validation



After entering a title for each posted photo, clicking *(Post Photos)* adds them to the break room photo gallery. The figure shows the two newly added photos first in the list in the gallery.

Copyright © 2026, Oracle and/or its affiliates.

Figure 16-72 Both New Photos with Titles Show Up Immediately in Break Room Photo Gallery



16.8.2.3 Using a Collection to Capture Image Titles

Use a session-scoped collection to store titles for uploaded images by their unique temporary file names.

When the user uploads photos, the APEX engine automatically stores them in the `APEX_APPLICATION_TEMP_FILES` table. This table is scoped by user session, so the current user only sees files she uploaded in the current session. To collect *additional* data like a photo title to accompany the uploaded images, use a collection to temporarily store the extra values. Make sure to store the *unique uploaded file name* in the collection to use as the join column, then add additional property values into other collection columns as needed. This page uses a collection named `UPLOADED_PHOTOS` to temporarily save the corresponding photo titles, storing:

- Unique uploaded file name in column `C001`, and
- Photo title in column `C002`.

> **Tip**
>
> For more info on working with session-scoped collections, see Using Temporary Collections.

16.8.2.4 Joining Collection with Temp Files Table

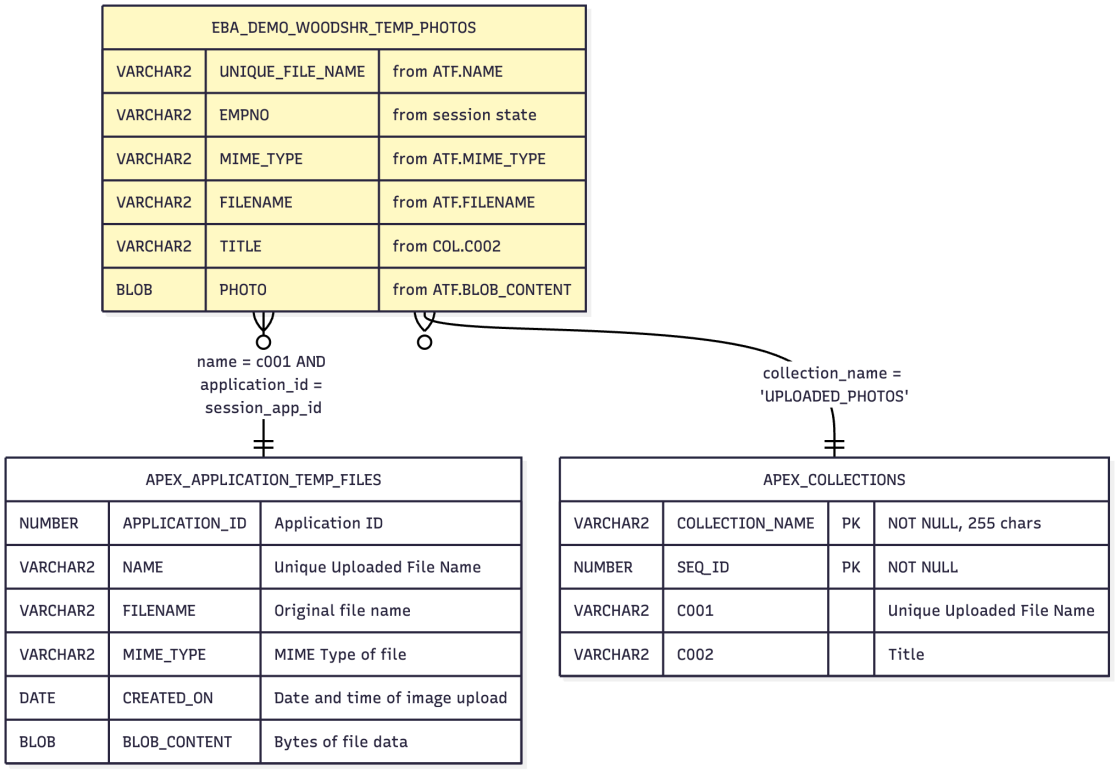
Join `APEX_APPLICATION_TEMP_FILES` to a collection on unique file name to process uploaded images with additional information related to them.

The `EBA_DEMO_WOODSHR_TEMP_PHOTOS` view shown below unifies the "sidecar" storage of image titles with the temporary storage of the uploaded images. It uses the unique uploaded file name to join the `APEX_APPLICATION_TEMP_FILES` table with the `UPLOADED_PHOTOS` collection. The collection stores the unique uploaded file name in column `C001` and the user-entered title for

Copyright © 2026, Oracle and/or its affiliates.

each uploaded image in column C002. The diagram illustrates of the EBA_DEMO_WOODSHR_TEMP_PHOTOS view joins APEX_APPLICATION_TEMP_FILES and the APEX_COLLECTIONS view (for COLLECTION_NAME = 'UPLOADED_PHOTOS')

Figure 16-73 View Adds Titles to Temp Files by Joining with UPLOADED_PHOTOS Collection



The view pulls in session state values of the currently logged-in user and the current application id using the NV function in a common table expression. It uses the NO_MERGE query optimizer hint to avoid having those functions evaluated multiple times.

```
create view eba_demo_woodshr_temp_photos as
with session_state as (
    select /*+ NO_MERGE */
           nv('G_APP_USER_EMPNO') as empno,
           nv('APP_ID')           as current_app_id
      from dual
)
select atf.name as unique_file_name,
       sst.empno,
       atf.mime_type,
       atf.filename,
       col.c002 as title,
       atf.blob_content as photo
  from session_state sst
  join apex_collections col
    on col.collection_name = 'UPLOADED_PHOTOS'
  join apex_application_temp_files atf
```


Copyright © 2026, Oracle and/or its affiliates.

```
 on atf.name = col.c001 /* Join on Unique File Name */
and atf.application_id = sst.current_app_id
```

16.8.2.5 Preparing and Saving Images with Titles

Preview the multi-file upload flow that stages images in a collection before saving them.

The *Add Break Room Photos* page uses application logic in a `EBA_DEMO_WOODSHR_FILE` package to:

- Clear the collection to start the process
- Prepare the collection with unique upload file names after the user uploads the images
- Preview the temporary images in a grid where the user can add the corresponding titles, and
- Save the images and titles to the break room photos table.

- Organizing Application Logic in a Package
  Encapsulate business logic in a PL/SQL package so you can maintain it and change its implementation without affecting callers.
- Clearing Collection to Start the Process
  Use a conditional execution chain to clear staged uploads only when starting a new multi-photo flow.
- Preparing Collection to Enter Photo Titles
  Prepare a collection with uploaded file names so users can add titles before posting photos.
- Previewing Images in a Grid to Enter Titles
  Preview staged uploads in a grid and collect required values before saving them.

16.8.2.5.1 Organizing Application Logic in a Package

Encapsulate business logic in a PL/SQL package so you can maintain it and change its implementation without affecting callers.

The `EBA_DEMO_WOODSHR_FILE` package contains several procedures and functions the page uses:

Table 16-3 EBA_DEMO_WOODSHR_FILE Program Units

Name	Purpose
`UPLOADED_BREAKROOM_PHOTOS`	Returns true if user has any pending uploaded photos.
`WIPE_UPLOADED_BREAKROOM_PHOTOS`	Clears out `UPLOADED_PHOTOS` collection
`PREP_UPLOADED_BREAKROOM_PHOTOS`	Adds unique uploaded file name for uploaded photos to a collection in preparation for collecting a corresponding title for each one.
`DOWNLOAD_UPLOADED_TEMPORARY_PHOTO`	Serves temporary uploaded image as inline data for previewing in the grid.
`SEQ_ID_FOR_UNIQUE_FILENAME`	Looks up the collection sequence id for the row having the unique file name value passed in.
`SAVE_UPLOADED_BREAKROOM_PHOTOS`	Stores uploaded images and titles in break room photos table.

Copyright © 2026, Oracle and/or its affiliates.

The code for these package procedures and functions appears in the following sections where each gets used.

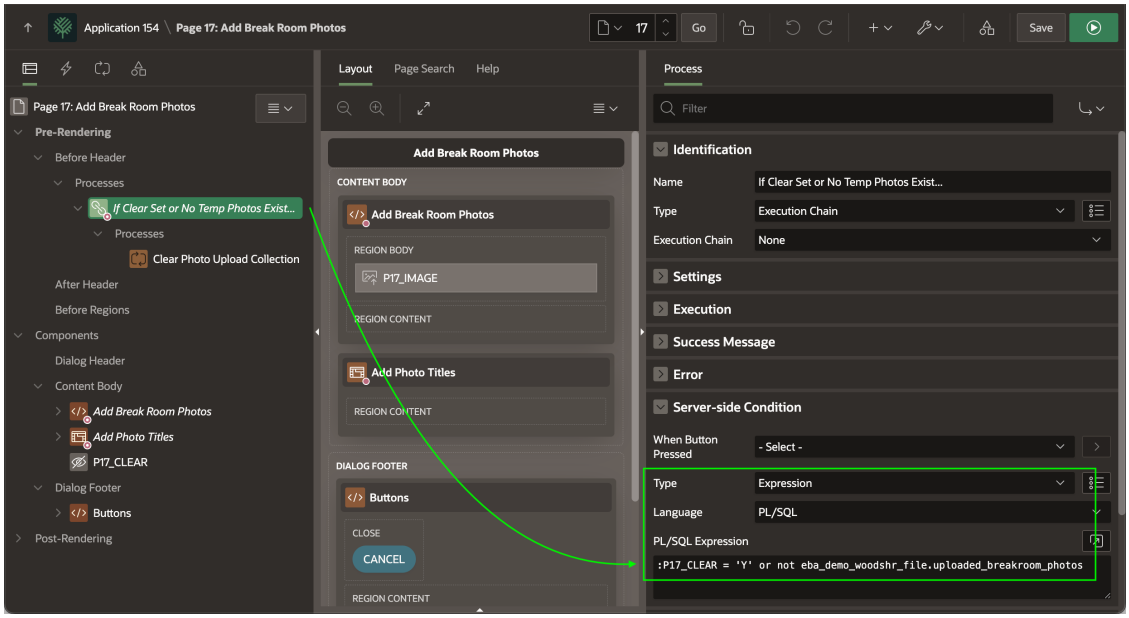
16.8.2.5.2 Clearing Collection to Start the Process

Use a conditional execution chain to clear staged uploads only when starting a new multi-photo flow.

The *Add Break Room Photos* page uses an execution chain page process in the *Pre-Rendering* section to group conditional logic with a descriptive name. As shown below, the containing "parent" chain uses the name "*If Clear Set or No Temp Photos Exist...*" to make the page more self-documenting. The server-side condition applied to this "container" group is the PL/SQL expression below. At runtime, any contained child processes *only* execute if the condition on their containing parent chain is true.

```
:P17_CLEAR = 'Y' or not eba_demo_woodshr_file.uploaded_breakroom_photos
```

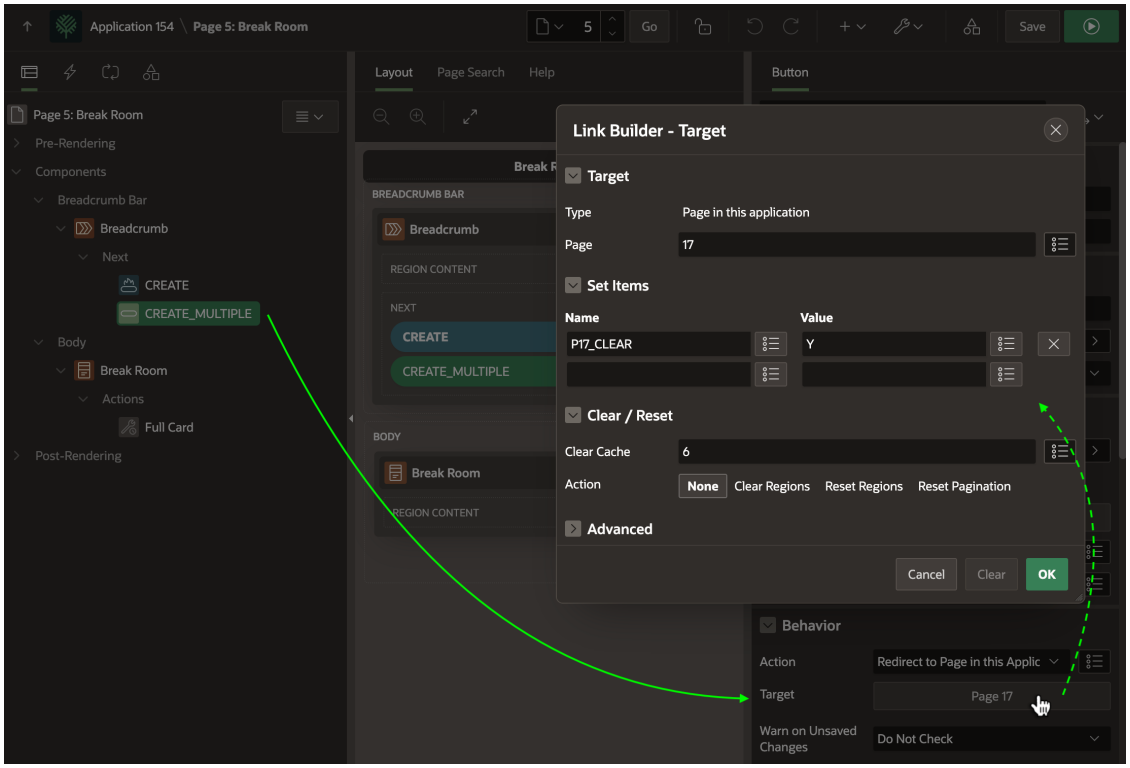
Figure 16-74 Using an Execution Chain to Group Conditional Logic



The *(Post Multiple Photos)* button on the calling *Break Room* page configures the value `Y` to pass to the `P17_CLEAR` parameter to cause the condition to be true.

Copyright © 2026, Oracle and/or its affiliates.

Figure 16-75 Calling Page Button Target Link Sets P17_CLEAR = Y to Clear Collection



The execution chain condition *also* evaluates the `UPLOADED_BREAKROOM_PHOTOS` function in the `EBA_DEMO_WOODSHR_FILE` package to check if any photos have already been uploaded. That function uses the simple logic shown below:

```
-- in package eba_demo_woodshr_file
function uploaded_breakroom_photos
return boolean
is
begin
    for j in (select 1
                from eba_demo_woodshr_temp_photos
               fetch first row only)
    loop
        return true;
    end loop;
    return false;
end uploaded_breakroom_photos;
```

Using the negation operator `NOT` before this function in the server-side condition ensures the collection does *not* get cleared when the user uploads the photos by submitting the page.

```
:P17_CLEAR = 'Y' or not eba_demo_woodshr_file.uploaded_breakroom_photos
```

In this case, the page needs to re-render to show the conditionally-visible Interactive Grid based on the `EBA_DEMO_WOODSHR_TEMP_PHOTOS` view. This lets the user enter corresponding photo titles, and requires the temporary collection rows be present.

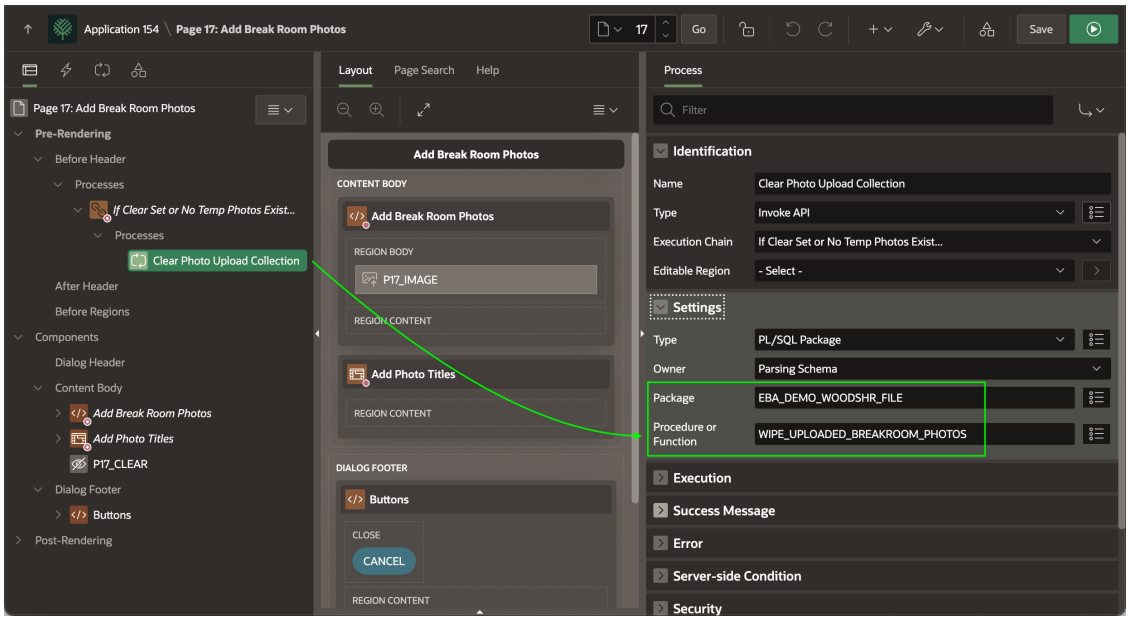
Copyright © 2026, Oracle and/or its affiliates.

The conditionally-executed chain contains a single child process *Clear Photo Upload Collection*. As shown below, it uses an Invoke API process to call the `WIPE_UPLOADED_BREAKROOM_PHOTOS` procedure:

```
-- in package eba_demo_woodshr_file
procedure wipe_uploaded_breakroom_photos
is
begin
    -- Delete any referenced temporary images
    delete from apex_application_temp_files
    where name in (select unique_file_name
                   from eba_demo_woodshr_temp_photos);
    -- Clear the collection

apex_collection.create_or_truncate_collection(c_uploaded_photos_collection);
end wipe_uploaded_breakroom_photos;
```

Figure 16-76 Invoke API Child Process Calls Procedure to Clear the Collection



16.8.2.5.3 Preparing Collection to Enter Photo Titles

Prepare a collection with uploaded file names so users can add titles before posting photos.

When the user clicks the *(Upload)* button, an *If Clicked Upload...* execution chain with its *When Button Pressed* condition set to `UPLOAD` runs the child process below to add the unique uploaded image names to the `UPLOADED_PHOTOS` collection. It's an Invoke API process that calls the `PREP_UPLOADED_BREAKROOM_PHOTOS` procedure. Its `p_unique_file_names` parameter gets its value from the *Image Upload* page item `P17_IMAGE`.

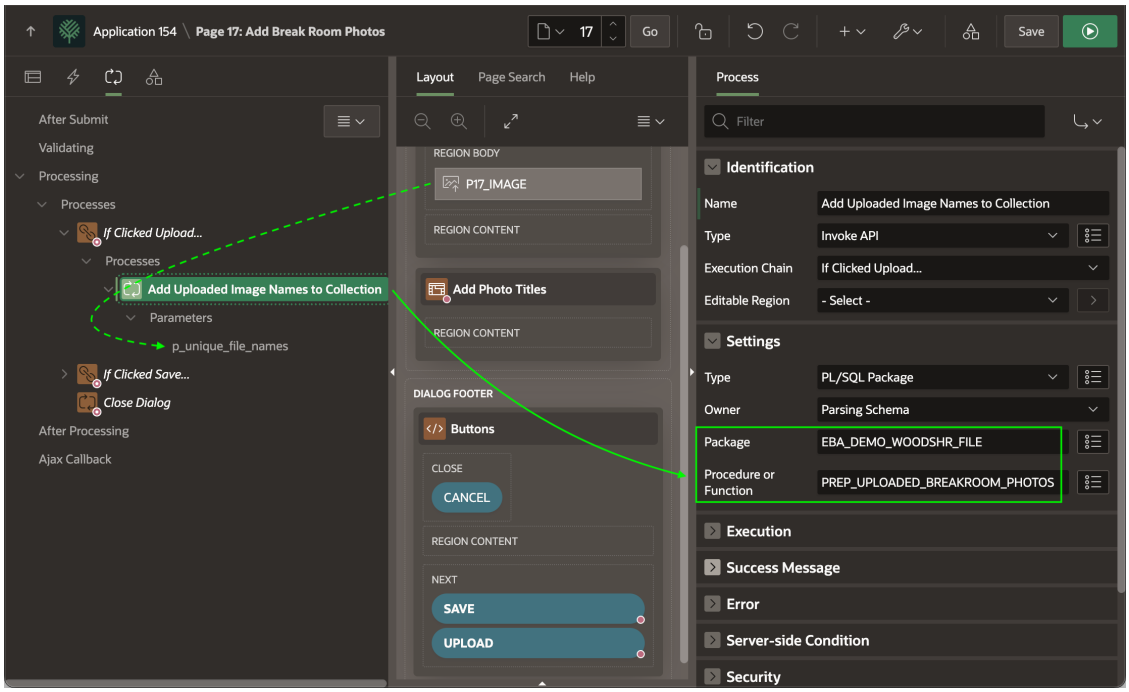
When uploading multiple files, the upload page item's value is a colon-delimited list of unique uploaded file names. Therefore, the code invoked uses `apex_string.split` passing the `':'` as

Copyright © 2026, Oracle and/or its affiliates.

the delimiter to process the unique file names to add them as collection members in column `C001`.

```
-- in package eba_demo_woodshr_file
procedure prep_uploaded_breakroom_photos(
    p_unique_file_names in varchar2)
is
begin
    for j in (select column_value as unique_file_name
                from apex_string.split(p_unique_file_names,':'))
    loop
        apex_collection.add_member(
            p_collection_name => c_uploaded_photos_collection,
            p_c001            => j.unique_file_name);
    end loop;
end prep_uploaded_breakroom_photos;
```

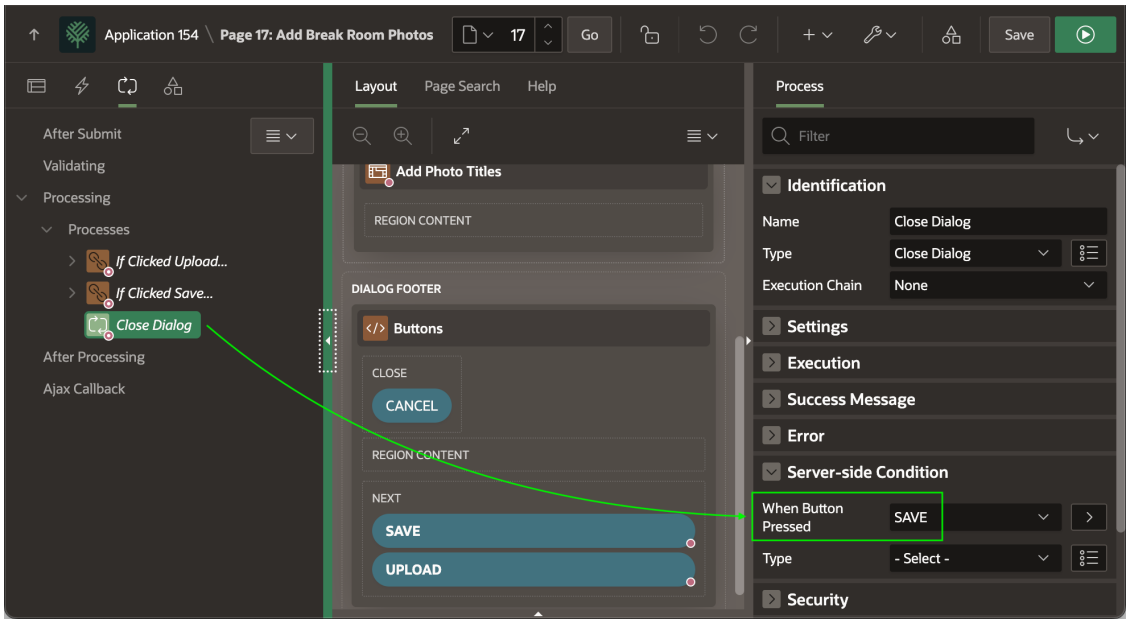
Figure 16-77 Upload Button Triggers Invoke API to Add Image Names to Collection



As shown below, the *Close Dialog* page process uses a server-side condition to only execute when user clicks the `SAVE` button, labelled *(Post Photos)* in the page. So, then the user first clicks the `UPLOAD` button to submit the image files, the dialog close is skipped and the user *remains* on this page to preview the uploaded images and enter the corresponding titles.

Copyright © 2026, Oracle and/or its affiliates.

Figure 16-78 Conditional Close Dialog Leaves User on Page After Image Upload to Set Titles



16.8.2.5.4 Previewing Images in a Grid to Enter Titles

Preview staged uploads in a grid and collect required values before saving them.

After uploading the photos, the end user sees a preview of each one in a grid where they can add a required title before posting.

- Toggling UI Based on Collection Contents
  Toggle the page between image upload and title entry based on whether staged photos exist.
- Serving Inline Temp Images for Preview
  Serve staged upload images from `APEX_APPLICATION_TEMP_FILES` so they can appear in a preview grid.
- Entering Images Titles in a Grid
  Build an editable grid that previews staged images and lets users add titles or remove photos.
- Invalidating Required Queried-Null Titles
  Invalidate required grid cells with no value when queried so users must enter titles before posting photos.
- Saving the Posted Images with Titles
  Save posted images with their entered titles when the user clicks (**Post Photos**).

16.8.2.5.4.1 Toggling UI Based on Collection Contents

Toggle the page between image upload and title entry based on whether staged photos exist.

When the *Add Break Room Photos* page re-renders after the *(Upload)* button click, the collection now contains one or more unique uploaded file names. This makes the `UPLOADED_BREAKROOM_PHOTOS` function return true. In turn, that causes the *If Clear Set or No Temp Photos Exist...* Pre-Rendering process to be skipped.

Copyright © 2026, Oracle and/or its affiliates.

It *also* changes the page from showing the *Image Upload* page item to show the *Add Photo Titles* interactive grid instead. As shown below, the *Add Break Room Photos* static content region containing the *Image Upload* page item `P17_IMAGE` uses the server-side condition:

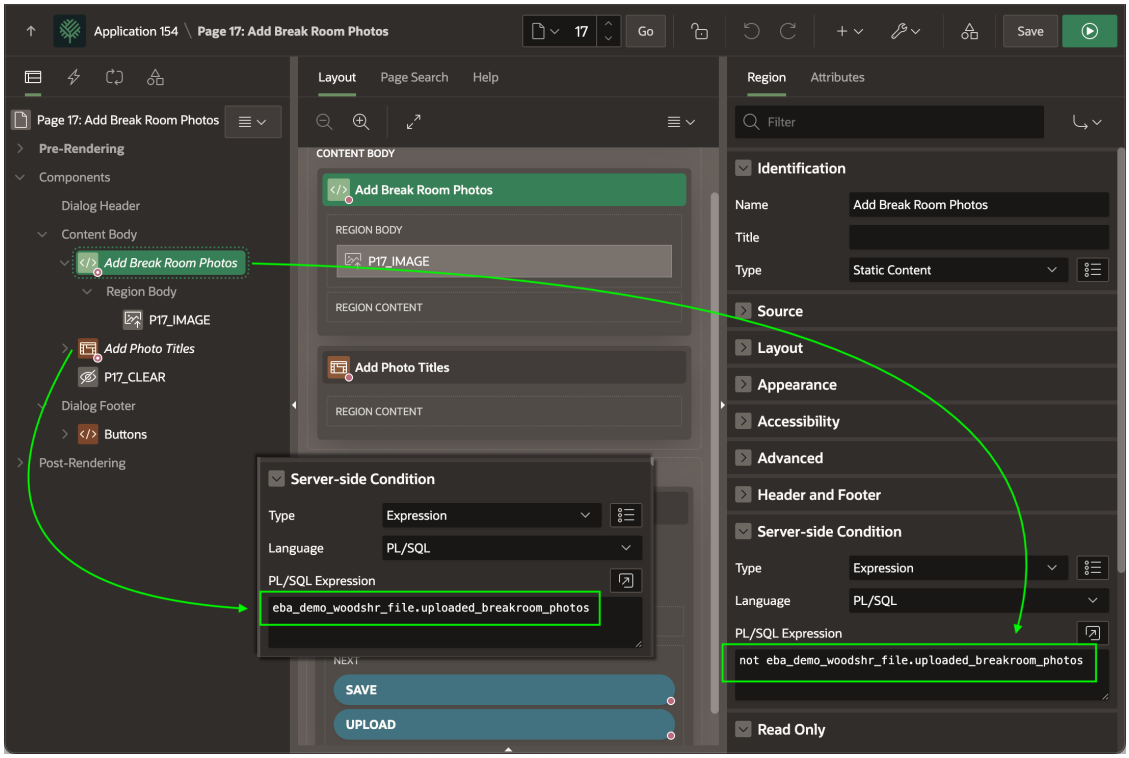
```
not eba_demo_woodshr_file.uploaded_breakroom_photos
```

In contrast, the *Add Photo Titles* interactive grid region uses the logically *opposite* condition:

```
eba_demo_woodshr_file.uploaded_breakroom_photos
```

So, the presence of rows in the collection toggles the page to show one or the other.

Figure 16-79 Collection Rows Determine if Image Upload or Titles Grid Displays



16.8.2.5.4.2 Serving Inline Temp Images for Preview

Serve staged upload images from `APEX_APPLICATION_TEMP_FILES` so they can appear in a preview grid.

As explained in Showing BLOB Images in Interactive Grid, to preview an uploaded image in the grid you need to reference it by URL in the `src` attribute of an HTML `<img>` tag. You can copy the inline image serving page from Serving Inline BLOB Images with a Page to make one with page id 9004 that serves an inline image from `APEX_APPLICATION_TEMP_FILES` table, based on its unique file name. The *Pre-Rendering* process calls the

Copyright © 2026, Oracle and/or its affiliates.

DOWNLOAD_UPLOADED_TEMPORARY_PHOTO procedure, passing in the unique uploaded file name to serve.

```
-- in package eba_demo_woodshr_file
procedure download_uploaded_temporary_photo(
    p_unique_filename in varchar2)
is
    l_file_blob apex_application_temp_files.blob_content%type;
    l_mime_type varchar2(255);
begin
    select blob_content, mime_type
      into l_file_blob, l_mime_type
      from apex_application_temp_files
     where name = p_unique_filename;

apex_http.download (
    p_blob         => l_file_blob,
    p_content_type => l_mime_type,
    p_is_inline    => true );

end download_uploaded_temporary_photo;
```

16.8.2.5.4.3 Entering Images Titles in a Grid

Build an editable grid that previews staged images and lets users add titles or remove photos.

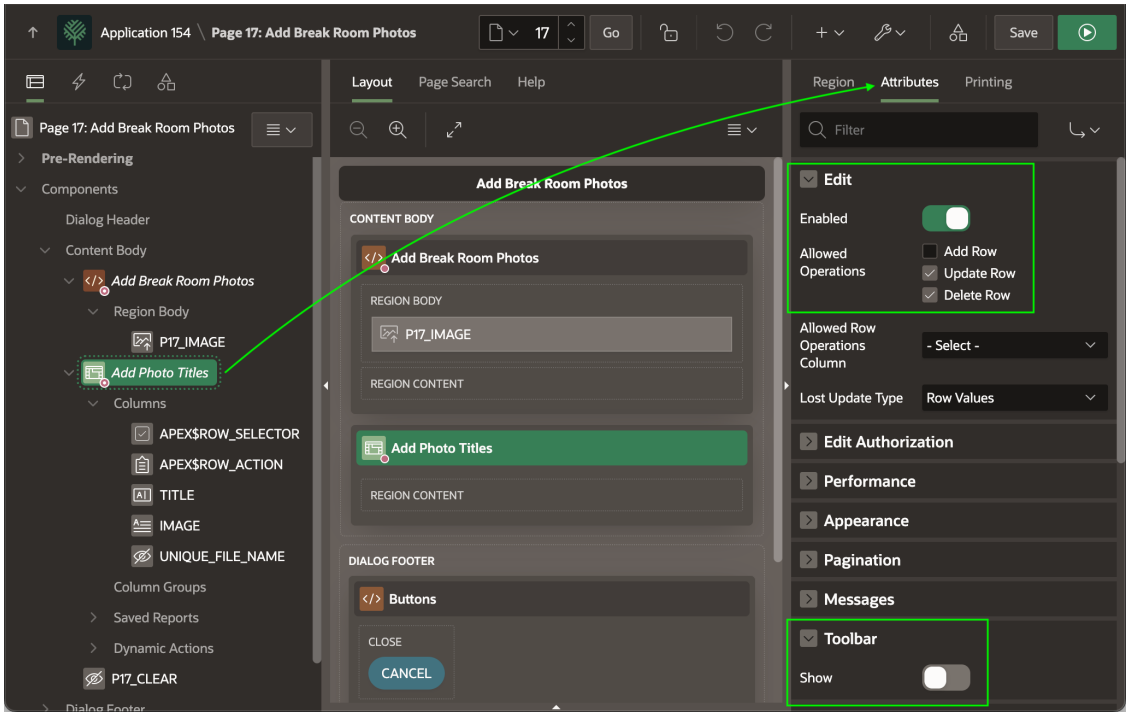
The *Add Photo Titles* editable grid uses the following query as its data source. It's based on the EBA_DEMO_WOODSHR_TEMP_PHOTOS view and produces the HTML <img> tags for previewing the uploaded images in temporary storage using the purpose-built inline image serving page 9004. The hidden UNIQUE_FILE_NAME column is the primary key, and the TITLE column is the only editable column.

```
select  unique_file_name,
        case
            when     photo is not null
                 and dbms_lob.getlength(photo) > 0
            then apex_string.format('<img src="%s">',
                 apex_page.get_url(
                    p_page => 9004,
                    p_x01  => unique_file_name,
                    p_plain_url => true))
        end as image,
        title
  from eba_demo_woodshr_temp_photos
```

As shown below, the grid enables editing and allows *Update* and *Delete* row operations to let the user edit the initially null TITLE column or delete an image they decide not to post. The toolbar functions serve no purpose here, so the *Toolbar* > *Show* switch is disabled.

Copyright © 2026, Oracle and/or its affiliates.

Figure 16-80 Image Preview Grid Allows Update and Delete, with No Toolbar



16.8.2.5.4.4 Invalidating Required Queried-Null Titles

Invalidate required grid cells with no value when queried so users must enter titles before posting photos.

When the interactive grid queries the `EBA_DEMO_WOODSHR_TEMP_PHOTOS` view, the image titles are initially null. While the `TITLE` column's *Value Required* switch is enabled, by default clicking the *(Post Photos)* button does *not* raise an error as you might expect. The grid validates data that the user has ***changed*** or ***created*** in the page, and considers queried data to already be valid. So, if the user does *not* type into the queried-null `TITLE` column cells and clicks to proceed with posting her images, no validation occurs.

The solution is to proactively *invalidate* any empty `TITLE` cells when the grid loads its data. Start by assigning an *HTML DOM ID* of `addPhotoTitles` to the grid region. Then, use a dynamic action event handler on the *Selection Changed [Interactive Grid]* event to run the following small bit of JavaScript. It gets the `model` object for the grid using its static ID. The grid's *model* holds the in-memory rows of data being edited in the browser.

The code uses a `forEach` to run the inline function on each model row. If any row has a null or empty `TITLE` column value, it "dirties" that row by setting its value to a space and then back to what it was before. By using this approach, the user sees the expected validation errors if they leave the required image titles blank.

```
// Run by Selection Changed [Interactive Grid]
// dynamic action event handler
const model = apex.region("addPhotoTitles")
                  .widget()
                  .interactiveGrid("getViews","grid")
                  .model;
```


Copyright © 2026, Oracle and/or its affiliates.

```
model.forEach(function (rec) {
    const titleVal = model.getValue(rec, "TITLE");
    if (titleVal === null || titleVal === "") {
        // Dirty the TITLE so IG will validate on Save
        model.setValue(rec, "TITLE", " ", true);
        model.setValue(rec, "TITLE", titleVal, true);
    }
});
```

16.8.2.5.4.5 Saving the Posted Images with Titles

Save posted images with their entered titles when the user clicks (**Post Photos**).

Once the user enters a title in the grid for every uploaded image – and possibly deletes any they decide not to submit – they click *(Post Photos)*. This button named `SAVE` matches the *When Button Pressed* condition on the *If Clicked Save...* execution chain, so its child processes only run on that click.

- Writing Grid Updates to Collection
  Use custom Interactive Grid DML code to write title edits and deleted rows back to the collection.
- Inserting Photos with Titles
  Insert the staged photos and titles into the permanent break room photos table.
- Clearing Collection When Done

16.8.2.5.4.5.1 Writing Grid Updates to Collection

Use custom Interactive Grid DML code to write title edits and deleted rows back to the collection.

The first child process is *Add Photo Titles - Save Interactive Grid*. It's a native *Interactive Grid – Automatic Row Processing (DML)* process that saves the user's grid edits to the underlying data source. Since this grid uses a *collection*, the native process needs a bit of additional help. Set the *Target Type* to *PL/SQL Code* to use the following custom logic to save the changes. It executes once for each created, modified, or deleted grid row. For each row, `:APEX$ROW_STATUS` indicates if it's a *Create* (`C`), *Update* (`U`), or a *Delete* (`D`). Since this *particular* grid only allows *Update* and *Delete* operations, the code only handles `U` and `D` values.

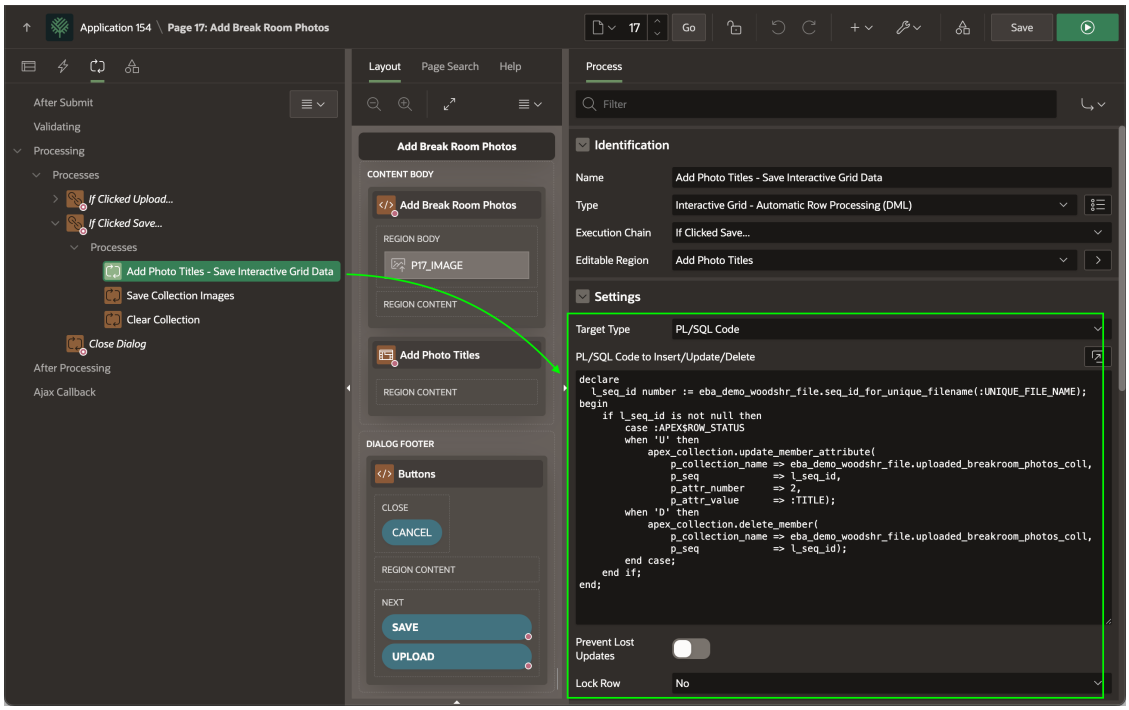
If the row is an update, it calls `UPDATE_MEMBER_ATTRIBUTE` to save the value of the `C002` image title in the current collection row being modified. For a delete, it calls `DELETE_MEMBER` instead.

```
-- Handler code for Grid DML with custom PL/SQL processing
declare
  l_seq_id number :=
eba_demo_woodshr_file.seq_id_for_unique_filename(:UNIQUE_FILE_NAME);
begin
    if l_seq_id is not null then
        case :APEX$ROW_STATUS
        when 'U' then
            apex_collection.update_member_attribute(
                p_collection_name =>
eba_demo_woodshr_file.uploaded_breakroom_photos_coll,
                p_seq             => l_seq_id,
                p_attr_number     => 2,
                p_attr_value      => :TITLE);
        when 'D' then
            apex_collection.delete_member(
```


Copyright © 2026, Oracle and/or its affiliates.

```
                p_collection_name =>
eba_demo_woodshr_file.uploaded_breakroom_photos_coll,
                p_seq             => l_seq_id);
        end case;
    end if;
end;
```

Figure 16-81 Saving Grid Edits Into Collection Using PL/SQL Code Target Type



For both update and delete, a collection sequence id is required. So, it starts by calling `SEQ_ID_FOR_UNIQUE_FILENAME` to use the `UNIQUE_FILE_NAME` column value in the current row to lookup the corresponding sequence id.

```
-- in package eba_demo_woodshr_file
function seq_id_for_unique_filename(
    p_unique_filename in varchar2)
return number
is
begin
    for j in (select seq_id
                from apex_collections
               where collection_name = c_uploaded_photos_collection
                 and c001 /* unique_file_name */ = p_unique_filename)
    loop
        return j.seq_id;
    end loop;
    return null;
end seq_id_for_unique_filename;
```

16.8.2.5.4.5.2 Inserting Photos with Titles

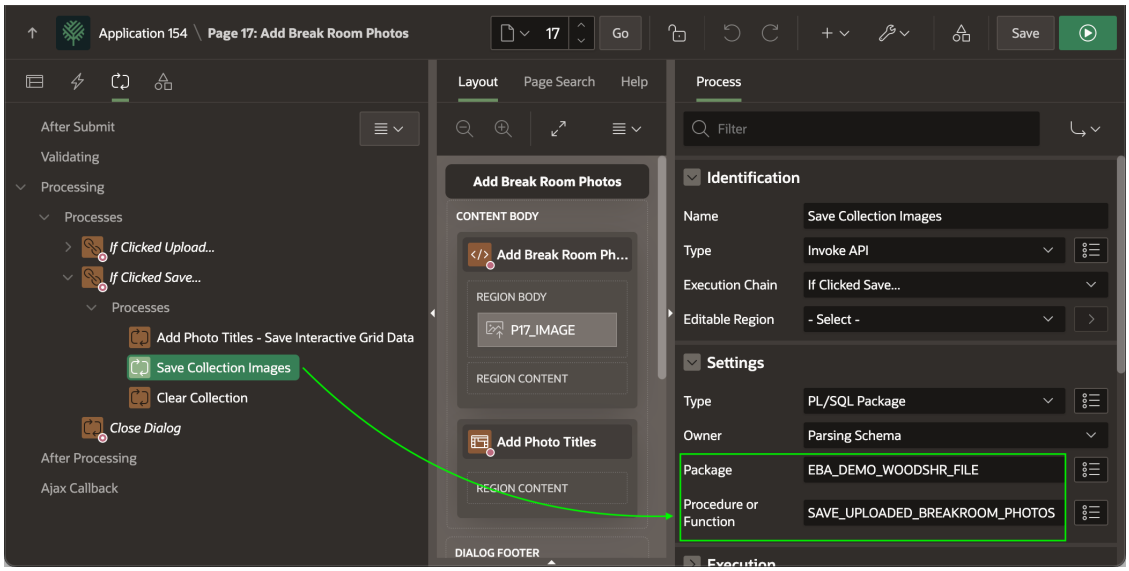
Copyright © 2026, Oracle and/or its affiliates.

Insert the staged photos and titles into the permanent break room photos table.

Once all the grid changes are saved in the collection, the second child process *Save Collection Images* uses an Invoke API to call the `SAVE_UPLOADED_BREAKROOM_PHOTOS` procedure. It uses the `EBA_DEMO_WOODSHR_TEMP_PHOTOS` view insert all the temporary photos and titles into the `EBA_DEMO_EMP_BREAKROOM_PHOTOS` table.

```
-- in package eba_demo_woodshr_file
procedure save_uploaded_breakroom_photos
is
begin
    insert into eba_demo_emp_breakroom_photos(
           empno,
           mime_type,
           file_name,
           title,
           image)
    select empno,
           mime_type,
           filename,
           title,
           photo
      from eba_demo_woodshr_temp_photos;
end save_uploaded_breakroom_photos;
```

Figure 16-82 Invoke API Calls Package Procedure to Save Temporary Photos

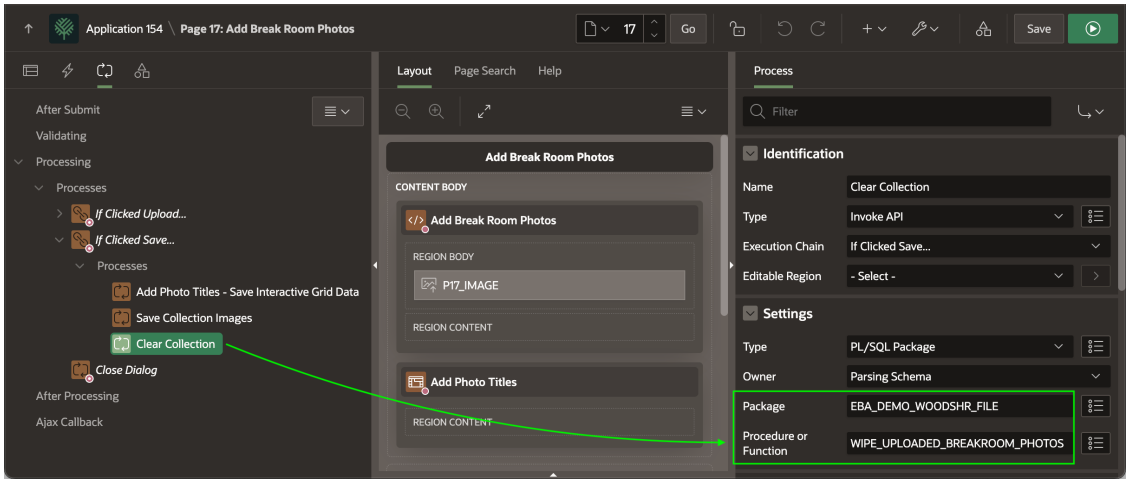


16.8.2.5.4.5.3 Clearing Collection When Done

The final **Clear Collection** child process uses an Invoke API to call `WIPE_UPLOADED_BREAKROOM_PHOTOS`. This deletes any temporary images the collection references, then truncates the collection. The figure shows the Clear Collection child page process selected in the Processing tab, and it highlights the package and procedure name it calls.

Copyright © 2026, Oracle and/or its affiliates.

Figure 16-83 When Done Processing Temporary Images, Clear the Collection



16.9 Using an OCI Object Bucket for Files

Use an OCI object bucket to store application files outside your transactional database.

Oracle Cloud Infrastructure (OCI) offers an *Object Bucket* service to efficiently store and retrieve files. While storing images, PDFs, and other files in a BLOB column the *simplest* approach, your app can store them in an object bucket *instead*. This requires some additional effort, but can potentially improve performance and require less storage in your transactional database.

- Understanding Object Bucket URL Format
  Understand how an OCI object bucket URL includes the region, namespace, bucket name, and object name.
- Defining OCI Web Credential for REST APIs
  Create an OCI Native Authentication web credential to call object bucket REST APIs.
- Listing Buckets with REST Data Source
  Create and test a REST Data Source that lists object buckets in an OCI namespace.
- Using REST Data Source for Bucket Files
  Create a REST Data Source for listing and retrieving files in an OCI object bucket.
- Serving Inline File from Object Bucket
  Serve an object bucket file inline using an APEX page.
- Browsing and Selecting Bucket Files
  Browse object bucket files in a Content Row region and act on the ones the user selects.
- Storing an Uploaded File in Object Bucket
  Upload a temporary file to an OCI object bucket and store its bucket file name in your table.
- Exploring Pre-Authenticated Requests
  Compare ways to serve object bucket files, including application proxy pages and pre-authenticated request URLs.

Copyright © 2026, Oracle and/or its affiliates.

16.9.1 Understanding Object Bucket URL Format

Understand how an OCI object bucket URL includes the region, namespace, bucket name, and object name.

As shown below, your object bucket has a *Namespace* like *x1y2z3a4b5c6* that appears in URLs when accessing the bucket via REST API. A `companion-bucket` in the Frankfurt OCI data center has a URL like:

`https://objectstorage.eu-frankfurt-1.oraclecloud.com/n/`*`x1y2z3a4b5c6`*`/b/`**`companion-bucket`**

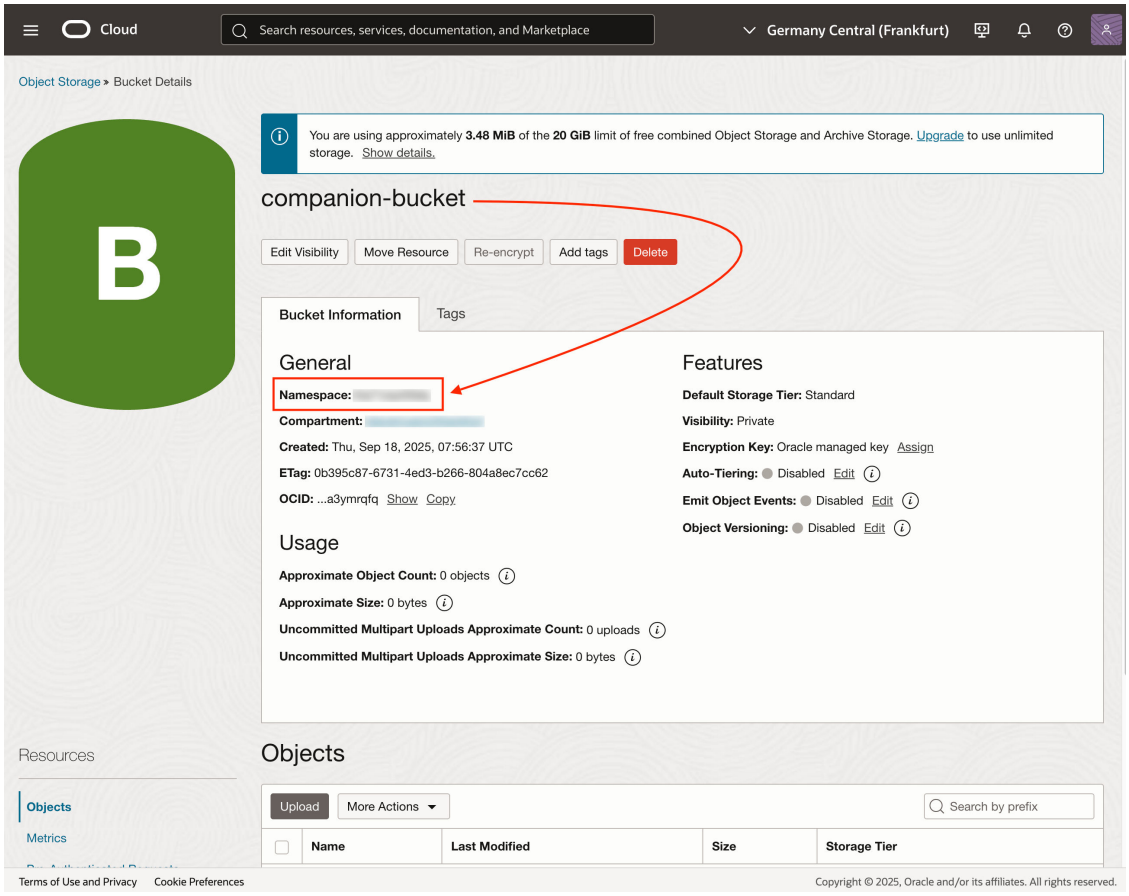
While a file `myfile.jpg` in that bucket will have URL:

`https://objectstorage.eu-frankfurt-1.oraclecloud.com/n/`*`x1y2z3a4b5c6`*`/b/companion-bucket/o/`**`myfile.jpg`**

> **Tip**
>
> The `eu-frankfurt-1` in URLs in this section reflects the OCI region in Frankfurt, Germany. Use your own OCI bucket region when putting these ideas into practice in your own application.

Copyright © 2026, Oracle and/or its affiliates.

Figure 16-84 Object Bucket Has a Namespace Used in REST APIs



16.9.2 Defining OCI Web Credential for REST APIs

Create an OCI Native Authentication web credential to call object bucket REST APIs.

Assume in your OCI tenancy you have:

- configured an object bucket named `companion-bucket`
- identified a user with privileges to manage the bucket, and
- created an API Key for that user to access OCI REST APIs.

To use the REST APIs for an object bucket or other OCI services, start by creating a *Web Credential*. In *Workspace Utilities*, on the *Web Credentials* list page, create a new one of type *OCI Native Authentication* as shown below. In this example, it's named *OCI Object Bucket* and has a *Static ID* of `oci_object_bucket`. You use this Static ID in PL/SQL calls to work with the bucket.

Fill in all web credential details including:

- *OCI User ID* – Oracle Cloud ID (OCID) for the user with privileges to manage the bucket
- *OCI Private Key* – Contents of the `.pem` file you downloaded when creating the API Key
- *OCI Public Key Fingerprint* – Additional security info provided at the time of API Key creation

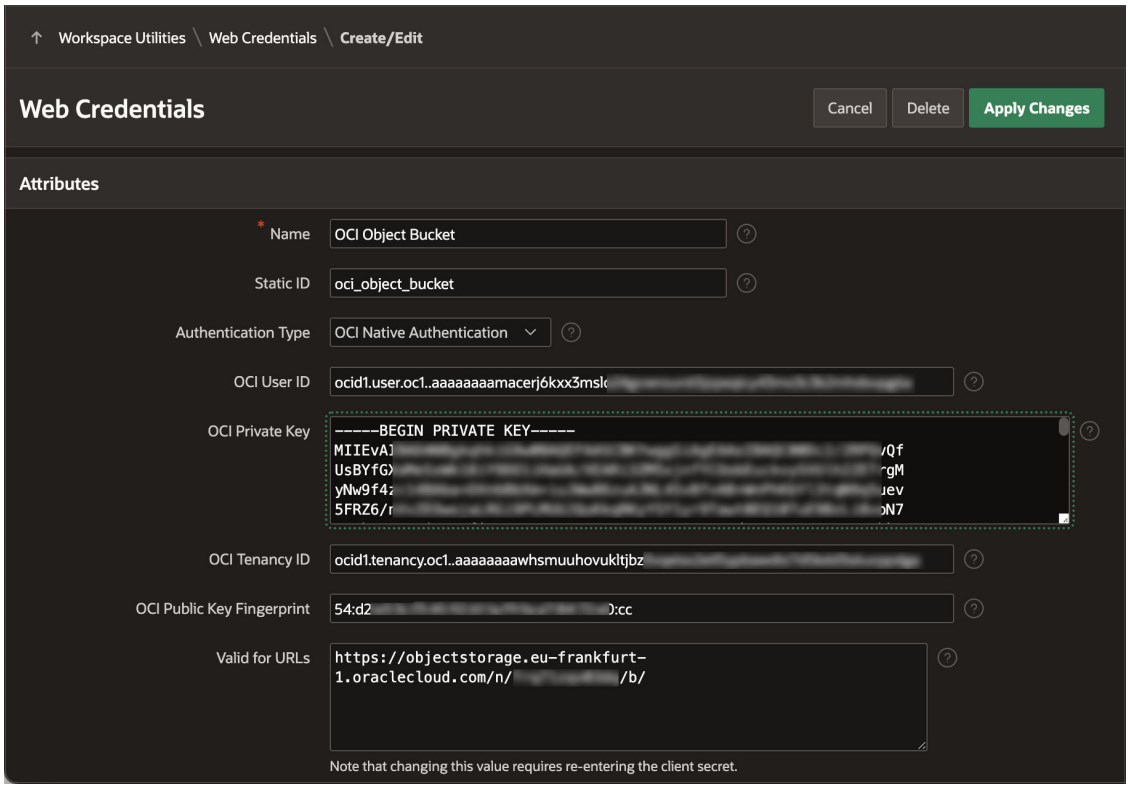
Copyright © 2026, Oracle and/or its affiliates.

Optionally, enter a base URL to limit the use of this credential to object buckets in the *x1y2z3a4b5c6* namespace using a string like the following in *Valid for URLs*:

```
https://objectstorage.eu-frankfurt-1.oraclecloud.com/n/x1y2z3a4b5c6/b/
```

The figure shows the Web Credentials Create/Edit dialog in App Builder with all details required to create a new OCI Native Authentication Web Credential named *OCI Object Bucket*.

Figure 16-85 Defining OCI Native Authentication Web Credential



16.9.3 Listing Buckets with REST Data Source

Create and test a REST Data Source that lists object buckets in an OCI namespace.

Create an *Object Buckets* REST Data Source of type *Oracle Cloud Infrastructure (OCI)* for the base URL of all object buckets in your namespace:

```
https://objectstorage.eu-frankfurt-1.oraclecloud.com/n/x1y2z3a4b5c6/b/
```

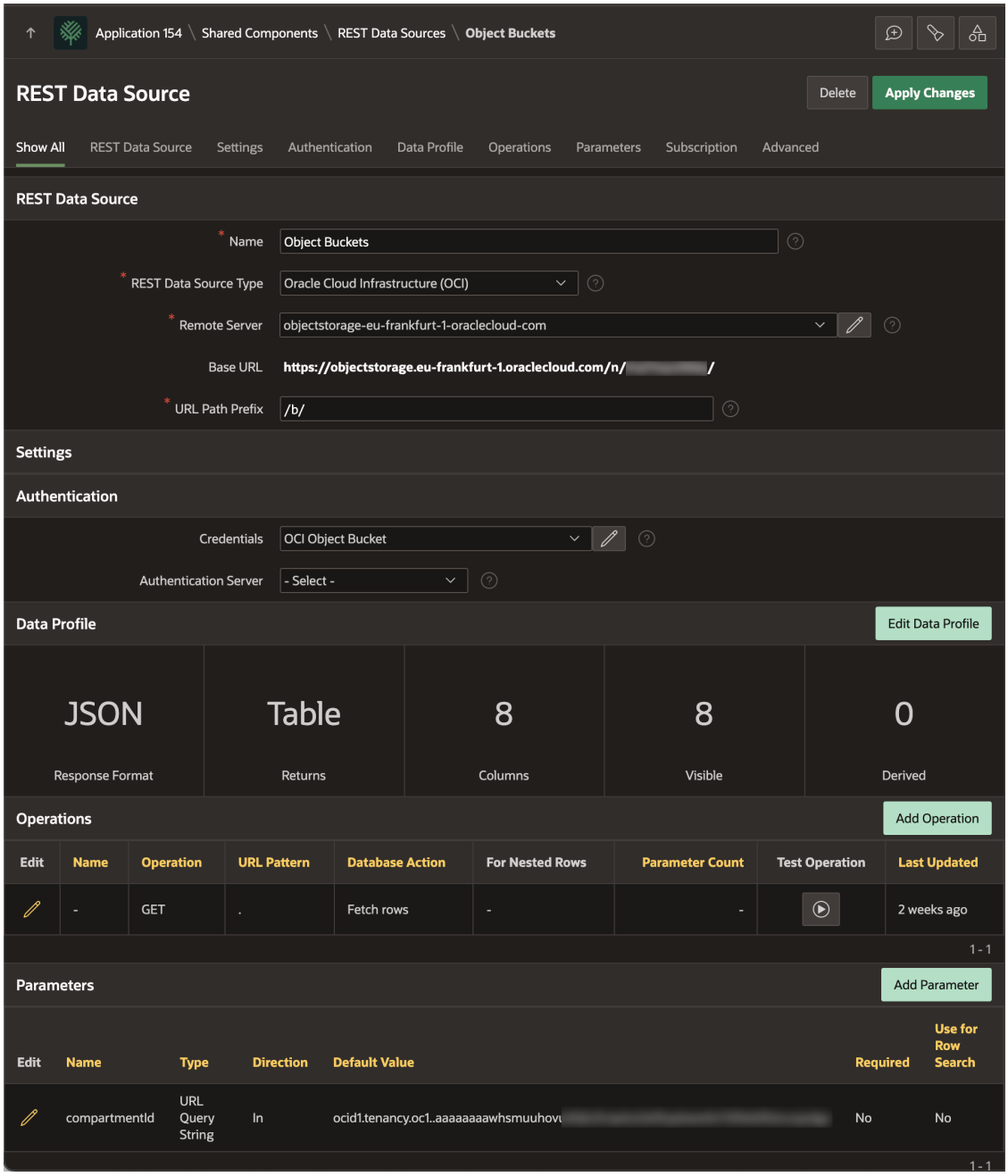
On the *Authentication* step of the *Create REST Data Source Wizard*, use the *OCI Object Bucket* web credential created before, and click the *(Advanced >)* button to add the `compartmentId` parameter of type *URL Query String* having your object bucket's compartment OCID as its static value. Then proceed with the automatic discovery of the data profile and complete the wizard.

Copyright © 2026, Oracle and/or its affiliates.

> **Tip**
>
> After creating the first REST Data Source, edit its *Remote Server* to remove the trailing `/b/`, and adjust the *URL Path Prefix* of the REST Data Source to add the `/b/` there instead. This ensures your setup will be similar to the screenshots in this section.

The figure shows the REST Data Source edit page in App Builder for the *Object Buckets* component. Notice the `compartmentId` parameter and the use of the *OCI Object Bucket* Web Credential.

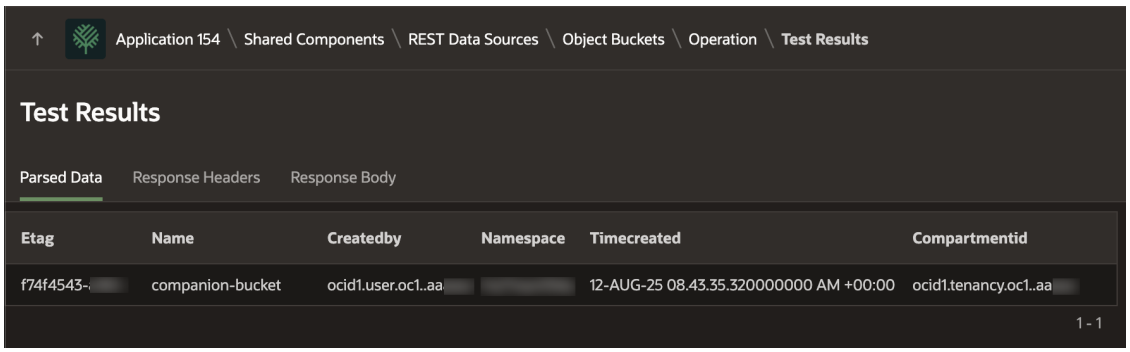
Figure 16-86 Defining Object Buckets REST Data Source



Copyright © 2026, Oracle and/or its affiliates.

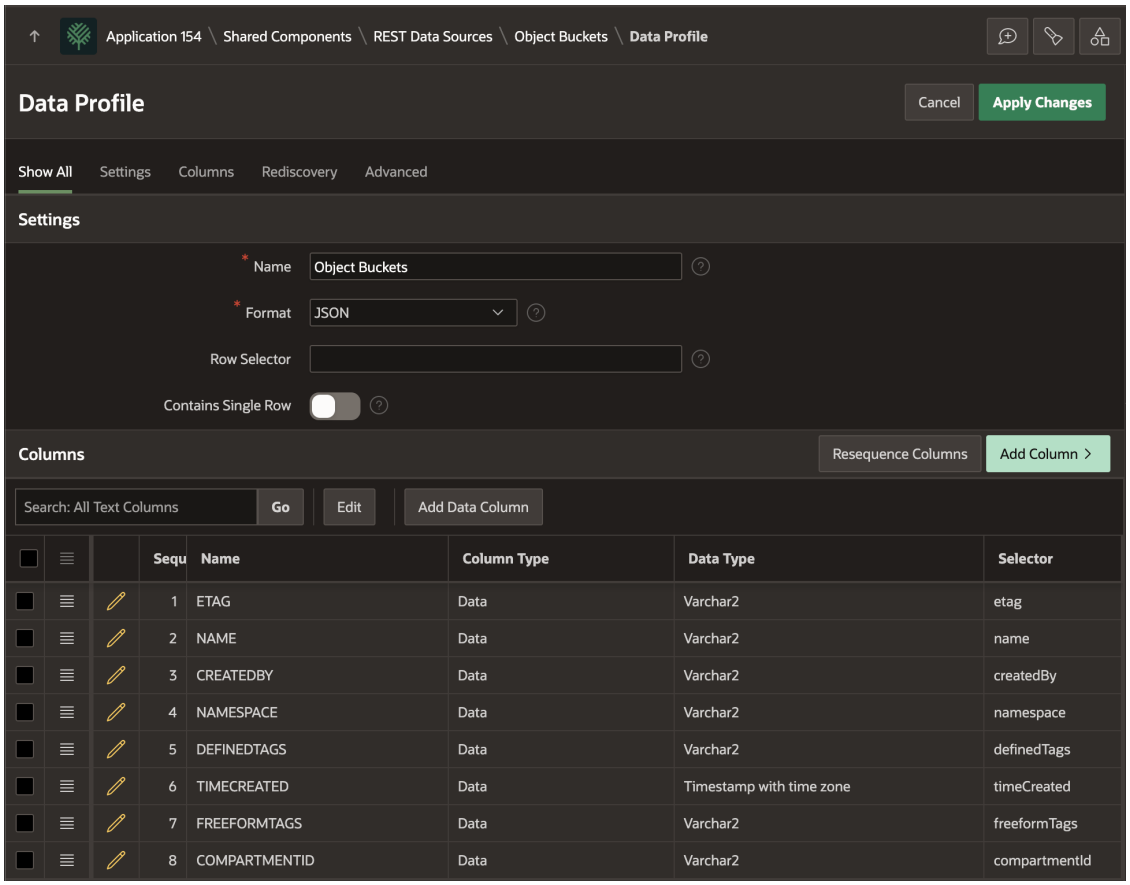
To test that the REST Data Source is working, click the "play" button for the `GET` *Fetch rows* operation in the *Test Operation* column of the *Operations* grid. You see the list of your object buckets to validate that the REST Data Source is working.

Figure 16-87 Testing Object Buckets REST Data Source



Examining the automatically-discovered Data Profile, as shown below, you confirm the 8 columns of information available for object buckets.

Figure 16-88 Data Profile for Object Buckets REST Data Source



Copyright © 2026, Oracle and/or its affiliates.

16.9.4 Using REST Data Source for Bucket Files

Create a REST Data Source for listing and retrieving files in an OCI object bucket.

Create a *Bucket Objects* REST Data Source of type *Oracle Cloud Infrastructure (OCI)* for the base URL of a named object bucket in your namespace, using `:bucketname` in the URL to implicitly define a URL Pattern parameter by that name.

```
https://objectstorage.eu-frankfurt-1.oraclecloud.com/n/
x1y2z3a4b5c6/b/:bucketname/o/
```

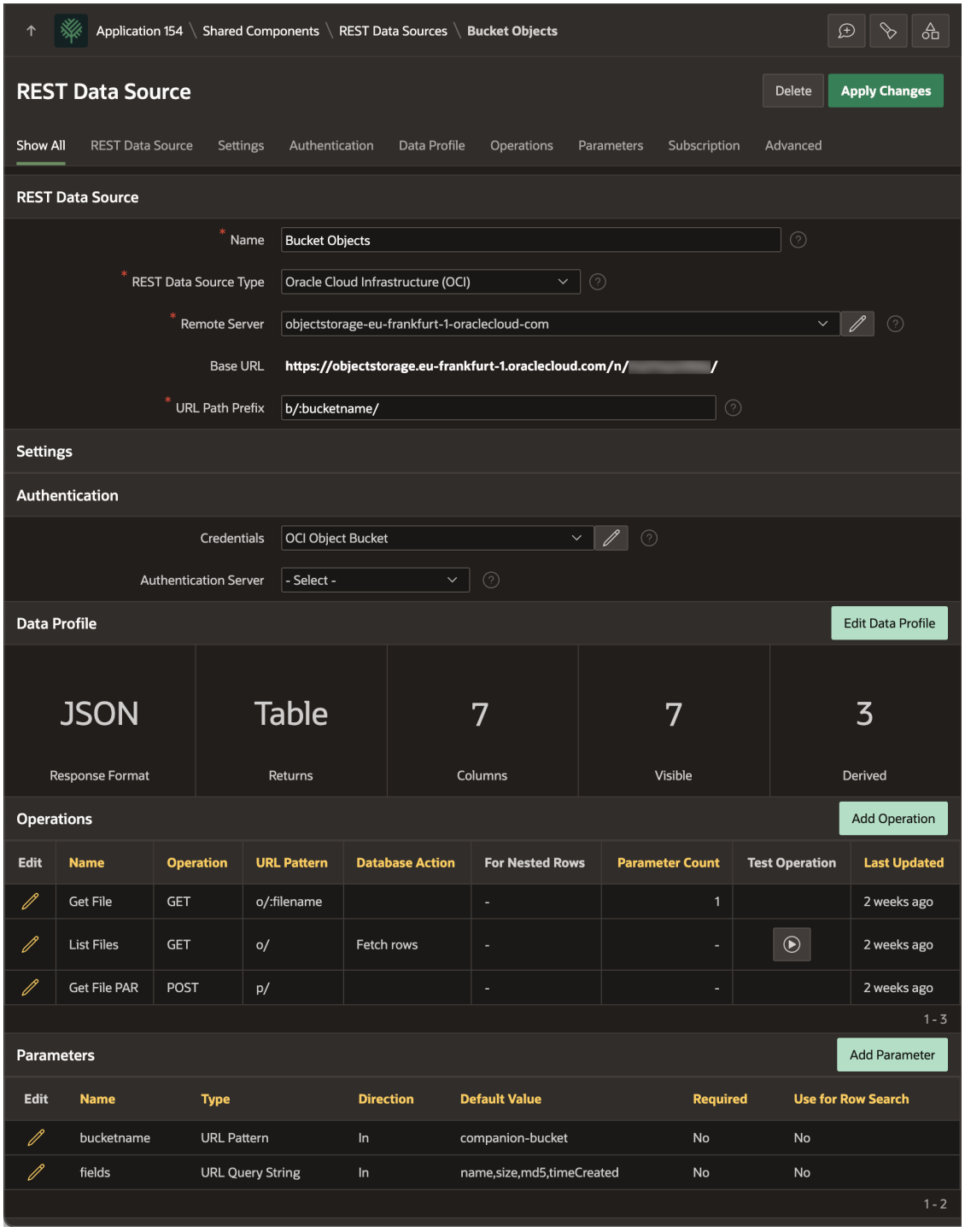
Enter a default bucket name value (e.g. `companion-bucket`) for the `bucketname` parameter so the automatic data profile discovery can do its job. On the *Authentication* step of the *Create REST Data Source Wizard*, use the *OCI Object Bucket* web credential created before. Then proceed with the automatic discovery of the data profile and complete the wizard.

Edit the new REST Data Source to add the following additional configuration shown below:

- Static `fields` *URL Query String* parameter with value `name,size,md5,timeCreated`
- `GET` operation named *Get File* with *URL Pattern* `o/:filename`, *URL Pattern* parameter `filename`, and static ID `get_file`
- `POST` operation named *Get File PAR* with *URL Pattern* `p/` with static ID `get_file_par`

Copyright © 2026, Oracle and/or its affiliates.

Figure 16-89 Defining Bucket Objects REST Data Source



16.9.5 Serving Inline File from Object Bucket

Serve an object bucket file inline using an APEX page.

You can create a page to serve inline object bucket images using the `DOWNLOAD_BUCKET_FILE` procedure. Just call it from a Pre-Rendering *Invoke API* page process, passing in the bucket file name to download inline. It calls the private helper procedure `GET_BUCKET_FILE` to retrieve

Copyright © 2026, Oracle and/or its affiliates.

the BLOB contents and MIME type of the bucket file and then downloads it using `APEX_HTTP.DOWNLOAD`. This page will be nearly identical to the ones explained in Serving Inline BLOB Images with a Page and Serving Inline Temp Images for Preview. The pages all differ only in the "download" procedure they call.

```
-- In package eba_demo_woodshr_oci
--------------------------------------------------------------------------------
-
-- Download a file from an object bucket
--------------------------------------------------------------------------------
-
procedure download_bucket_file(
    p_bucket_name         in varchar2,
    p_file_name           in varchar2,
    p_module_static_id    in varchar2 default 'bucket_objects',
    p_operation_static_id in varchar2 default 'get_file')
is
    l_blob   blob;
    l_mime   varchar2(4000);
begin
    get_bucket_file(
        p_bucket_name         => p_bucket_name,
        p_file_name           => p_file_name,
        p_module_static_id    => p_module_static_id,
        p_operation_static_id => p_operation_static_id,
        p_file_contents       => l_blob,
        p_mime_type           => l_mime);

    apex_http.download(
        p_blob         => l_blob,
        p_content_type => l_mime,
        p_is_inline    => true,
        p_filename     => p_file_name);
end download_bucket_file;
```

The `GET_BUCKET_FILE` procedure calls private helper procedure `BUCKET_URL_FOR` to get the absolute URL and web credential static id to use for the file "get" operation. It sets the `Accept` and `Accept-Encoding` headers to let the object bucket know it wants to receive the binary content verbatim, without compression. It uses the `MAKE_REST_REQUEST_B` function to perform an HTTP `GET` operation with the absolute URL of the file, getting the binary file contents as the return value. Finally, it retrieves the MIME Type of the file by peeking at the `Content-Type` response header.

> **Tip**
>
> A REST Data Source operation does not support a binary response, so use `MAKE_REST_REQUEST_B` to work with a REST API that returns a binary file.

```
-- In package eba_demo_woodshr_oci
--------------------------------------------------------------------------------
-
-- Get file and mimetype from an object bucket file
--------------------------------------------------------------------------------
```


Copyright © 2026, Oracle and/or its affiliates.

```
-
procedure get_bucket_file(
    p_bucket_name         in varchar2,
    p_file_name           in varchar2,
    p_file_contents      out blob,
    p_mime_type          out varchar2,
    p_module_static_id    in varchar2 default 'bucket_objects',
    p_operation_static_id in varchar2 default 'get_file')
is
    l_url    varchar2(32767);
    l_cred   varchar2(4000);
begin
    bucket_url_for(
        p_module_static_id     => p_module_static_id,
        p_operation_static_id  => p_operation_static_id,
        p_bucket_name          => p_bucket_name,
        p_file_name            => p_file_name,
        p_credential_static_id => l_cred,
        p_file_url             => l_url);

    apex_web_service.set_request_headers(
        p_name_01  => 'Accept',          p_value_01 => '*/*',
        p_name_02  => 'Accept-Encoding', p_value_02 => 'identity',
        p_reset    => true);

    p_file_contents := apex_web_service.make_rest_request_b(
                p_credential_static_id => l_cred,
                p_http_method          => 'GET',
                p_url                  => l_url );

    if    apex_web_service.g_status_code <> 200
       or p_file_contents is null
       or dbms_lob.getlength(p_file_contents) = 0
    then
        raise_application_error(-20001,
            'Download failed: HTTP '||apex_web_service.g_status_code);
    end if;

    p_mime_type := nvl(apex_web_service.get_request_header('Content-Type'),
                     'application/octet-stream');
end get_bucket_file;
```

The `BUCKET_URL_FOR` helper method queries application metadata for the REST Data Source operation whose static ID passed in `p_operation_static_id`. The `p_module_static_id` parameter value identifies the REST data source context for this lookup. The lets the REST Data Source definition itself supply the runtime code with the fully qualified URL and credential static ID without having to hard-code that information anywhere.

```
-- In package eba_demo_woodshr_oci
--------------------------------------------------------------------------------
-
-- Return absolute URL of object bucket file using REST Data Source
-- operation as a template
--------------------------------------------------------------------------------
-
```


Copyright © 2026, Oracle and/or its affiliates.

```
procedure bucket_url_for(
    p_module_static_id      in varchar2,
    p_operation_static_id   in varchar2,
    p_bucket_name           in varchar2,
    p_file_name             in varchar2,
    p_credential_static_id out varchar2,
    p_file_url             out varchar2)
is
    c_app_id constant number := sys_context('APEX$SESSION','APP_ID');
begin
    select replace(
             replace(m.url_endpoint||o.url_pattern, ':filename',
p_file_name),
             ':bucketname', p_bucket_name),
           m.credential_static_id
      into p_file_url, p_credential_static_id
      from apex_appl_web_src_operations o
      join apex_appl_web_src_modules   m on o.module_id = m.module_id
     where o.operation_static_id = p_operation_static_id
       and m.module_static_id    = p_module_static_id
       and o.application_id      = c_app_id;
end bucket_url_for;
```

16.9.6 Browsing and Selecting Bucket Files

Browse object bucket files in a Content Row region and act on the ones the user selects.

By combining a Content Row region and the *Bucket Objects* REST Data Source, you can build a page that browses object bucket files and lets a user select them for further processing.

- Displaying Bucket Files with Content Row
  Display object bucket files in a Content Row region with built-in selection support.
- Configuring Content Row for Bucket Files
  Use a REST Data Source as the Content Row source and map its columns into the row display.
- Enhancing REST Data with SQL Expressions
  Add SQL Expression columns to a REST Data Source to compute display values for bucket files.
- Defining and Using Global Substitutions
  Define an application-level substitution for constant values you need to reuse across the app.
- Configuring Selection, Paging, and Avatar
  Configure avatars, row selection, and pagination for a Content Row file browser.

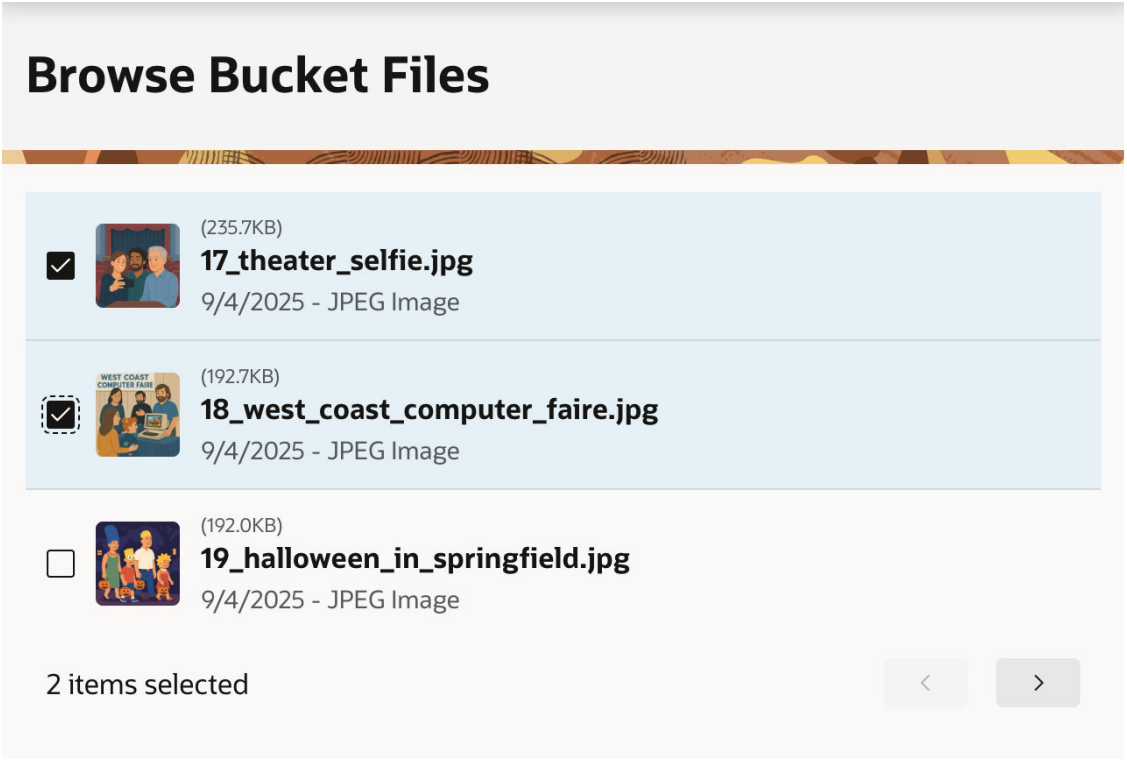
16.9.6.1 Displaying Bucket Files with Content Row

Display object bucket files in a Content Row region with built-in selection support.

As shown below, a *Content Row* region works well to display a paginated list with an optional "avatar". This visual representation can be an icon or an image. This region type also has native support for single and multiple selection. Just configure the name of a page item to hold the primary keys of the selected rows. When necessary, the combination of *Content Row* with a *Bucket Objects* REST Data Source is a simple way to let users browse and select bucket files in your app with no code.

Copyright © 2026, Oracle and/or its affiliates.

Figure 16-90 Browsing and Selecting Bucket Files with a Content Row Region



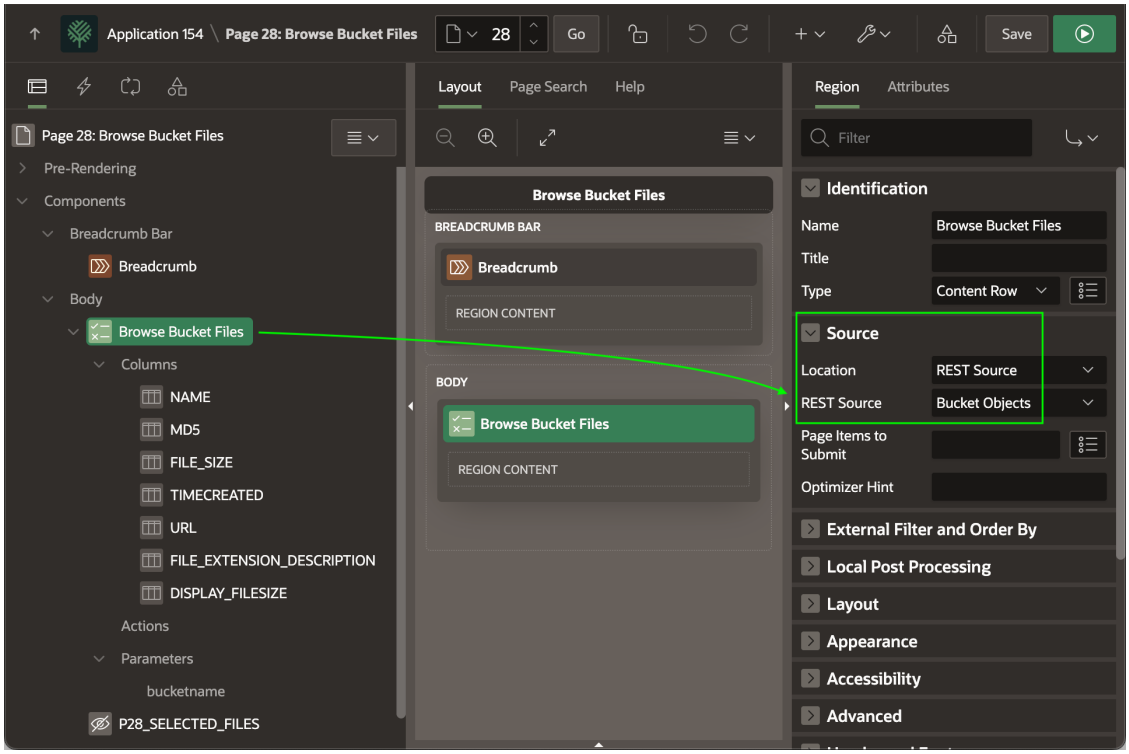
16.9.6.2 Configuring Content Row for Bucket Files

Use a REST Data Source as the Content Row source and map its columns into the row display.

Like all regions, the *Content Row* can directly use a REST Data Source for the data to display. As shown below, just select *REST Source* for *Location* and choose *Bucket Objects* in the *REST Source* list.

Copyright © 2026, Oracle and/or its affiliates.

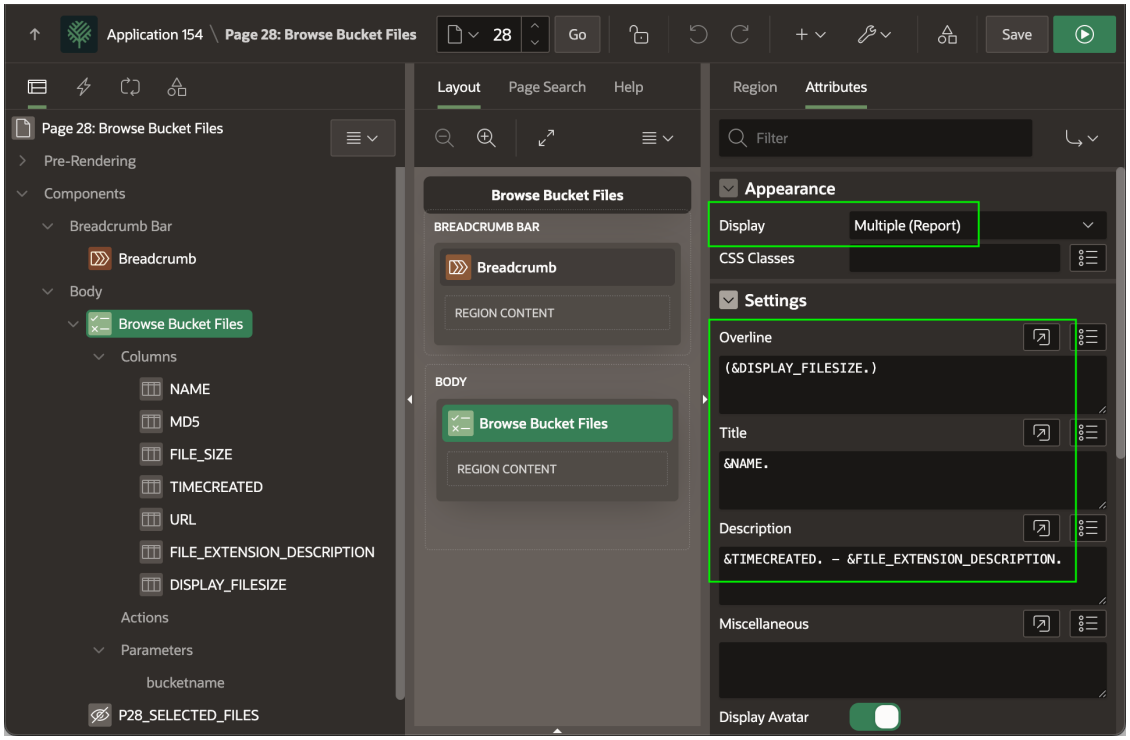
Figure 16-91 Content Row Region Uses Bucket Objects REST Data Source



All the available data profile columns appear under the *Columns* heading in the rendering tree. As shown below, reference any of the columns on the *Attributes* tab using `&NAME.` notation in appropriate slots like *Overline*, *Title*, and *Description*. The setting of *Multiple (Report)* for the *Appearance* > *Display* setting confirms you want the Content Row template component to work with multiple rows. This also enables the *Pagination* settings.

Copyright © 2026, Oracle and/or its affiliates.

Figure 16-92 Configuring What Appears in Each Content Row Slot



16.9.6.3 Enhancing REST Data with SQL Expressions

Add SQL Expression columns to a REST Data Source to compute display values for bucket files.

The object bucket REST API response provides the following basic information about the files:

- `name` – File name
- `md5` – Checksum for file contents
- `size` – Size of the file in bytes
- `timeCreated` – Date and time the file was added to the bucket.

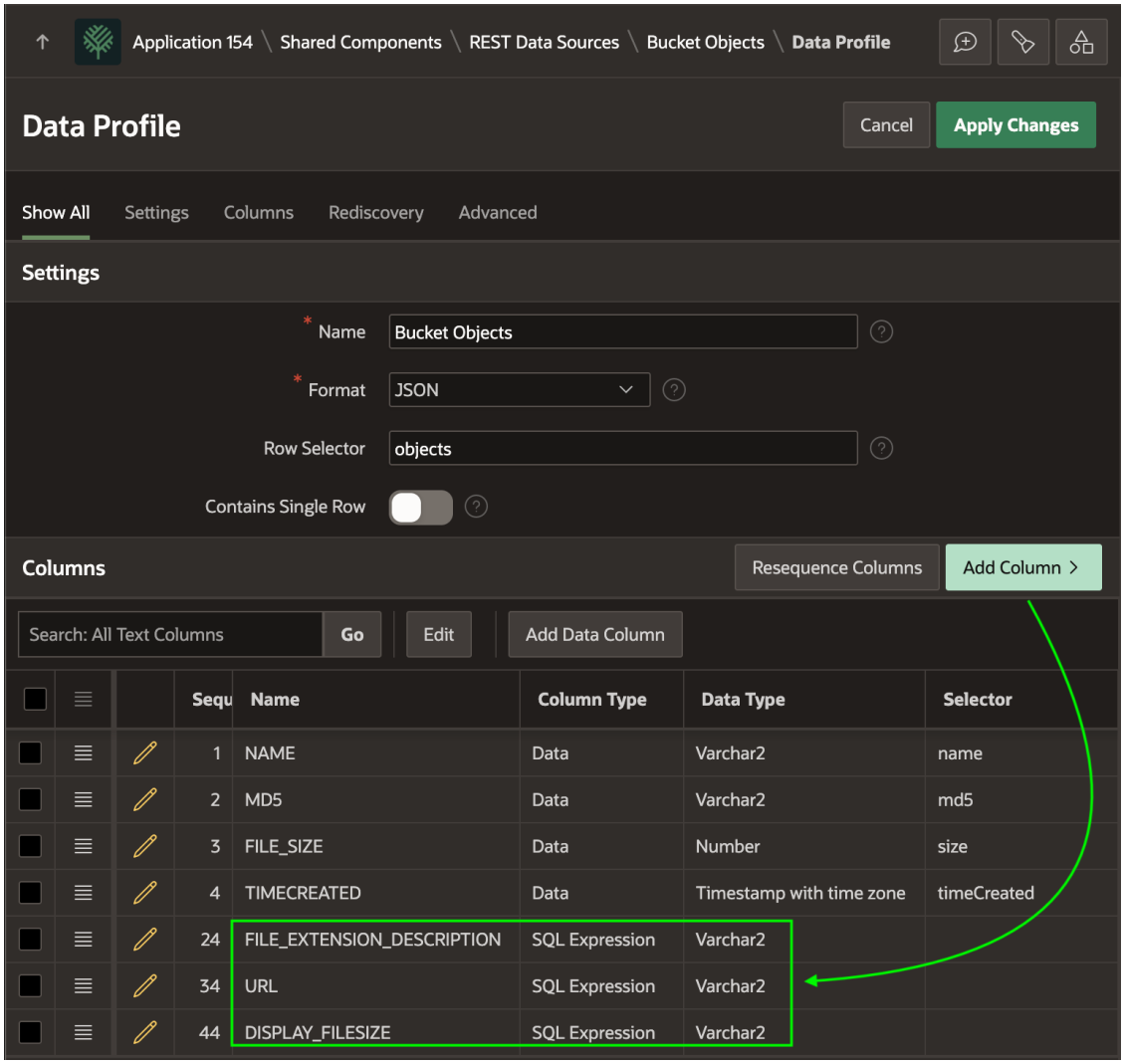
For a more user-friendly file display, you can add additional data profile columns to the REST Data Source that you compute using SQL. These expressions or inline queries can reference other data profile columns by name as needed.

Notice below that *Bucket Objects* has three additional SQL Expression columns in its data profile:

- `FILE_EXTENSION_DESCRIPTION`
- `DISPLAY_FILESIZE`
- `URL`

Copyright © 2026, Oracle and/or its affiliates.

Figure 16-93 Data Profile Columns for Bucket Objects REST Data Source



The `FILE_EXTENSION_DESCRIPTION` column uses the following `CASE` statement to return a description of the file based on its file extension:

```
case lower(substr(name, instr(name,'.',-1)+1))
  when 'pdf'  then 'PDF Document'
  when 'jpg'  then 'JPEG Image'
  when 'jpeg' then 'JPEG Image'
  when 'png'  then 'PNG Image'
  when 'gif'  then 'GIF Image'
  when 'txt'  then 'Text File'
  when 'zip'  then 'ZIP Archive'
  when 'csv'  then 'CSV File'
  when 'doc'  then 'Word Document'
  when 'docx' then 'Word Document'
  when 'xls'  then 'Excel Spreadsheet'
  when 'xlsx' then 'Excel Spreadsheet'
  when 'ppt' then 'PowerPoint Presentation'
  when 'pptx' then 'PowerPoint Presentation'
```


Copyright © 2026, Oracle and/or its affiliates.

```
  else 'Other'
end
```

The `DISPLAY_FILESIZE` column uses the `TO_DISPLAY_FILESIZE` function in the `APEX_STRING_UTIL` package to return a file size display string like *235.7KB*.

```
apex_string_util.to_display_filesize( FILE_SIZE )
```

The `URL` columns uses `APEX_PAGE.GET_URL` to generate the absolute URL that serves the object bucket file as an inline image to the browser.

```
apex_page.get_url(
   p_page         => 9003,
   p_x01          => NAME,
   p_absolute_url => true,
   p_plain_url    => true)
```

16.9.6.4 Defining and Using Global Substitutions

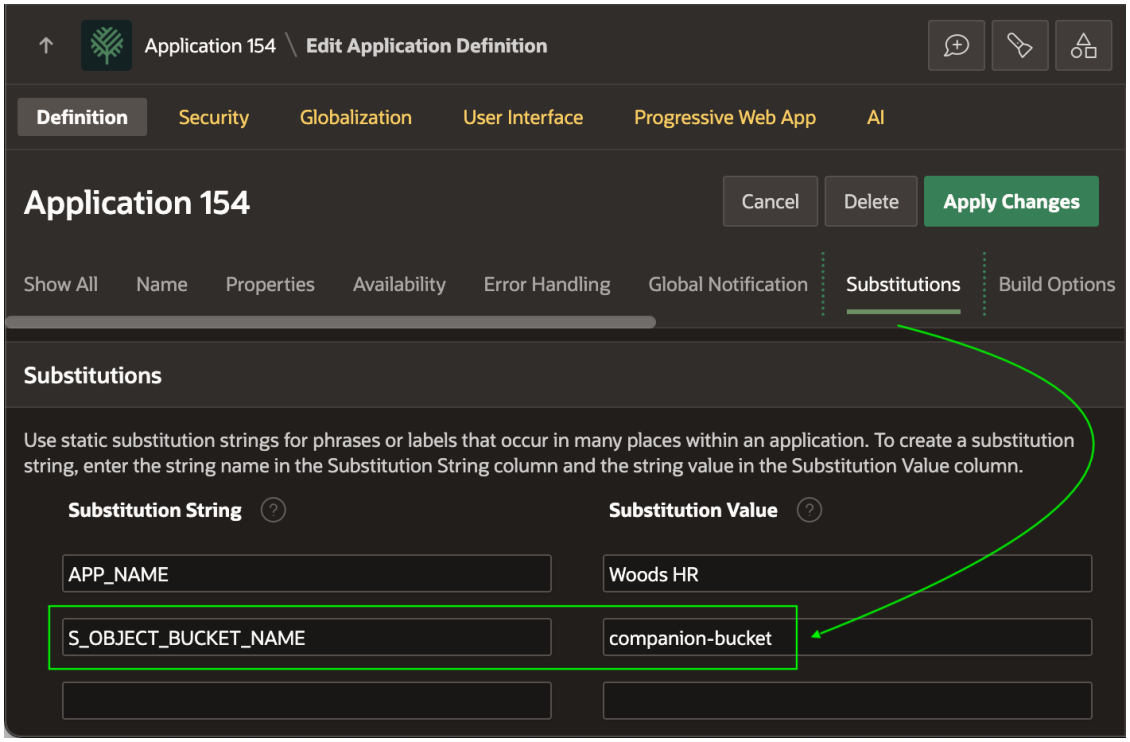
Define an application-level substitution for constant values you need to reuse across the app.

When your app needs to work with a constant value that you might need to reference from many places, define an application-level substitution. In this use case, the name of the object bucket the app uses is a good candidate. Visit the *Substitutions* tab of the application definition under *Shared Components* to configure one.

As shown below, an `S_OBJECT_BUCKET_NAME` substitution string has the corresponding value of `companion-bucket`. Should the object bucket name ever change in the future, you can update this value in one place, and all of its references throughout your app start using the new value.

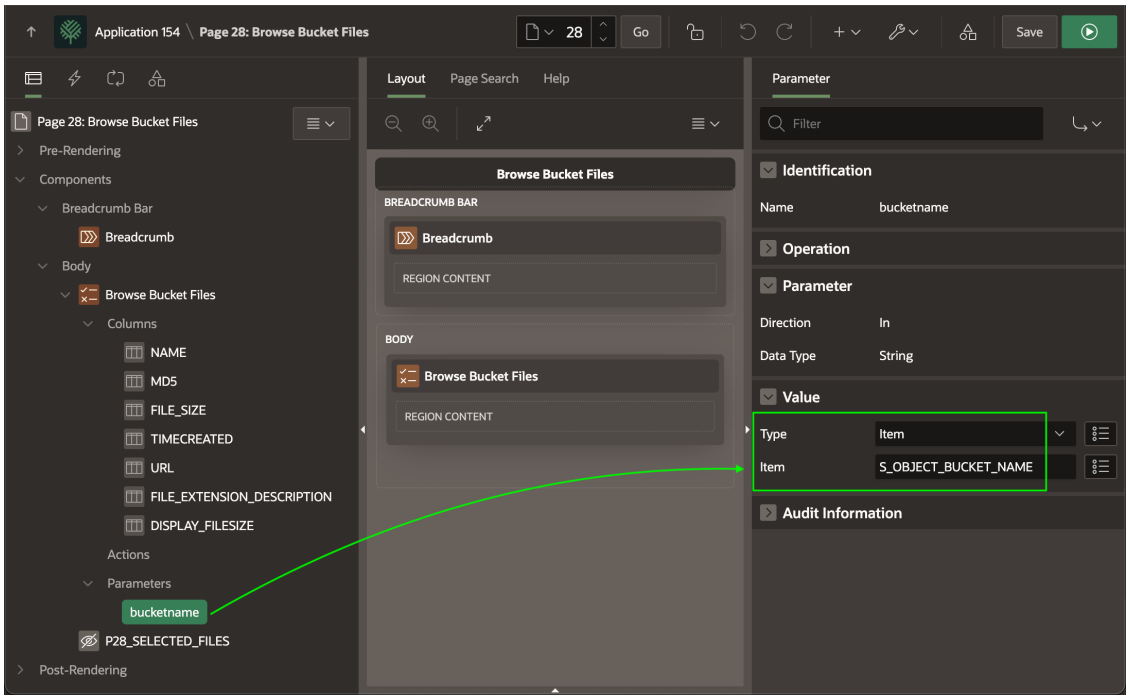
Copyright © 2026, Oracle and/or its affiliates.

Figure 16-94 Use Application Definition Substitutions for Global Constants



Here, you reference it using an *Item* named `S_OBJECT_BUCKET_NAME` when configuring `bucketname`REST Data Source parameter value as shown below.

Figure 16-95 Referencing Global Constants as Items



Copyright © 2026, Oracle and/or its affiliates.

16.9.6.5 Configuring Selection, Paging, and Avatar

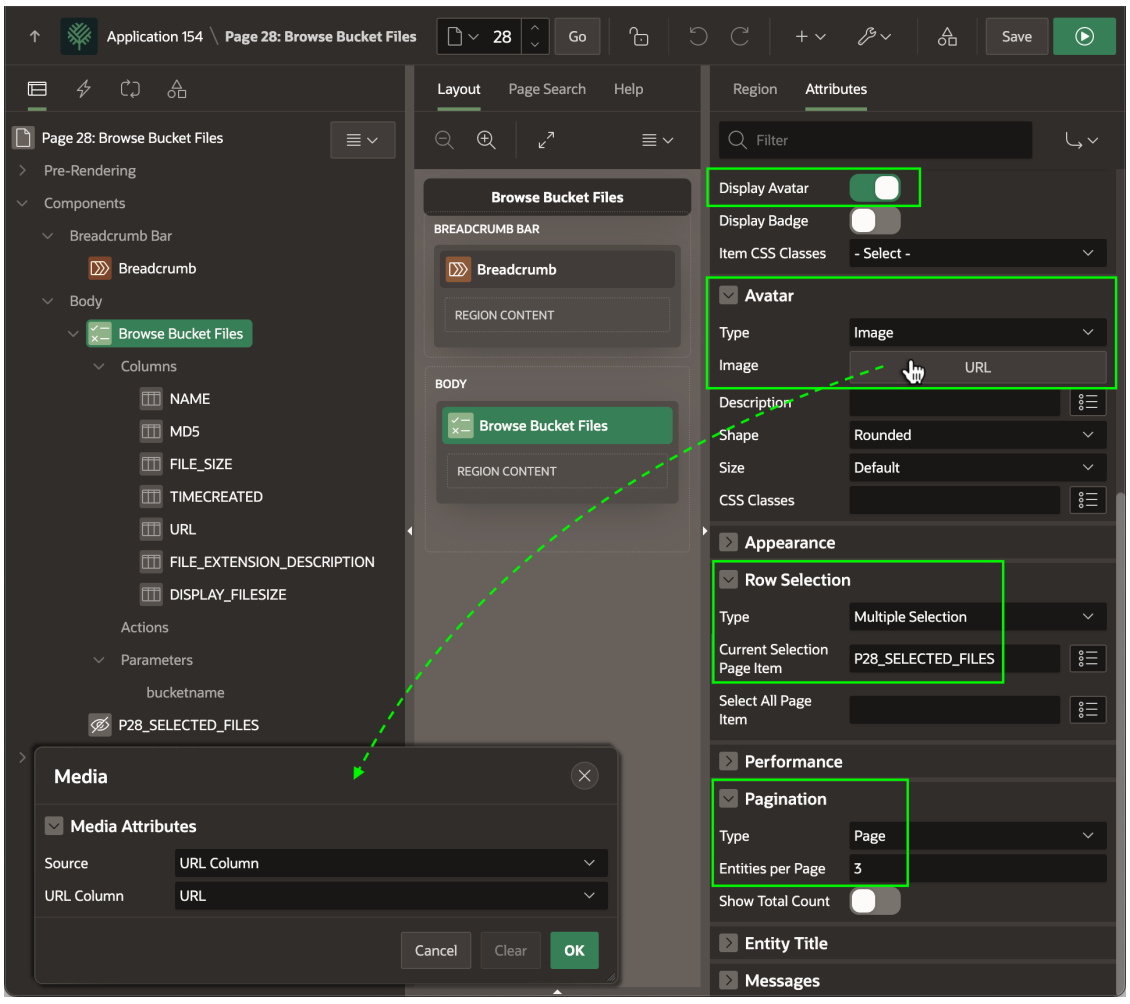
Configure avatars, row selection, and pagination for a Content Row file browser.

On the *Attributes* tab, set the Content Row region to display an image avatar by enabling the *Display Avatar* switch and *Image* as the *Type*. Clicking the *Image* button, as shown below, the *Media* dialog appears to configure that the image source is the *URL Column* named `URL`.

In the *Row Selection* section, choose *Multiple Selection* to let users pick several files at once, and configure the *Current Selection Page Item* to the name of a *Hidden* page item like `P28_SELECTED_FILES`. Its value is a colon-separated list of primary keys for the selected rows. In this use case, the `NAME` column is the primary key.

In the *Pagination* section, choose the type of pagination and if using *Page*-style, indicate the number of *Entities per Page*.

Figure 16-96 Configuring Avatar Image URL, Selection Behavior, and Pagination



16.9.7 Storing an Uploaded File in Object Bucket

Upload a temporary file to an OCI object bucket and store its bucket file name in your table.

Copyright © 2026, Oracle and/or its affiliates.

The PUT_BREAKROOM_PHOTO_TEMP_FILE procedure in the EBA_DEMO_WOODSHR_OCI package adds a temporary file from APEX_APPLICATION_TEMP_FILES to an object bucket. It uses the unique uploaded file name passed in the p_unique_file_name parameter to retrieve the BLOB content, MIME Type, and file name from APEX_APPLICATION_TEMP_FILES. In the *Woods HR* app, to ensure the file name is unique in the bucket, it prepends the primary key of the EBA_DEMO_EMP_BREAKROOM_PHOTO row passed in the p_breakroom_photo_id parameter. It calls procedure PUT_BUCKET_FILE to put the file name in the object bucket, then updates the EBA_DEMO_EMP_BREAKROOM_PHOTO row with the final bucket file name.

> **Tip**
>
> Any pages that work with bucket-stored break room photos use the EMP_BREAKROOM_PHOTO_B view explained in Studying Supporting Tables and Views. It only contains the FILE_NAME column from the underlying EBA_DEMO_EMP_BREAKROOM_PHOTOS table, since the photo data and the MIME Type come from the file in the object bucket.

```
-- In package eba_demo_woodshr_oci
-------------------------------------------------------------------------------
-
-- Put a breakroom photo temp file into an object bucket
-------------------------------------------------------------------------------
-
procedure put_breakroom_photo_temp_file(
    p_bucket_name         in varchar2,
    p_unique_file_name    in varchar2,
    p_breakroom_photo_id  in number,
    p_module_static_id    in varchar2 default 'bucket_objects',
    p_operation_static_id in varchar2 default 'get_file')
is
    l_temp_image_blob blob;
    l_file_name       varchar2(1000);
begin
    for j in (select blob_content, mime_type, filename
                from apex_application_temp_files
               where name = p_unique_file_name)
    loop
        if     j.blob_content is not null
           and dbms_lob.getlength(j.blob_content) > 0
        then
            for k in (select file_name, mime_type
                        from eba_demo_emp_breakroom_photos
                       where id = p_breakroom_photo_id)
            loop
                l_file_name := p_breakroom_photo_id||'_'||j.filename;
                put_bucket_file(
                    p_bucket_name         => p_bucket_name,
                    p_file_name           => l_file_name,
                    p_mime_type           => j.mime_type,
                    p_file_contents       => j.blob_content,
                    p_module_static_id    => p_module_static_id,
                    p_operation_static_id => p_operation_static_id);
            end loop;
```


Copyright © 2026, Oracle and/or its affiliates.

```
            -- Record the filename the uploaded file gets in the bucket
            update eba_demo_emp_breakroom_photos
               set file_name = l_file_name
             where id = p_breakroom_photo_id;
        end if;
    end loop;
end put_breakroom_photo_temp_file;
```

The `PUT_BUCKET_FILE` procedure calls private helper procedure `BUCKET_URL_FOR` explained in Serving Inline File from Object Bucket to get the absolute URL and web credential static id to use for the file "put" operation. It proceeds to set the `Content-Type` header to let the object bucket know the MIME Type of the file being stored, then uses `MAKE_REST_REQUEST` to perform an HTTP `PUT` operation with the absolute URL of the file. Notice that it passes the binary file contents as the value of the `p_body_blob` parameter.

> **Tip**
>
> A REST Data Source operation does not currently support sending a binary request payload, so you need to use `MAKE_REST_REQUEST` with the `p_body_blob` parameter when the REST API to invoke expects the binary contents as the request body.

```
-- In package eba_demo_woodshr_oci
--------------------------------------------------------------------------------
-
-- Put a file into an object bucket
--------------------------------------------------------------------------------
-
procedure put_bucket_file(
    p_bucket_name         in varchar2,
    p_file_name           in varchar2,
    p_mime_type           in varchar2,
    p_file_contents       in blob,
    p_module_static_id    in varchar2 default 'bucket_objects',
    p_operation_static_id in varchar2 default 'get_file')
is
    l_clob   clob;
    l_url    varchar2(32767);
    l_cred   varchar2(4000);
begin
    bucket_url_for(
        p_module_static_id     => p_module_static_id,
        p_operation_static_id  => p_operation_static_id,
        p_bucket_name          => p_bucket_name,
        p_file_name            => p_file_name,
        p_credential_static_id => l_cred,
        p_file_url             => l_url);

    apex_web_service.set_request_headers(
        p_name_01  => 'Content-Type',
        p_value_01 => p_mime_type,
        p_reset    => true);

    l_clob := apex_web_service.make_rest_request(
```


Copyright © 2026, Oracle and/or its affiliates.

```
                    p_credential_static_id => l_cred,
                    p_http_method          => 'PUT',
                    p_url                  => l_url,
                    p_body_blob            => p_file_contents);

    if apex_web_service.g_status_code not in (200,201) then
        raise_application_error(-20001,
            'Upload failed: HTTP '||apex_web_service.g_status_code);
    end if;
end put_bucket_file;
```

16.9.8 Exploring Pre-Authenticated Requests

Compare ways to serve object bucket files, including application proxy pages and pre-authenticated request URLs.

When your application pages reference images or other assets from the object bucket by URL, you have multiple options:

- An authenticated application page can serve inline images from the bucket, or
- You can use a *pre-authenticated request* (PAR) URL to the bucket file.

Using the first strategy, your application page acts as a proxy to the bucket file. In the second approach, the browser downloads the file directly from the object bucket. A hybrid strategy can serve images using a local cache table of PAR URLs, encapsulating the details of using cached PAR URLs or creating new ones as necessary.

- Understanding Object Bucket Visibility
  Understand when to use public or private object buckets and how your app accesses private files.
- Creating Pre-Authenticated Requests
  Create short-lived pre-authenticated request URLs so clients can access private bucket files directly.
- Using Pre-Authenticated Requests
  Use cached PAR URLs to let OCI serve private bucket images directly when possible.

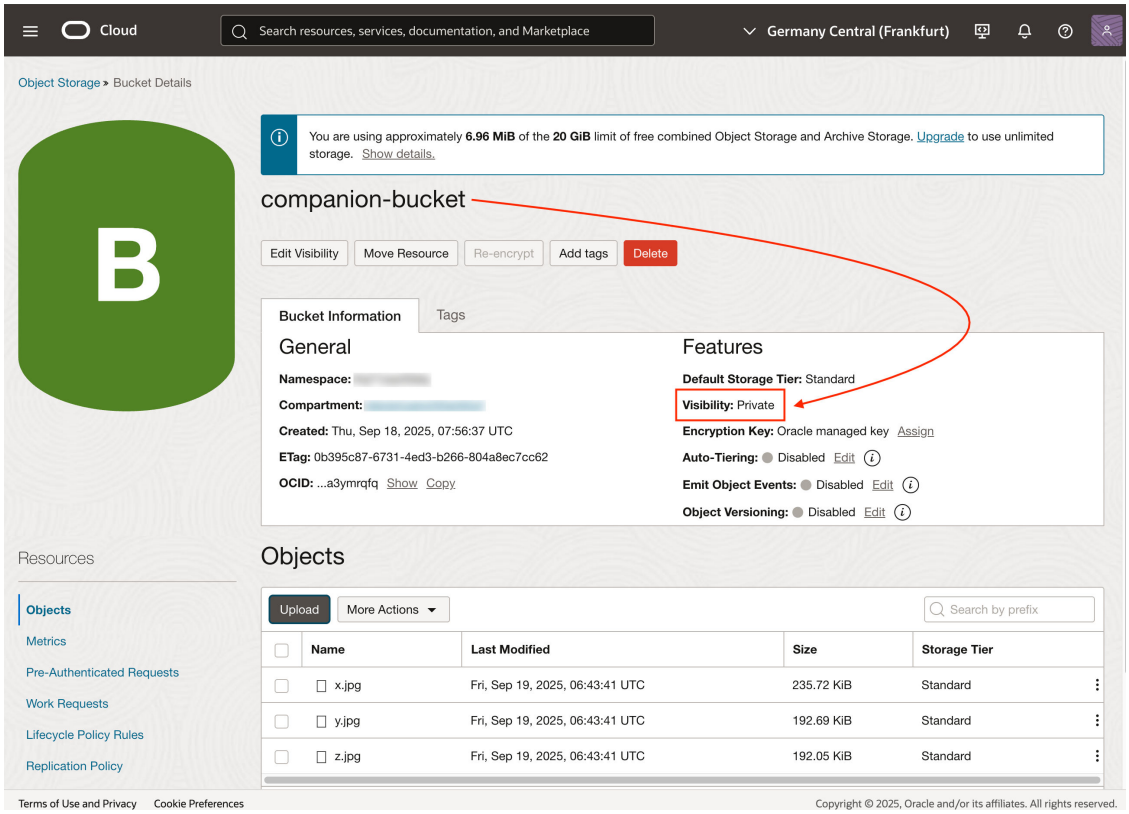
16.9.8.1 Understanding Object Bucket Visibility

Understand when to use public or private object buckets and how your app accesses private files.

You set the *visibility* of an Oracle Cloud Infrastructure (OCI) object bucket. A *public* bucket is for web site assets that any internet user can access. Use a *private* bucket for other use cases: it only allows *authenticated* access. An *OCI Native Authentication* web credential provides your application secure access to the bucket to put and get files. The figure shows the Oracle Cloud Infrastructure object bucket details page for `companion-bucket` and highlights its *Visibility* setting of Private.

Copyright © 2026, Oracle and/or its affiliates.

Figure 16-97 Use a Private Object Bucket Unless It Stores Public Web Site Assets



16.9.8.2 Creating Pre-Authenticated Requests

Create short-lived pre-authenticated request URLs so clients can access private bucket files directly.

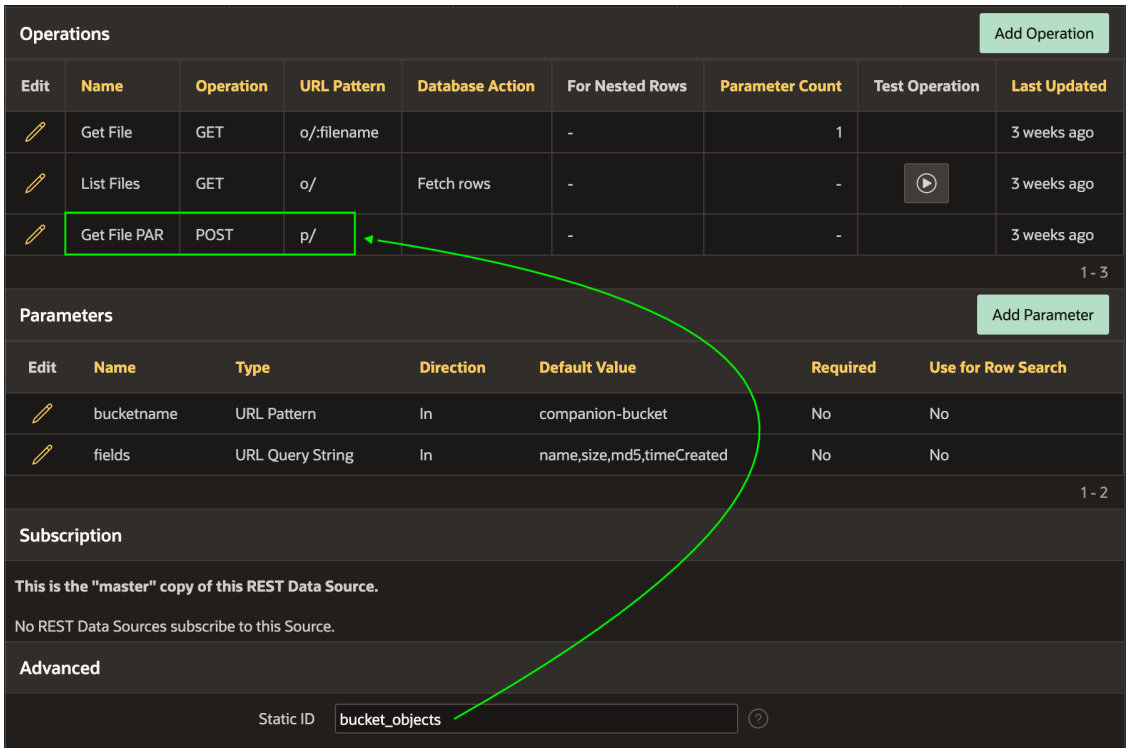
A user with privileges to manage the object bucket can create a *pre-authenticated request* (PAR) for a file. It defines an *alternative*, impossible-to-guess URL a client can use for a limited time to access a bucket file. The PAR can expire after a few minutes, or at any date in the future. During its lifetime, any client aware of the PAR URL can use it to access the particular file.

Notice below that the *Bucket Objects* REST Data Source with Static ID `bucket_objects` has a *Get File PAR* `POST` operation defined. The operation's Static ID is `get_file_par`. When its *URL Pattern* of `p/` gets appended to the REST Data Source's base URL it becomes:

```
https://objectstorage.eu-frankfurt-1.oraclecloud.com/n/x1y2z3a4b5c6/b/
companion-bucket/p/
```


Copyright © 2026, Oracle and/or its affiliates.

Figure 16-98 Get File PAR Operation Has Base URL for Creating Pre-Authenticated Request



The `CREATE_OBJECT_PAR` procedure below shows how your app can create a pre-authenticated request for the file identified by the `p_bucket_name` and `p_file_name` passed in. It accepts a `p_ttl_seconds` parameter to set the number of seconds the PAR URL will be usable, defaulting to 120 (two minutes). It calls the `BUCKET_URL_FOR` helper function to get the absolute URL to the create PAR endpoint and web credential static ID to use to make the `POST` call below.

Then, it computes the expiration date by adding the `p_ttl_seconds` to the `systimestamp` and formats the result using the ISO 8601 standard. It includes this as the value of the `timeExpires` property in the JSON document it creates to send as the Create PAR request body. That JSON also includes the `accessType` of `ObjectRead` and `objectName` with the file name. The procedure returns the new `p_par_url` and `p_expires_utc` expiration date in `OUT` parameters.

```
-- In package eba_demo_woodshr_oci
--------------------------------------------------------------------------------
-
-- Create a short-lived PAR (per object) and return URL + expiry.
--------------------------------------------------------------------------------
-
procedure create_object_par(
    p_bucket_name         in  varchar2,
    p_file_name           in  varchar2,
    p_ttl_seconds         in  pls_integer default 120,
    p_par_url             out varchar2,
    p_expires_utc         out timestamp with time zone,
    p_module_static_id    in  varchar2 default 'bucket_objects',
    p_operation_static_id in  varchar2 default 'get_file_par')
```


Copyright © 2026, Oracle and/or its affiliates.

```
is
    l_url         varchar2(32767);
    l_cred        varchar2(4000);
    l_body        clob;
    l_resp        clob;
    l_access_uri  varchar2(32767);
    l_full_path   varchar2(32767);
    l_host        varchar2(1024);
    l_expires_str varchar2(64);
    l_resp_json   json_object_t;
begin
    bucket_url_for(
        p_module_static_id     => p_module_static_id,
        p_operation_static_id  => p_operation_static_id,
        p_bucket_name          => p_bucket_name,
        p_file_name            => p_file_name,
        p_credential_static_id => l_cred,
        p_file_url             => l_url);

    -- Compute expiration date by adding p_ttl_second to
    -- systimestamp and formatting using ISO 8601 for REST payload
    p_expires_utc := (systimestamp at time zone 'UTC')
                     + numtodsinterval(greatest(p_ttl_seconds, 1), 'SECOND');
    l_expires_str := to_char(p_expires_utc, 'YYYY-MM-DD"T"HH24:MI:SS"Z"');

    -- Build the JSON payload to send in the Create PAR request body
    select json_object(
             'name'        value ('par-'||replace(p_file_name,'/','_')),
             'accessType'  value 'ObjectRead',
             'objectName'  value p_file_name,
             'timeExpires' value l_expires_str
           returning clob)
      into l_body
      from dual;

    apex_web_service.set_request_headers(
        p_name_01 => 'Content-Type', p_value_01 => 'application/json',
        p_name_02 => 'Accept',       p_value_02 => 'application/json',
        p_reset   => true);

    l_resp := apex_web_service.make_rest_request(
                p_credential_static_id => l_cred,
                p_http_method          => 'POST',
                p_url                  => l_url,
                p_body                 => l_body);

    if apex_web_service.g_status_code not in (200,201) then
        raise_application_error(-20001,
            'Create PAR failed: HTTP '||apex_web_service.g_status_code);
    end if;

    l_resp_json  := json_object_t(l_resp);
    l_access_uri := l_resp_json.get_string('accessUri');
    l_full_path  := l_resp_json.get_string('fullPath');

    if l_full_path is not null then
```


Copyright © 2026, Oracle and/or its affiliates.

```
        p_par_url := l_full_path;
    elsif l_access_uri is not null then
        l_host := regexp_substr(l_url, '^https?://[^/]+');
        p_par_url := case when substr(l_access_uri,1,1)='/' then l_host||
l_access_uri
                          else l_access_uri end;
    else
        raise_application_error(-20001,
            'Create PAR failed: neither accessUri nor fullPath returned');
    end if;
end create_object_par;
```

16.9.8.3 Using Pre-Authenticated Requests

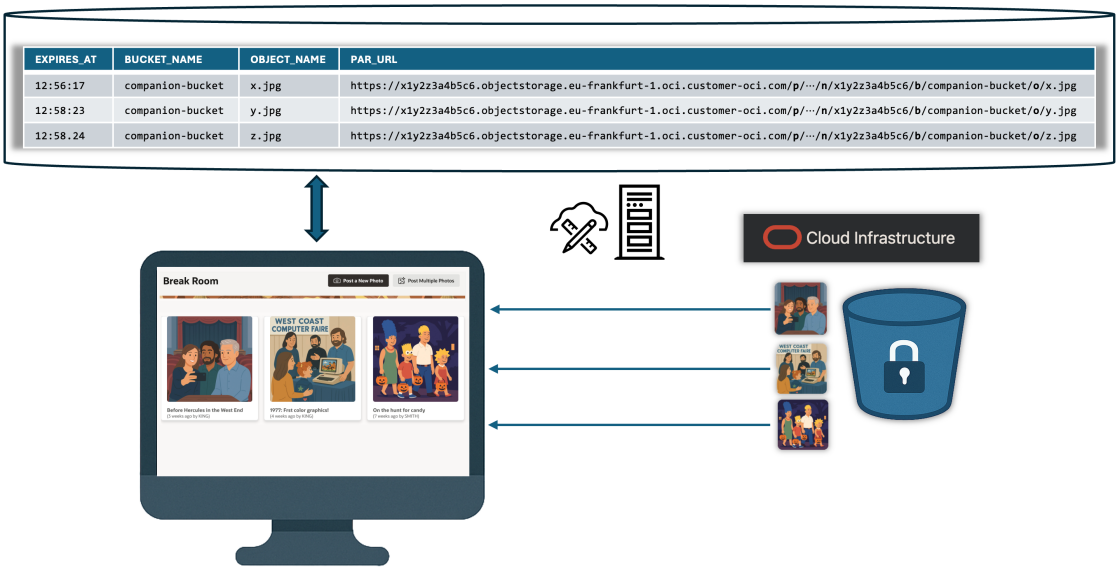
Use cached PAR URLs to let OCI serve private bucket images directly when possible.

You could optimize app image asset serving using PAR URLs your application creates. As shown below, your app could use a local table to store PAR URLs and expiration dates for image files like `x.jpg`, `y.jpg`, and `z.jpg` files. It could use these PAR URLs directly if they are still valid, or alternatively create a inline image serving page that encapsulates working with the PAR URL for a requested file:

- If the file's PAR URL in the cache table is still valid, use it
- If not, create a new PAR URL for the file requested, and save it in the cache table
- Redirect the browser to the PAR URL.

An automation could run periodically to clean up the expired entries in the cache table. Using an image serving strategy involving PAR URLs, you can defer most or all of the image servring to the object bucket in OCI. The diagram shows how a local table could store the `PAR_URL` and `EXPIRES_AT` time for different file `OBJECT_NAME` entries. Your APEX app could use the cached PAR URLs to delegate image downloading directly to the OCI object bucket.

Figure 16-99 Strategy to Use PAR URLs to Serve Object Bucket Images Directly



Copyright © 2026, Oracle and/or its affiliates.

17

Loading External Data

Load external data from Excel, XML, JSON, or CSV files using declarative pages, page processes, or PL/SQL APIs.

Create a Data Load Definition to set up a profile that describes how external data in Excel, XML, JSON, or CSV files should be loaded. Use it to load external data into a table or collection:

- *declaratively* – using the native *Data Load* page process, or
- *programmatically* – with the `LOAD_DATA` function in the `APEX_DATA_LOADING` package.

Pair the *file profile* it represents with `APEX_DATA_PARSER.PARSE` to *preview* data or process it in custom ways. The same function can also infer a file profile at *runtime* to visualize or process external data with an unknown structure.

Create a Data Loading page using the wizard that lets users upload an external data file, preview the data, and load it. It combines all of the above features. When necessary, the APEX_ZIP package lets you process the contents of a compressed archive for bulk data loading use cases.

- Exploring External Data Examples
  Review external employee data examples in Excel, CSV, JSON, and XML formats.
- Creating a Data Load Definition
  Create a *Data Load Definition* using the wizard in *Shared Components*.
- Augmenting Data Load Profile Columns
  Add data profile columns to a Data Load Definition using SQL expressions or queries.
- Generating and Using a Data Loading Page
  Generate a Data Loading page with the Create Page Wizard to let users load data by choosing a file.
- Parsing Data Without Data Load Definition
  Parse external data without a Data Load Definition by letting APEX discover the file structure at runtime.
- Loading Multiple Files Including ZIPs
  Load employee data from multiple uploaded files, including ZIP archives, using Data Load Definitions and PL/SQL.

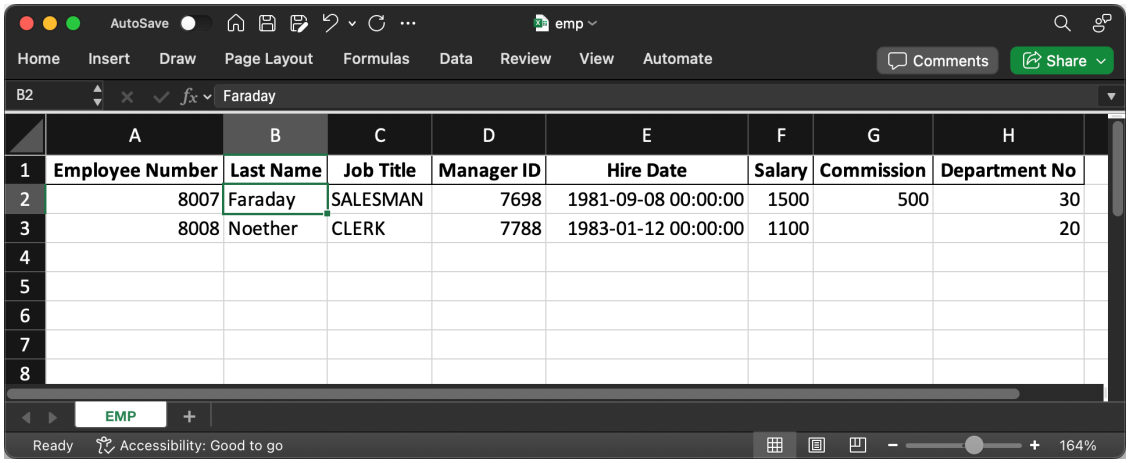
17.1 Exploring External Data Examples

Review external employee data examples in Excel, CSV, JSON, and XML formats.

In the Woods HR application, HR reps sometimes need to load employee data from external files. An Excel spreadsheet, as shown below, is the most common format they encounter. The figure shows employee information in two rows of a spreadsheet featuring column headings in the first row: *Employee Number*, *Last Name*, *Job Title*, *Manager ID*, *Hire Date*, *Salary*, *Commission*, and *Department No*.

Copyright © 2026, Oracle and/or its affiliates.

Figure 17-1 External Data in an Excel Spreadsheet



	A	B	C	D	E	F	G	H
1	Employee Number	Last Name	Job Title	Manager ID	Hire Date	Salary	Commission	Department No
2	8007	Faraday	SALESMAN	7698	1981-09-08 00:00:00	1500	500	30
3	8008	Noether	CLERK	7788	1983-01-12 00:00:00	1100		20
4								
5								
6								
7								
8								

However, sometimes the employee data arrives in a text file in comma-separated value (CSV) format:

```
WORKER_NO,LAST_NAME,POSITION,BOSS_NO,ENTRY_DATE,PAY_RATE,BONUS_AMOUNT,SECTION_
NO
8001,HOPPER,CLERK,7902,2025-03-14T00:00:00,900,,20
8002,CURIE,SALESMAN,7698,2025-11-05T00:00:00,1350,1200,30
```

Certain external systems export their data in JSON format:

```
{
    "items": [
        {
            "employeeId": 8003,
            "fullName": "GALLO",
            "positionTitle": "MANAGER",
            "reportsTo": 7839,
            "startDate": "2025-04-22",
            "salaryAmount": 2450,
            "commissionValue": "",
            "departmentCode": 10
        },
        {
            "employeeId": 8004,
            "fullName": "DARWIN",
            "positionTitle": "SALESMAN",
            "reportsTo": 7698,
            "startDate": "2025-09-10",
            "salaryAmount": 1600,
            "commissionValue": 300,
            "departmentCode": 30
        }
    ]
}
```


Copyright © 2026, Oracle and/or its affiliates.

Others use the Extensible Markup Language (XML) instead:

```
<?xml version="1.0"?>
<ROWSET>
    <ROW>
        <STAFF-ID>8005</STAFF-ID>
        <SURNAME>MENDEL</SURNAME>
        <ROLE>ANALYST</ROLE>
        <JOIN-DATE>2025-02-11T00:00:00</JOIN-DATE>
        <BASE-PAY>3200</BASE-PAY>
        <DIVISION>10</DIVISION>
    </ROW>
    <ROW>
        <STAFF-ID>8006</STAFF-ID>
        <SURNAME>LOVELACE</SURNAME>
        <ROLE>SALESMAN</ROLE>
        <SUPERVISOR-ID>7839</SUPERVISOR-ID>
        <JOIN-DATE>2025-08-27T00:00:00</JOIN-DATE>
        <BASE-PAY>2750</BASE-PAY>
        <ADDITIONAL_COMP>1300</ADDITIONAL_COMP>
        <DIVISION>30</DIVISION>
    </ROW>
</ROWSET>
```

Using a *Data Load Definition* shared component, your app can make quick work of loading external data in any of these formats.

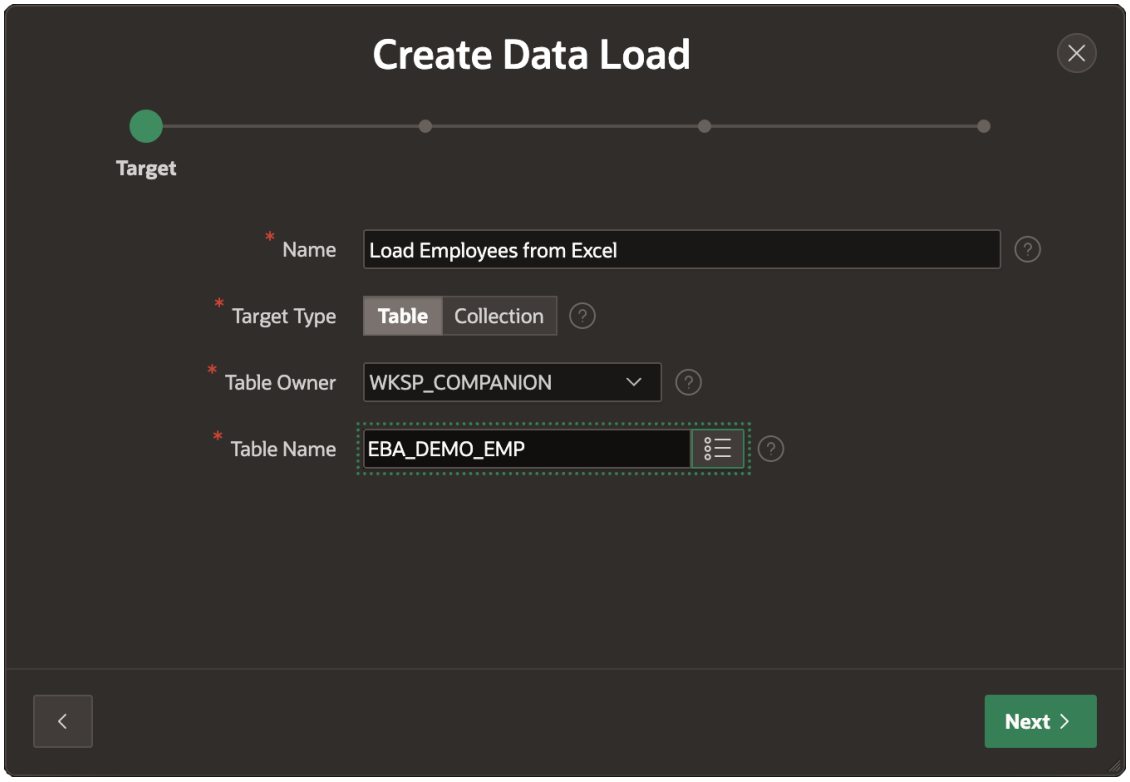
17.2 Creating a Data Load Definition

Create a *Data Load Definition* using the wizard in *Shared Components*.

As shown below, you can target a table or collection, and specify the target object name. Here, your new *Load Employees from Excel* definition loads employee data into the `EBA_DEMO_EMP` table.

Copyright © 2026, Oracle and/or its affiliates.

Figure 17-2 Creating a Data Load Definition for an Excel Spreadsheet



On the second step, upload an example file. It can be an Excel spreadsheet, CSV, JSON, or XML file.

> **Tip**
>
> When a spreadsheet has *multiple* sheets, pick the one to load with the *Select Sheet* list that appears.

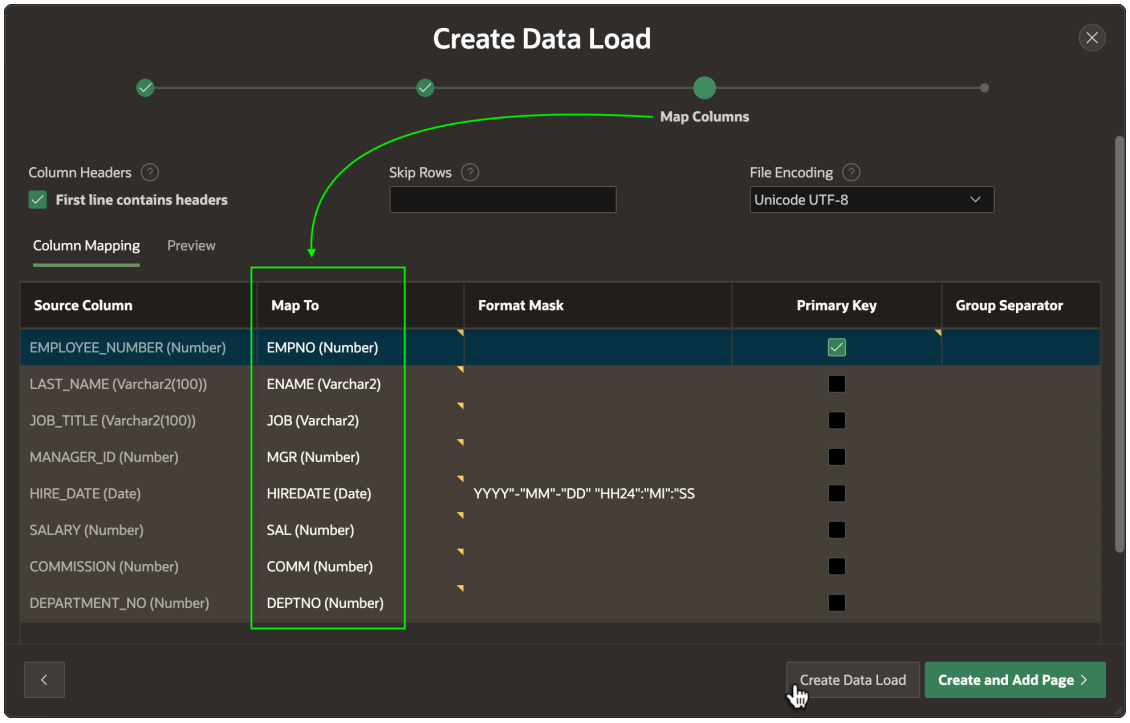
When you proceed, the wizard uploads and analyzes the file's contents. It detects names, data types, and format masks as needed.

Then, as shown below, the *Map Columns* step shows the discovered *Source Columns* in a grid. Use the select list in each row of the *Map To* column to pick a target column for the data you want to load.

> **Tip**
>
> To ignore a source column, leave *Map To* blank in that row.

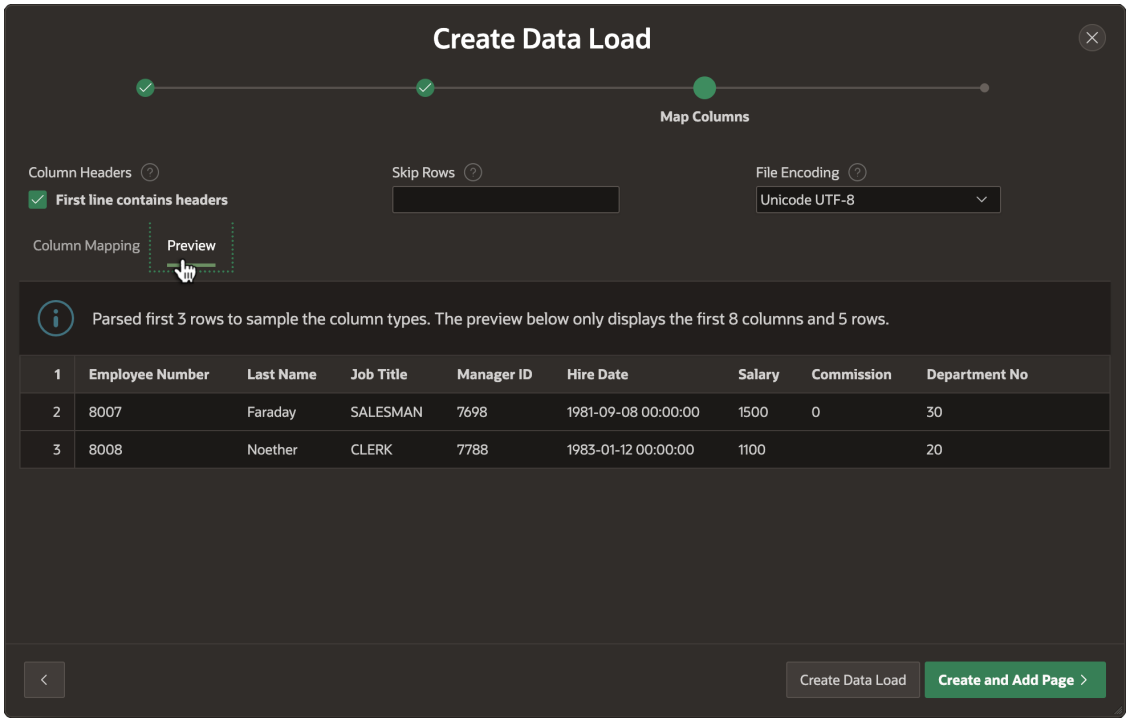
Copyright © 2026, Oracle and/or its affiliates.

Figure 17-3 Mapping Source Columns from External Data to Target Columns



To preview the data, visit the *Preview* tab as shown below. Then click *(Create Data Load)* to complete the wizard.

Figure 17-4 Reviewing Data Load Definition Results By Previewing Data



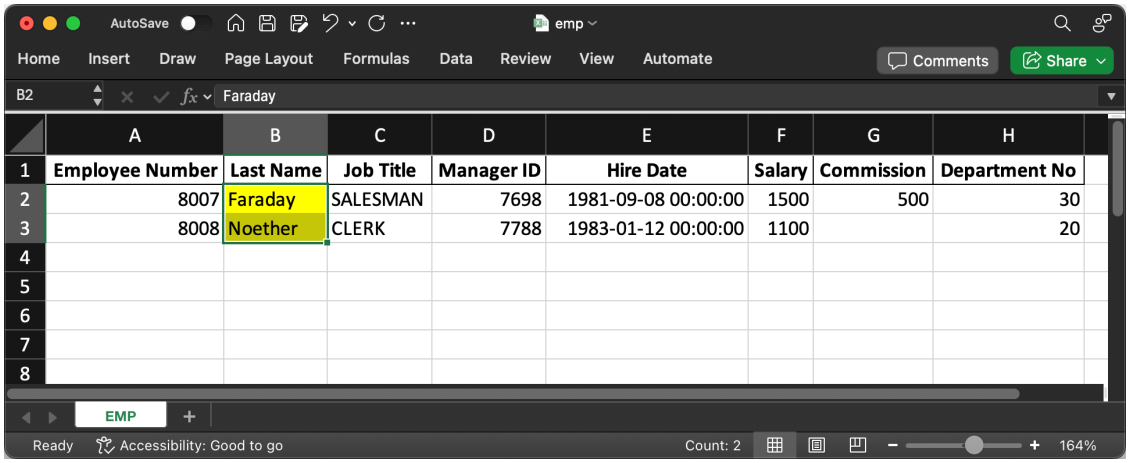
Copyright © 2026, Oracle and/or its affiliates.

17.3 Augmenting Data Load Profile Columns

Add data profile columns to a Data Load Definition using SQL expressions or queries.

For example, as shown below, the *Last Name* column in the Excel spreadsheet contains names in mixed case like *Faraday* and *Noether*.

Figure 17-5 Excel Spreadsheet Has Employee Names in Mixed Case



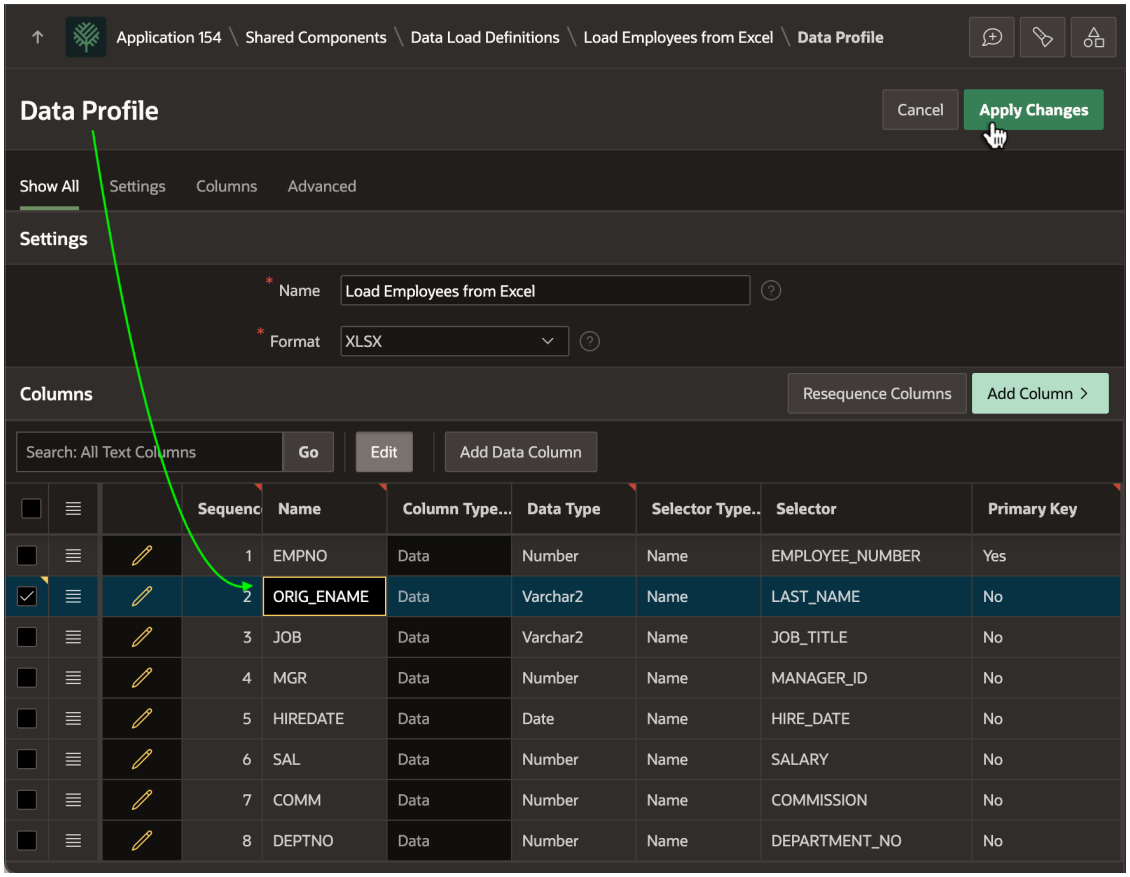
	A	B	C	D	E	F	G	H
1	Employee Number	Last Name	Job Title	Manager ID	Hire Date	Salary	Commission	Department No
2	8007	Faraday	SALESMAN	7698	1981-09-08 00:00:00	1500	500	30
3	8008	Noether	CLERK	7788	1983-01-12 00:00:00	1100		20
4								
5								
6								
7								
8								

The *Woods HR* app expects the `ENAME` column data to be uppercase, so adjust the *Load Employees from Excel* data load definition to convert each employee name before loading it. Edit the data load definition and open its Data Profile. As shown below, start by using the grid to edit the existing `ENAME` column to change its name to `ORIG_ENAME`, then click *(Apply Changes)*.

> **Tip**
>
> To stay on the same page when clicking *(Apply Changes)*, tick the *Stay on Page* checkbox in the margin prior to clicking. App Builder remembers your preference.

Copyright © 2026, Oracle and/or its affiliates.

Figure 17-6 Renaming ENAME to ORIG_ENAME Data Profile Column



Next, click the *(Add Column >)* button in the Data Profile to add a new `ENAME` column, of type *SQL Expression* with *Data Type* `VARCHAR2`. Use the expression below, referencing other data profile columns names as necessary. The new `ENAME` column derives its value by uppercasing the value in the `ORIG_ENAME` column loaded from the spreadsheet.

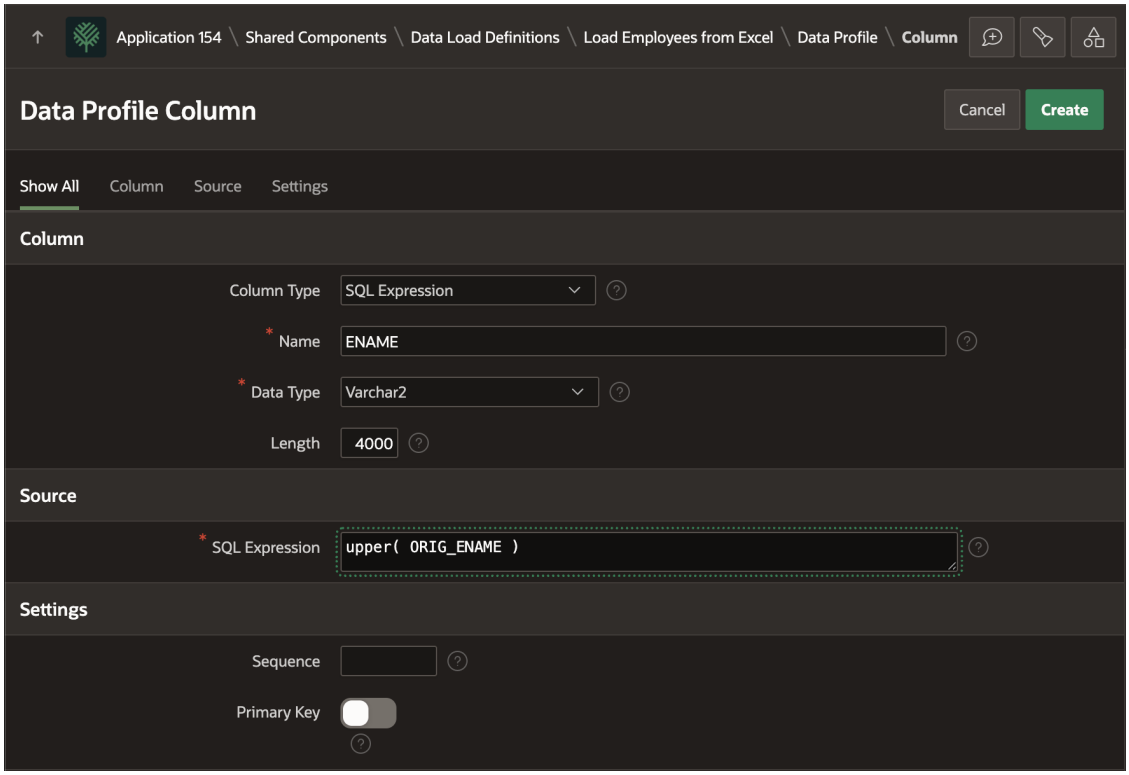
```
upper( ORIG_NAME )
```

Tip

The SQL Expression or SQL Query you enter can also be a constant value or reference the value of a *global* application item by name using the built-in `V` function (e.g. `V('G_APP_USER_EMPNO')`). Use this technique in a SQL Expression column named *`YOUR_NAME`* to load a custom value into the target table's *`YOUR_NAME`* column.

Copyright © 2026, Oracle and/or its affiliates.

Figure 17-7 Adding a SQL Expression Data Profile Column for ENAME



17.4 Generating and Using a Data Loading Page

Generate a Data Loading page with the Create Page Wizard to let users load data by choosing a file.

- Creating a Data Loading Page
  Create a data loading page by choosing *Data Loading* in the Create Page Wizard.
- Using a Data Loading Page
  Run a generated Data Loading page to upload, preview, and load external employee data.
- Understanding Data Loading Pages
  Understand the features a generated Data Loading page combines to upload, preview, and load data.

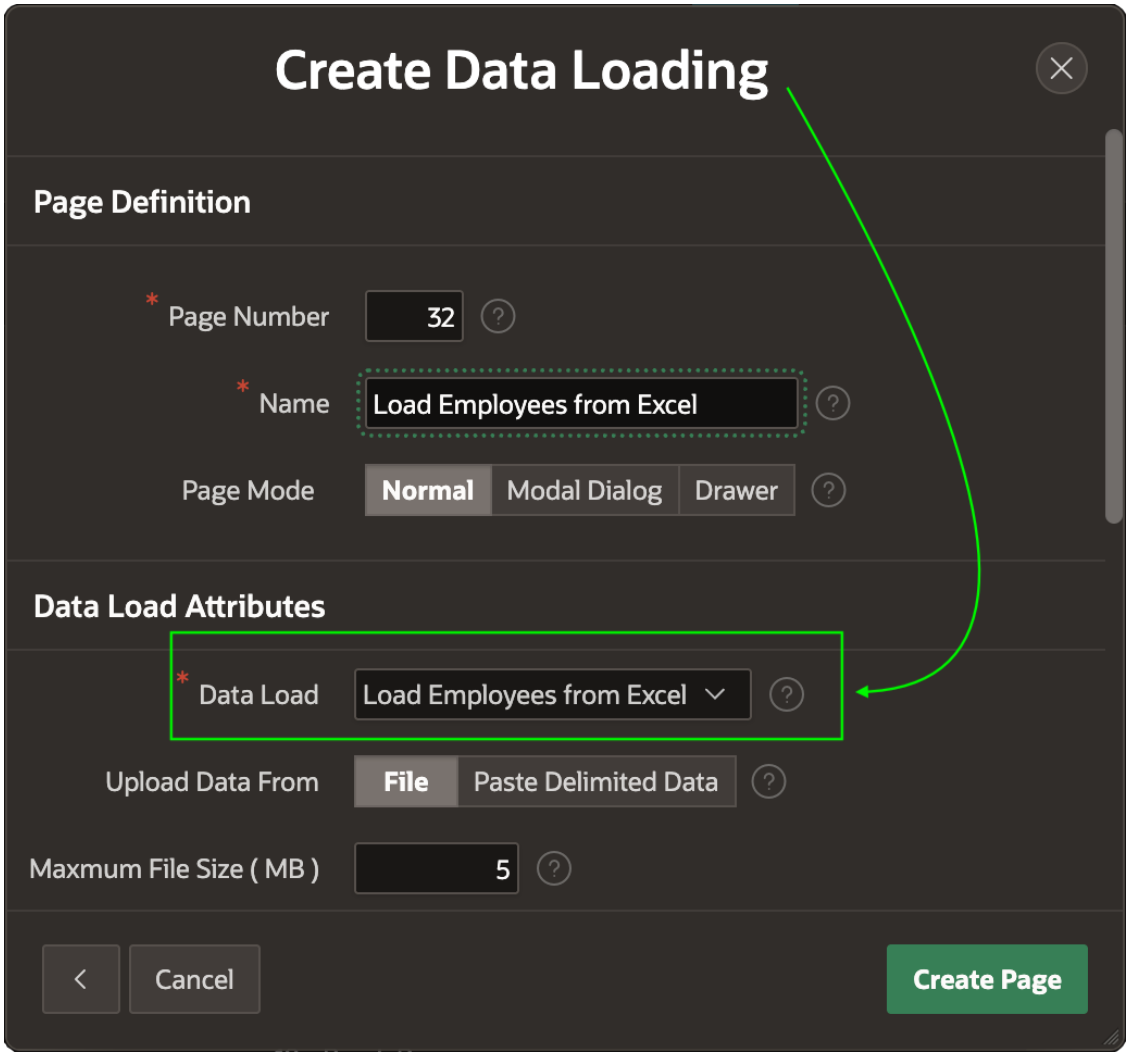
17.4.1 Creating a Data Loading Page

Create a data loading page by choosing *Data Loading* in the Create Page Wizard.

As shown below, just pick an existing *Data Load Definition* like *Load Employees from Excel* and click *(Create Page)*. While the default *Upload Data From* setting of *File* is what you need here, notice it's also possible to create a page where the user can paste in the data to upload.

Copyright © 2026, Oracle and/or its affiliates.

Figure 17-8 Creating a Data Load Page for an Existing Data Load Definition



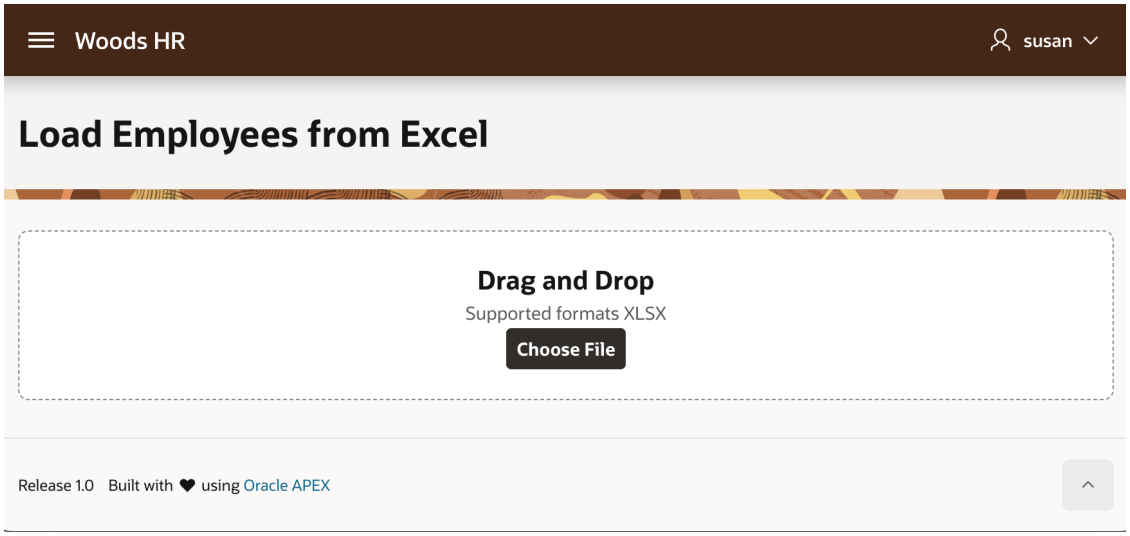
17.4.2 Using a Data Loading Page

Run a generated Data Loading page to upload, preview, and load external employee data.

Running the generated Data Loading page, it displays a file drop zone as shown below.

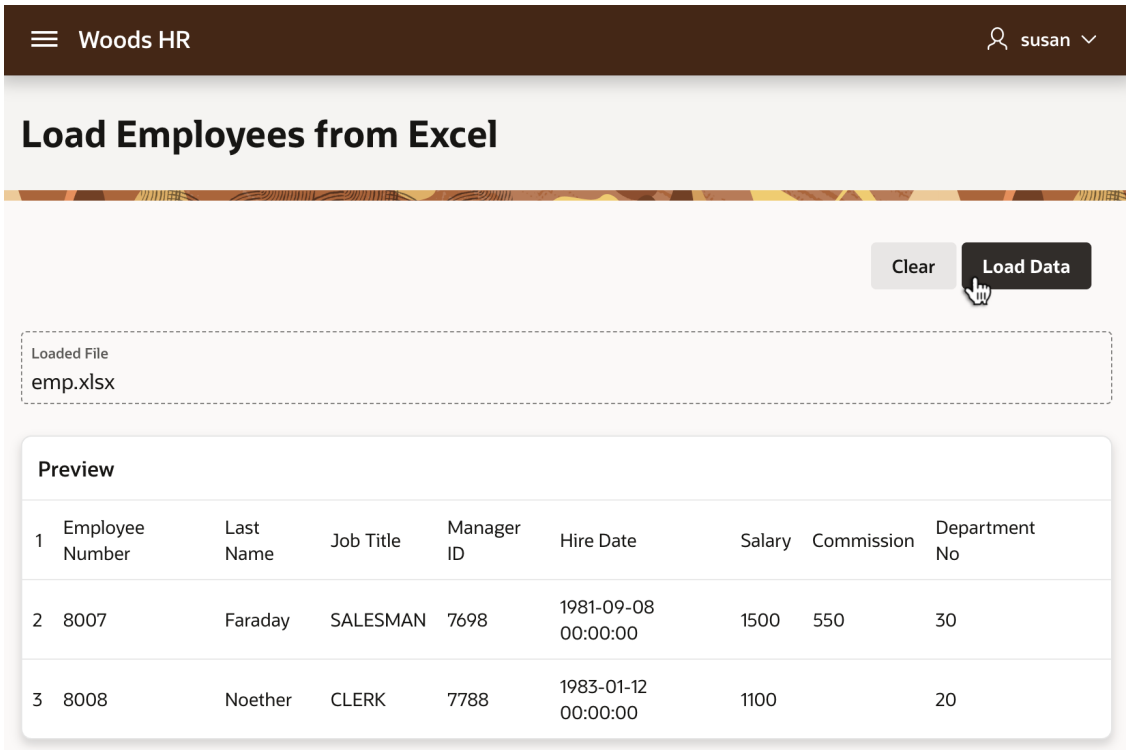
Copyright © 2026, Oracle and/or its affiliates.

Figure 17-9 Data Loading Page File Drop Zone



After dragging and dropping an `emp.xlsx` Excel spreadsheet file onto the drop zone, the page uploads the file straight away and previews the data to load. Clicking *(Load Data)* completes the operation. The figure shows the generated Data Loading page's preview of the uploaded spreadsheet data.

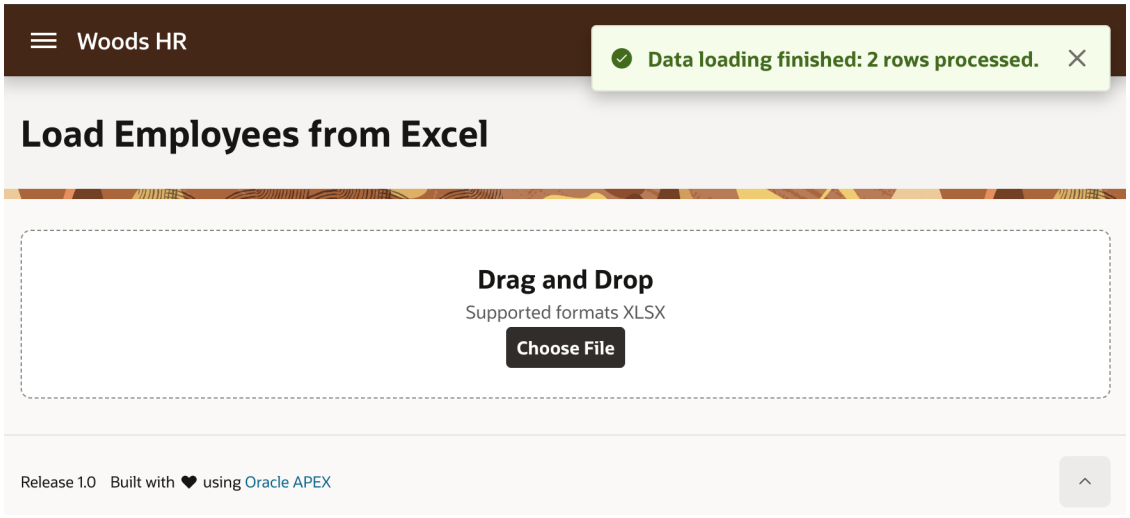
Figure 17-10 Preview of Excel Spreadsheet Data to Load



The success message shown below confirms the number of rows processed.

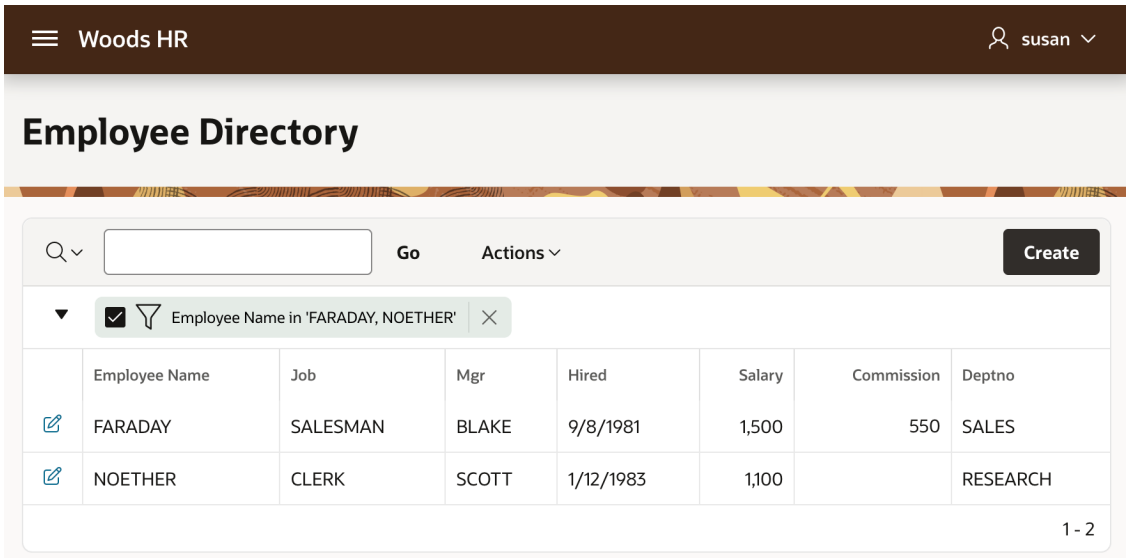
Copyright © 2026, Oracle and/or its affiliates.

Figure 17-11 Success Message Confirms Number of Rows Loaded



When Susan, the HR Representative, visits the *Employee Directory* page, she sees the two new employees, with *Employee Name* values in uppercase, so the SQL Expression data profile column explained in Augmenting Data Load Profile Columns functioned as expected. The figure shows the Woods HR Employee Directory page, with its Interactive Report filtered to show employees with names FARADAY and NOETHER.

Figure 17-12 Loaded Employees as HR Rep Susan in Employee Directory



17.4.3 Understanding Data Loading Pages

Understand the features a generated Data Loading page combines to upload, preview, and load data.

A generated *Data Loading* page integrates the following features:

Copyright © 2026, Oracle and/or its affiliates.

- *File Upload* page item to let the user upload a file
- Validation to ensure the user only submits expected file types
- Worksheet selector when spreadsheet has multiple sheets
- Data Preview to verify the data is parsing correctly
- Declarative data loading using the native *Data Load* page process.
- Specifying the File and Uploading It
  Understand how the generated Data Loading page uploads a selected file and stores it temporarily..
- Validating the Uploaded File Type
  Understand how the generated Data Loading page validates the uploaded file before loading it.
- Letting User Select Worksheet
  Understand how the generated Data Loading page lets users choose a worksheet when an Excel file has more than one.
- Previewing External Data Before Loading
  Understand how the generated Data Loading page previews uploaded data before loading it.
- Loading Data with Native Page Process
  Understand how the generated Data Loading page uses a native Data Loading process to load the uploaded file.

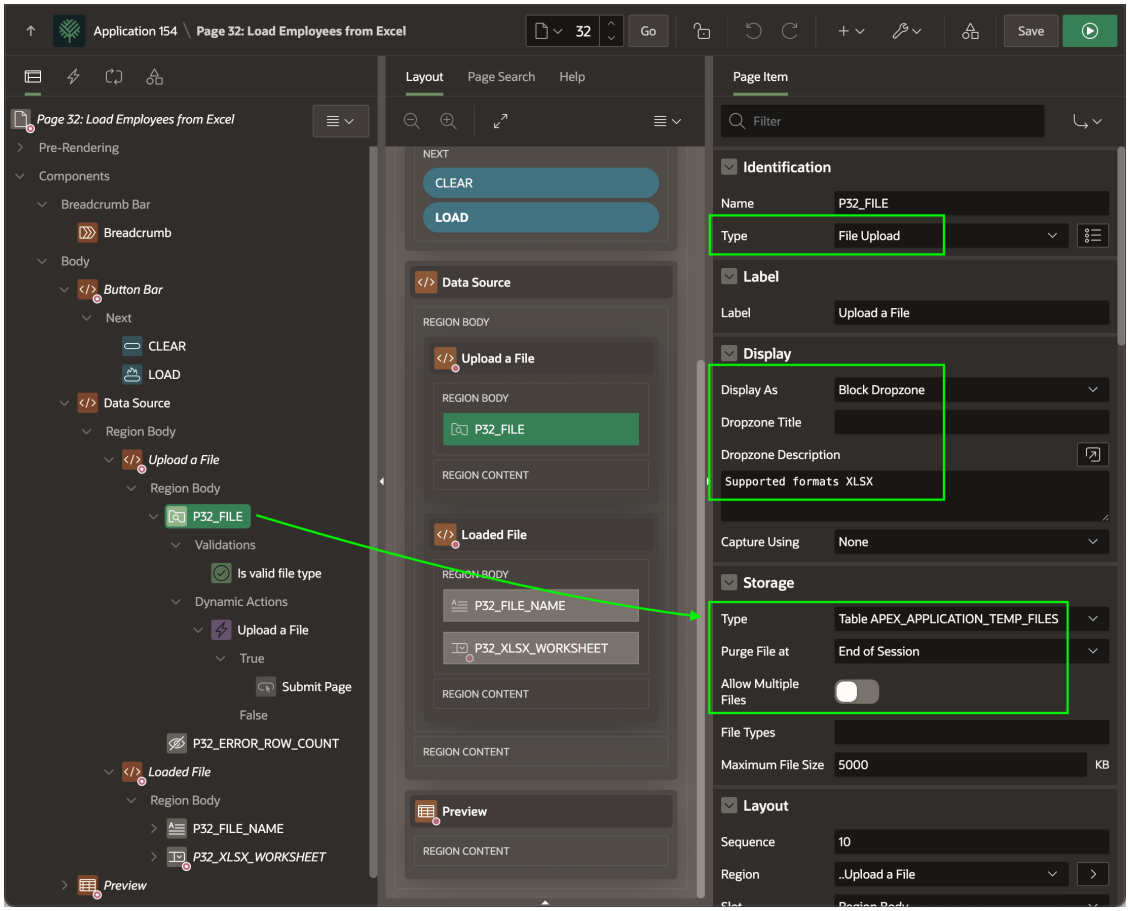
17.4.3.1 Specifying the File and Uploading It

Understand how the generated Data Loading page uploads a selected file and stores it temporarily..

The *Load Employees from Excel* page shown below is a wizard-created page. It contains the `P32_FILE` *File Upload* page item configured to display as a *Block Dropzone*. It stores the uploaded file in the *Table APEX_APPLICATION_TEMP_FILES* for the duration of the user session. By default, it's set to upload a single file at a time.

Copyright © 2026, Oracle and/or its affiliates.

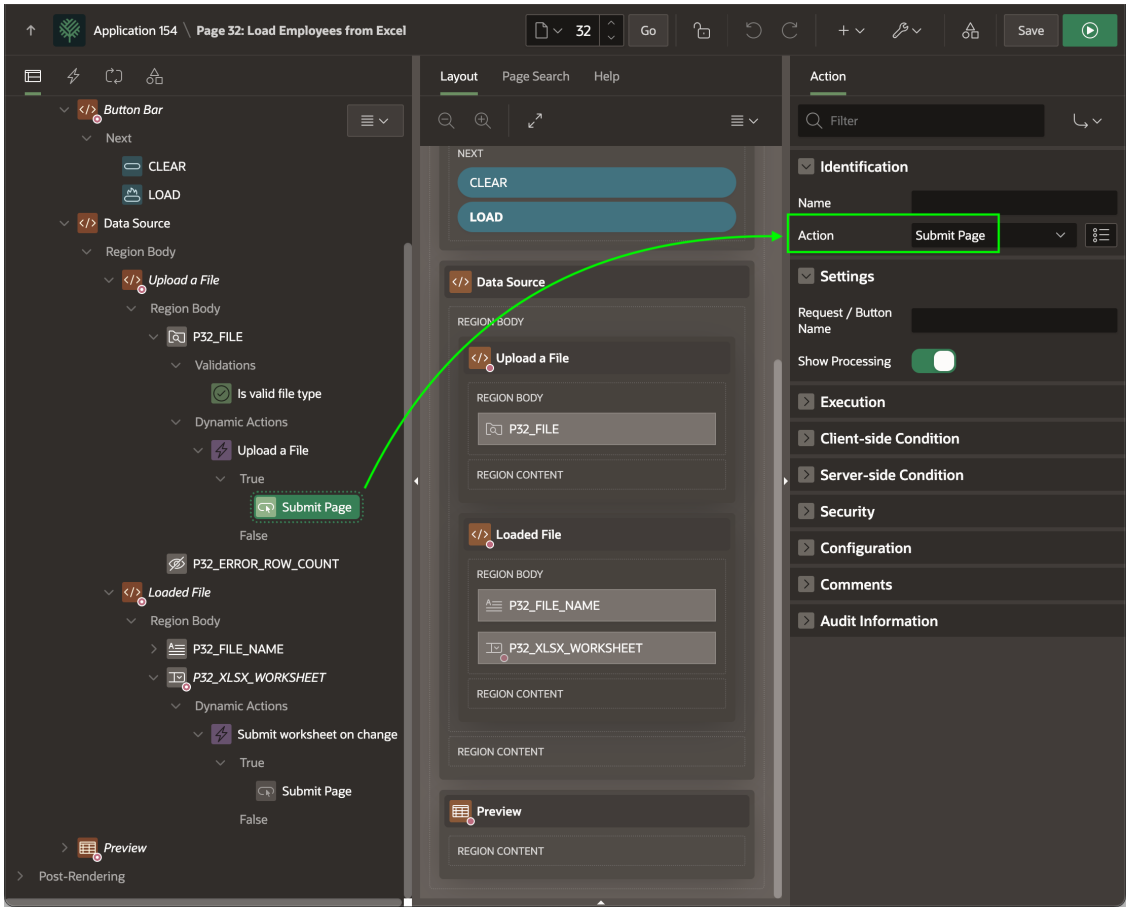
Figure 17-13 File Upload Item in Data Loading Page



The *Upload a File* dynamic action event handler reacts to the `P32_FILE` *File Upload* item's *Change* event. This happens when the user chooses a file to upload. It triggers an immediate page submit using the native *Submit Page* action. This uploads the file to the server. The figure shows the **Submit Page** dynamic action step selected in the component tree and highlights the **Submit Page** action it's configured to perform.

Copyright © 2026, Oracle and/or its affiliates.

Figure 17-14 Dynamic Action Immediately Submits File Upload

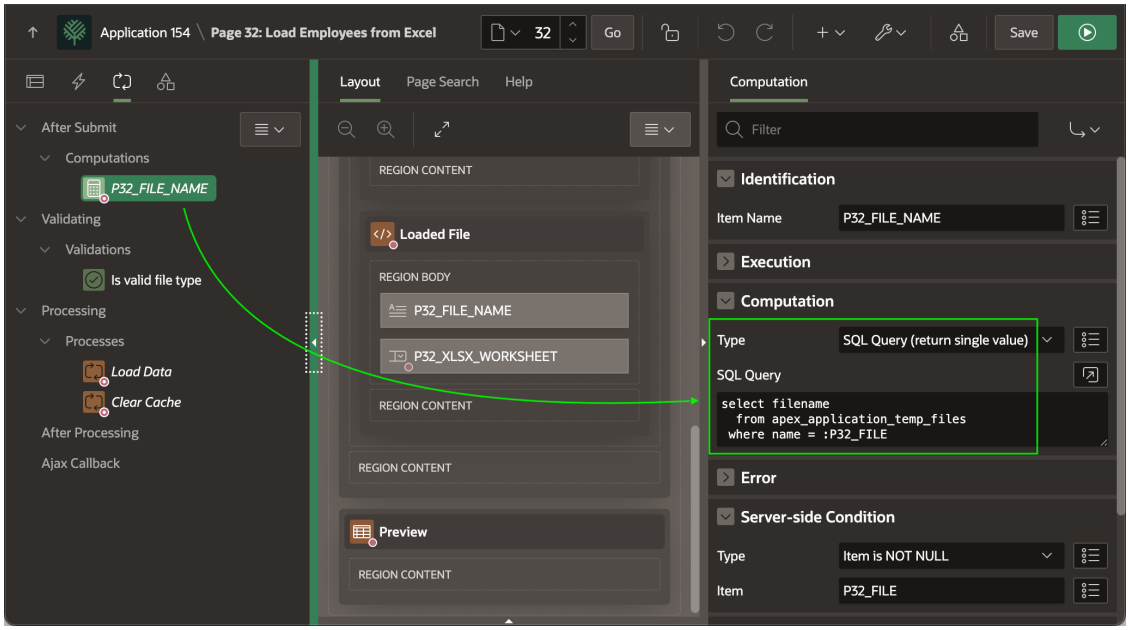


An *After Submit* computation shown below populates the display only `P32_FILE_NAME` page item with the uploaded file name. It selects the `FILENAME` column from the `APEX_APPLICATION_TEMP_FILES` table using the unique uploaded file name in the `P32_FILE` *File Upload* item.

Then, since the page contains no branches to navigate elsewhere the page re-renders after processing the page submit.

Copyright © 2026, Oracle and/or its affiliates.

Figure 17-15 After Submit Computation Populates Uploaded File Name



17.4.3.2 Validating the Uploaded File Type

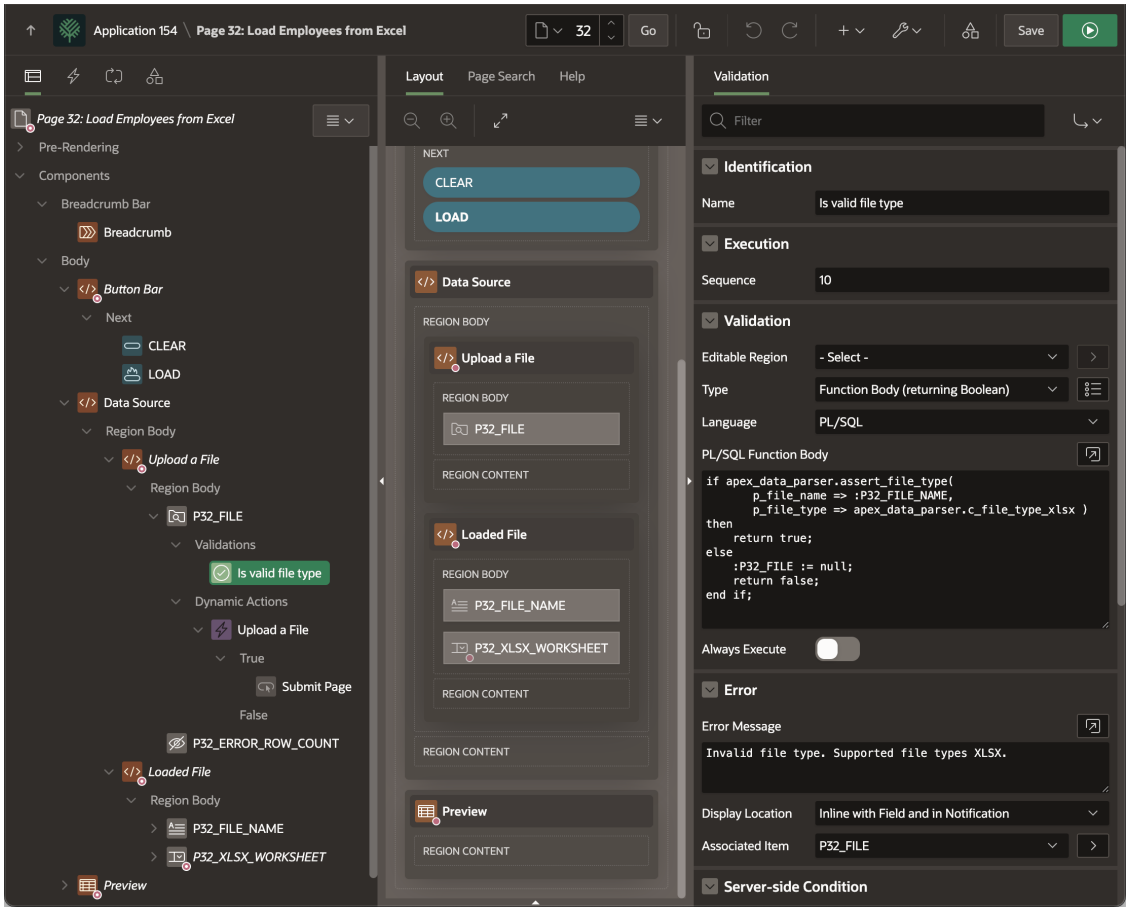
Understand how the generated Data Loading page validates the uploaded file before loading it.

A validation on the `P32_FILE` *File Upload* page item ensures only the expected file type is submitted. As shown below, it uses the following PL/SQL code that calls `ASSERT_FILE_TYPE` in the `APEX_DATA_PARSER` package to test if the submitted file is an Excel spreadsheet. It passes in the value of `P32_FILE_NAME`, the *File Upload* item. It contains the unique uploaded file name that identifies the file in the `APEX_APPLICATION_TEMP_FILES` table. Notice it references the `C_FILE_TYPE_XLSX` constant in the same package to indicate the type of file to check. If true, the validation succeeds. If false, it clears the *File Upload* item by assigning `null` to it and returns false to signal the validation failed.

```
if apex_data_parser.assert_file_type(
       p_file_name => :P32_FILE_NAME,
       p_file_type => apex_data_parser.c_file_type_xlsx )
then
    return true;
else
    :P32_FILE := null;
    return false;
end if;
```


Copyright © 2026, Oracle and/or its affiliates.

Figure 17-16 Validating Allowed File Types on Submit



17.4.3.3 Letting User Select Worksheet

Understand how the generated Data Loading page lets users choose a worksheet when an Excel file has more than one.

The `P32_XLSX_WORKSHEET` select list has a server-side condition so it displays *only* when the uploaded file contains multiple worksheets. The *Function Body* type condition below references the unique uploaded file name in `P32_FILE` in a SQL query that counts the number of worksheets. It joins the `APEX_APPLICATION_TEMP_FILES` table with the results of the `GET_XLSX_WORKSHEETS` function in the `APEX_DATA_PARSER` package. Then, it returns *true* to display the select list if the sheet count is greater than one.

```
declare
    l_sheet_count number;
begin
    select count(*)
      into l_sheet_count
      from apex_application_temp_files f,
           table( apex_data_parser.get_xlsx_worksheets(
                    p_content => f.blob_content ) ) p
     where f.name = :P32_FILE;

     -- display if the XSLX file contains multiple worksheets
```


Copyright © 2026, Oracle and/or its affiliates.

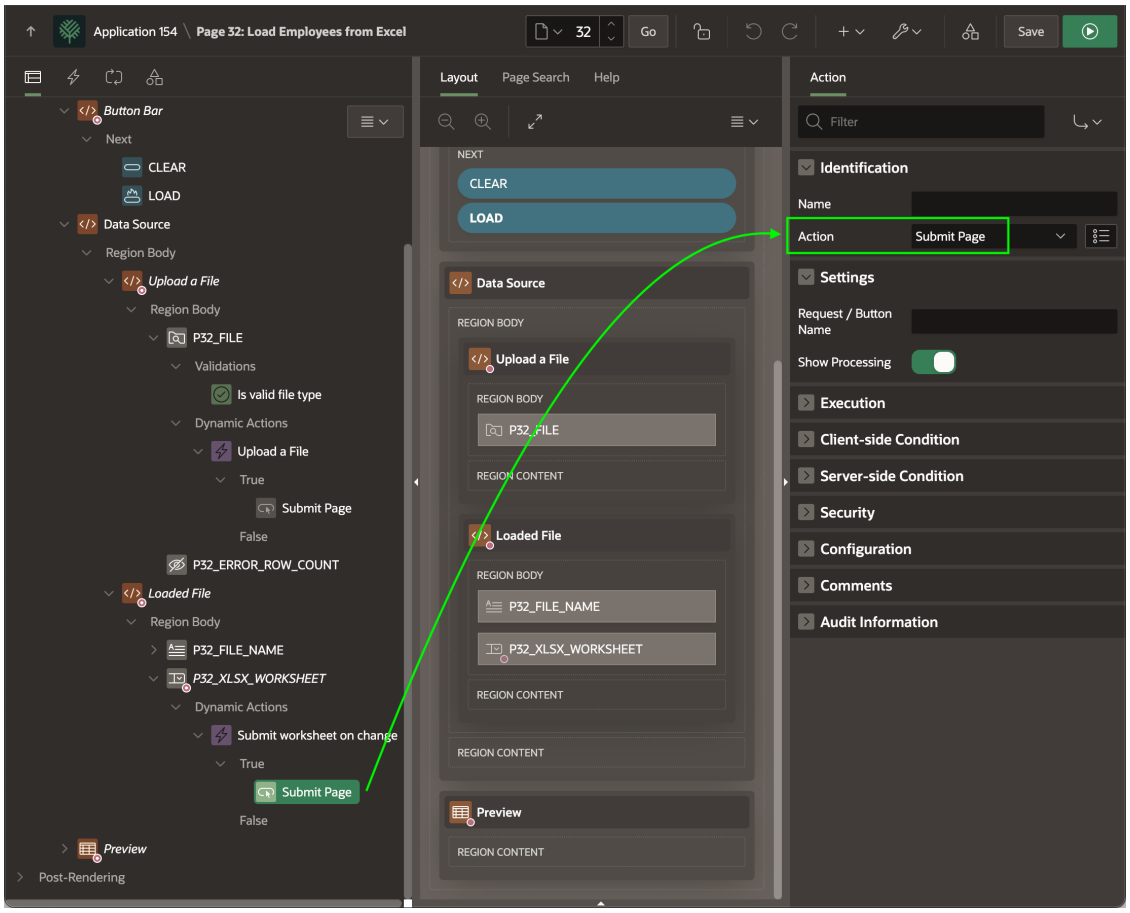
```
    return ( l_sheet_count > 1 );
exception
    when others then
        return false;
end;
```

The select list uses the same function and `P32_FILE` bind variable in the following query to show the list of worksheet names for the user to pick from:

```
select p.sheet_display_name,
       p.sheet_file_name
  from apex_application_temp_files f,
       table( apex_data_parser.get_xlsx_worksheets( p_content =>
f.blob_content ) ) p
 where f.name = :P32_FILE
```

Since the data to load is different on each sheet, a dynamic action shown below reacts to the value change in the `P32_XLSX_WORKSHEET_NAME` select list to submit the page. When the page re-renders after this submit, the *Preview* region displays the data preview for the newly selected sheet. The figure shows the **Submit Page** dynamic action step selected in the components tree and highlights its **Submit Page** action.

Figure 17-17 Selecting Worksheet to Load When Multiple Ones Exist



Copyright © 2026, Oracle and/or its affiliates.

17.4.3.4 Previewing External Data Before Loading

Understand how the generated Data Loading page previews uploaded data before loading it.

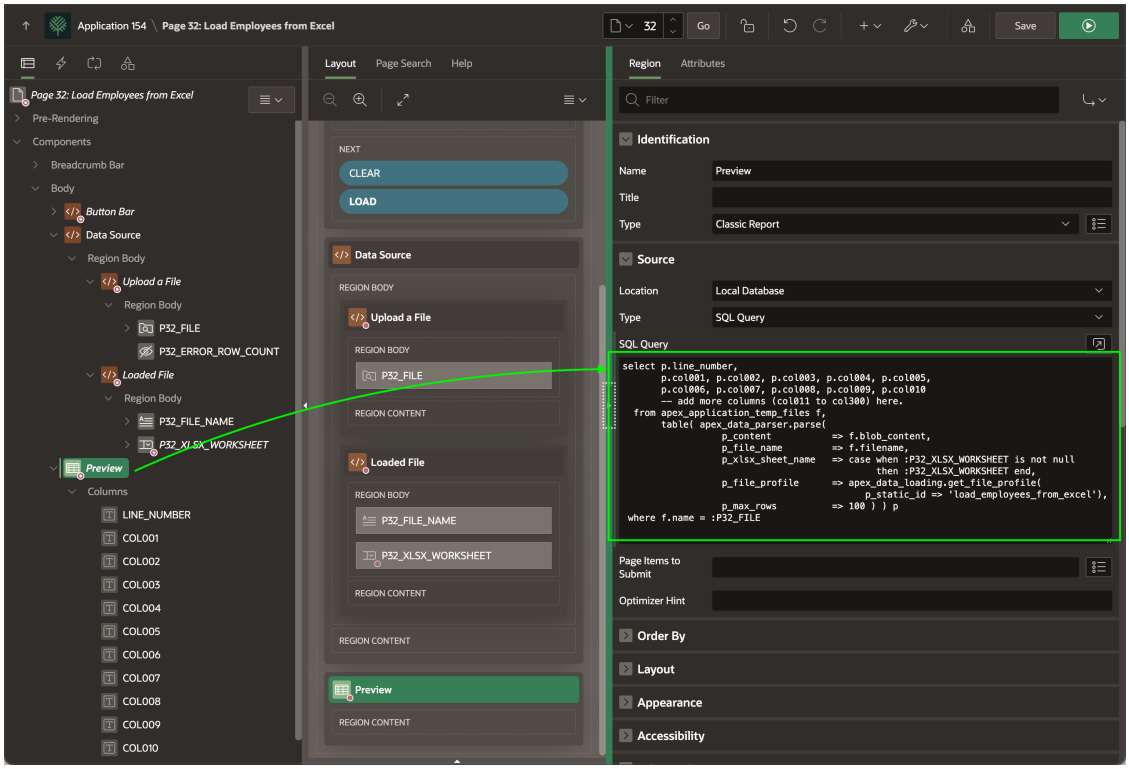
The *Preview* region is a *Classic Report* with a server-side condition to display only when the `P32_FILE` Item is not `null`. In other words, it shows only after the user uploads a file. Its query below shows up to 100 rows of data from the file using `APEX_DATA_PARSER.PARSE` to read the data using the *file profile* in the Data Load Definition with static ID `load_employees_from_excel`. The query uses the unique uploaded file name in the *File Upload* page item `P32_FILE` to access the uploaded spreadsheet `BLOB_CONTENT` from the `APEX_APPLICATION_TEMP_FILES` table. It passes in the user-selected sheet name by referencing the `P32_XLSX_WORKSHEET_NAME` page item.

```
select p.line_number,
       p.col001, p.col002, p.col003, p.col004, p.col005,
       p.col006, p.col007, p.col008, p.col009, p.col010
       -- add more columns (col011 to col300) here.
  from apex_application_temp_files f,
       table( apex_data_parser.parse(
                  p_content         => f.blob_content,
                  p_file_name       => f.filename,
                  p_xlsx_sheet_name => case when :P32_XLSX_WORKSHEET is not
null
                                          then :P32_XLSX_WORKSHEET end,
                  p_file_profile    => apex_data_loading.get_file_profile(
                                         p_static_id =>
'load_employees_from_excel'),
                  p_max_rows        => 100 ) ) p
 where f.name = :P32_FILE
```

The figure shows the Preview classic report region selected in the component tree and highlights its data source SQL query it uses to preview the data from the uploaded file.

Copyright © 2026, Oracle and/or its affiliates.

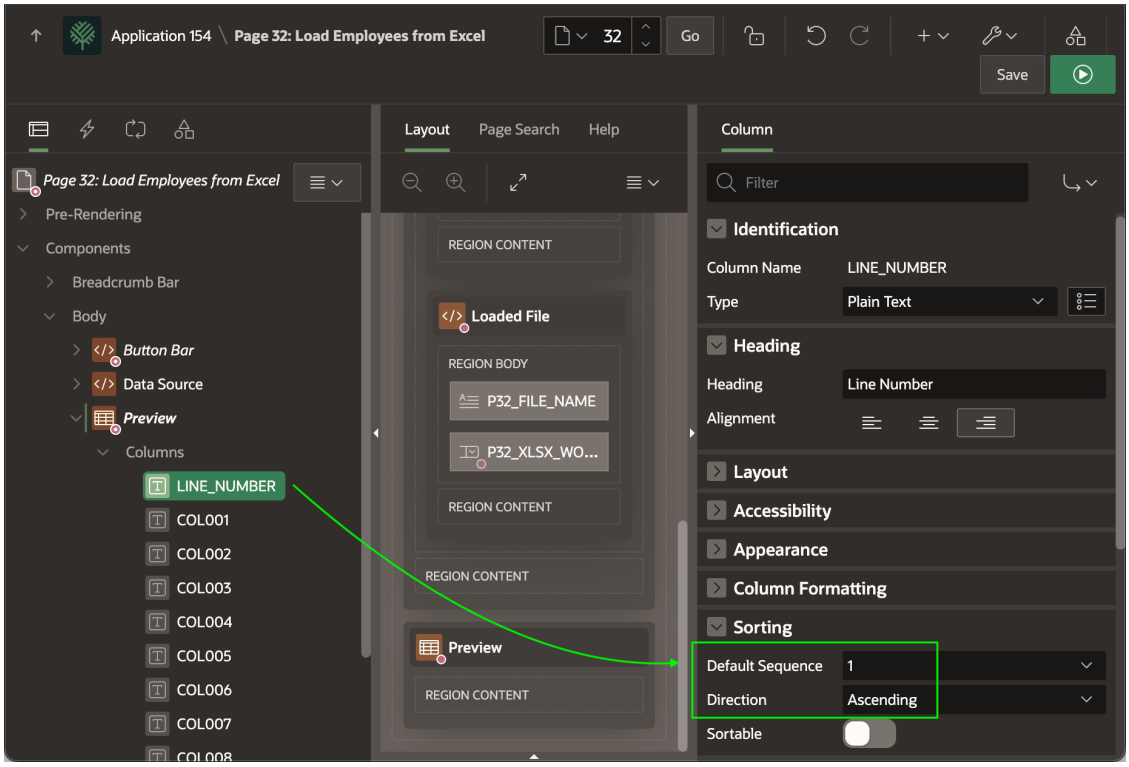
Figure 17-18 Previewing the Data to Load with APEX_DATA_PARSER.PARSE



The *Preview* region has its *Heading* > *Type* set to *None* on the *Attributes* tab of the Property Editor. As shown below, ensure the *Sorting* > *Default Sequence* of the `LINE_NUMBER` column is set to one (1) with *Ascending* sort direction so the user sees the data preview in line number order.

Copyright © 2026, Oracle and/or its affiliates.

Figure 17-19 Ensuring Preview Sorts by Line Number



17.4.3.5 Loading Data with Native Page Process

Understand how the generated Data Loading page uses a native Data Loading process to load the uploaded file.

After reviewing the data preview, when the user clicks the *(Load)* button to load the data, it submits the page and executes the *Load Data* page process. As shown below, its server-side condition ensures it runs only when the `LOAD` button is pressed.

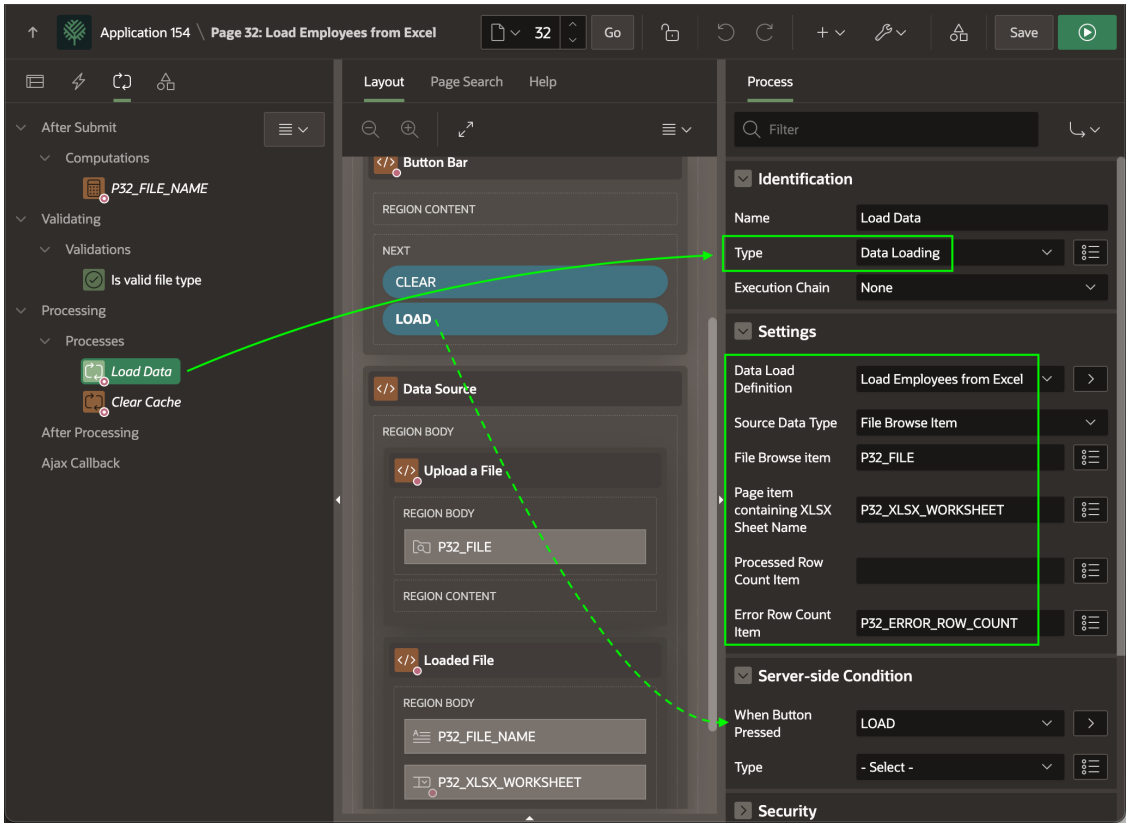
The *Load Data* page process is a native *Data Loading* process type, with its *Data Load Definition* set to *Load Employees from Excel*.

While it can also load data from a text area, or a SQL query returning a CLOB or BLOB, here it's set to use the contents of the `P32_FILE` *File Upload* item's temporary file. It configures the `P32_XSLX_WORKSHEET` item to supply the worksheet name, if any.

> **Tip**
>
> If you configure a *Success Message* on the *Load Data* page process, that takes precedence. Otherwise a *Data Loading* page process shows a default success message indicating the number of rows loaded.

Copyright © 2026, Oracle and/or its affiliates.

Figure 17-20 Configuring Native Data Load Page Process



17.5 Parsing Data Without Data Load Definition

Parse external data without a Data Load Definition by letting APEX discover the file structure at runtime.

In Previewing External Data Before Loading, the `APEX_DATA_PARSER.PARSE` function referenced an existing Data Load Definition by static ID. This is the most efficient approach if you know the structure of the data to load ahead of time. However, for more dynamic use cases you can *also* call `PARSE` to automatically discover the structure of a data file on the fly and return the data it contains in a row set with columns names from `C001` to `C300`. It handles CSV, XML, JSON, and Excel data.

After running the `PARSE` function to retrieve the data, call:

- `GET_COLUMNS` for a collection of the discovered column names and data types
- `GET_FILE_PROFILE` to retrieve the file profile in JSON format

To work with the file profile information in a PL/SQL record structure instead, pass the file profile JSON to `JSON_TO_PROFILE` to get a `T_FILE_PROFILE` record structure.

17.6 Loading Multiple Files Including ZIPs

Load employee data from multiple uploaded files, including ZIP archives, using Data Load Definitions and PL/SQL.

Copyright © 2026, Oracle and/or its affiliates.

Suppose HR representatives need to load external employee data from *multiple* files at a time. They work with different supplier systems that provide the data in Excel format, CSV, JSON, and XML. Sometimes multiple files arrive in a ZIP archive. A *Load Employees* page in the *Woods HR* app can meet this requirement by combining:

- *Data Load Definitions* for each kind of source file (Excel, CSV, JSON, and XML)
- A *File Upload* page item with *Allow Multiple Files* enabled
- A custom `EMPLOYEES_FROM_FILES` procedure that uses:
  - A `FOR` loop over the uploaded files, calling
  - `APEX_DATA_LOADING` package to load data using a Data Load Definition, and
  - `APEX_ZIP` package to process the contents of a ZIP file

- Adding XML, CSV, JSON Data Load Definitions
  Add Data Load Definitions for XML, CSV, and JSON files that all load into the same target table.
- Setting File Upload to Handle Multiple Files
  Configure a File Upload item for multiple files and track processing results in hidden page items.
- Processing Multiple Uploaded Files
  Process multiple uploaded files with PL/SQL that chooses the right Data Load Definition and expands ZIP archives.
- Uploading Multiple Data Load Files
  Upload multiple employee data files and load them together, including files inside ZIP archives.

17.6.1 Adding XML, CSV, JSON Data Load Definitions

Add Data Load Definitions for XML, CSV, and JSON files that all load into the same target table.

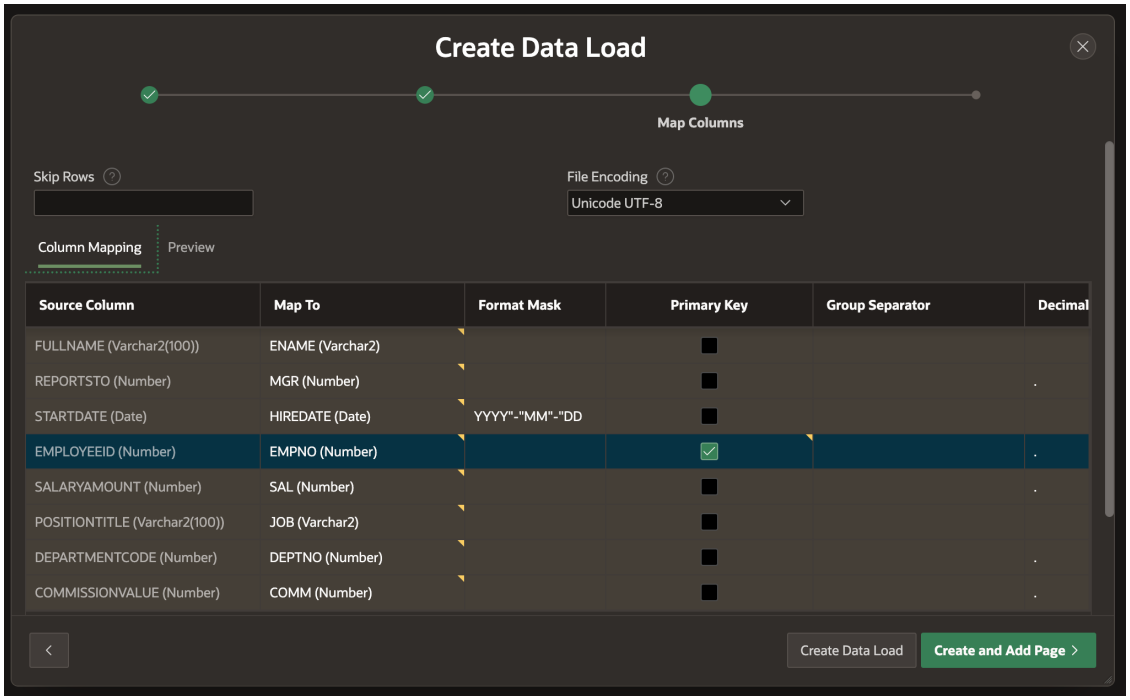
By running through the *Create Data Load Definition* wizard three more times, using the XML, CSV, and JSON files from Exploring External Data Examples, you can add the following additional shared components to your initial one for Excel spreadsheets:

- *Load Employees from XML* with Static ID `load_employees_from_xml`
- *Load Employees from CSV* with Static ID `load_employees_from_csv`
- *Load Employees from JSON* with Static ID `load_employees_from_json`

As shown below for the JSON file example, even though each file uses different nomenclature for the fields, they all target the `EBA_DEMO_EMP` table, so their *Map To* columns are the same.

Copyright © 2026, Oracle and/or its affiliates.

Figure 17-21 Creating a Data Load Definition for an Example JSON File



17.6.2 Setting File Upload to Handle Multiple Files

Configure a File Upload item for multiple files and track processing results in hidden page items.

The page's `P33_FILE` *File Upload* page item has *Allow Multiple Files* enabled. It displays as a *Block Dropzone* with a custom title and description. Uploaded files are stored only for the duration of the ***request*** in the `APEX_APPLICATION_TEMP_FILES` table.

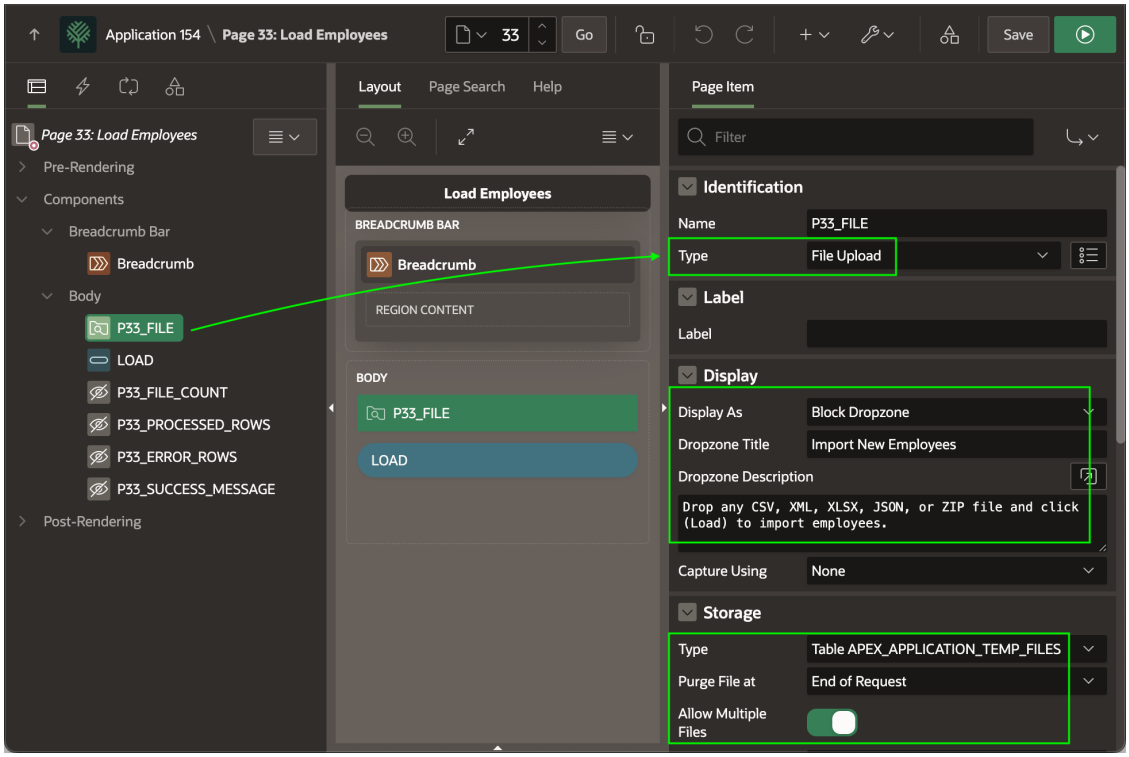
> **Tip**
>
> In this context, *request* means the HTTP `POST` sent to the APEX server when the user submits the page. Setting **Purge File at** to **End of Request** tells APEX to delete the uploaded file from temporary storage after it finishes processing that submit.

Four hidden page items act like local page variables to hold the values of:

- Number of files processed – `P33_FILE_COUNT`
- Number of employee rows processed – `P33_PROCESSED_ROWS`
- Number of error rows – `P33_ERROR_ROWS`
- Custom success message – `P33_SUCCESS_MESSAGE`

Copyright © 2026, Oracle and/or its affiliates.

Figure 17-22 Configuring File Upload Item to Allow Multiple Files



17.6.3 Processing Multiple Uploaded Files

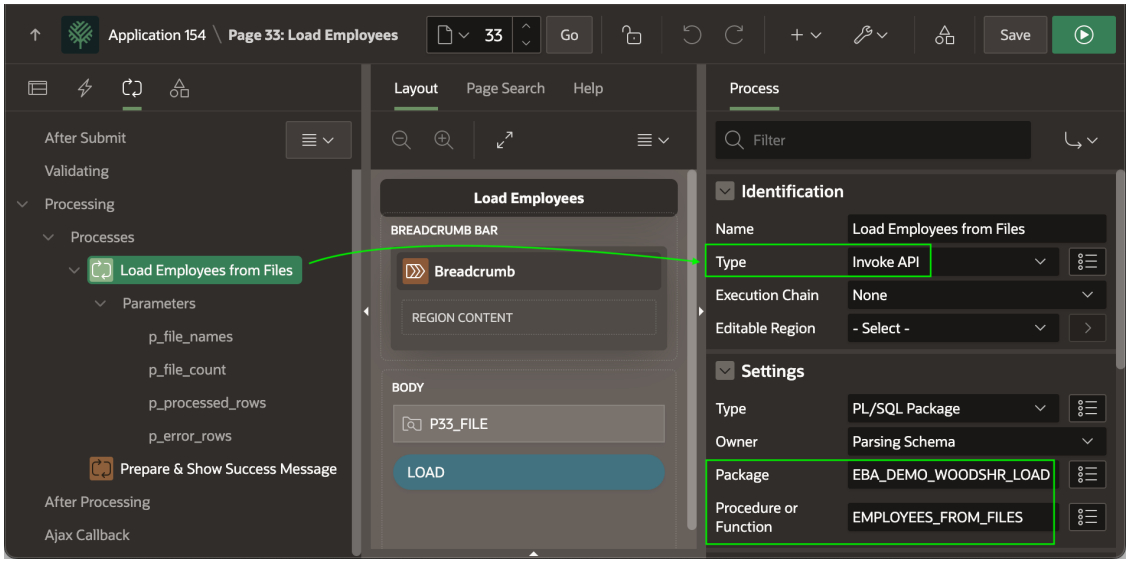
Process multiple uploaded files with PL/SQL that chooses the right Data Load Definition and expands ZIP archives.

When the user clicks the *(Load)* button it submits the page and the *Load Employees from Files* page process executes. It's an *Invoke API* page process that calls the `EMPLOYEES_FROM_FILES` procedure in the `EBA_DEMO_WOODSHR_LOAD` package explained below. The `p_file_names` *`IN`* parameter gets its value from the *File Upload* page item `P33_FILE`, and the three *`OUT`* parameters return their values into respective hidden page items:

- `p_file_count => P33_FILE_COUNT`
- `p_processed_rows => P33_PROCESSED_ROWS`
- `p_error_rows => P33_ERROR_ROWS`

Copyright © 2026, Oracle and/or its affiliates.

Figure 17-23 Calling Custom Data Load File Processing with Invoke API Page Process



The app defines two translatable text messages:

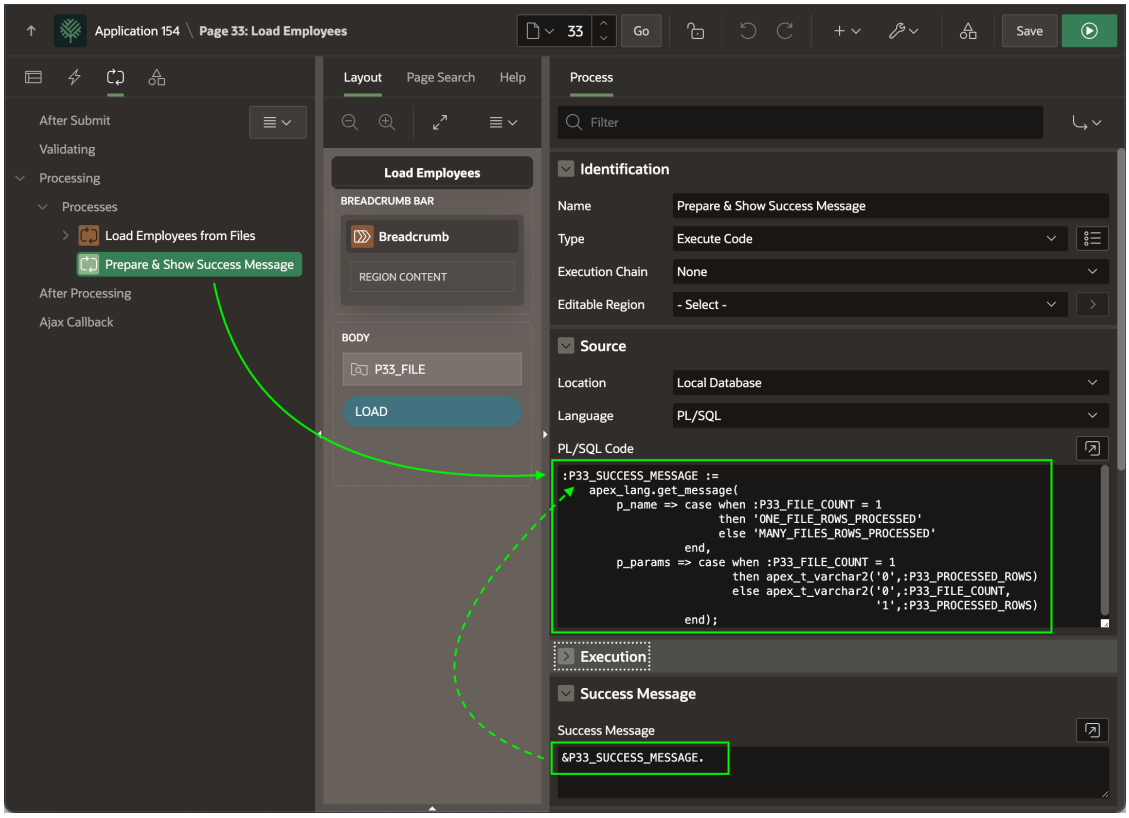
- `ONE_FILE_ROWS_PROCESSED => "Loaded 1 file, new employees: %0"`
- `MANY_FILES_ROWS_PROCESSED => "Loaded %0 files, new employees: %1"`

A subsequent *Execute Code* page process references these translatable messages to compute the value of the hidden `P33_SUCCESS_MESSAGE`. It uses `APEX_LANG.GET_MESSAGE` and `CASE` expressions with to supply an appropriate number of placeholder values for a conditional translatable text message key.

```
:P33_SUCCESS_MESSAGE :=
    apex_lang.get_message(
        p_name => case when :P33_FILE_COUNT = 1
                       then 'ONE_FILE_ROWS_PROCESSED'
                       else 'MANY_FILES_ROWS_PROCESSED'
                  end,
        p_params => case when :P33_FILE_COUNT = 1
                         then apex_t_varchar2('0',:P33_PROCESSED_ROWS)
                         else apex_t_varchar2('0',:P33_FILE_COUNT,
                                              '1',:P33_PROCESSED_ROWS)
                  end);
```


Copyright © 2026, Oracle and/or its affiliates.

Figure 17-24 Computing a Custom Success Message with Conditional Text



The source for the `EMPLOYEES_FOR_FILES` procedure appears below. It loops over the uploaded files in temporary storage from the *File Upload* item passed in, and calls the `DATA_LOAD_FOR_FILE` helper procedure to perform the data load on the current file.

```
-- In package eba_demo_woodshr_load
procedure employees_from_files(
    p_file_names      in varchar2,
    p_file_count     out number,
    p_processed_rows out number,
    p_error_rows     out number)
is
begin
    p_file_count     := 0;
    p_processed_rows := 0;
    p_error_rows     := 0;

    -- Loop over uploaded files in temp storage from File Upload item
    for j in (select blob_content, filename, mime_type
                from apex_application_temp_files
               where name in (select column_value
                                from apex_string.split(p_file_names,':')))
    loop
        -- Load the employees from the current file
        data_load_for_file(
            p_content        => j.blob_content,
            p_filename       => j.filename,
```


Copyright © 2026, Oracle and/or its affiliates.

```
            p_mime_type      => j.mime_type,
            p_file_count     => p_file_count,
            p_processed_rows => p_processed_rows,
            p_error_rows     => p_error_rows);
    end loop;
end employees_from_files;
```

The `DATA_LOAD_FOR_FILE` procedure looks at the extension of the passed-in file name as well as the MIME Type of the file if supplied to decide the suffix of the Data Load Definition to use. Depending on the match, the `l_data_load_def_suffix` ends up being `csv`, `json`, `xml`, `excel`, or `zip`. If nothing matches, it will be null.

If we're processing a ZIP file, then it calls the `DATA_LOAD_FROM_ZIP` helper procedure to process the contents of the zip file. Otherwise, if the data load suffix is not null, it appends the suffix to "`load_employees_from_`" to produce the static ID of the Data Load Definition to use. Finally, it passes this data load definition static id to `APEX_DATA_LOADING.LOAD_DATA` to perform the data load based on the *file profile* it specifies.

It keeps a running total of the number of files loaded and employees processed using the `processed_rows` field value from the `t_data_load_result` return value.

```
-- In package eba_demo_woodshr_load
-------------------------------------------
-- Load the employees data using the filename
-- extension or the mime type to determine
-- whether we're processing a zip file, or
-- a CSV, JSON, XML, or Excel file to data load
-------------------------------------------
procedure data_load_for_file(
    p_content        in blob,
    p_filename       in varchar2,
    p_mime_type      in varchar2,
    p_file_count     in out number,
    p_processed_rows in out number,
    p_error_rows     in out number)
is
    c_extension            constant varchar2(255)  :=
extension_of(p_filename);
    l_data_load_def_suffix          varchar2(255);
    l_result                        apex_data_loading.t_data_load_result;
begin
    l_data_load_def_suffix :=
        case
            when    c_extension in ('txt','csv')
                 or p_mime_type in ('text/plain','text/csv')
            then
                'csv'
            when    c_extension = 'json'
                 or p_mime_type = 'application/json'
            then
                'json'
            when    c_extension = 'xml'
                 or p_mime_type = 'text/xml'
            then
                'xml'
            when    c_extension in ('xls','xlsx')
```


Copyright © 2026, Oracle and/or its affiliates.

```
                or p_mime_type in ('application/vnd.ms-excel',
                                   'application/vnd.openxmlformats-
officedocument.spreadsheetml.sheet')
            then
                'excel'
            when    c_extension = 'zip'
                 or p_mime_type = 'application/zip'
            then
                'zip'
        end;
    if l_data_load_def_suffix = 'zip' then
        data_load_from_zip(
            p_zip_content    => p_content,
            p_file_count     => p_file_count,
            p_processed_rows => p_processed_rows,
            p_error_rows     => p_error_rows);
    elsif l_data_load_def_suffix is not null then
        l_result := apex_data_loading.load_data(
                        p_static_id => 'load_employees_from_'||
l_data_load_def_suffix,
                        p_data_to_load => p_content);
        p_file_count     := p_file_count + 1;
        p_processed_rows := p_processed_rows + l_result.processed_rows;
        p_error_rows     := p_error_rows + l_result.error_rows;
    end if;
end data_load_for_file;
```

The `DATA_LOAD_FROM_ZIP` uses `APEX_ZIP.GET_FILES` and a `FOR` loop to process the list of files contained in the zip file. It calls `DATA_LOAD_FOR_FILE` to get the job done.

```
-- In package eba_demo_woodshr_load
-----------------------------------------------
-- Load the employees data files contained in
-- the supplied zip file blob
-----------------------------------------------
procedure data_load_from_zip(
    p_zip_content    in blob,
    p_file_count     in out number,
    p_processed_rows in out number,
    p_error_rows     in out number)
is
    l_files        apex_zip.t_files;
    l_file_content blob;
begin
    -- Get list of files from the zip
    l_files := apex_zip.get_files(p_zip_content);

    -- Process the individual files inside the zip
    for j in 1..l_files.count loop

        -- Get the contents of the current file in the zip file
        l_file_content := apex_zip.get_file_content(p_zip_content,l_files(j));

        -- Load employees from the current file
        data_load_for_file(
```


Copyright © 2026, Oracle and/or its affiliates.

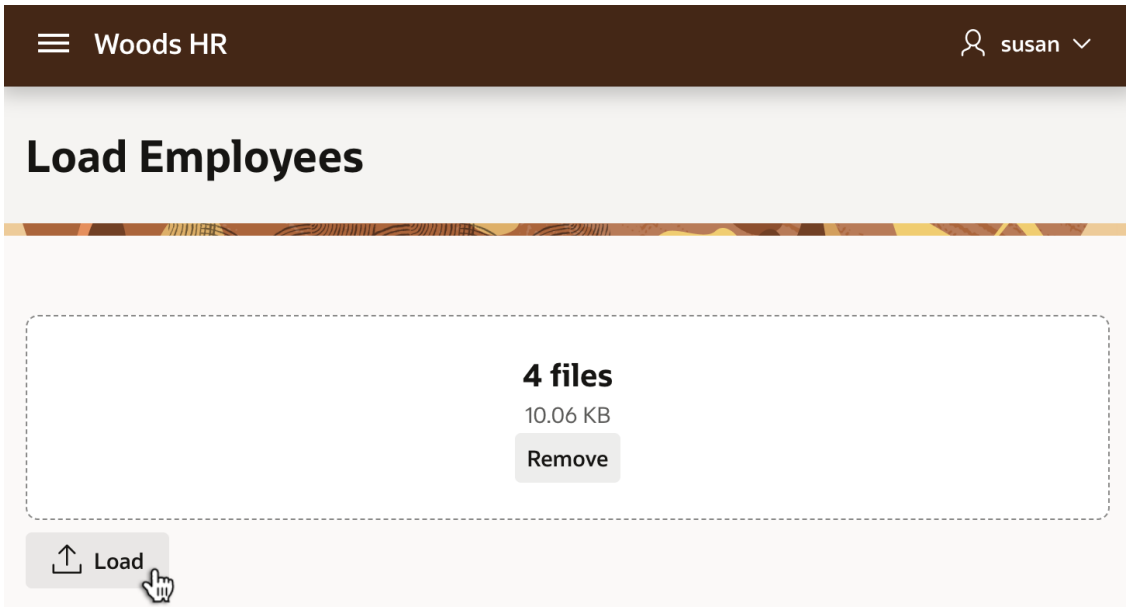
```
            p_content        => l_file_content,
            p_filename       => l_files(j),
            p_mime_type      => null,
            p_file_count     => p_file_count,
            p_processed_rows => p_processed_rows,
            p_error_rows     => p_error_rows);
    end loop;
```

17.6.4 Uploading Multiple Data Load Files

Upload multiple employee data files and load them together, including files inside ZIP archives.

When an HR Rep like *Susan* visits the *Load Employees* page, as shown below, she can now pick an Excel file, CSV file, JSON file, and XML file all at once, and click *(Load)* to perform the bulk employee data load.

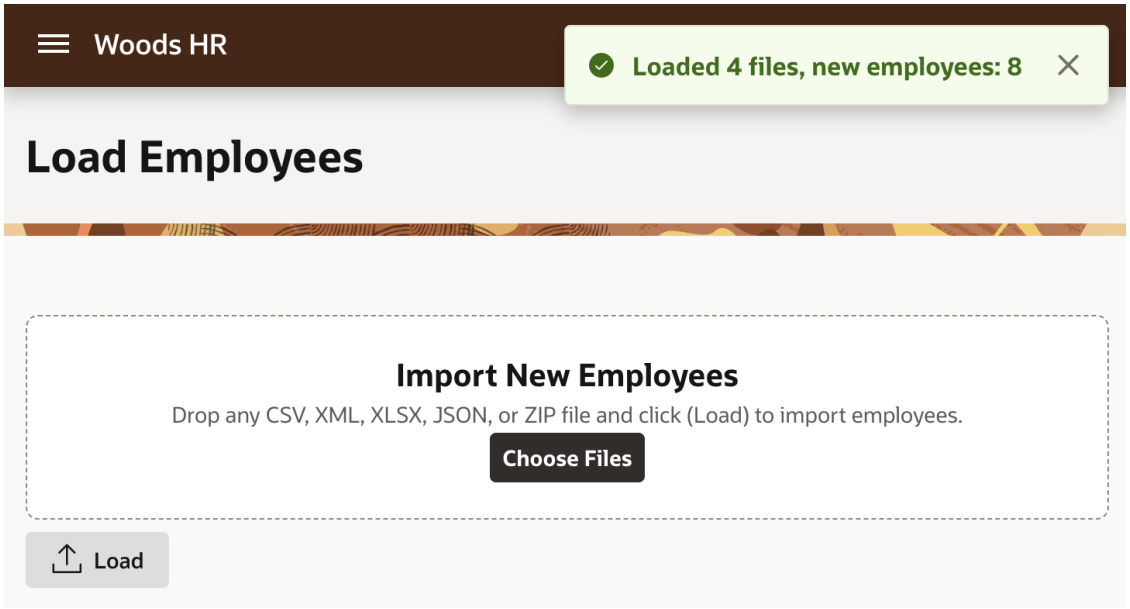
Figure 17-25 Selecting Four Different Kinds of Files at Once for Employee Data Loading



The custom, conditional success message confirms that four files were loaded containing eight total new employees.

Copyright © 2026, Oracle and/or its affiliates.

Figure 17-26 Success Message Confirms Eight Employees Loaded from Four Files



The same works if Susan submits a mix of files and zip files. All the different XML, CSV, JSON, and Excel files containing employee data get loaded according to their respective Data Load Definitions, including ones that were part of any submitted ZIP files.

Copyright © 2026, Oracle and/or its affiliates.

18

Grouping Logic and Background Execution

Group related page processes in an execution chain and offload longer-running tasks to the background.

Use an *Execution Chain* page process to group a sequence of child processes. Its server-side condition determines whether the children execute. They can be any kind of page process to execute code, invoke APIs and REST services, start workflows, print reports, send emails or push notification, load data, and more. Since chains can nest within chains, your logic becomes a self-documenting outline when you use clear, descriptive names.

Execution chains also improve user experience. Offload long-running work by adding child processes to a chain and enabling its *Run in Background* switch. The APEX runs the work in the background moves on to the next page process in the user session. Background jobs run in a distinct session, with a private copy of the page's session state. Do anything you need to, including processing user-uploaded files, reporting progress, and notifying users when complete.

You can limit background jobs per user and declaratively serialize them to avoid resource contention or to prevent multiple executions for the same context value. Workspace administrators set overall limits and can grant background jobs more resources when needed.

- Exploring Employee Excellence for HR Reps
  Preview the Employee Excellence page, where HR reps send welcome packets and start background award reviews.
- Outlining Logic Using Execution Chains
  Organize page processing logic into a clear outline using nested execution chains.
- Offloading Work to the Background
  Offload longer-running page processing to the background so users can keep working.
- Serializing Background Process Jobs
  Serialize background jobs by type or context value to prevent conflicting work.
- Understanding Cloned Session State
  Understand how background jobs use an isolated copy of the user’s session state.
- Reporting Background Progress
  Show users background job progress with the `APEX_APPL_PAGE_BG_PROC_STATUS` view.
- Loading Data in the Background
  Load uploaded data in the background and notify users when processing completes.
- Limiting and Throttling Background Jobs
  Limit or throttle background execution chain jobs at the chain, application, or workspace level.
- Adjusting Resources for Background Jobs
  Configure a custom job class when background process jobs need different resource limits.
- Debugging Background Processes
  Debug background execution chains by viewing jobs, session state, and failure details.

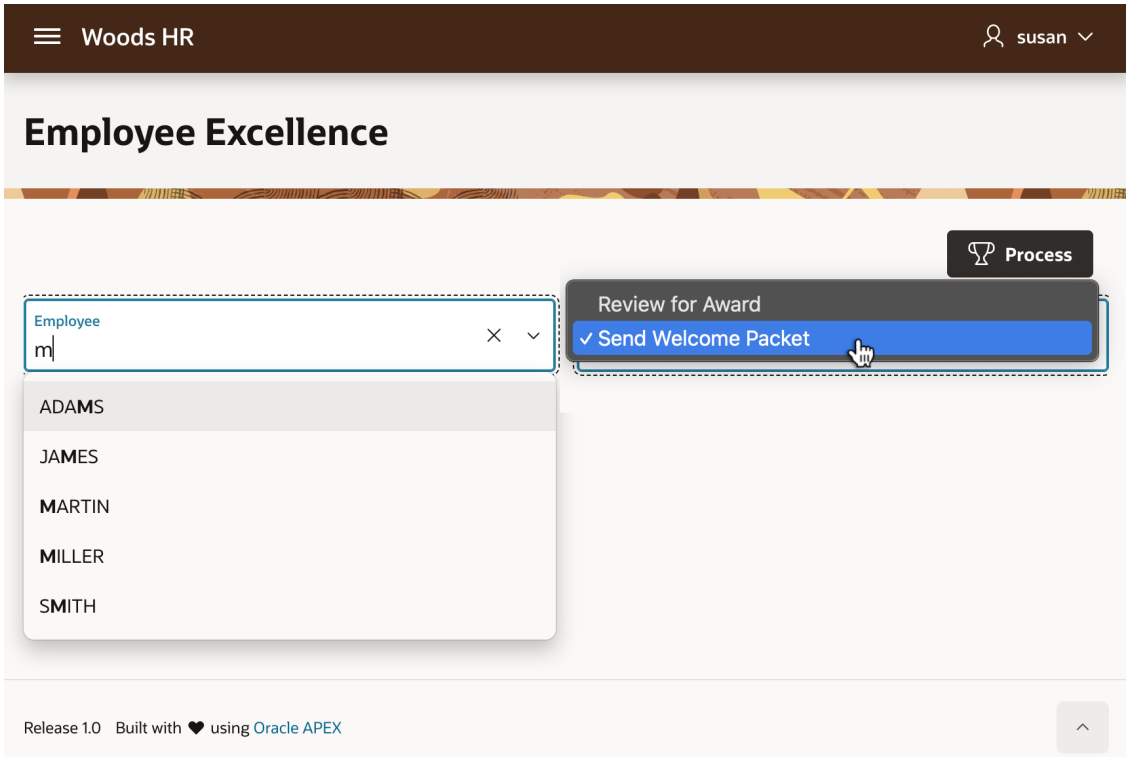
Copyright © 2026, Oracle and/or its affiliates.

18.1 Exploring Employee Excellence for HR Reps

Preview the Employee Excellence page, where HR reps send welcome packets and start background award reviews.

HR reps like *Susan* at *Woods Clinic* strive to promote and reward staff success. She uses the *Employee Excellence* page below to start new hires off right with a *Welcome Packet*. It contains appropriate materials, tailored by role, on how to succeed at the company. Susan can also initiate an AI-assisted employee review to decide whether a staff member qualifies to receive an *Employee Excellence Award*. The figure shows Susan's view of the Employee Excellence page where she can search for an employee and send them a welcome packet or review them for an award.

Figure 18-1 Woods HR App's Employee Excellent Page for HR Reps



While preparing and sending the welcome packet is quick, the *Review for Award* action is a longer-running process involving:

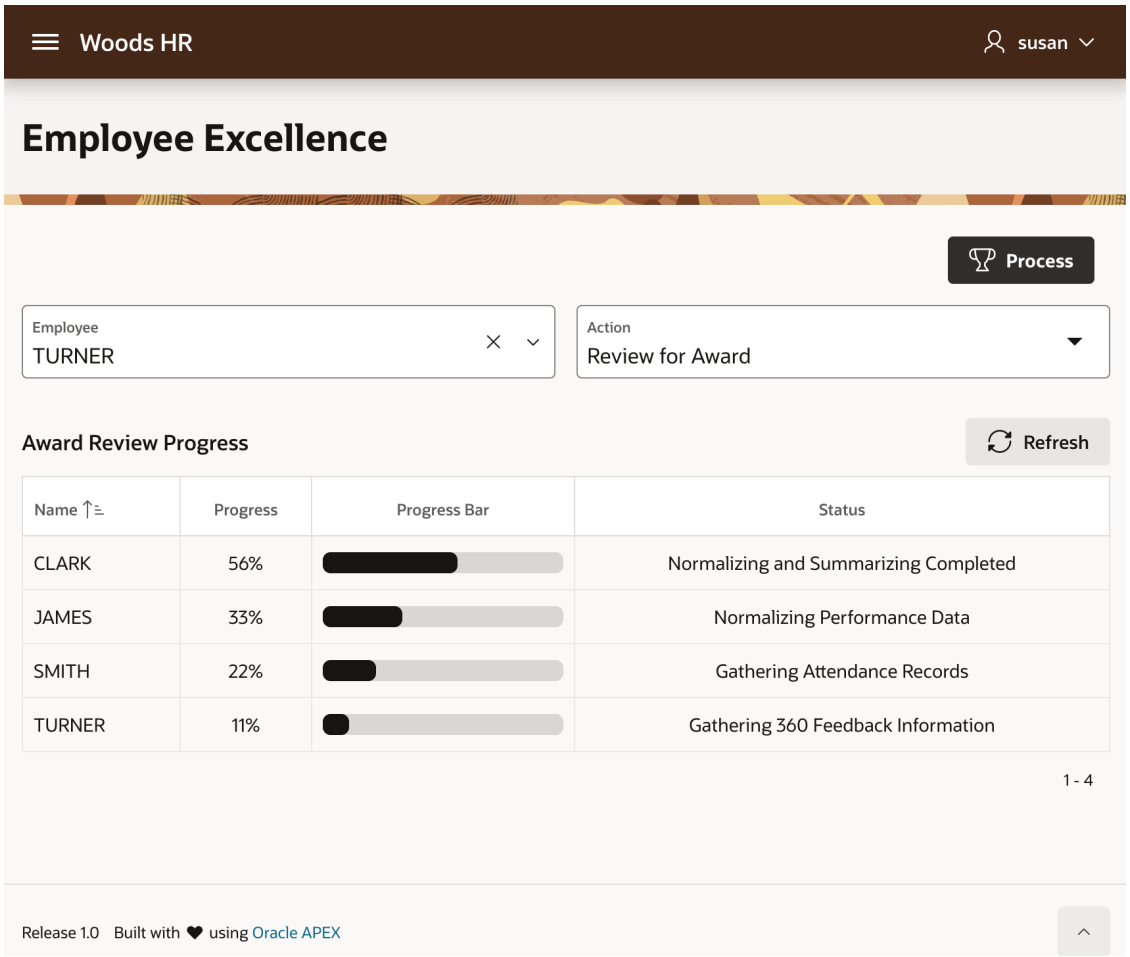
- Gathering peer and manager feedback on the employee and metrics from various external systems
- Normalizing and summarizing the different employee information sources using AI
- Analyzing the employee's progress against a company rubric with additional AI assistance
- Predicting award eligibility from historical data using machine learning, and
- Generating and emailing an award certificate if the employee qualifies.

The *Review for Award* process runs in the background so *Susan* can do other tasks while the system progresses the reviews without her involvement. If the recipient qualifies, they receive

Copyright © 2026, Oracle and/or its affiliates.

their award as an email attachment when the process completes without further input. *Susan* can return at any time to see the ongoing progress of any reviews she's initiated on the page as shown below. The figure shows Susan's view of the Employee Excellence page when there are ongoing award review processes running in the background. An Award Review Progress section of the page shows the progress of each review as a percent and a progress bar, as well as its current status.

Figure 18-2 Viewing Progress of Longer-Running Award Review



18.2 Outlining Logic Using Execution Chains

Organize page processing logic into a clear outline using nested execution chains.

The *Employee Excellence* page uses execution chains to group and nest processing logic into a self-documenting outline.

- Setting Up Context After Submit
  Set hidden page items after submit so later processing steps can use the selected employee context.
- Making Processing Conditional
  Use an execution chain server-side condition to run child processes only in specific cases.
- Sequencing Child Processes
  Control child process order and move processes between execution chains.

Copyright © 2026, Oracle and/or its affiliates.

- Nesting Execution Chains
  Nest execution chains to structure complex logic, adding conditions where needed.
- Choosing Descriptive Process Names
  Choose clear process names and conditions to make execution chains easier to understand and maintain.

18.2.1 Setting Up Context After Submit

Set hidden page items after submit so later processing steps can use the selected employee context.

When a user clicks the *(Process)* button to submit the page, an *After Submit* process *Get Employee Name and Email* retrieves the employee's name and email into two hidden page items using the selected employee id with the query:

```
select ename, :G_DEMO_EMAIL
  into :P34_EMPLOYEE_ENAME, :P34_EMPLOYEE_EMAIL
  from eba_demo_emp
 where empno = :P34_EMPLOYEE_EMPNO
```

Various steps of the page processing logic below reference these hidden page item values, along with the value of the `P34_EMPLOYEE_EMPNO` *Select One* page item the user uses to pick an employee to process.

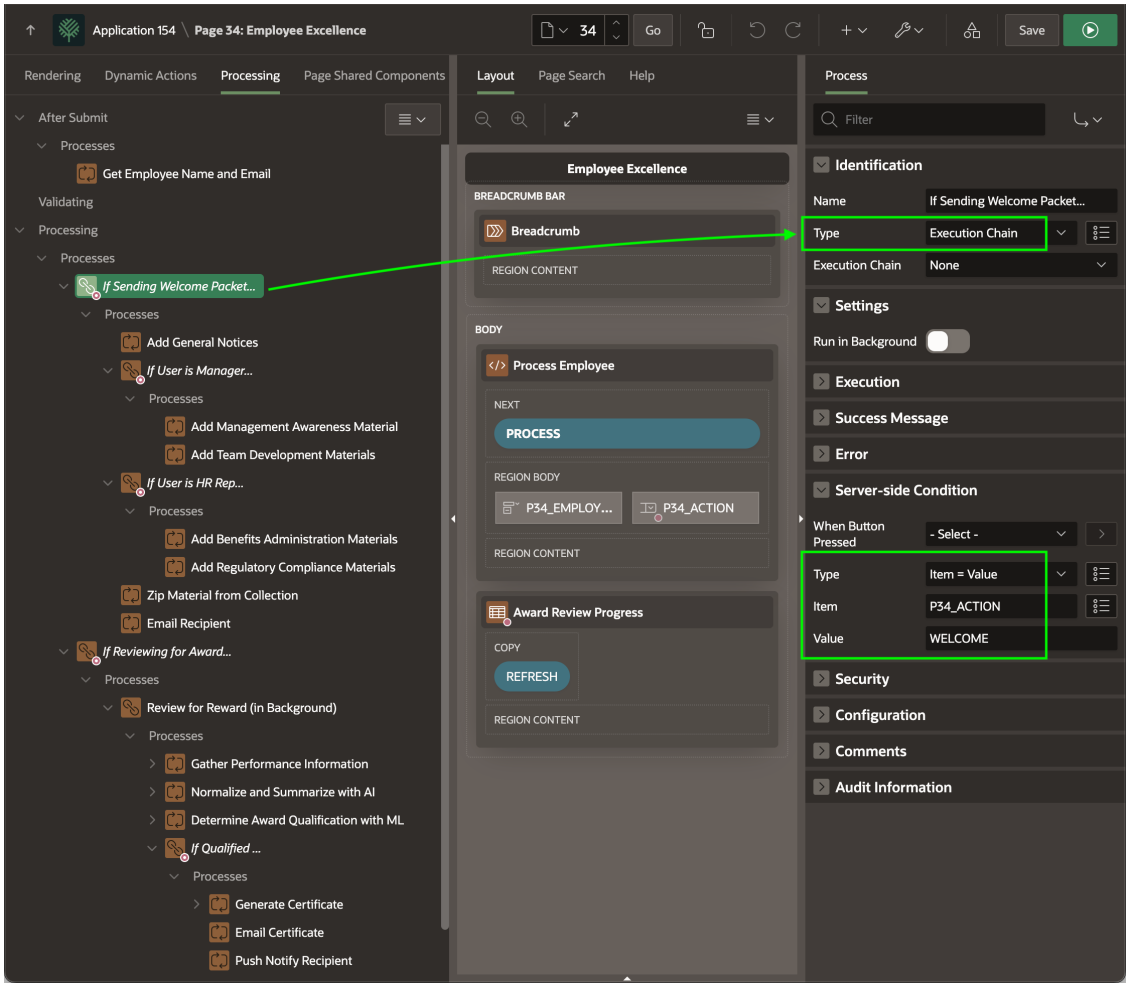
18.2.2 Making Processing Conditional

Use an execution chain server-side condition to run child processes only in specific cases.

In the *Processing* section shown below, the *If Sending Welcome Packet...* page process sets *Type* to *Execution Chain* and configures its *Server-side Condition* to run its child processes only when the `P34_ACTION` select list page item's value equals "`WELCOME`".

Copyright © 2026, Oracle and/or its affiliates.

Figure 18-3 Execution Chains Let You Organize Page Processing Logic into an Outline



18.2.3 Sequencing Child Processes

Control child process order and move processes between execution chains.

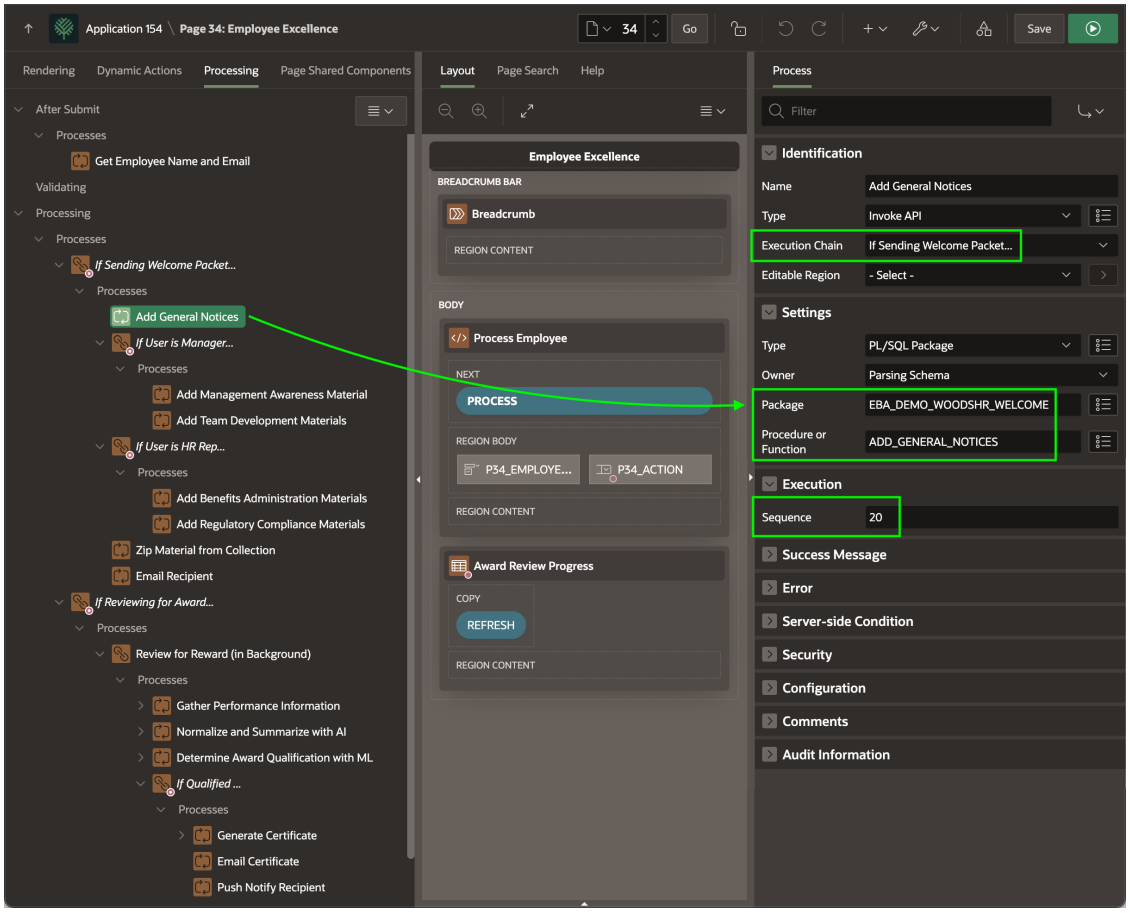
The *If Sending Welcome Packet...* execution chain's child processes execute in sequence. You can reorder a child process among other processes at the same level using drag and drop. You can also edit the value of its *Sequence* property shown below.

To change which execution chain a child process belongs to, just change its *Execution Chain* property to the new parent. Or set that to *None* to restore the process to the root level of the processing logic "tree".

In the *If Sending Welcome Packet...* chain, the *Add General Notices* child process executes first. It uses an *Invoke API* process to call the `ADD_GENERAL_NOTICES` procedure in the `EBA_DEMO_WOODSHR_WELCOME` package.

Copyright © 2026, Oracle and/or its affiliates.

Figure 18-4 Child Processes Execute in Sequence Order in an Execution Chain



18.2.4 Nesting Execution Chains

Nest execution chains to structure complex logic, adding conditions where needed.

The *If User is Manager...* execution chain process runs next. Nested inside an outer execution chain, it shows how you can outline your business logic to any level of depth required. Attach conditional logic as needed to guide the flow. Its *Server-side Condition* calls the PL/SQL boolean expression below to determine if the selected employee manages others:

```
eba_demo_woodshr_welcome.manages_others(:P34_EMPLOYEE_EMPNO)
```

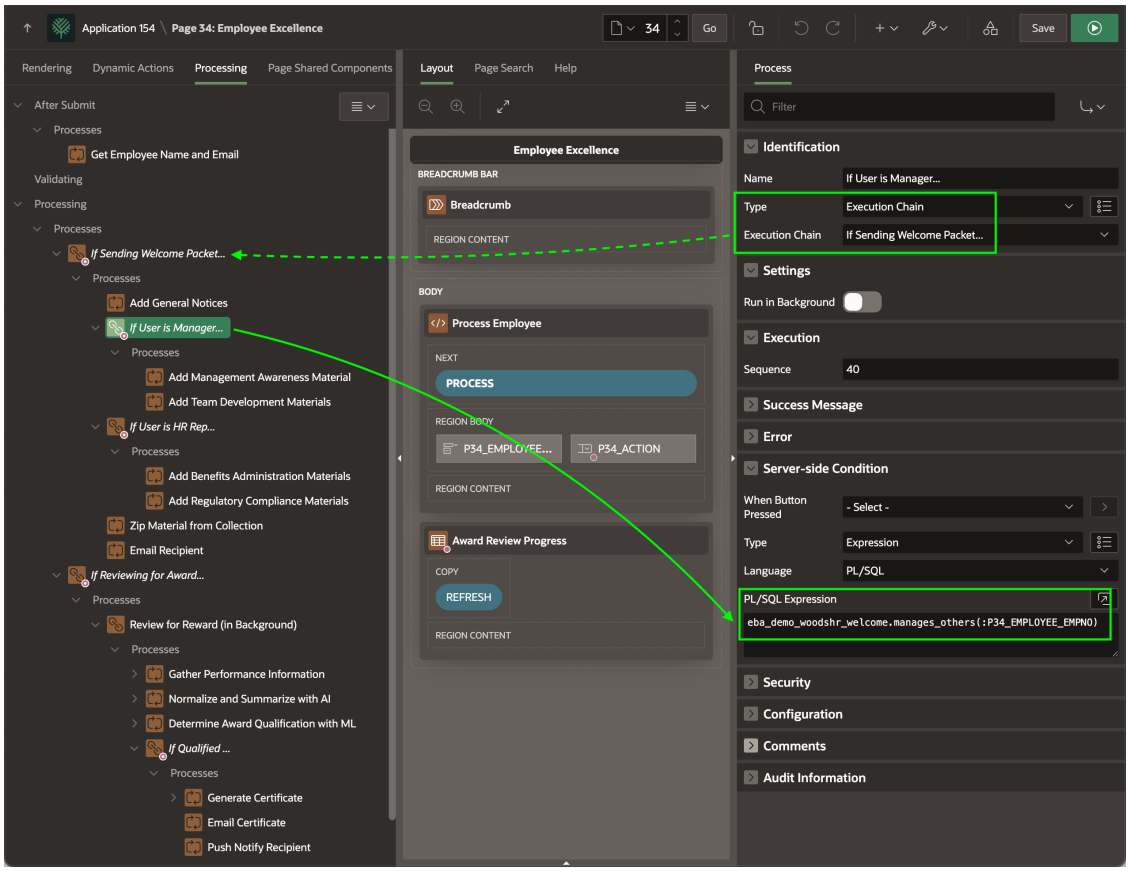
If *true*, its child processes execute to include management awareness and team development materials pertinent only to mangers.

> **Tip**
>
> To change which execution chain a child process belongs to, set its *Execution Chain* property to the new parent. To make it a top-level page process, set that property to *None*.

Copyright © 2026, Oracle and/or its affiliates.

The figure shows the selected **If User is Manager...** child process selected in the processing tree. Its **Execution Chain** property identifies the containing **If Sending Welcome Packet...** chain, and its PL/SQL condition controls when its own child processes run.

Figure 18-5 Nest Execution Chains to Form an Outline



18.2.5 Choosing Descriptive Process Names

Choose clear process names and conditions to make execution chains easier to understand and maintain.

By using descriptive names, appropriate server-side conditions, and nesting chains as needed, you simplify maintenance of your processing logic. The final steps in the readable outline of logic to send a welcome packet include:

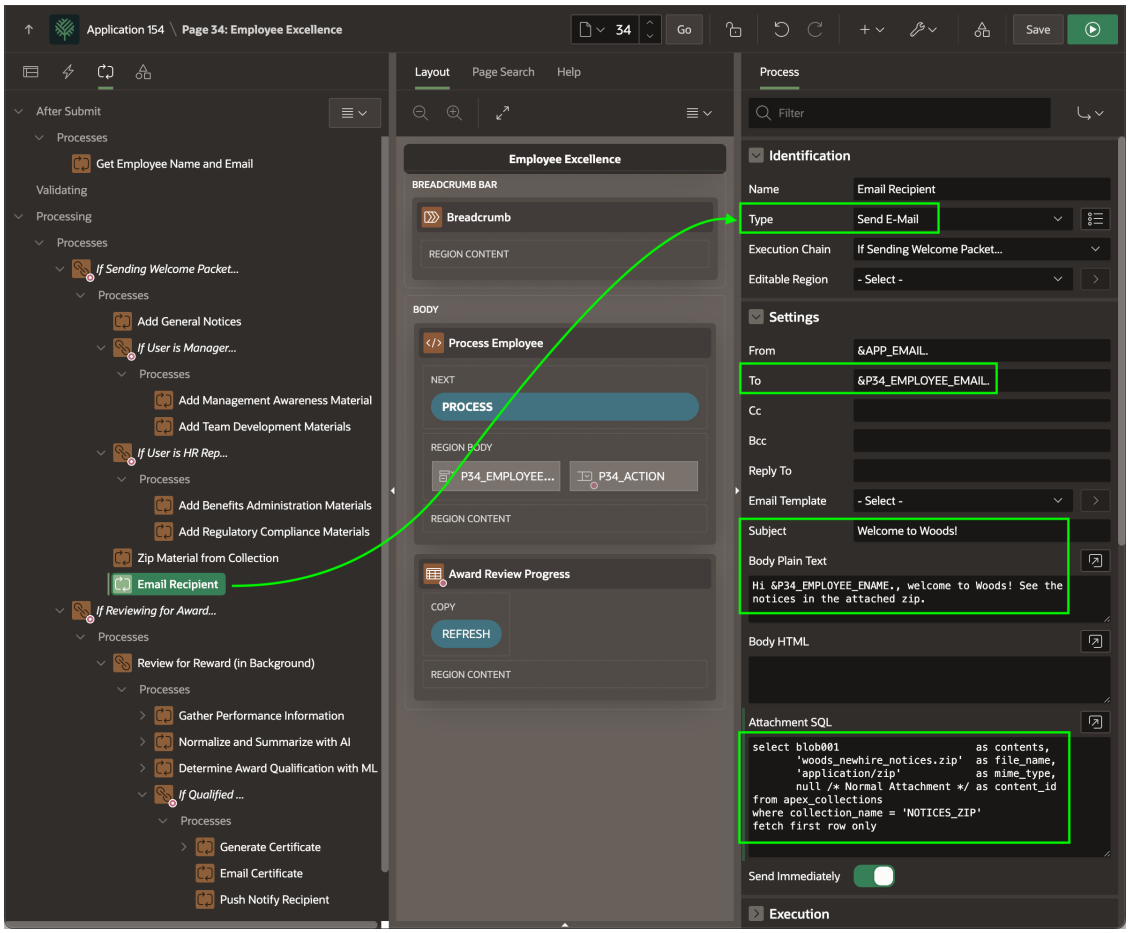
- *If User is HR Rep...* execution chain conditionally includes materials only relevant for HR reps.
- *Zip Materials from Collection* process invokes the `ZIP_NOTICES` procedure to prepare a zip file in a collection named `NOTICES_ZIP` containing the appropriate materials for the selected employee.
- *Email Recipient* step shown below sends the zip file to the employee as an email attachment.

Copyright © 2026, Oracle and/or its affiliates.

It uses a native *Send E-Mail* process type and configures `&P34_EMPLOYEE_EMAIL.` as the recipient in the *To* property. Using the *Attachment SQL* below, it includes the zip file by querying the BLOB from the `NOTICES_ZIP` collection.

```
select blob001                      as contents,
       'woods_newhire_notices.zip'  as file_name,
       'application/zip'            as mime_type,
       null /* Normal Attachment */ as content_id
from apex_collections
where collection_name = 'NOTICES_ZIP'
fetch first row only
```

Figure 18-6 Sending Email with a Native Page Process



18.3 Offloading Work to the Background

Offload longer-running page processing to the background so users can keep working.

Enable one switch to offload longer-running page processing to the background so the user doesn't wait. Backgrounded execution chains can invoke APIs, engage AI services and machine learning, generate reports, send email and push notifications, and more. The *Employee Excellence* page's *Review for Reward (in Background)* chain combines all of these.

Copyright © 2026, Oracle and/or its affiliates.

- Configuring Chain to Run in Background
  Configure an execution chain to run in the background and optionally associate it with a context value.
- Outlining the Award Review Process
  Outline a background award review with API calls, a qualification check, and conditional follow-up steps.
- Printing Personalized Certificate
  Generate a personalized PDF certificate in the background and store it temporarily for email delivery.
- Emailing User with Attachments
  Send an email with file attachments or inline images using Attachment SQL.
- Push Notifying User on Opted-in Devices
  Send push notifications to opted-in user devices from a native page process.

18.3.1 Configuring Chain to Run in Background

Configure an execution chain to run in the background and optionally associate it with a context value.

When you enable an execution chain's *Run in Background* switch, at runtime the APEX engine enqueues a background job to execute as soon as possible and proceeds to the next process in the list, if any. To learn the unique job execution ID the system assigns, provide the name of a page item for the *Return ID into Item* property. To associate a context value with the background job, as shown below, configure the name of a page item for the *Context Value Item* property.

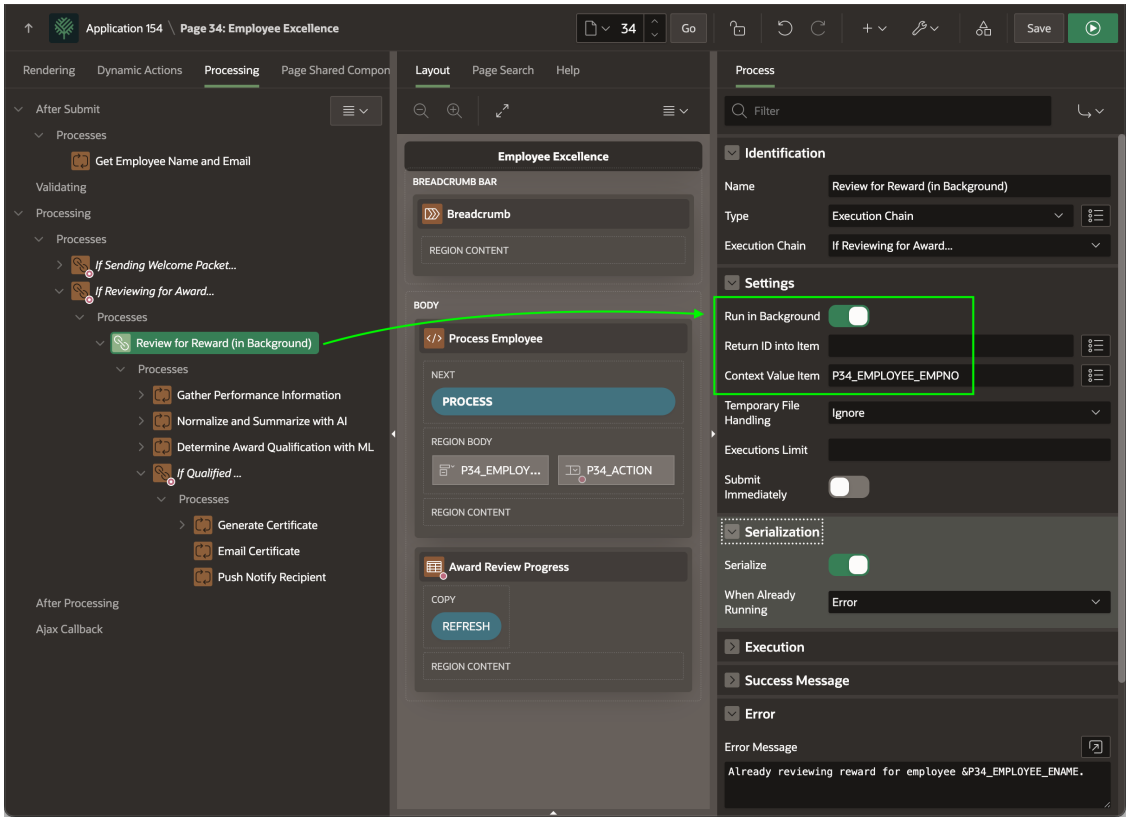
For example, the *Employee Excellence* page ignores the job's execution id by leaving *Return ID into Item* property blank, and configures `P34_EMPLOYEE_EMPNO` to provide the context value. This means that each *instance* of the *Review for Reward (in Background)* job is linked to the employee ID it is processing.

Tip

The context value you associate with a background job appears in the `CONTEXT_VALUE` column of the `APEX_APPL_PAGE_BG_PROC_STATUS` view.

Copyright © 2026, Oracle and/or its affiliates.

Figure 18-7 Setting Execution Chain to Run in the Background



18.3.2 Outlining the Award Review Process

Outline a background award review with API calls, a qualification check, and conditional follow-up steps.

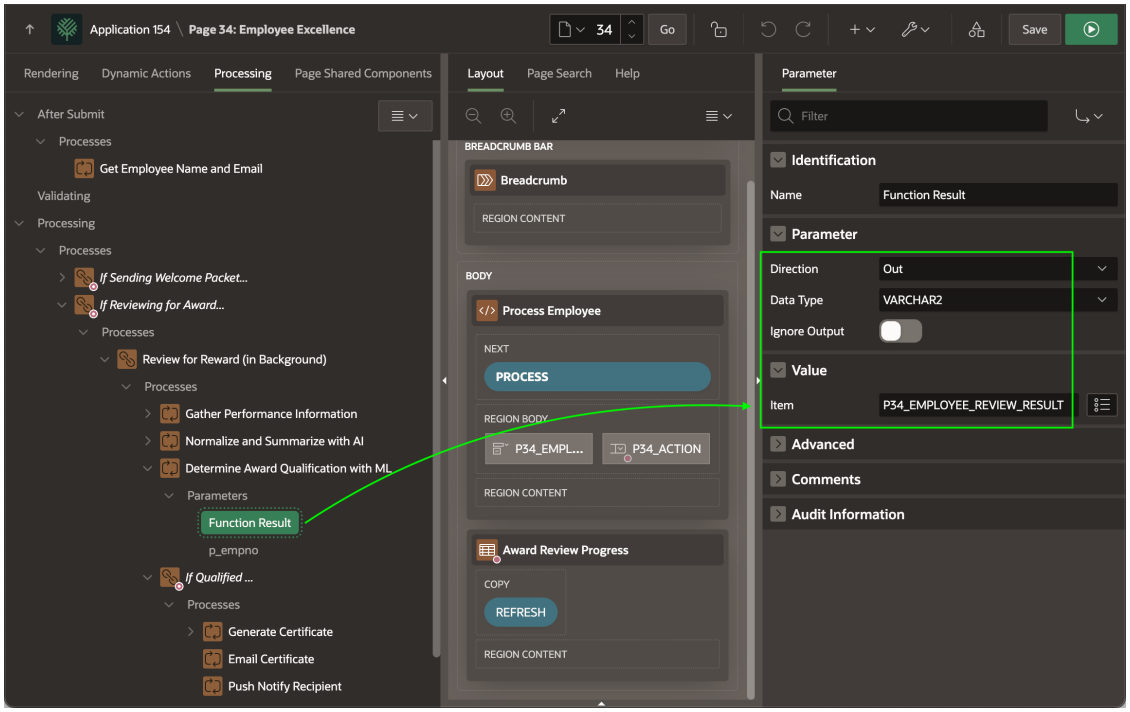
The first three steps in the *Review for Reward (in Background)* execution chain are *Invoke API* child processes. Each calls a respective procedure in the `EBA_DEMO_WOODSHR_REWARD` package:

- The first gathers employee peer and manager feedback and metrics from various external systems.
- The second normalizes and summarizes the different employee information sources using an AI service.
- The third uses a machine learning model trained on historical data to predict whether the candidate employee qualifies for an Excellence Award.

As shown below, the *Determine Award Qualification with ML* Invoke API process maps its function result into the hidden page item `P34_EMPLOYEE_REVIEW_RESULT`.

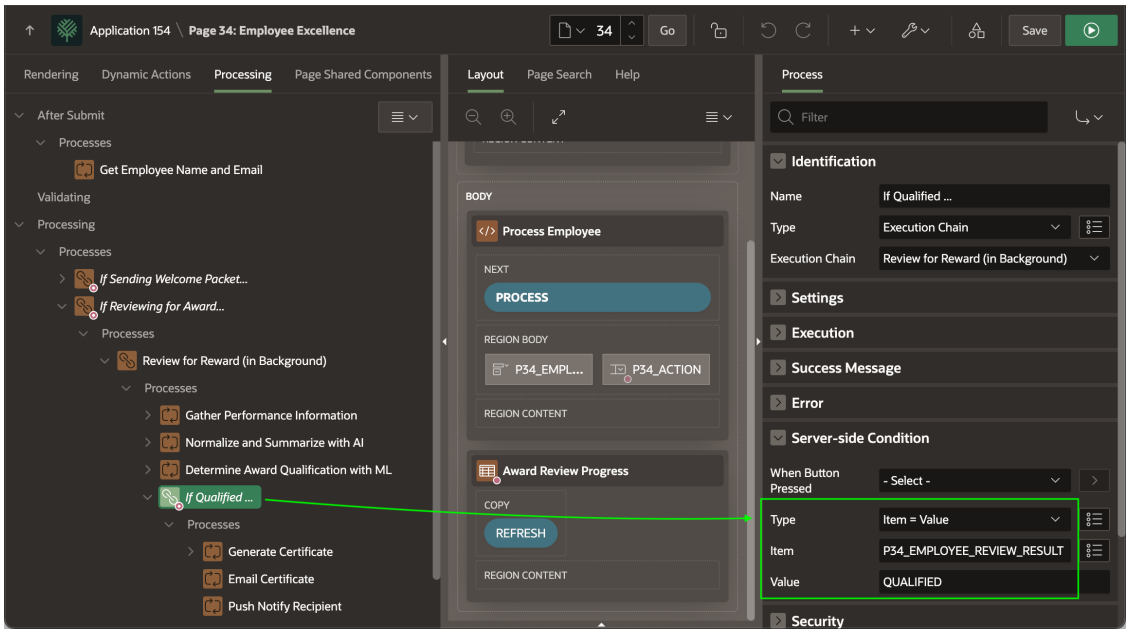
Copyright © 2026, Oracle and/or its affiliates.

Figure 18-8 Returning Function Result into Hidden Page Item



Next, the *If Qualified...* execution chain's child processes only execute if its *Server-side Condition* is true. It's configured to check for `P34_EMPLOYEE_REVIEW_RESULT` equals `"QUALIFIED"`. If true, it generates the certificate, emails it to the recipient, and sends a push notification to the recipient as well.

Figure 18-9 Checking Whether Candidate Employee Qualified for Excellence Award



Copyright © 2026, Oracle and/or its affiliates.

18.3.3 Printing Personalized Certificate

Generate a personalized PDF certificate in the background and store it temporarily for email delivery.

The *Generate Certificate* child process uses *Invoke API* to call the procedure below, passing in the candidate employee ID from `P34_EMPLOYEE_EMPNO`. It does the following key tasks:

- Sets employee ID into `G_EMPLOYEE_EXCELLENCE_EMPNO` app item the report query references
- Generates PDF certificate using report query by static ID and its associated report layout
- Creates an `EMP_EXCELLENCE_CERTIFICATE` collection
- Adds generated PDF to it for use in email attachment.

```
-- In package eba_demo_woodshr_reward
procedure generate_certificate(
    p_empno in number)
is
    l_certificate_pdf blob;
begin
    apex_background_process.set_status('Generating Certificate');
    -- Set application item referenced in Report Query to passed
empno
    apex_session_state.set_value('G_EMPLOYEE_EXCELLENCE_EMPNO',p_empno);
    -- Generate the document using a report query by static id
    l_certificate_pdf := apex_print.generate_document(
                            p_application_id => V('APP_ID'),
                            p_report_query_static_id =>
'emp_excellence_certificate');
    -- Create collection and add PDF blob for email attachment to it

apex_collection.create_or_truncate_collection('EMP_EXCELLENCE_CERTIFICATE');
    apex_collection.add_member(
        p_collection_name => 'EMP_EXCELLENCE_CERTIFICATE',
        p_blob001         => l_certificate_pdf);
    apex_background_process.set_progress(p_totalwork => c_steps, p_sofar =>
8);
    apex_background_process.set_status('Generating Certificate
Completed');
end generate_certificate;
```

18.3.4 Emailing User with Attachments

Send an email with file attachments or inline images using Attachment SQL.

The *Email Certificate* child process is a native *Send Email* type that delivers the *Employee Excellence Award* certificate to the recipient. It references the hidden `P34_EMPLOYEE_EMAIL` item in the *To* setting using substitution syntax. Notice it sends a simple email in plain text, foregoing the use of an email template and *Body HTML*. The *Body Plain Text* also uses substitution strings to personalize the text.

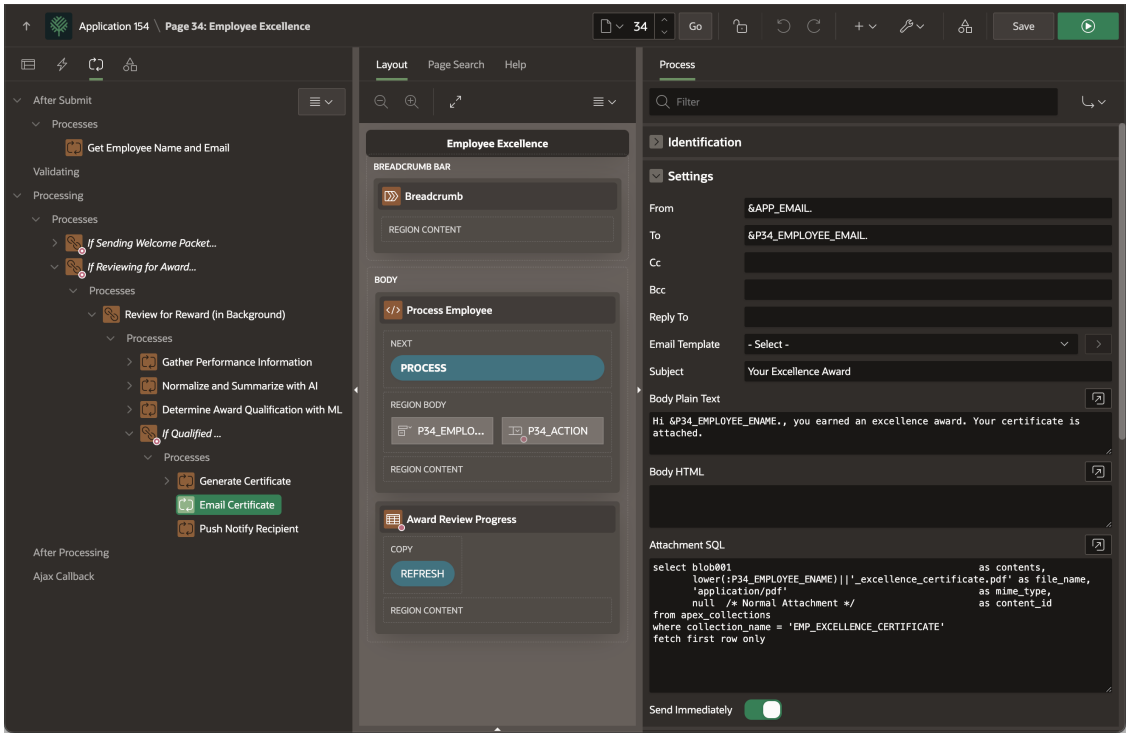
To attach one or more files, use an *Attachment SQL* statement with a four-column query that returns, in this order:

Copyright © 2026, Oracle and/or its affiliates.

- The BLOB of file content to attach
- The attachment file name
- The MIME type of the attachment
- `null` for the Content Id

> **Tip**
>
> To include an image to reference inline in the *Body HTML*, use a distinct, non-null value for *Content Id* in each row of the *Attachment SQL* query results that represents an embedded image. For example, if one row of the query returns a BLOB with file name `logo.jpg` of MIME type `image/jpeg` and a content id of `company_logo`, then the HTML body can reference it inline using:
>
> ```
> <img src="cid:company_logo" alt="Woods Clinic Logo">
> ```

Figure 18-10 Emailing Employee Excellence Award Certificate to Recipient

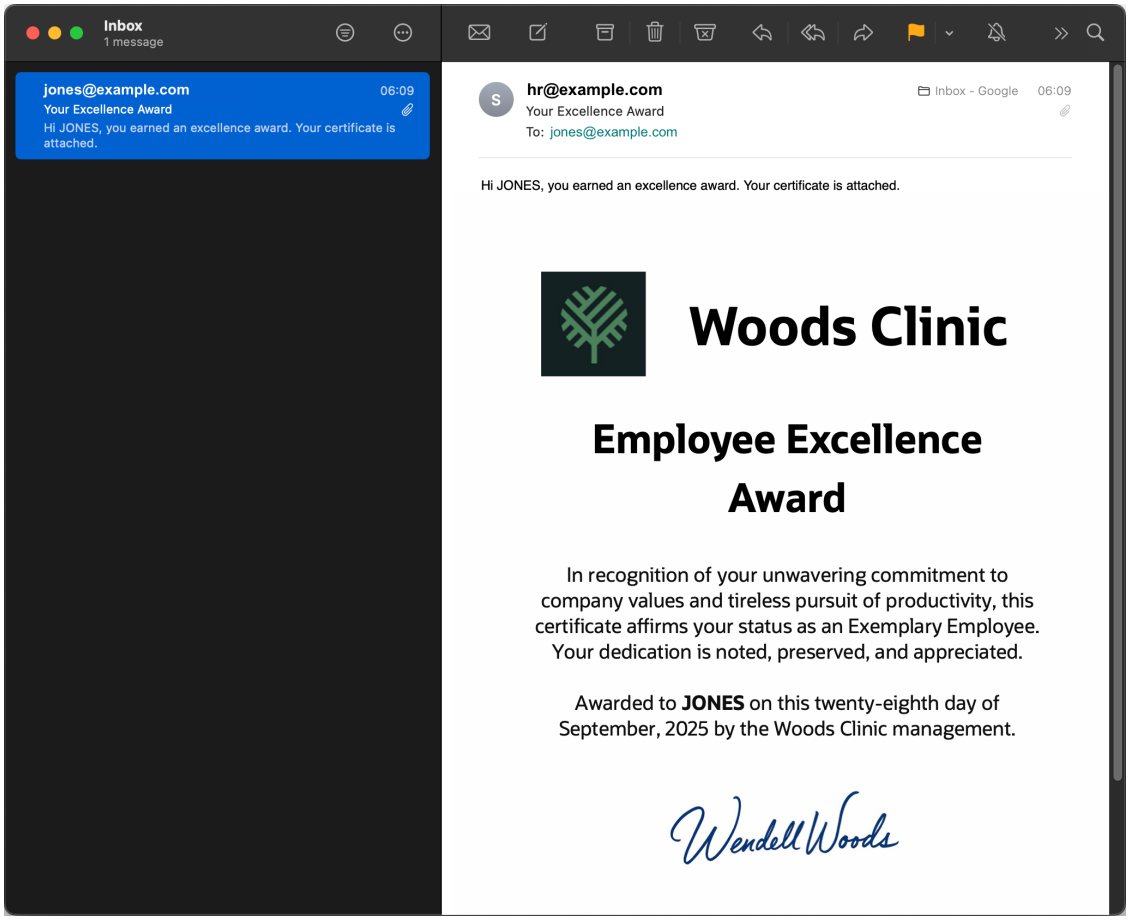


When the email arrives in JONES' inbox, it looks like the figure below. JONES email client displays the attachment straight away so she can read the great news and the text of the award signed by Wendell Woods himself:

> *In recognition of your unwavering commitment to company values and tireless pursuit of productivity, this certificate affirms your status as an Exemplary Employee. Your dedication is noted, preserved, and appreciated. Awarded to JONES on this twenty-eighth day of September, 2025 by the Woods Clinic management.*

Copyright © 2026, Oracle and/or its affiliates.

Figure 18-11 JONES Receives Her Employee Excellence Award Certificate



18.3.5 Push Notifying User on Opted-in Devices

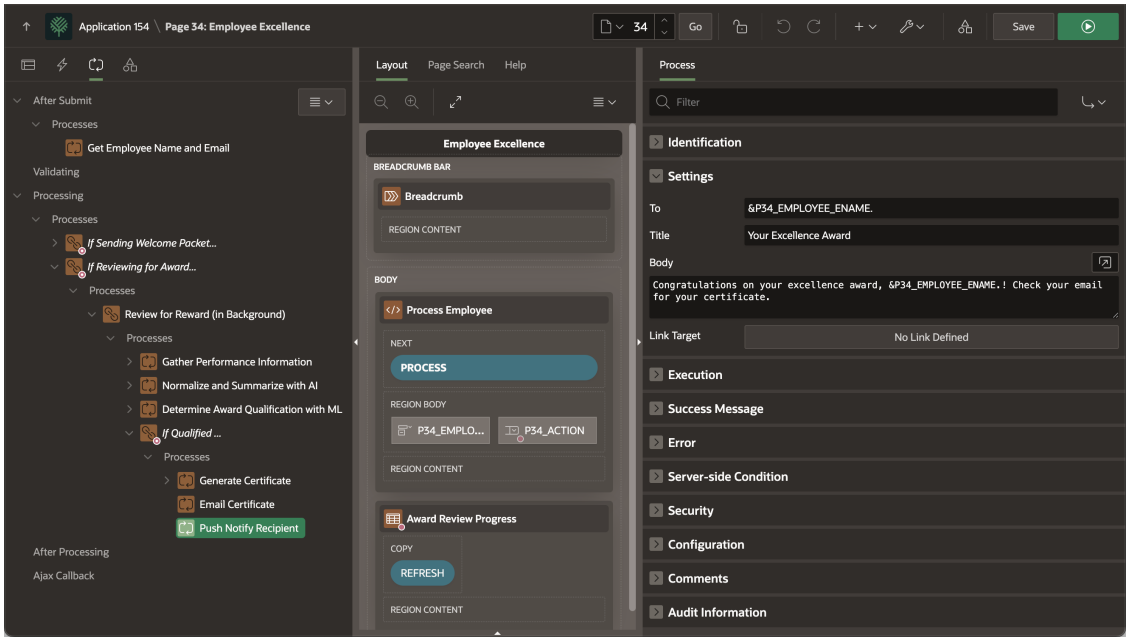
Send push notifications to opted-in user devices from a native page process.

The final child process notifies the *Employee Excellence Award* winner with a push notification. It uses a native *Send Push Notification* process type. Notice that *here* the *To* property uses the username (e.g. `SMITH`) instead of an email address.

> **Tip**
>
> Your app can confidently send push notifications knowing the APEX engine only *actually* delivers them to user devices that have *opted-in* to receive them.

Copyright © 2026, Oracle and/or its affiliates.

Figure 18-12 Sending Push Notification to Excellence Award Recipient



18.4 Serializing Background Process Jobs

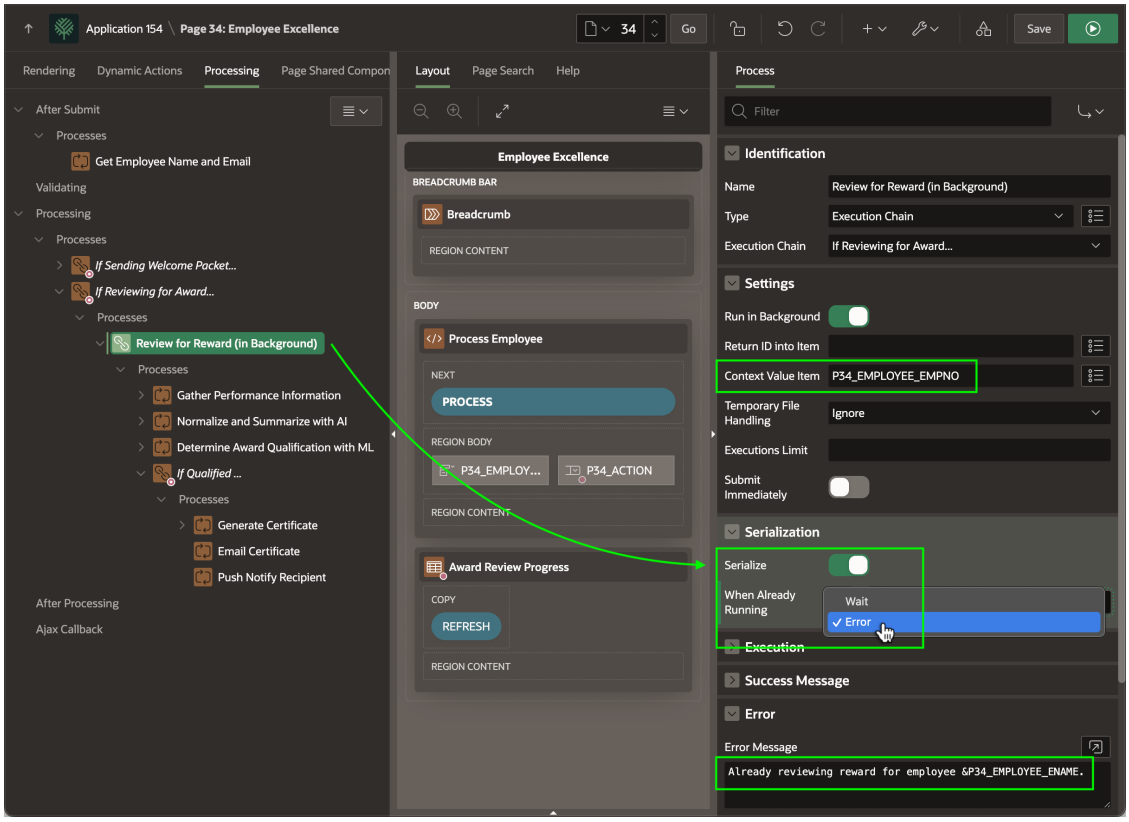
Serialize background jobs by type or context value to prevent conflicting work.

When you enable the *Serialize* switch for an execution chain that runs in the background, the APEX engine lets only one job of that type run at a time *for any user of the system*. This can be useful to avoid multiple background processes from modifying the same rows at the same time.

When you also configure a *Context Value Item*, the jobs serialize with respect to that context value. For example, the *Employee Excellence* page configures *Context Value Item* to come from Select One page item `P34_EMPLOYEE_EMPNO`. This means *Susan* or other HR reps can process the award reviews for employees 7876 (ADAMS) and 7499 (ALLEN) and 7698 (BLAKE) at the same time. The figure shows the **Review for Reward (in Background)** execution chain selected in the processing tree and highlights its **Serialization** properties and the configured error message a user would see if the job can't be scheduled.

Copyright © 2026, Oracle and/or its affiliates.

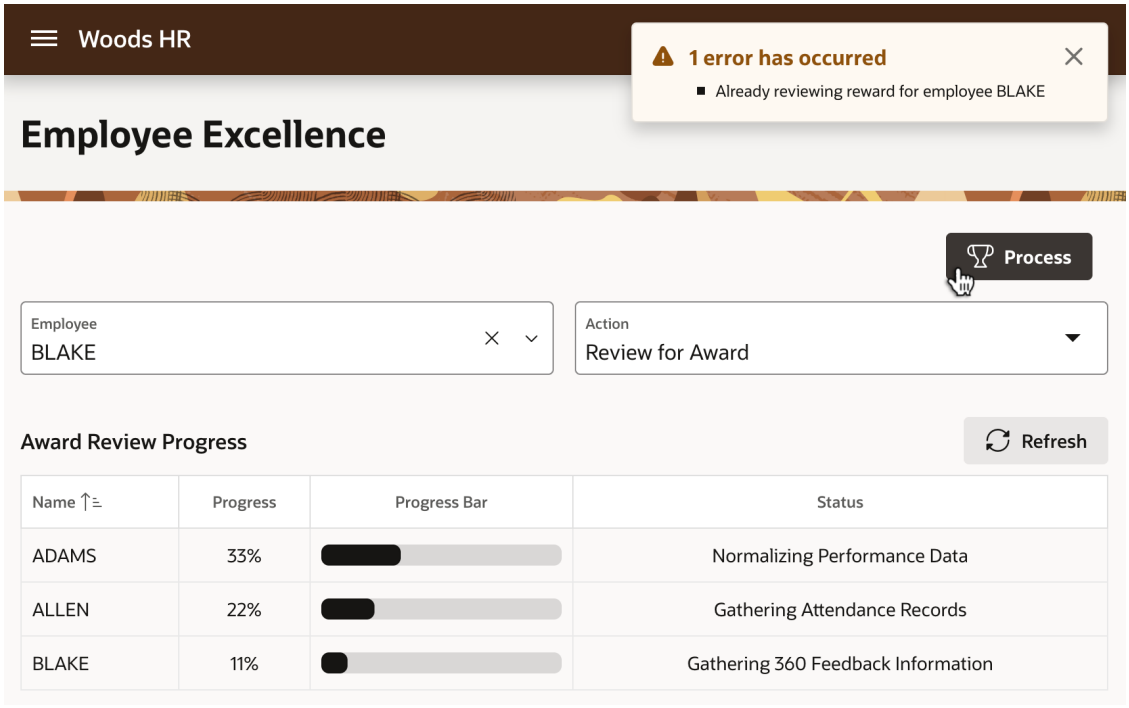
Figure 18-13 Serializing Background Process Job on Context Value



However, if some HR rep tries to initiate an award review process for an employee already being actively processed by a background process job, it must either wait or give an error. As shown above, the *Employee Excellence* page configures the *When Already Running* property of the execution chain to *Error* and configures an *Error Message*. This means *Susan*'s attempt to process BLAKE when that employee is already being processed, shows the error below. The figure shows Susan's view of the Employee Excellence page when she tries to initiate the award review process for BLAKE when one is already ongoing in the background for that employee.

Copyright © 2026, Oracle and/or its affiliates.

Figure 18-14 Background Process Error if Context Value Already Being Processed



18.5 Understanding Cloned Session State

Understand how background jobs use an isolated copy of the user's session state.

The APEX engine simplifies background processing by cloning the user session state to the scheduled job and ensuring any session state changes it makes affect only its own private copy.

- Isolating Background Session State
  APEX gives each background job its own session state so its changes stay separate from the user session.
- Configuring Temporary File Handling
  Configure how a background execution chain handles temporary files from upload items.

18.5.1 Isolating Background Session State

APEX gives each background job its own session state so its changes stay separate from the user session.

When the APEX engine enqueues an execution chain's background process job, it:

- Assigns a unique execution id to the new background process
- Creates a new session for *exclusive* use by the new process
- Clones the user session state to the new background process session.

From that moment, the background process has its own session state, separate from the user's. Each background process can read and write its own state as needed. However, changes in one background process do not affect the user session or any other background process.

Copyright © 2026, Oracle and/or its affiliates.

18.5.2 Configuring Temporary File Handling

Configure how a background execution chain handles temporary files from upload items.

Processing user-uploaded files in the background is a common use case when the processing may be long-running or the user does not need to wait for the result. Use the *Temporary File Handling* property on the execution chain to control the behavior of user-uploaded files in temporary storage. Supported values of this setting include:

- *Ignore* – files remain in the user session for processing,
- *Move* – files move to the background execution chain's new session, or
- *Copy* – files remain in the user session and are copied to the new background session.

When using *Move* or *Copy*, set the additional property *Temporary File Items* to a comma-separated list of *File Upload* or *Image Upload* items whose temporary files you want to affect. The background session works with the files in the same way as in the user session using the `APEX_APPLICATION_TEMP_FILES` table.

18.6 Reporting Background Progress

Show users background job progress with the `APEX_APPL_PAGE_BG_PROC_STATUS` view.

The `APEX_APPL_PAGE_BG_PROC_STATUS` view lets you access the information and status of all background jobs, including the dynamic status message and percent complete details your own code sets to keep users informed.

- Reporting Background Process to Users
  Report background process status and progress from PL/SQL as each step completes.
- Accessing Background Process Status
  Use the `APEX_APPL_PAGE_BG_PROC_STATUS` view to show users the progress of background processes.
- Querying Progress of Background Process
  Query active background jobs to show status messages and percent complete.
- Showing Background Process Progress Bar
  Show background progress as a bar using the Percent Graph report column type.
- Stretching Classic Report's Last Column
  Apply a CSS class to make the last Classic Report column fill the remaining width.
- Clamping Text to a Fixed Number of Lines

18.6.1 Reporting Background Process to Users

Report background process status and progress from PL/SQL as each step completes.

To report progress and current status of a background process, use `SET_PROGRESS` and `SET_STATUS` procedures in the `APEX_BACKGROUND_PROCESS` package. As shown below, your code sets the total amount of work as a number in some appropriate units. It also sets the work completed so far using another number in the same units.

The package code related to the *Employee Excellence* award review processing establishes a total of 9 units of `TOTALWORK`, and increments the `SOFAR` value as it completes each of the 9 different steps. The `GATHER_PERFORMANCE_INFORMATION` below accomplishes three of the nine

Copyright © 2026, Oracle and/or its affiliates.

total steps. At each step the code updates the current background process status and progress.

> **Tip**
>
> The `APEX_BACKGROUND_PROCESS` procedures implicitly work on the currently executing background process id. If necessary, call the `GET_CURRENT_EXECUTION` function in this package to access a `T_EXECUTION` record with key information about the current executing background process including the execution `ID`, `STATE`, `LAST_STATUS_MESSAGE`, `SOFAR`, and `TOTALWORK`.

```
-- In package eba_demo_woodshr_reward
c_steps constant pls_integer := 9;

procedure gather_performance_information(
    p_empno in number)
is
begin
    apex_background_process.set_progress(p_totalwork => c_steps, p_sofar =>
0);
    apex_background_process.set_status('Gathering Project Management
Details');
    --     ▒
    apex_background_process.set_progress(p_totalwork => c_steps, p_sofar=> 1);
    apex_background_process.set_status('Gathering 360 Feedback
Information');
    --     ▒
    apex_background_process.set_progress(p_totalwork => c_steps, p_sofar =>
2);
    apex_background_process.set_status('Gathering Attendance
Records');
    --     ▒
    apex_background_process.set_progress(p_totalwork => c_steps, p_sofar =>
3);
    apex_background_process.set_status('Gathering Completed');
end gather_performance_information;
```

18.6.2 Accessing Background Process Status

Use the `APEX_APPL_PAGE_BG_PROC_STATUS` view to show users the progress of background processes.

If you configure an execution chain's *Return ID into Item* property, use the execution ID saved in the target page item to query this view by `EXECUTION_ID` to find that particular background job's status.

The diagram below shows how foreign key values in the `APEX_APPL_PAGE_BG_PROC_STATUS` view relate to `APEX_APPLICATION_PAGE_PROC` and `APEX_WORKSPACE_SESSIONS` views:

- `PROCESS_ID` – page process definition the background job is an instance of
- `CURRENT_PROCESS_ID` – currently executing child page process definition in the chain
- `SESSION_ID` – session that *initiated* the background process

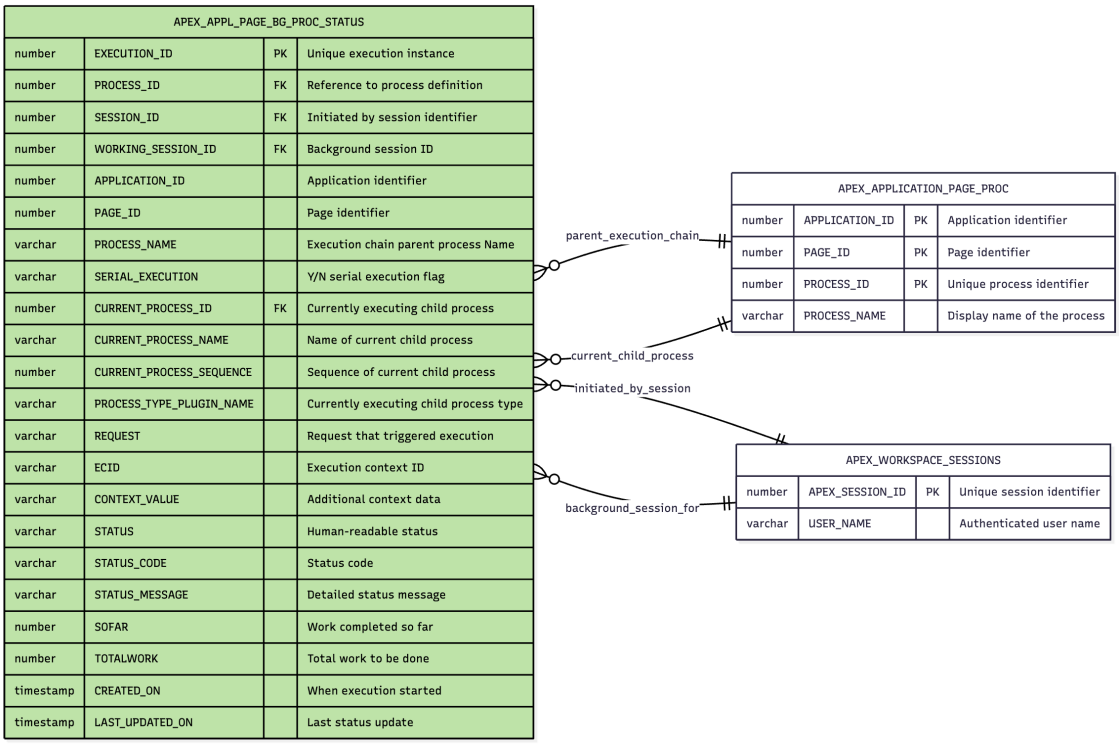
Copyright © 2026, Oracle and/or its affiliates.

- `WORKING_SESSION_ID` – private session for the background process

> **Tip**
>
> Notice that for filtering convenience, the view includes the `PROCESS_NAME` and `CURRENT_PROCESS_NAME` columns already.

The figure shows how the dictionary view `APEX_APPL_PAGE_BG_PROC_STATUS` relates to two other views containing information about the page processes and sessions via the foreign key columns described above.

Figure 18-15 Views Providing Progress of Background Processes



18.6.3 Querying Progress of Background Process

Query active background jobs to show status messages and percent complete.

The *Employee Excellence* page's *Award Review Progress* classic report region uses the following query to show the progress of all active background processes related to the page. It filters on the current page's process name: `Review for Reward (in Background)`. Notice it only includes rows with `STATUS_CODE` values of `ENQUEUED`, `SCHEDULED`, and `EXECUTING`.

The query computes the percent complete by dividing the `SOFAR` value by the `TOTALWORK`. The `PROGRESS` select list column multiplies this ratio by 100 to display the value as a percentage. The `PROGRESS_BAR` column leaves the value as the decimal the *Percent Graph* report column type expects.

To show the employee *name* being processed, it joins the `EBA_DEMO_EMP` table using the `CONTEXT_VALUE` associated with each background process job.

Copyright © 2026, Oracle and/or its affiliates.

> **Tip**
>
> Other `STATUS_CODE` values include `SUCCESS`, `FAILED`, and `ABORTED`.

```
select e.ename,
       case when bgp.sofar is not null and bgp.totalwork is not null
            then round(bgp.sofar/bgp.totalwork*100)||'%'
            else '
       end progress,
       nvl(round(bgp.sofar/bgp.totalwork*100),0) progress_bar,
       coalesce(bgp.status_message,'
  from apex_appl_page_bg_proc_status bgp
  join eba_demo_emp e
    on e.empno = to_number(bgp.context_value)
 where    bgp.process_name = 'Review for Reward (in Background)'
      and bgp.application_id = :APP_ID
      and bgp.page_id        = :APP_PAGE_ID
      /* Background Job is Waiting or Doing Work */
      and bgp.status_code in ( 'ENQUEUED','SCHEDULED', 'EXECUTING')
```

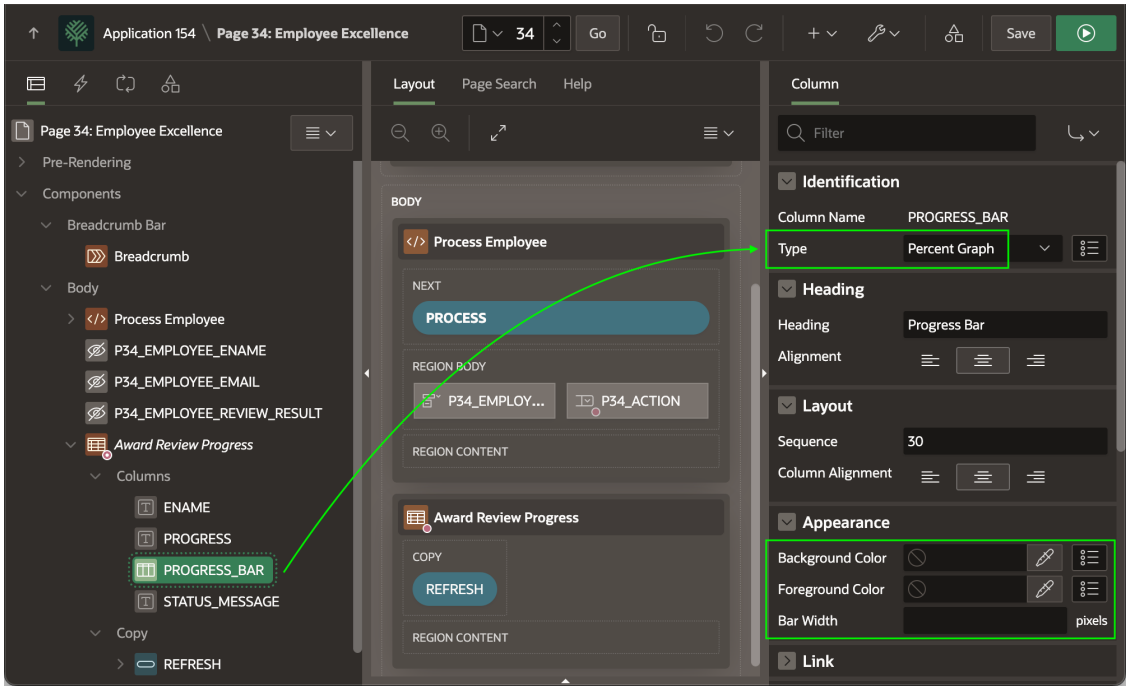
18.6.4 Showing Background Process Progress Bar

Show background progress as a bar using the Percent Graph report column type.

Use the report column type of *Percent Graph* to show a decimal value between 0 and 1 as a progress bar. If needed, configure custom background and foreground colors and an explicit bar width in the *Appearance* section. The figure shows the `PROGRESS_BAR` column of the **Award Review Progress** region selected in the component tree and highlights the column **Type** is set to **Percent Graph** as well as the available **Appearance** options.

Copyright © 2026, Oracle and/or its affiliates.

Figure 18-16 Displaying Progress Percentage as a Inline Bar Graph

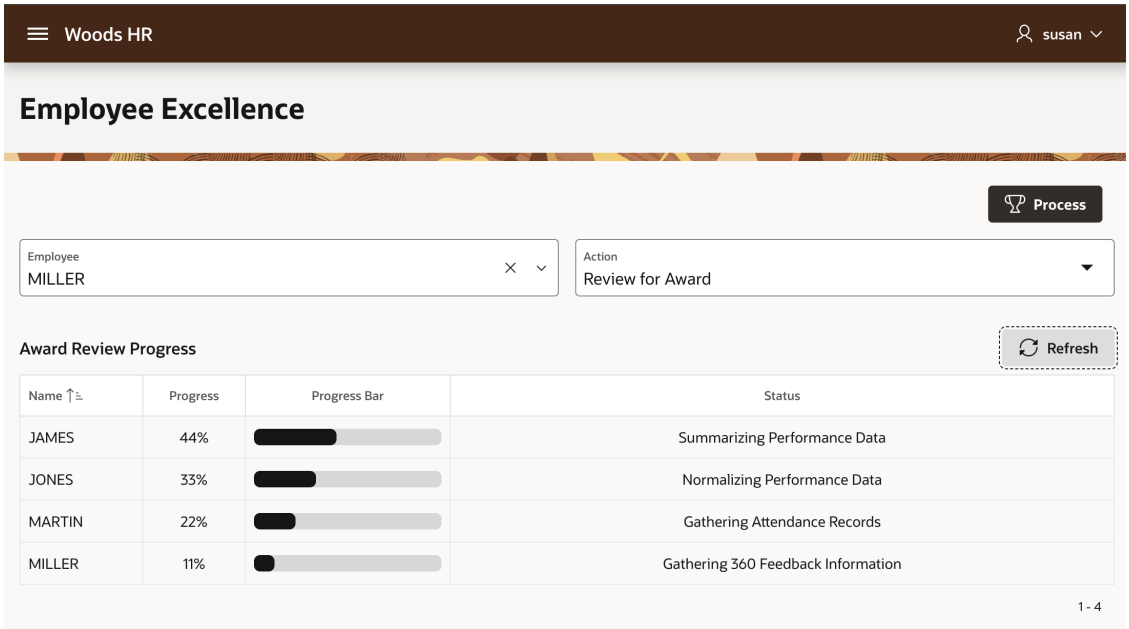


18.6.5 Stretching Classic Report's Last Column

Apply a CSS class to make the last Classic Report column fill the remaining width.

The *Award Review Progress* classic report in the *Employee Excellence* page shown below has the employee name being processed, and the percentage progress both as a percent and a bar graph. The *Status* column uses the rest of the horizontal space available in the page width.

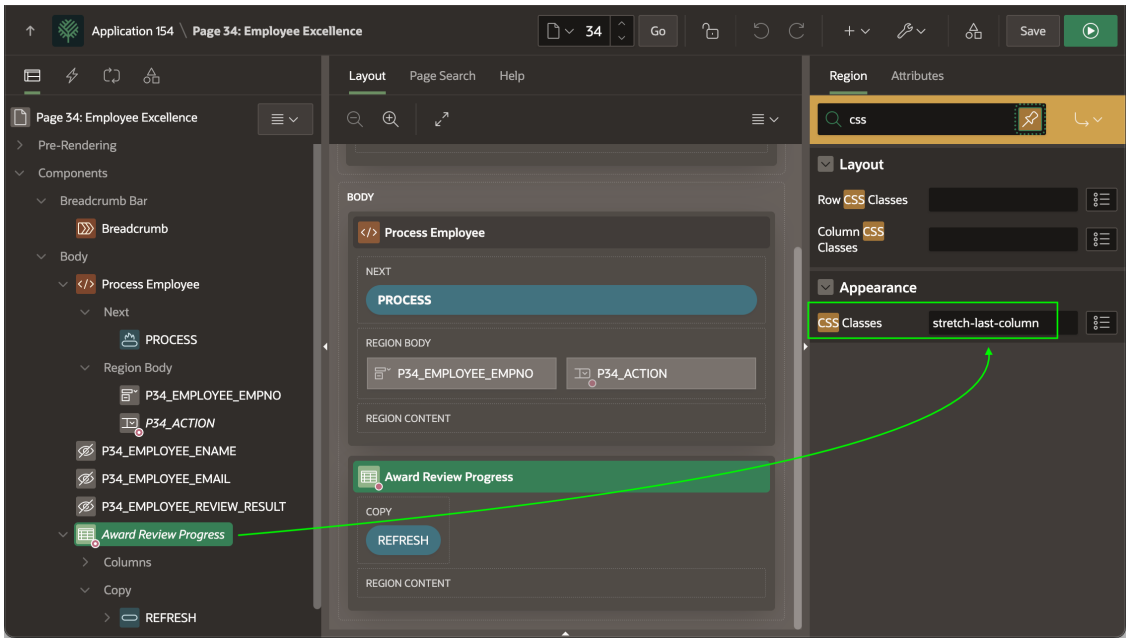
Figure 18-17 Stretching Last Classic Report Column



Copyright © 2026, Oracle and/or its affiliates.

As shown below, the report configures this behavior by applying a custom CSS class named `stretch-last-column` in the *CSS Classes* region property.

Figure 18-18 Stretching Classic Report's Last Column Using CSS Class



This `stretch-last-column` CSS class solves a common need in Classic Report regions: stretching the last column to use the remaining width. As explained in Configuring Application-Wide Style Rules, it's part of the *Woods HR* application's `app.css` file. The technique uses two CSS rules that override the browser's default table layout.

First, it applies `table-layout: fixed` and `width: 100%` to the table. By default, browsers use automatic layout, sizing columns by content and space, often with unpredictable results. Fixed layout makes the table obey your width values and stretch across its container.

Next, it uses CSS selectors to handle columns with or without explicit *Cell Width* specified. The selector `:not([data-col-width])` matches columns that lack a defined width. Those non-last columns get a fixed width (`120px` in this example) so they stay readable but don't hog space. The last column gets `width: auto`, which under fixed layout means "use the rest."

You can easily apply the same class to classic reports across the app. When you need custom sizing, a report column's *Cell Width* setting still wins. The CSS rules appear below:

```
/* In app-wide CSS Static Application File app.css */
/* Generic stretch-last-column class for APEX Classic Reports */
.stretch-last-column .t-Report-report {
    width: 100% !important;
    table-layout: fixed;
}

/* Force table wrapper to full width */
.stretch-last-column .t-Report-tableWrap {
    width: 100%;
}
```


Copyright © 2026, Oracle and/or its affiliates.

```
/*
 * Give all non-last columns without explicit widths a reasonable default
 * The key is using a percentage that works regardless of column count
 */
.stretch-last-column .t-Report-report th:not(:last-child):not([data-col-
width]),
.stretch-last-column .t-Report-report td:not(:last-child):not([data-col-
width]) {
    width: 120px; /* Fixed pixel width - adjust this value as needed */
}

/* Let the last column stretch to fill remaining space (unless it has
explicit width) */
.stretch-last-column .t-Report-report th:last-child:not([data-col-width]),
.stretch-last-column .t-Report-report td:last-child:not([data-col-width]) {
    width: auto;
}
```

18.6.6 Clamping Text to a Fixed Number of Lines

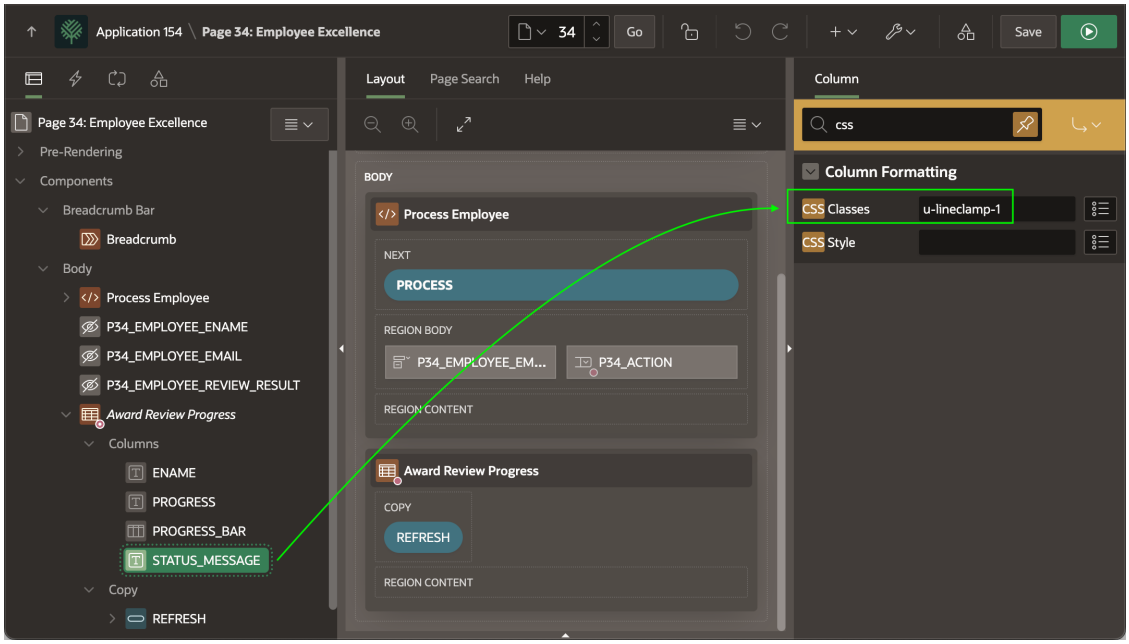
Universal Theme includes many pre-defined CSS classes to easily customize how pages look. Reference the complete set using the *Reference* section of the Universal Theme Reference Application online. One group in particular comes in handy for reports that display text of *unpredictable* length. The *Status* column of background processes falls in this category.

Regardless of text length, sometimes it's useful to limit a report column's display to a fixed number of lines. The *line clamp* classes are the solution. By applying the `u-lineclamp-1` class to the `STATUS_MESSAGE` report column, the *Award Review Progress* region only display a single line of text. As shown below, set it on the *CSS Classes* property of the report column.

> **Tip**
>
> To clamp text to *n* lines, use class `u-lineclamp-n` with `n` from 1 to 5.

Copyright © 2026, Oracle and/or its affiliates.

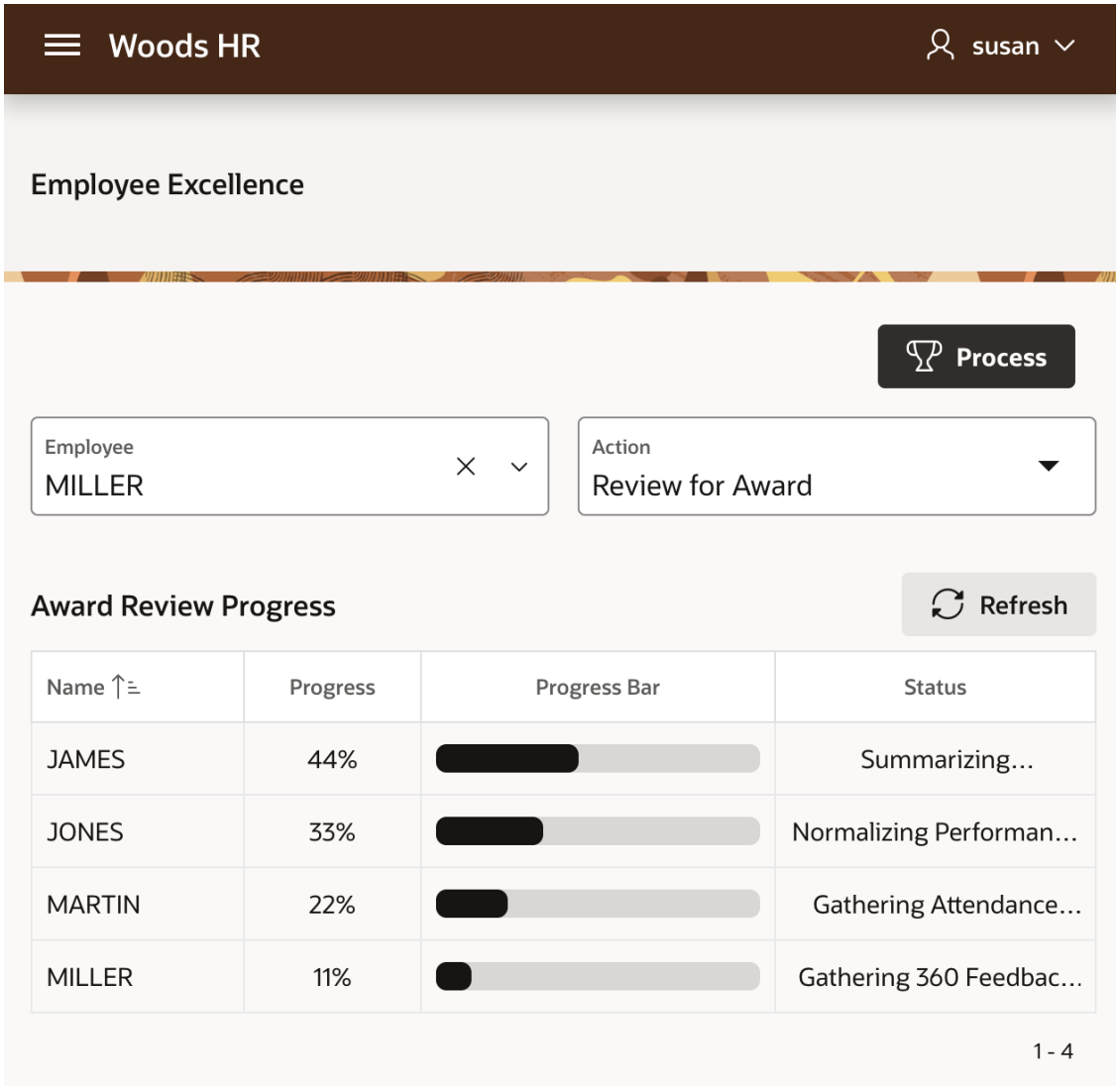
Figure 18-19 Clamping Status Message Text to a Single Line



At runtime, if text in the *Status* column exceeds the "clamped" number of lines the browser automatically adds an ellipsis instead of word-wrapping onto more lines.

Copyright © 2026, Oracle and/or its affiliates.

Figure 18-20 Ellipsis Displays when Text Length Exceeds Clamped Number of Lines



18.7 Loading Data in the Background

Load uploaded data in the background and notify users when processing completes.

To load user-uploaded data in the background, use an execution chain with:

- *Run in Background* enabled,
- *Temporary File Handling* set to *Move* or *Copy*, and
- A *Data Loading* child process

If appropriate, the execution chain can include *Send Email* or *Send Push Notification* child processes to notify the user that the data load has completed. Since the background job has access to a private copy of the user session state at the time the background job began, it can reference page item values like `P34_EMPLOYEE_EMAIL` and `P34_EMPLOYEE_NAME` as well as `APP_USER` when performing these kinds of notifications.

Copyright © 2026, Oracle and/or its affiliates.

18.8 Limiting and Throttling Background Jobs

Limit or throttle background execution chain jobs at the chain, application, or workspace level.

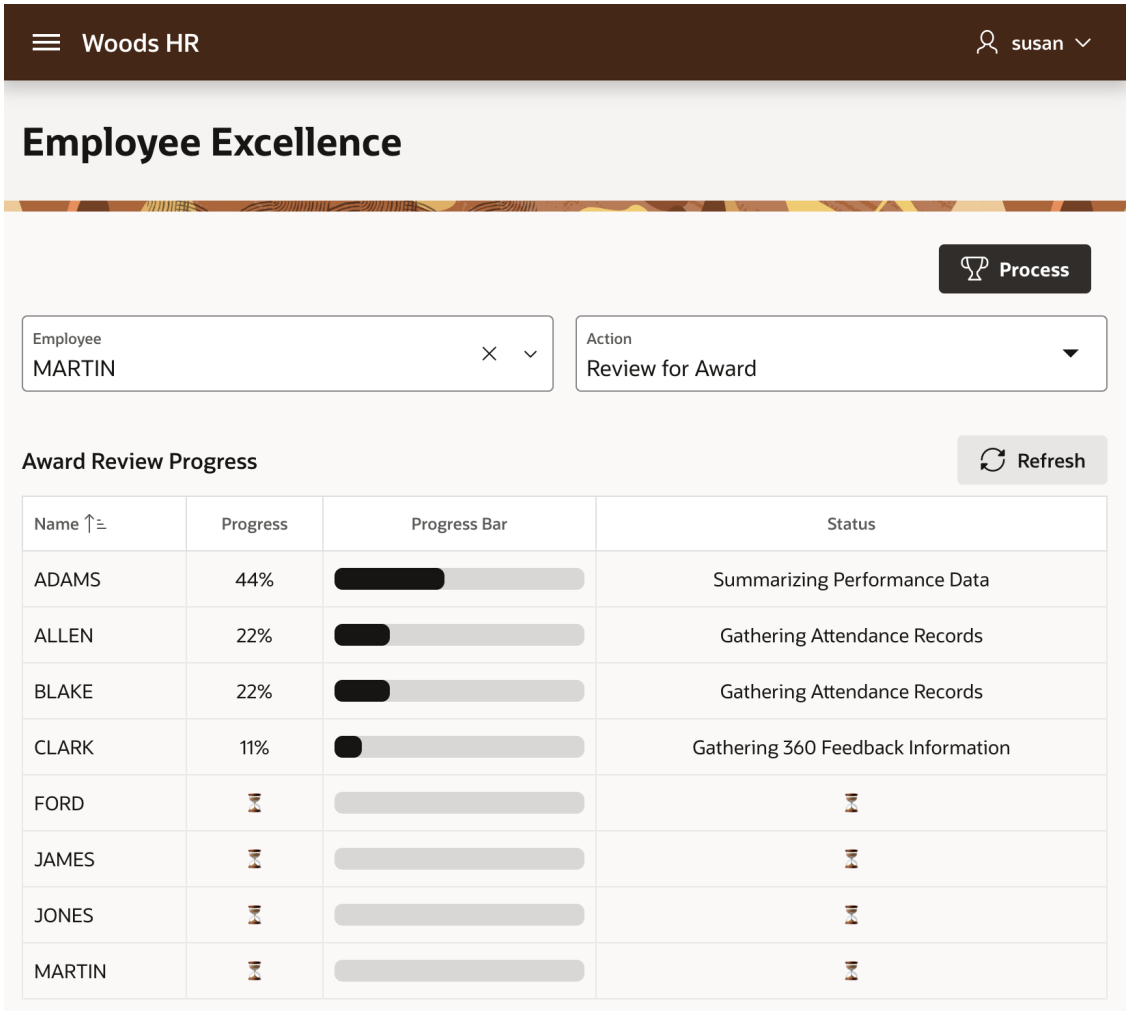
Both you and your APEX instance administrator can limit the number background execution chain jobs. Ranging from most specific to most general, these controls include the:

- *Executions Limit* on the execution chain
  - Raises an error if user exceeds this number of background executions of that chain.
- *Maximum Background Page Process Jobs* on the application definition
  - Throttles queued background jobs to stay within this limit. Others wait their turn to execute.
- *Maximum Background Page Process Jobs* on the workspace
  - Lets instance administrator throttle queued background jobs for all applications in the workspace. Others wait for their turn to execute. If set, it overrides the instance parameter `MAX_PROCESS_SCHEDULER_JOBS_DEFAULT` value.

For example, if the *Woods HR* application sets a *Maximum Background Page Process Jobs* to 4, then as shown below, employee award review processes wait their turn to be scheduled to stay within this limit. The figure shows Susan's view of the Employee Excellence page. The *Award Review Progress* section shows four executing background processes and four others with an hourglass emoji in their **Progress** and **Status** column indicating they are waiting to be scheduled for background execution.

Copyright © 2026, Oracle and/or its affiliates.

Figure 18-21 Enqueued Background Jobs Wait Their Turn to Execute



18.9 Adjusting Resources for Background Jobs

Configure a custom job class when background process jobs need different resource limits.

APEX instance administrators who apply resource constraints to ensure a good service for all workspaces and applications can configure a *distinct* job class for background process jobs if necessary. A DBA user or one with `APEX_ADMINISTRATOR_ROLE` can execute the following PL/SQL to configure a custom background job process class. For example, this class might enable background jobs to use more resources for longer-running processing use cases.

```
-- Run as SYS or user with APEX_ADMINISTRATOR_ROLE
begin
   apex_instance_admin.set_parameter(
      p_parameter => 'BACKGROUND_PROCESS_JOB_CLASS',
      p_value     => 'CUSTOM_BG_PROC_CLASS'
   );
end;
```


Copyright © 2026, Oracle and/or its affiliates.

18.10 Debugging Background Processes

Debug background execution chains by viewing jobs, session state, and failure details.

When using background execution chains, use the *Debug Viewer* and *Monitor Activity* pages to view background jobs, terminate them if needed, explore their session state, and investigate the cause of jobs that fail with an unexpected error.

- Reviewing App Background Processes
  Review and terminate application background processes from the Debug Viewer.
- Investigating Background Process Errors
  Investigate failed background jobs by reviewing their debug trace logs and error details.
- Viewing Background Process Session State
  View a background job's working session to inspect its session state during debugging.

18.10.1 Reviewing App Background Processes

Review and terminate application background processes from the Debug Viewer.

Use *Debug* > *View Debug* from the Developer Toolbar to open the *Debug Viewer* page. As shown below, on its *Session* tab, choose *Background Executions* in the *View* select list and click *(Set)* to review background processes for the current application.

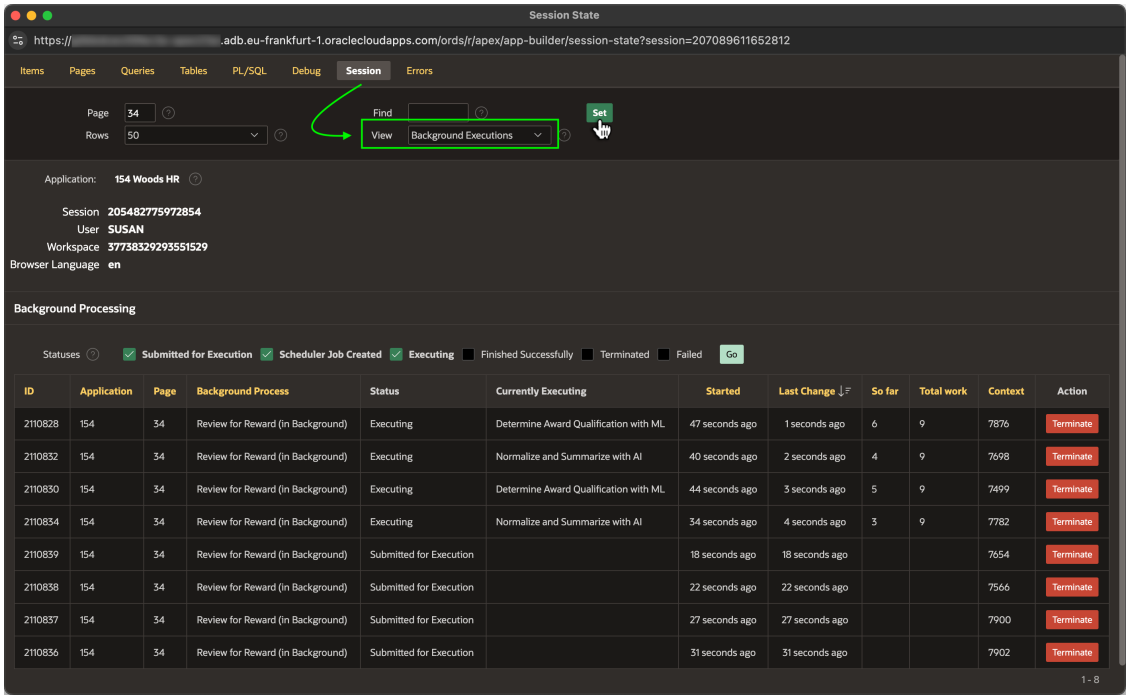
Notice you can see each job's status, *Currently Executing* status message, *So far*, *Total work*, and *Context* values. Recall that the *Review for Reward (in Background)* execution chain configures the employee ID to provide this context value, so the values shown represent the employee being reviewed for a *Employee Excellence Award* by each particular background process.

Use the *Statuses* checkboxes to refine the list to see the processes that interest you. By default, it shows jobs that have been enqueued, scheduled, or are executing.

Click the *(Terminate)* button in any row to abort that background job immediately.

Copyright © 2026, Oracle and/or its affiliates.

Figure 18-22 Inspecting Background Executions for the Current Session

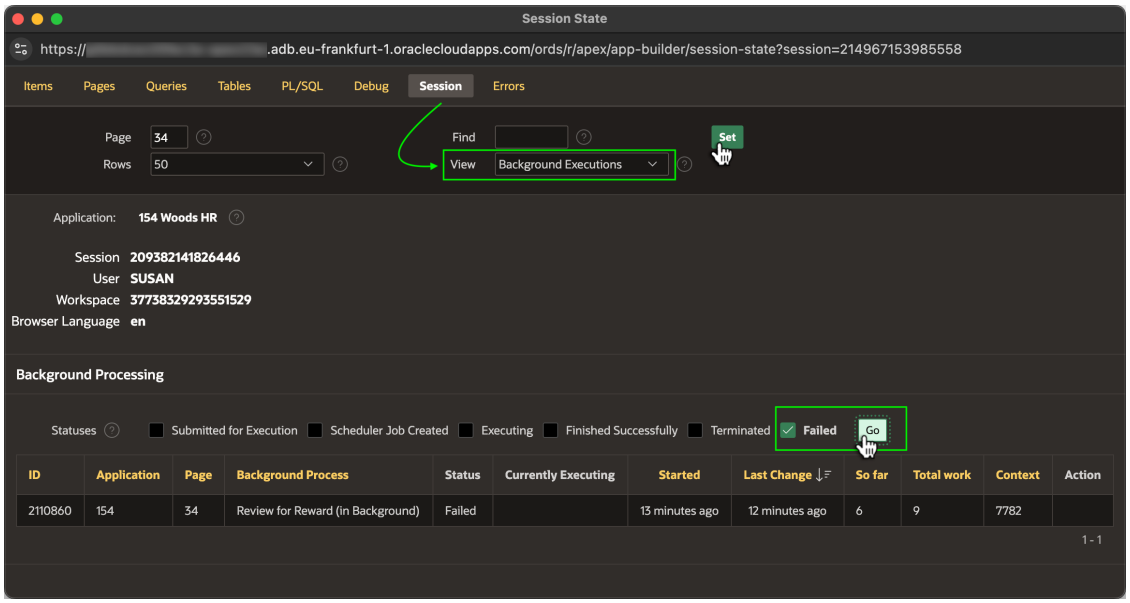


18.10.2 Investigating Background Process Errors

Investigate failed background jobs by reviewing their debug trace logs and error details.

When a background job encounters an unexpected error, start with the *Debug Viewer* page's *Session* tab. Set *View* to *Background Executions* and click *(Set)*. Then filter for jobs with a *Failed* status as shown below.

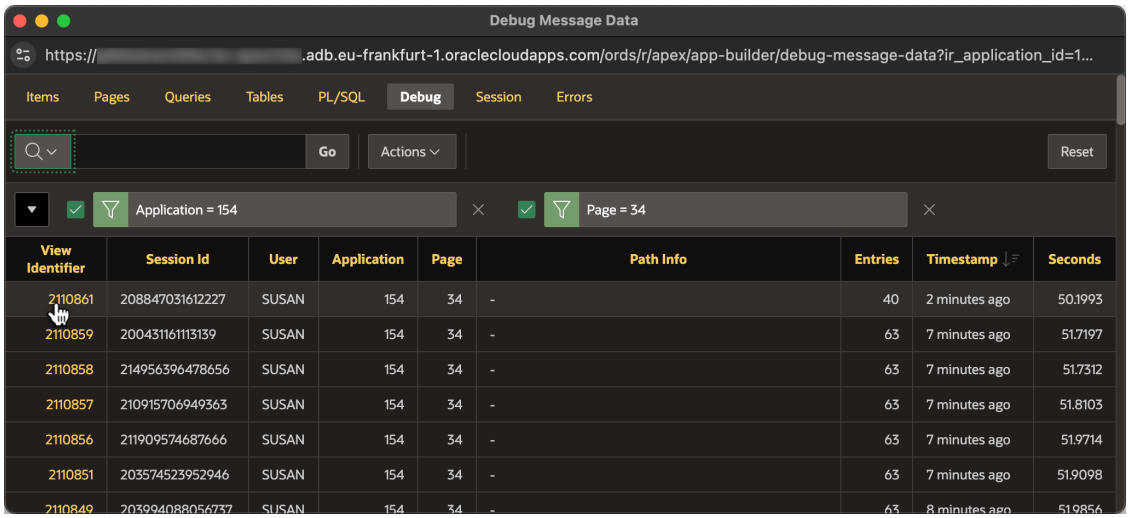
Figure 18-23 Viewing Failed Background Execution Jobs



Copyright © 2026, Oracle and/or its affiliates.

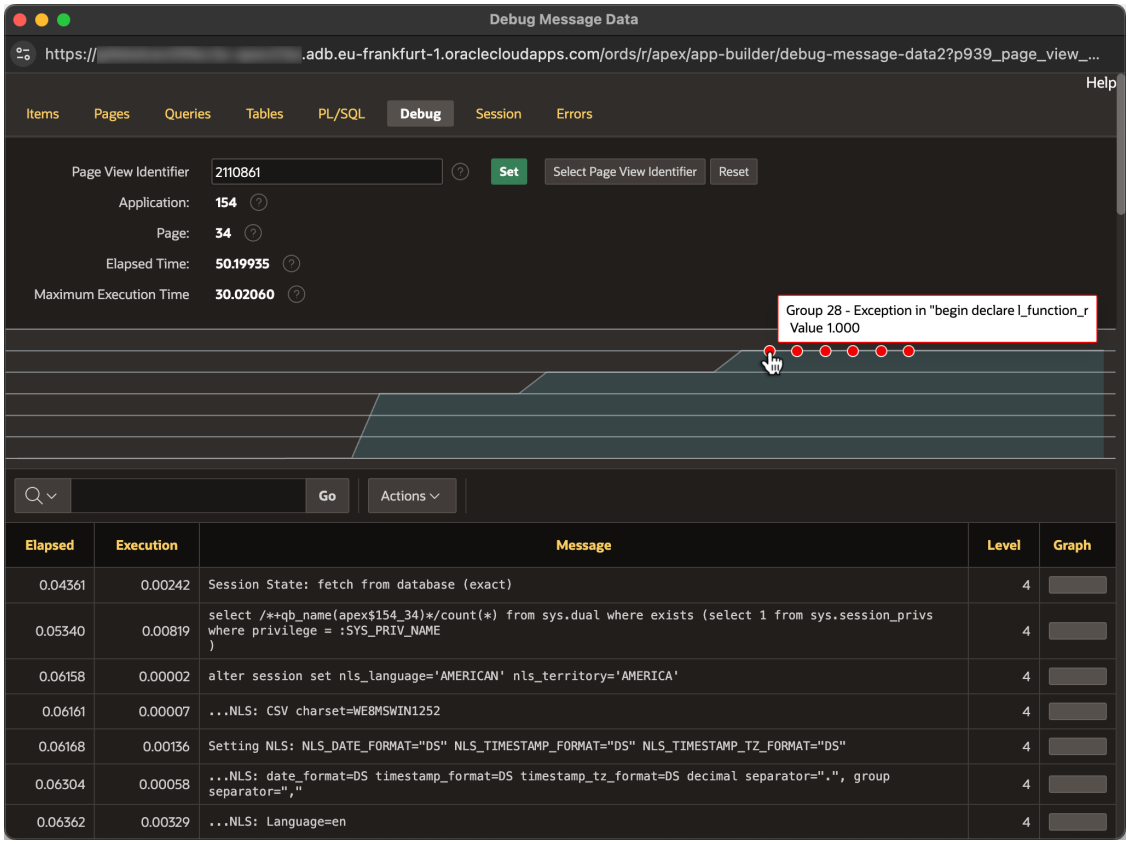
Then, on the *Debug* tab sort descending by *Timestamp* and drill into the debug trace log for the candidate process as shown below.

Figure 18-24 Identifying the Debug Trace Log for the Background Job



Any errors that occurred show as red dots on the debug trace timeline. As shown below, click the first red dot to jump to the relevant part of the debug trace log.

Figure 18-25 Jumping to First Error in the Debug Trace Timeline

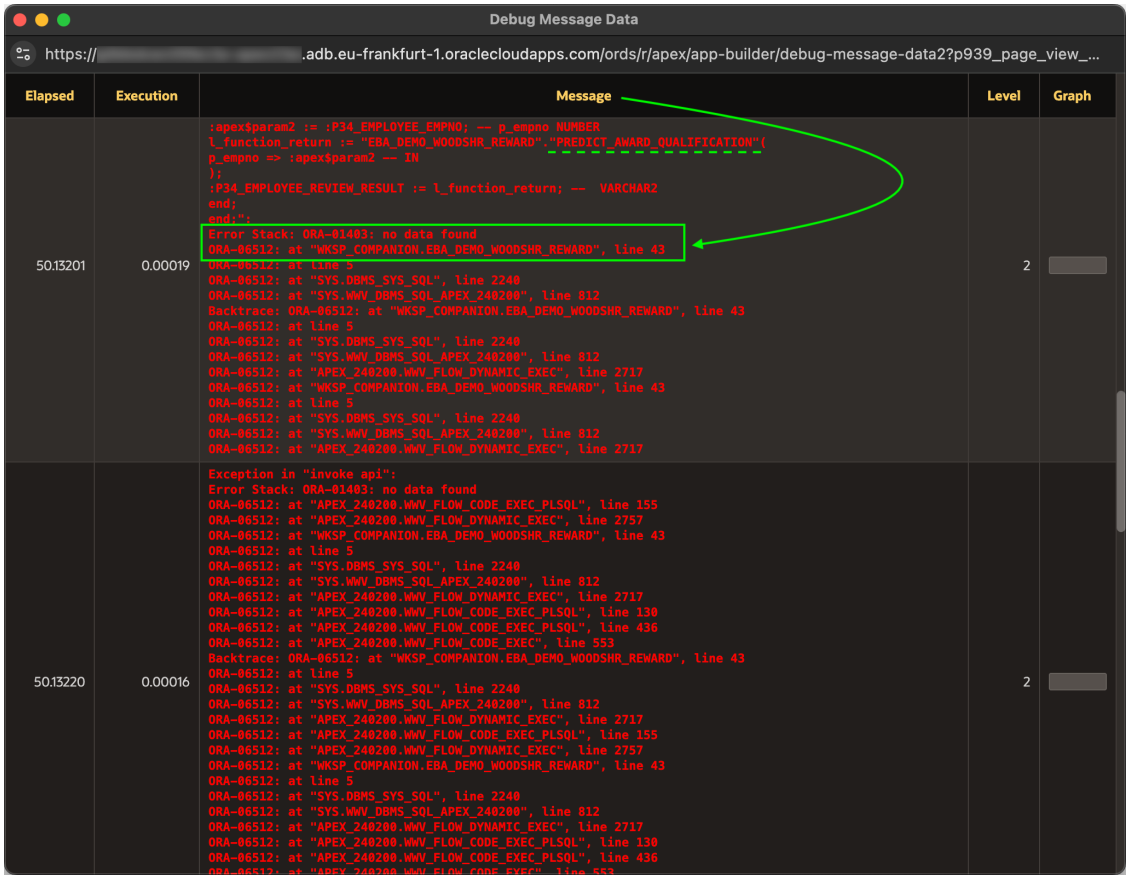


Copyright © 2026, Oracle and/or its affiliates.

Review the error information. As shown below, the stack trace helps you see that line 43 of the `EBA_DEMO_WOODSHR_REWARD` package encountered a `NO_DATA_FOUND` exception during the execution of the Invoke API that calls `PREDICT_AWARD_QUALIFICATION`. With this info, you can more quickly identify the cause in your code.

When needed, you can enable debug mode and add additional `APEX_DEBUG.INFO` debugging messages into your code to add extra debugging context information to the logs to further narrow down the culprit.

Figure 18-26 Investigating Details of a Background Job in FAILED State



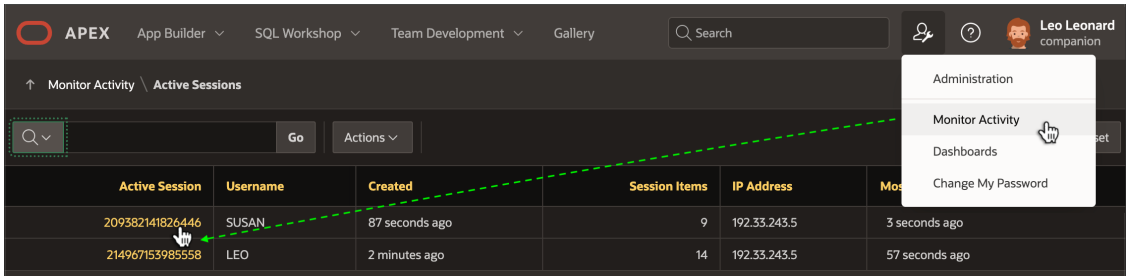
18.10.3 Viewing Background Process Session State

View a background job's working session to inspect its session state during debugging.

To view the session state of a background process, start by opening the *Monitor Activity* page from the "person with wrench" menu button in the App Builder as shown below. Then identify a user session like `SUSAN`'s and click its session id link to drill down for more information.

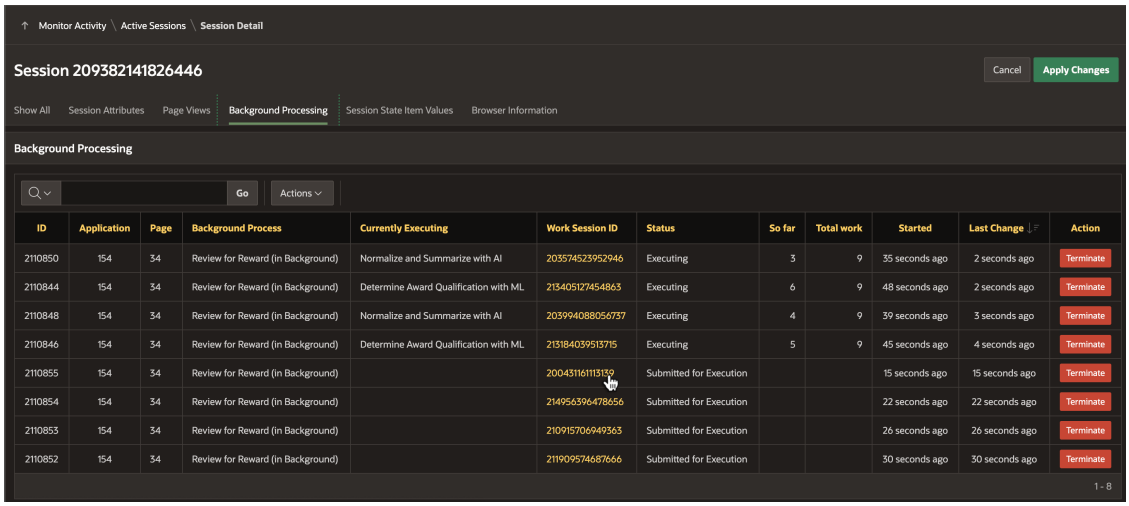
Copyright © 2026, Oracle and/or its affiliates.

Figure 18-27 Drilling Down into an Active Session for a User



On the *Session Detail* page, select the *Background Processing* tab as shown below to see all its scheduled and running jobs. To view the session state of a particular background job's session, just click on the appropriate session id link in the *Work Session ID* column.

Figure 18-28 Reviewing Background Executions for Specific User Session

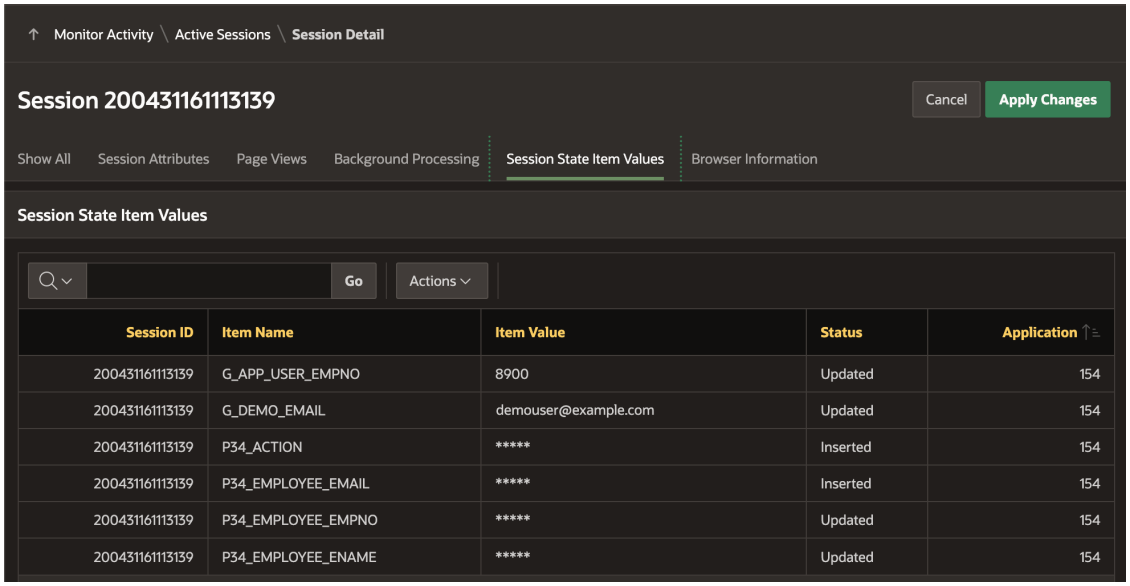


This opens the *Session Detail* page for the background job's working session. From there, select the *Session State Item Values* tab. As shown below, you can then inspect the values of any session state that are not encrypted.

> **Tip**
>
> Seeing the values of *encrypted* session state for debugging purposes requires temporarily adding `APEX_DEBUG.INFO` calls into your code while you are triaging the cause of a particular problem. These additional messages can include values using `%s` placeholders like:
>
> ```
> apex_debug.info('### DEBUGGING (%s) (%s)', expression1, expression2);
> ```


Copyright © 2026, Oracle and/or its affiliates.

Figure 18-29 Viewing Session State for a Background Process Work Session



Copyright © 2026, Oracle and/or its affiliates.

19

Executing Repetitive Processing

Use an *Automation* in Shared Components to define repetitive processing.

- Understanding Key Automation Concepts
  An *Automation* lets you define a recurring job to run on a schedule or to invoke on demand.
- Exploring an Automation Example
  Explore a Woods HR automation that gives one employee a random Saturday salary increase.

19.1 Understanding Key Automation Concepts

An *Automation* lets you define a recurring job to run on a schedule or to invoke on demand.

Sequencing Automation Actions

Your automation performs a sequence of one or more *actions*. Each can execute code in the local database or a remote one. It can also send an email or push a notification to a user. When your app runs on Autonomous AI Database, server-side geocoding is available, too, as explained in Geocoding in the Background.

Running Actions Once, or Once per Row

The automation's action sequence can run a single time, or automatically repeat to process each row of a query you define. In the latter case, actions can reference the query column names as bind variables to reference values in the current row.

Processing Conditionally

You can configure a *Server-side condition* on both the automation itself, and any of its actions to perform work only under certain conditions.

Determining Frequency with Schedule Expression

If scheduled, a *Schedule Expression* indicates how often the job executes. For common situations, use the *Interval Builder* dialog. Otherwise, you can also enter any valid Database Scheduler syntax for more complex requirements.

Enabling or Disabling the Job as Needed

You control the automation's *Schedule Status*. If enabled, then the APEX engine runs the job on your schedule. If disabled, then it only executes if you run it interactively from App Builder's *Automations* list page or when you call `APEX_AUTOMATION.EXECUTE`.

Controlling Error Handling

By default the automation ignores errors that happen in an action, and the automation continues processing the following action and remaining rows. You can adjust this behavior to terminate the automation instead, and *either* keep it scheduled for its next run or disable it.

Copyright © 2026, Oracle and/or its affiliates.

Consulting Automation Logs

The APEX engine maintains an automation log you can consult in App Builder, or query using the `APEX_AUTOMATION_LOG` view. Your automation can log additional information, warnings, and errors to the log using the `LOG_INFO`, `LOG_WARN`, or `LOG_ERROR` procedures in the `APEX_AUTOMATION` package.

Controlling Automations Programmatically

The `APEX_AUTOMATION` package has a procedure to `EXECUTE` an automation. When doing so, one signature offers a `p_run_in_background` boolean parameter to control whether it runs in the current user session (`false`) or a new session in the background (`true`). Other procedures let you `RESCHEDULE`, `ENABLE`, `DISABLE`, or `TERMINATE` an automation as well. In addition, your action code can call `SKIP_CURRENT_ROW` to immediately continue to the next row to process, or `EXIT` to end the automation prematurely without processing remaining rows.

Configuring Supporting Logic

In addition to the automation's action logic, you can optionally configure names of an *Initialization Procedure*, a *Cleanup Procedure*, and a *Before Row Processing Procedure* as required. These can be existing procedures, or defined as part of the automation in its *Additional Code Execution* section.

Working with Session State

Automations can read and write the values of application items as temporary storage. The automation's optional *Initialization Procedure* can assign an initial value to an application item, that the *Before Row Processing Procedure* as well as actions can reference and modify if needed.

In addition, if the automation processes the rows of a query, then actions can reference the current row's data using column names as bind variables. Where useful, actions can also *update* the current row's column values, but the APEX engine ***does not automatically save*** any such changes. Your action code needs appropriate logic to update any values in the database.

Executing with No Logged-in User

Unless you use `APEX_AUTOMATION.EXECUTE` to execute an automation in the current user session, they run in the background using a private background session with no logged-in user. In this situation, `:APP_USER` returns `nobody`.

19.2 Exploring an Automation Example

Explore a Woods HR automation that gives one employee a random Saturday salary increase.

The *Woods HR* application uses an automation to randomly increase the salary of a lucky employee every Saturday.

- Winning Salary Increase Every Saturday
  Review the weekly salary increase rule that rewards one randomly selected lower-paid employee.
- Identifying Automation Rows to Process
  Use an automation source query to choose lower-paid employees and prepare row values for actions.

Copyright © 2026, Oracle and/or its affiliates.

- Configuring Supporting Automation Logic
  Use initialization code to prepare shared values before automation actions run.
- Defining Automation Actions in Sequence
  Define automation actions that decide which rows to process, perform conditional updates, and notify affected users.
- Establishing an Automation's Schedule
  Configure an automation schedule with the Interval Builder or Database Scheduler syntax.
- Running an Automation on Demand
  Use the *Automations* list page to run one on demand.
- Viewing the Automation Log
  Use the *Execution Log* tab of the *Automations* list to review recent runs.

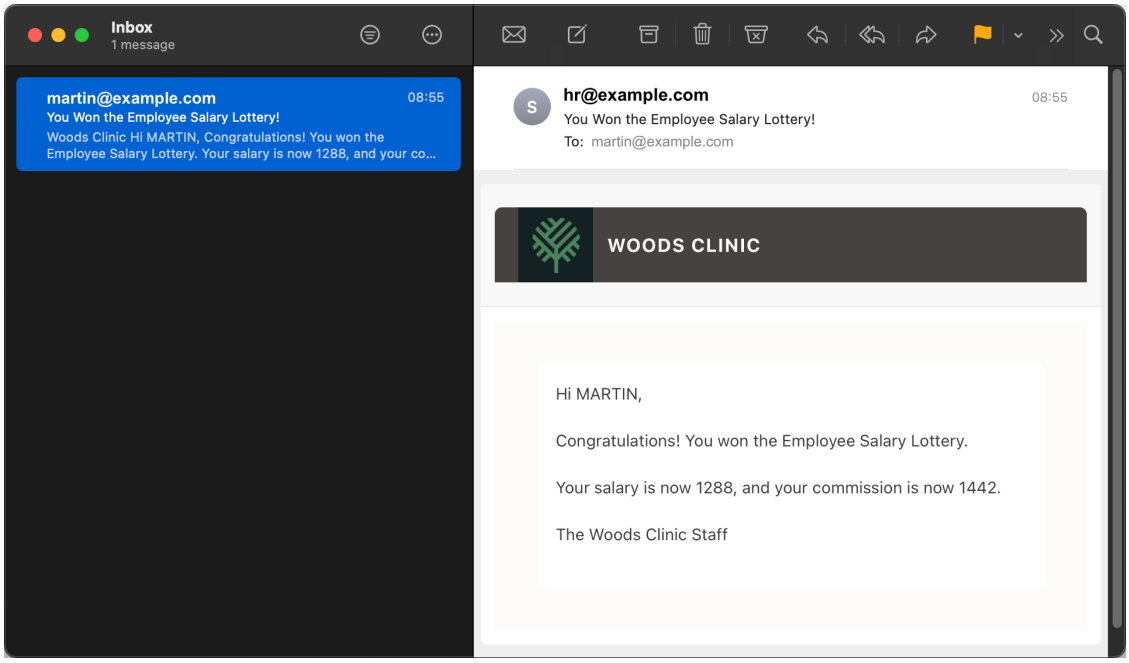
19.2.1 Winning Salary Increase Every Saturday

Review the weekly salary increase rule that rewards one randomly selected lower-paid employee.

Woods Clinic management likes to reward lower income earners on the staff by randomly increasing the salary of one lucky employee by 3% every Saturday. If the weekly winner is salesperson, then they also gets a 3% increase in their commission. Employees love the program! The figure shows employee MARTIN's email client receiving an email from `hr@example.com` with the subject line *You Won the Employee Salary Lottery!* It contains the email text:

> Hi MARTIN, Congratulations! You won the Employee Salary Lottery. Your salary is now 1288, and your commission is now 1442. The Woods Clinic Staff.

Figure 19-1 Martin Gets an Email That She's the Lucky Winner of a Raise



Copyright © 2026, Oracle and/or its affiliates.

19.2.2 Identifying Automation Rows to Process

Use an automation source query to choose lower-paid employees and prepare row values for actions.

The *Saturday Employee Lottery* automation identifies the rows to process using a source query. Since lottery candidates are lower-earning employees, it determines the salary cutoff for the bottom 25% of earners with the continuous percentile analytic function `PERCENTILE_CONT`. As shown below, the query's `cutoff` common table expression computes the salary limit for the group, then references that `SALARY_LIMIT` in the WHERE clause to retrieve employees earning that amount or less. It selects key columns from the `EBA_DEMO_EMP` table that automation actions use while processing each employee the query returns. The query aliases the `ROWNUM` pseudo-column to `RN` so each row gets an ordinal number.

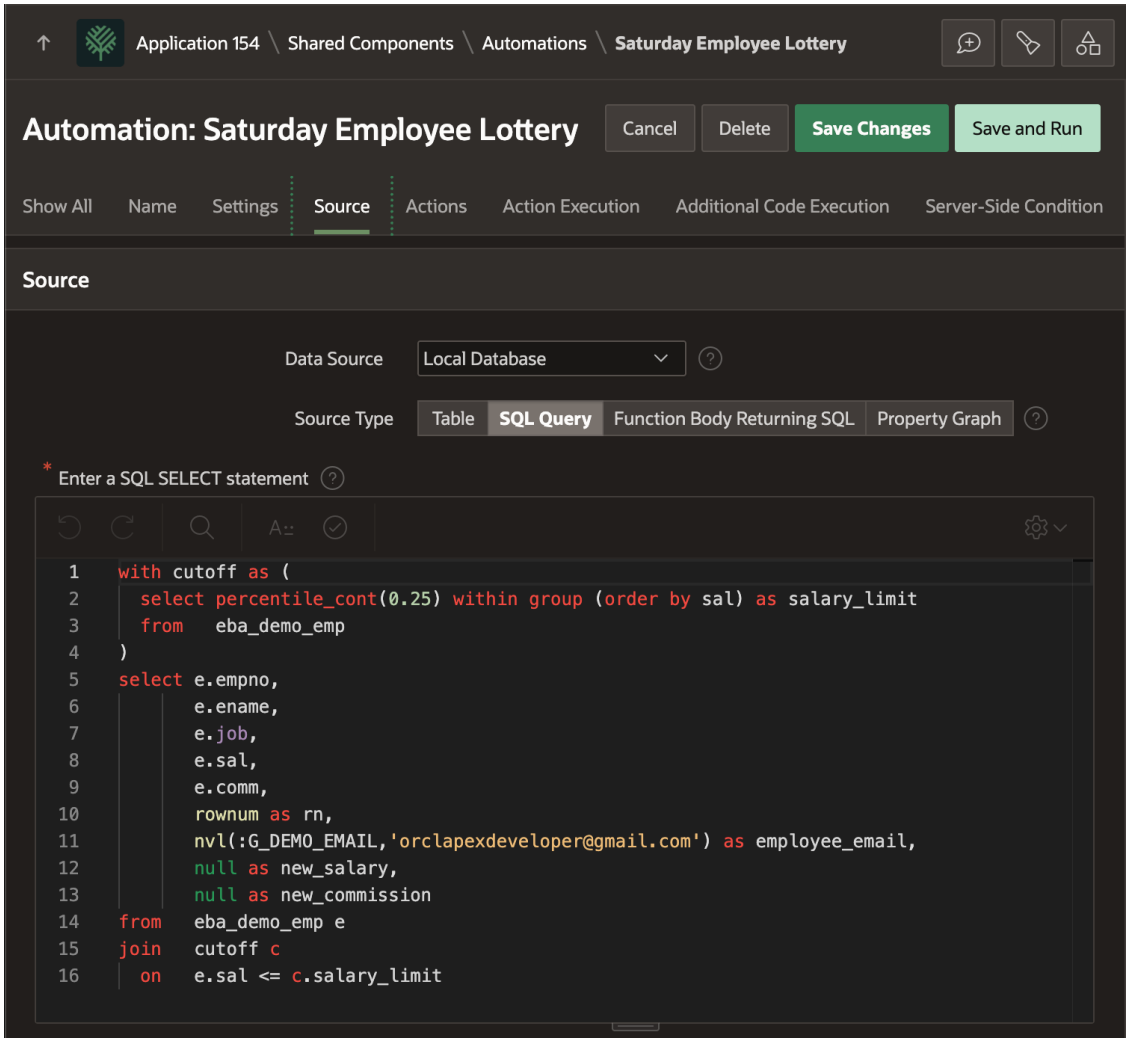
> **Tip**
>
> The query adds two extra `NULL` columns, aliased to `NEW_SALARY` and `NEW_COMMISSION`. Actions can assign values to these columns as temporary storage, passing data to later actions in the sequence. These values exist only in memory while the row is processed and are always `VARCHAR2` type.

```
with cutoff as (
  select percentile_cont(0.25)
         within group (order by sal) as salary_limit
  from   eba_demo_emp
)
select e.empno,
       e.ename,
       e.job,
       e.sal,
       e.comm,
       rownum as rn,
       nvl(:G_DEMO_EMAIL,'demo.user@example.com') as employee_email,
       null as new_salary,
       null as new_commission
from   eba_demo_emp e
join   cutoff c
  on   e.sal <= c.salary_limit
```

The figure shows the **Source** tab of the Automation edit page, showing the SQL statement above as the automation's data source.

Copyright © 2026, Oracle and/or its affiliates.

Figure 19-2 Saturday Employee Lottery Automation Source Query



19.2.3 Configuring Supporting Automation Logic

Use initialization code to prepare shared values before automation actions run.

The *Saturday Employee Lottery* automation uses an *Initialization Procedure* to pick the random number that identifies the weekly winner. As shown below, it defines the `INIT_EMP_LOTTERY` procedure in the *Additional Code Execution* section of the automation definition. It counts the low-earning employees using the same criteria as the source query. Then it assigns a random whole number between 1 and that count to the application item `G_LOTTERY_PICK`. Finally, it adds a message to the automation log.

Copyright © 2026, Oracle and/or its affiliates.

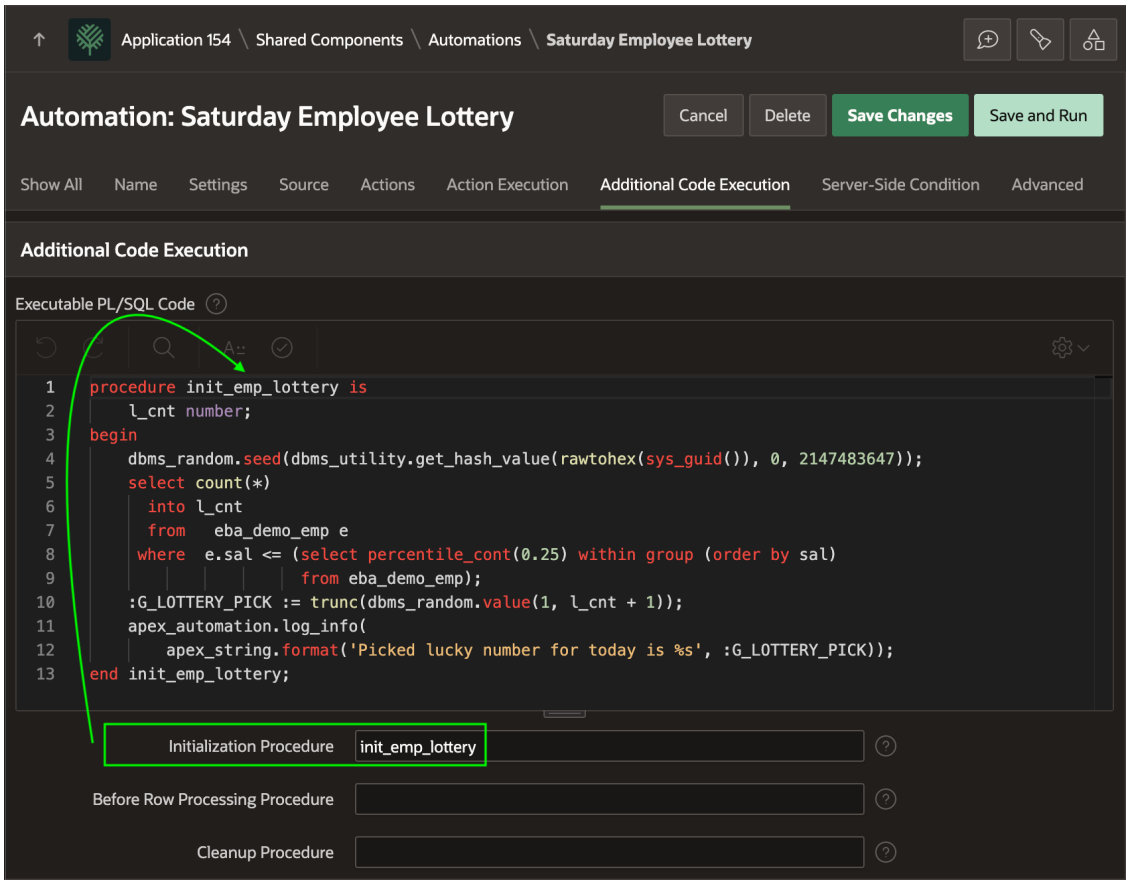
> **Tip**
>
> Because this procedure is defined *inline* in the automation, it can directly reference and assign application items as bind variables. If the *Initialization Procedure* were externally-defined, that would need to use `SET_VALUE` in the `APEX_SESSION_STATE` package instead and reference application items using `V('ITEM_NAME')`.

```
procedure init_emp_lottery is
    l_cnt number;
begin
    dbms_random.seed(dbms_utility.get_hash_value(
                         rawtohex(sys_guid()), 0, 2147483647));
    select count(*)
      into l_cnt
      from   eba_demo_emp e
     where  e.sal <= (select percentile_cont(0.25)
                             within group (order by sal)
                      from eba_demo_emp);
    :G_LOTTERY_PICK := trunc(dbms_random.value(1, l_cnt + 1));
    apex_automation.log_info(
        apex_string.format('Picked lucky number for today is %s',
                           :G_LOTTERY_PICK));
end init_emp_lottery;
```

The figure shows the Additional Code Execution tab of the Automation edit page, with the PL/SQL source code mentioned above in the **Executable PL/SQL Code** text box and the **Initialization Procedure** set to the procedure name `init_emp_lottery`.

Copyright © 2026, Oracle and/or its affiliates.

Figure 19-3 Additional Code Execution Section Defines an Inline Initialization Procedure



19.2.4 Defining Automation Actions in Sequence

Define automation actions that decide which rows to process, perform conditional updates, and notify affected users.

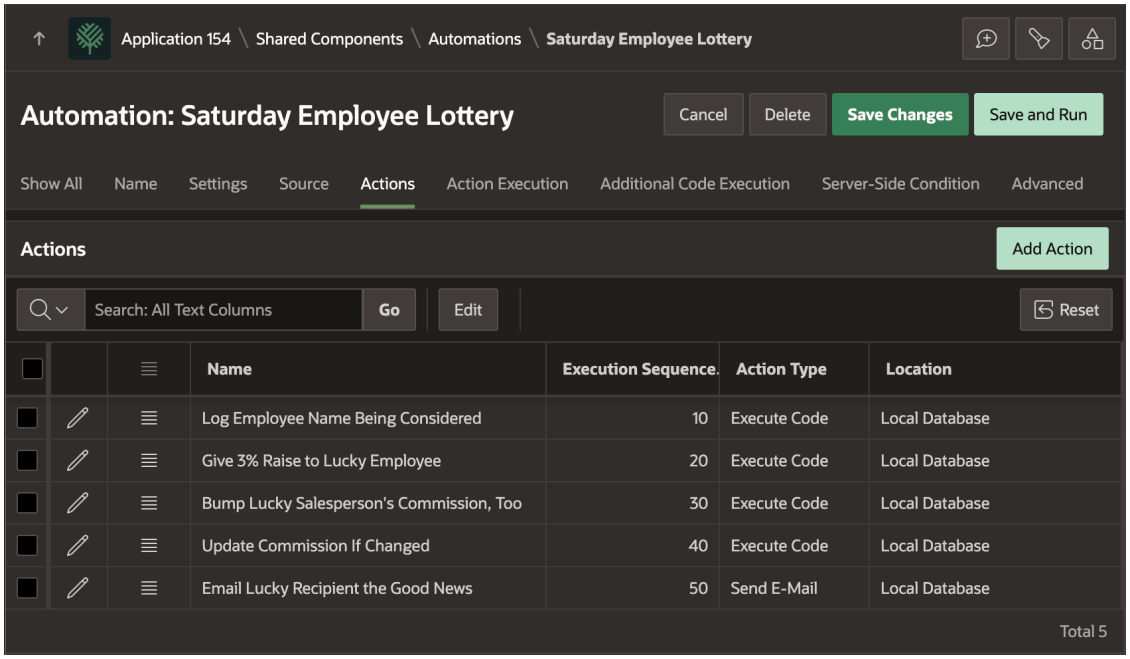
The *Saturday Employee Lottery* automation includes the sequence of five actions shown below:

1. *Log Employee Name Being Considered*
2. *Give 3% Raise to Lucky Employee*
3. *Bump Lucky Salesperson's Commission, Too*
4. *Update Commission If Changed*
5. *Email Lucky Recipient the Good News*.

The figure shows the **Actions** tab of the Automation edit page showing the five actions described above, in execution sequence order, listing the **Action Type** for each one. Each one is explained in more detail below.

Copyright © 2026, Oracle and/or its affiliates.

Figure 19-4 Five Actions in the Saturday Employee Lottery Automation



Log Employee Name Being Considered

Log Employee Name Being Considered writes a message to the automation log with information about the employee being processed. It is an *Execute Code* action with the following code. If the ordinal `RN` value matches the randomly selected winning row number, then it logs a message to that effect. Otherwise, it calls `SKIP_CURRENT_ROW` to continue on to process the next row. This avoids the execution of the remaining actions in the sequence for the non-winning employee row.

```
if :RN = :G_LOTTERY_PICK then
    apex_automation.log_info(
        apex_string.format('Processing bottom quartile lucky employee %s with
rn=%s',
                           :ENAME, :RN));
else
    apex_automation.skip_current_row(
        apex_string.format('Skipping bottom quartile employee %s with rn=%s',
                           :ENAME, :RN));
end if;
```

Give 3% Raise to Lucky Employee

If the employee is the lucky one, *Give 3% Raise to Lucky Employee* computes their 3% salary raise, updates their `SAL` in the `EBA_DEMO_EMP` table, and writes a message to the automation log. It is an *Execute Code* action with the following Server-side condition so it only executes for the winning employee.

```
:RN = :G_LOTTERY_PICK
```


Copyright © 2026, Oracle and/or its affiliates.

It has the following code. Notice how it's using the NEW_SALARY column in the current row as temporary storage for the new salary it calculates. The *Send E-Mail* action below passes this NEW_SALARY value to an email template placeholder.

```
:NEW_SALARY := round(to_number(:SAL) * 1.03);
apex_automation.log_info(
    apex_string.format('Lucky employee %s gets 3% salary raise from %s to %s',
                       :ENAME, :SAL, :NEW_SALARY));
update eba_demo_emp
   set sal = :NEW_SALARY
 where empno = :EMPNO;
apex_automation.log_info(
    apex_string.format('Updated Lucky employee %s salary from %s to %s',
                       :ENAME, :SAL, :NEW_SALARY));
```

Bump Lucky Salesperson's Commission, Too

If the lucky employee is a salesperson, *Bump Lucky Salesperson's Commission, Too* computes their 3% commission raise and writes a message to the automation log. It is an *Execute Code* action with the following Server-side condition so it only executes for a winning salesperson with an existing commission.

```
:RN = :G_LOTTERY_PICK and :JOB = 'SALESMAN' and :COMM is not null
```

It has the following code. Notice again how it's using the NEW_COMMISSION column in the current row as temporary storage for the new commission it computes. The *Update Commission If Changed* and the *Send E-Mail* action below both reference this value.

```
:NEW_COMMISSION := round(to_number(:COMM) * 1.03);
apex_automation.log_info(
    apex_string.format('Lucky Salesperson %s also gets 3% commission bump
from %s to %s',
                       :ENAME, :COMM, :NEW_COMMISSION));
```

Update Commission If Changed

If the commission has changed, *Update Commission If Changed* reflects the change in the database and writes a message to the automation log. It is an *Execute Code* action with the following Server-side condition to only run if a new commission got set.

```
:NEW_COMMISSION is not null
```

It has the following code. Notice how it's referencing the NEW_COMMISSION column in the current row that the previous action *might* have assigned a value to. It shows how temporary storage columns can be assigned by earlier actions in preparation for a later action to save the changes.

```
update eba_demo_emp
   set comm = :NEW_COMMISSION
 where empno = :EMPNO;
apex_automation.log_info(
    apex_string.format('Updated Lucky Salesperson %s commission from %s to
```

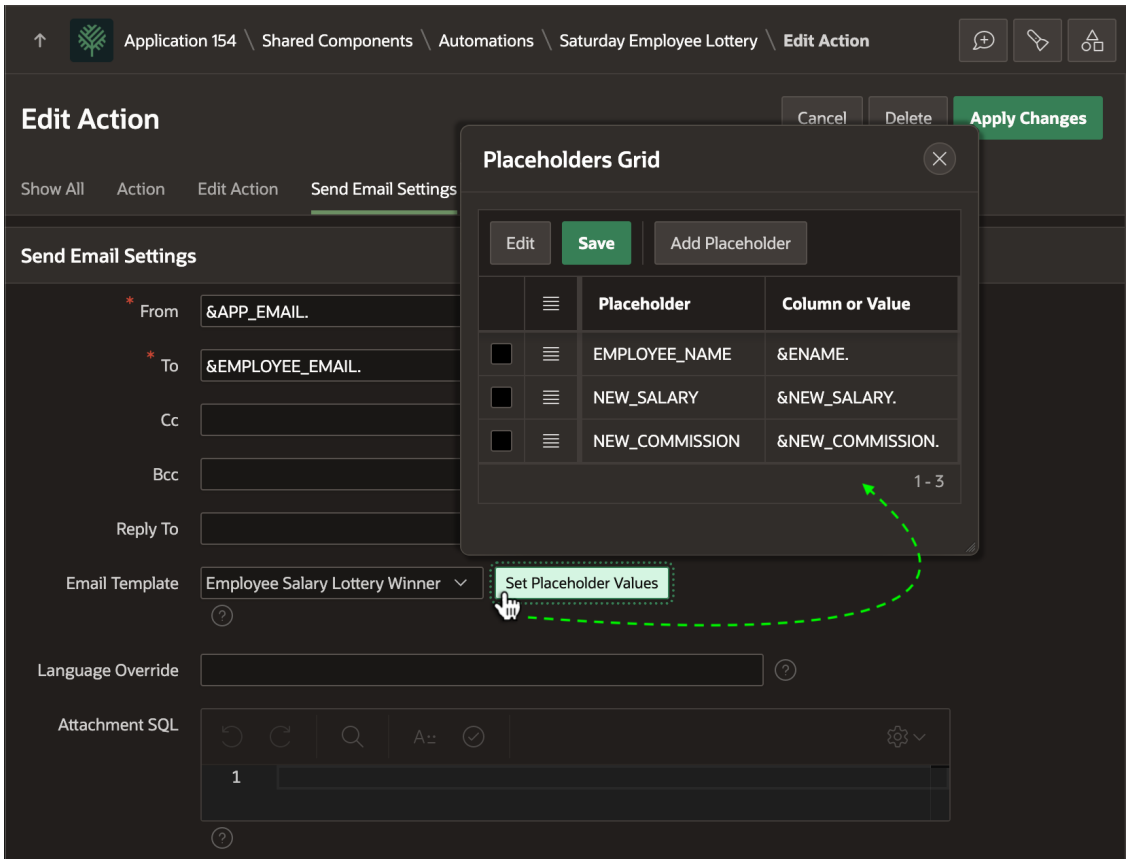

Copyright © 2026, Oracle and/or its affiliates.

```
%s',
    :ENAME, :COMM, :NEW_COMMISSION));
```

Email Lucky Recipient the Good News

If the employee is the lucky one, *Email Lucky Recipient the Good News* sends them an congratulatory email with their new salary, and commission if relevant. It is a *Send E-Mail* action with Server-side condition `:RN = :G_LOTTERY_PICK` configured as shown below. Notice how it's using the *Email Salary Lottery Winner* email template. The email template placeholders get their values using substitution syntax from the columns names in the current row.

Figure 19-5 Configuring Email Template and Placeholder Values for a Send E-Mail Action



19.2.5 Establishing an Automation's Schedule

Configure an automation schedule with the Interval Builder or Database Scheduler syntax.

For common scheduling requirements, the *Interval Builder* dialog is a quick way to configure your automation's *Schedule Expression*. For more complex requirements, enter any valid Database Scheduler syntax.

- Using Interval Builder to Schedule Job
  Use the Interval Builder to create an automation schedule expression.

Copyright © 2026, Oracle and/or its affiliates.

- Using More Complex Schedule Expressions
  Use Database Scheduler syntax for advanced automation schedules and test future run dates with PL/SQL.

19.2.5.1 Using Interval Builder to Schedule Job

Use the Interval Builder to create an automation schedule expression.

Your automation's *Schedule Expression* uses a compact syntax to establish when the job runs and how often. Use the *Interval Builder* dialog to configure the most common use cases. For example, as shown below, choosing the following options:

- *Frequency* – `Weekly`
- *Interval* – `1`
- *Execution Day* – `Sat`
- *Execution Hour* – `20`
- *Execution Minute* – `50`

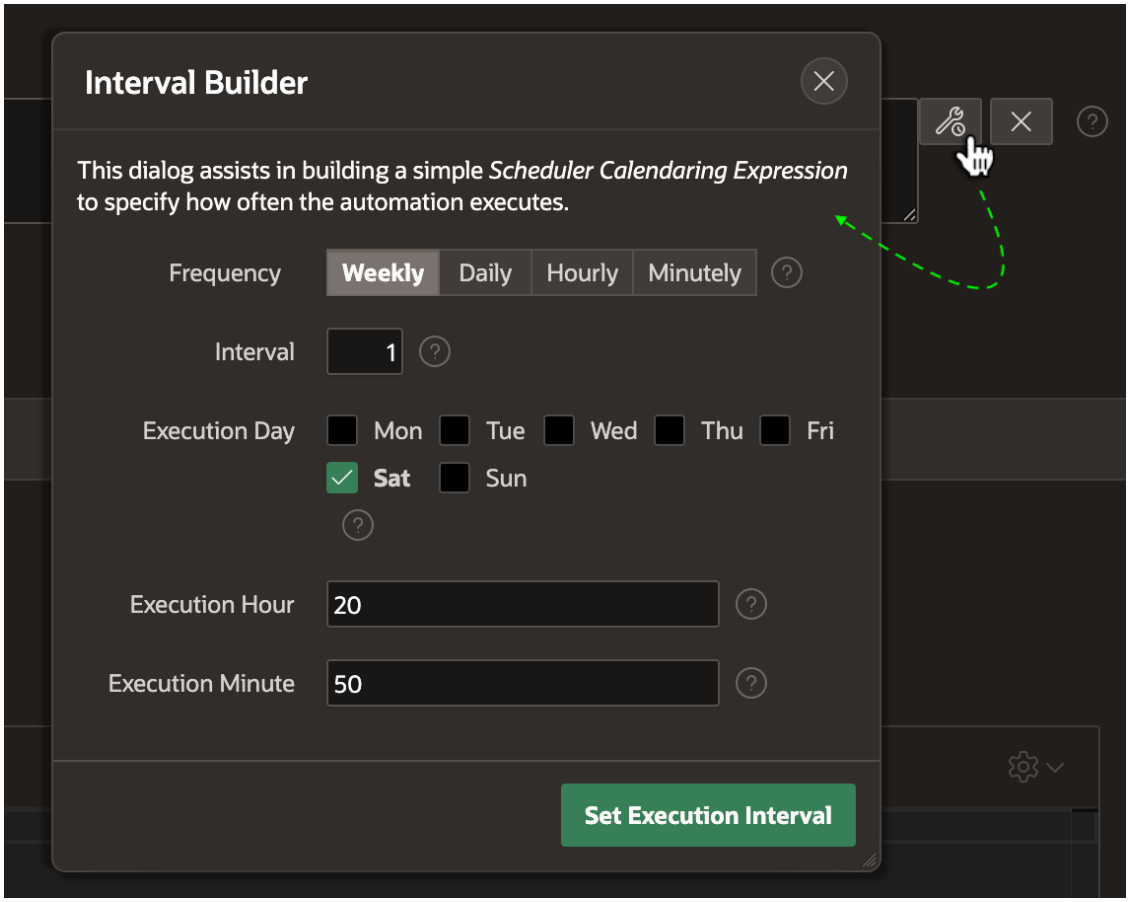
yields an expression that means "Weekly on Saturday at 20:50":

```
FREQ=WEEKLY;INTERVAL=1;BYDAY=SAT;BYHOUR=20;BYMINUTE=50
```

The figure shows the **Interval Builder** dialog you can open using the button in the Automation edit page containing a wrench and a clock.

Copyright © 2026, Oracle and/or its affiliates.

Figure 19-6 Configuring the Automation Scheduler Expression with Interval Builder



19.2.5.2 Using More Complex Schedule Expressions

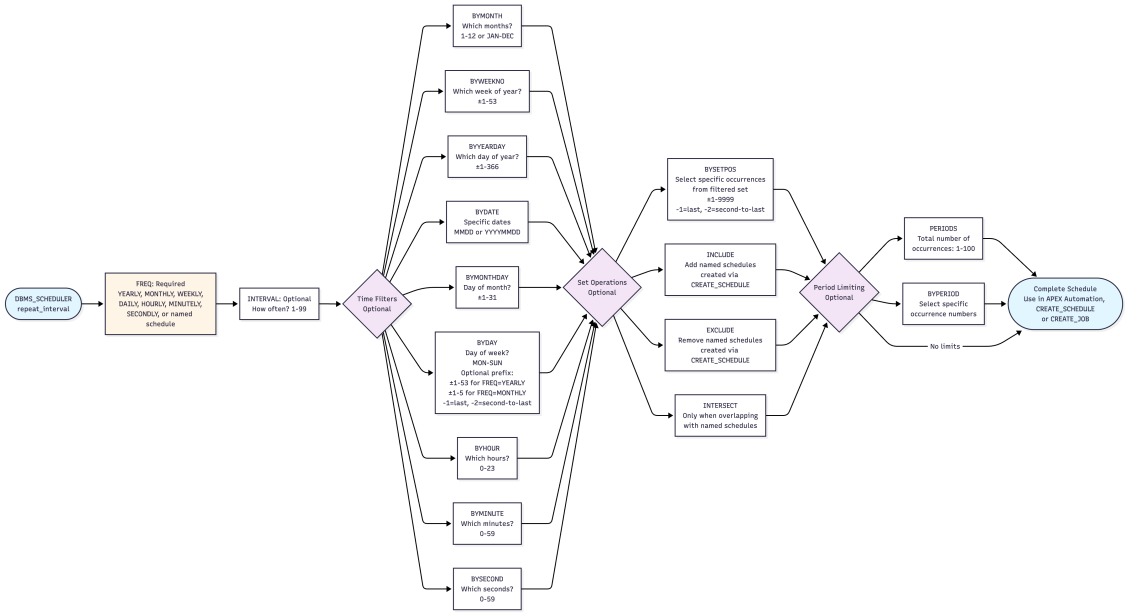
Use Database Scheduler syntax for advanced automation schedules and test future run dates with PL/SQL.

The automation's *Schedule Expression* supports any Database Scheduler syntax. These are the same ones the `DBMS_SCHEDULER` package accepts to create named schedules and create scheduled jobs. The diagram below summarizes the options you can use. You specify an expression using semicolon-separated list of one or more *`KEYWORD=VALUE`* elements. Supported keywords include:

- `FREQ` – Frequency of repetition (*required)*
- `INTERVAL` – Repetition counter (*n* = repeat every *n*th frequency unit)
- <*Time Filter*> – Identifies specific time elements to fine-tune exactly when it repeats
- <*Set Operations*> – Combine with a named schedule, or pick occurrences within a cycle
- <*Period Limiting*> – Limit the total number of cycles, or pick specific cycle occurrences.

Copyright © 2026, Oracle and/or its affiliates.

Figure 19-7 Syntax for DBMS_SCHEDULER Calendar Expressions



Use the following PL/SQL block in the *SQL Commands* page to test your scheduler expression. It helps you validate that your candidate syntax gives the expected results. Call `next_five_dates` with any expression you want to experiment with.

```
declare
    procedure next_five_dates(
        p_calendar_string in varchar2)
    is
        l_next_date timestamp with time zone;
        l_return_date_after timestamp with time zone := systimestamp;
        l_start_date timestamp with time zone := systimestamp;
    begin
        dbms_output.put_line(p_calendar_string);
        for i in 1..5 loop
            dbms_scheduler.evaluate_calendar_string(
                calendar_string   => p_calendar_string,
                start_date        => l_start_date,
                return_date_after => l_return_date_after,
                next_run_date     => l_next_date);
            dbms_output.put_line('Next ' || i || ': ' ||
                                 to_char(l_next_date,
                                         'DY DD-MON-YYYY HH24:MI:SS TZR'));
            l_return_date_after := l_next_date;
        end loop;
    end;
begin
    -- "Every week on Saturday at 20:50"
    next_five_dates('FREQ=WEEKLY;INTERVAL=1;BYDAY=SAT;BYHOUR=20;BYMINUTE=50');
    -- "Last Saturday of the month at 20:50"
    next_five_dates('FREQ=MONTHLY;INTERVAL=1;BYDAY=-1
SAT;BYHOUR=20;BYMINUTE=50');
end;
```


Copyright © 2026, Oracle and/or its affiliates.

Running the example above produces output like the following:

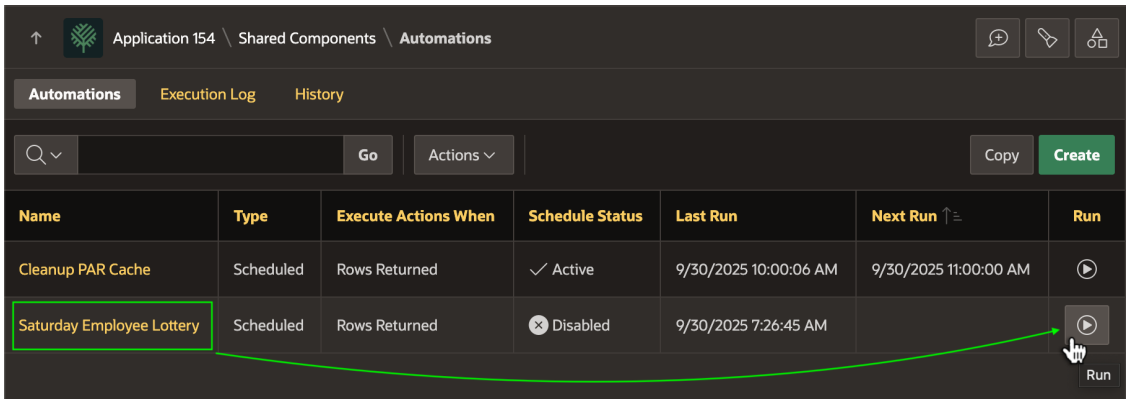
```
FREQ=WEEKLY;INTERVAL=1;BYDAY=SAT;BYHOUR=20;BYMINUTE=50
Next 1: SAT 04-OCT-2025 20:50:16 +00:00
Next 2: SAT 11-OCT-2025 20:50:16 +00:00
Next 3: SAT 18-OCT-2025 20:50:16 +00:00
Next 4: SAT 25-OCT-2025 20:50:16 +00:00
Next 5: SAT 01-NOV-2025 20:50:16 +00:00
FREQ=MONTHLY;INTERVAL=1;BYDAY=-1 SAT;BYHOUR=20;BYMINUTE=50
Next 1: SAT 25-OCT-2025 20:50:16 +00:00
Next 2: SAT 29-NOV-2025 20:50:16 +00:00
Next 3: SAT 27-DEC-2025 20:50:16 +00:00
Next 4: SAT 31-JAN-2026 20:50:16 +00:00
Next 5: SAT 28-FEB-2026 20:50:16 +00:00
```

19.2.6 Running an Automation on Demand

Use the *Automations* list page to run one on demand.

As shown below, clicking the "play" button in the *Run* column of the report executes the job. If the automation's *Schedule Status* is *Disabled*, then the job runs a single time. If it's *Active*, then it runs and uses the *Schedule Expression* you configured to determine the *Next Run* as well. The figure shows the Automations list under Shared Components in App Builder. It highlights the "play" button to run the *Saturday Employee Lottery* automation.

Figure 19-8 Running an Automation from the List Page



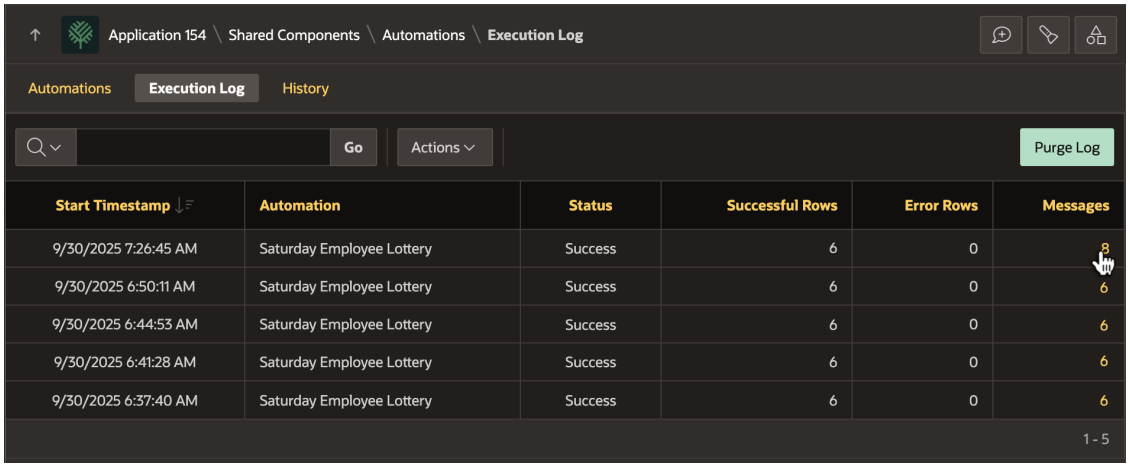
19.2.7 Viewing the Automation Log

Use the *Execution Log* tab of the *Automations* list to review recent runs.

As shown below, click on the count in the *Messages* column to review log messages for that run. The figure shows the **Execution Log** tab of the Automations page under *Shared Components*. It lists recent automation runs, how many rows were processed, and provides a link to view automation messages written to the log during each run.

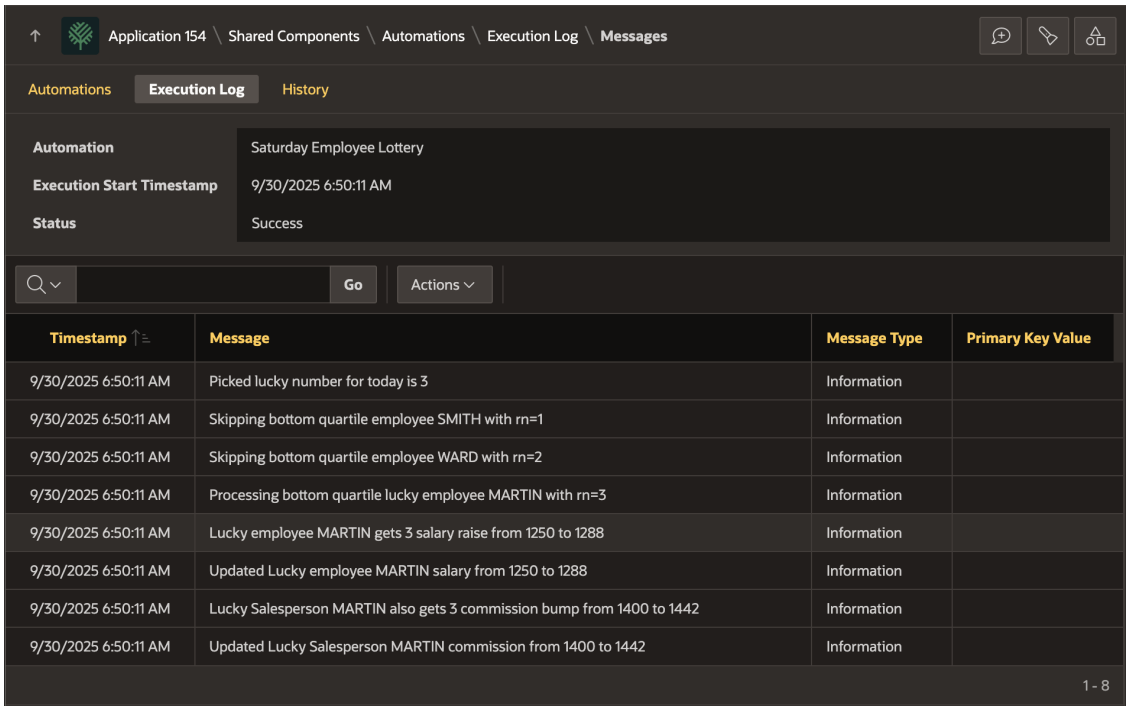
Copyright © 2026, Oracle and/or its affiliates.

Figure 19-9 Viewing the Automation Execution Log List



In the *Messages* page, you can inspect them, sort them, and download them to any of the formats an Interactive Report supports. The figure shows the **Messages** page you visit when clicking on a particular automation run. Logged messages display in an Interactive Report.

Figure 19-10 Inspecting Automation Execution Log Messages



Copyright © 2026, Oracle and/or its affiliates.

20
Notifying End Users

Notify end users with messages, emails, and push notifications from declarative settings or code.

- Displaying Success and Error Messages
  Show users success and error messages, and customize errors with an Error Handling Function.
- Sending Emails with Templates
  Send email from a native page process or workflow activity.
- Delivering Push Notifications
  Send push notifications from native processes or PL/SQL to users who opted in.

20.1 Displaying Success and Error Messages

Show users success and error messages, and customize errors with an Error Handling Function.

Use success and error messages to show users when operations have succeeded or failed. To tailor how end users see *unexpected* errors or ones raised by constraint violations or trigger exceptions, use a custom *Error Handling Function*.

- Configuring Success and Error Messages
  Configure page process messages with inline text, substitution syntax, or translatable text messages.
- Using Validations for Multiple Errors
  All validation errors show at once so users can review and fix them in one step.
- Auto-Dismissing Success Messages
  Configure success messages to dismiss automatically, with care for users who may need more time.
- Showing Messages with Dynamic Actions
  Show success or error messages from dynamic actions, including eager validation messages returned by server-side checks.
- Showing Messages Programmatically
  Show and clear success or error messages from PL/SQL or JavaScript code.
- Customizing Error Message Display
  Use an *Error Handling Function* to customize how users see errors raised from database constraints, trigger exceptions, or the APEX engine.

20.1.1 Configuring Success and Error Messages

Configure page process messages with inline text, substitution syntax, or translatable text messages.

Copyright © 2026, Oracle and/or its affiliates.

Configure success and error messages declaratively on page processes using their *Success Message* and *Error Message* settings. You can enter text inline with optional substitution syntax like:

```
Your order number is &P113_ORDER_ID. and will ship &P113_SHIP_DATE.
```

Alternatively, reference a translatable text message with a name like `ORDER_CONFIRMATION` and optional placeholders like `ordnum` and `ships` using the syntax:

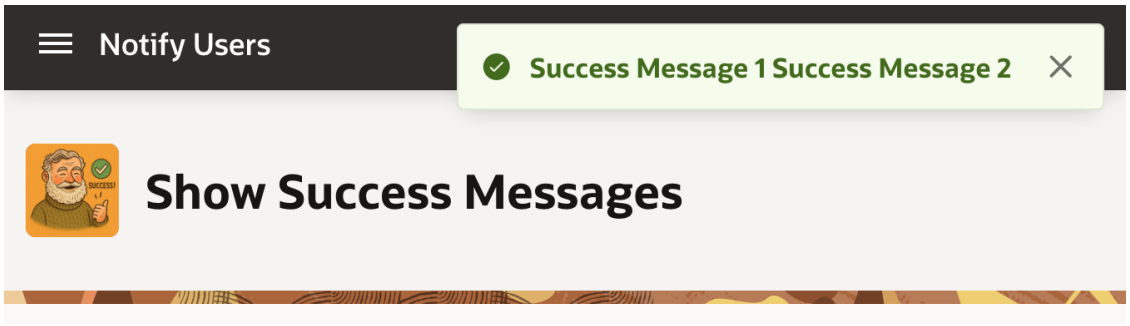
```
&{ORDER_CONFIRMATION ordnum="&P113_ORDER_ID." ships="&P113_SHIP_DATE."}.
```

If the translatable text message includes HTML markup, then use the `!RAW` modifier to avoid escaping the text for display:

```
&{ORDER_CONFIRMATION ordnum="&P113_ORDER_ID." ships="&P113_SHIP_DATE."}!RAW.
```

If multiple page process include a success message, they get concatenated as shown below. The figure shows a success notification with a green check mark icon and concatenated messages from two page processes: "*Success Message 1 and Success Message 2*".

Figure 20-1 Multiple Success Messages Get Concatenated



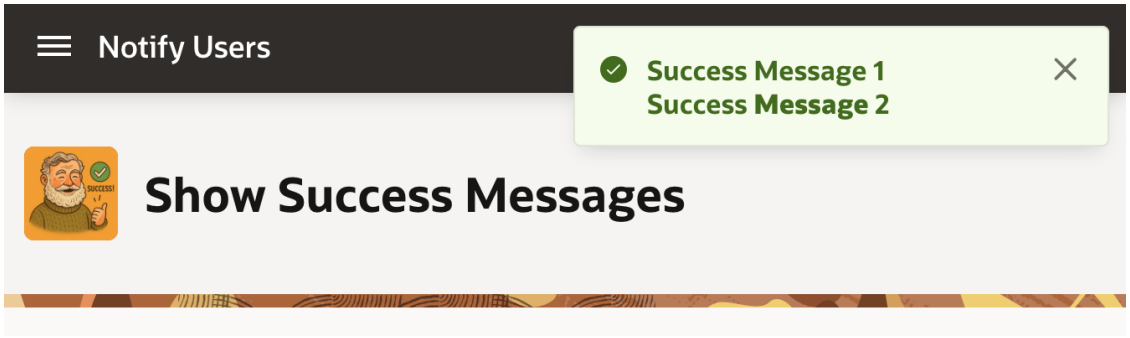
Since the success message can include HTML markup, you can optionally use `<br>` tags or other formatting like the following:

```
<br>Success <strong>Message</strong> 2
```

This produces the result shown below. The figure shows a success notification with a green check mark icon. "Success Message 1" appears on the first line, and "Success Message 2" appears on the second line with **Message** in bold.

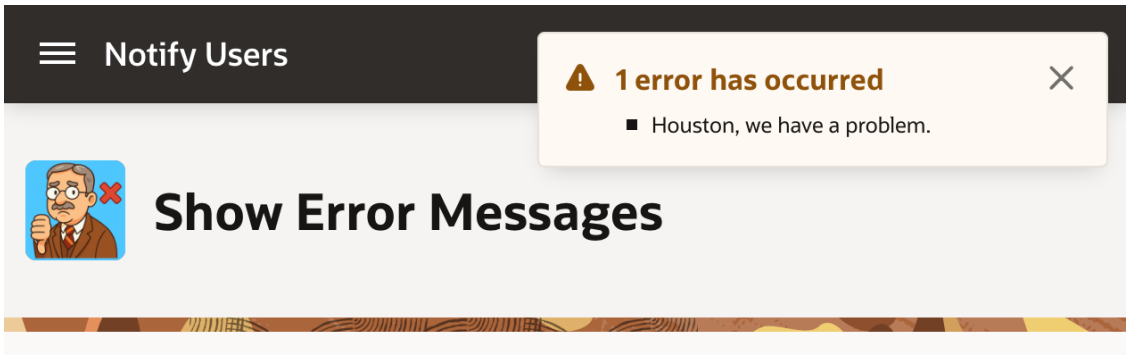
Copyright © 2026, Oracle and/or its affiliates.

Figure 20-2 Messages Can Include HTML Markup



In contrast, multiple page process' *Error Messages* do not accumulate. The first error halts processing to display the error message as shown below. The figure shows an error notification with an exclamation point inside an orange warning triangle. The first line reads "1 error has occurred", and the second line contains a single bullet: "Houston, we have a problem".

Figure 20-3 First Page Process Error Halts Processing and Displays Message



Tip

To prepare a declarative success or error message that is more *dynamic* than text substitutions allow, use a preceding page process to assign the value of a `P113_COMPUTED_MESSAGE` hidden page item, then reference it as the *Error Message* in that same process or a following one using `&P113_COMPUTED_MESSAGE.`.

20.1.2 Using Validations for Multiple Errors

All validation errors show at once so users can review and fix them in one step.

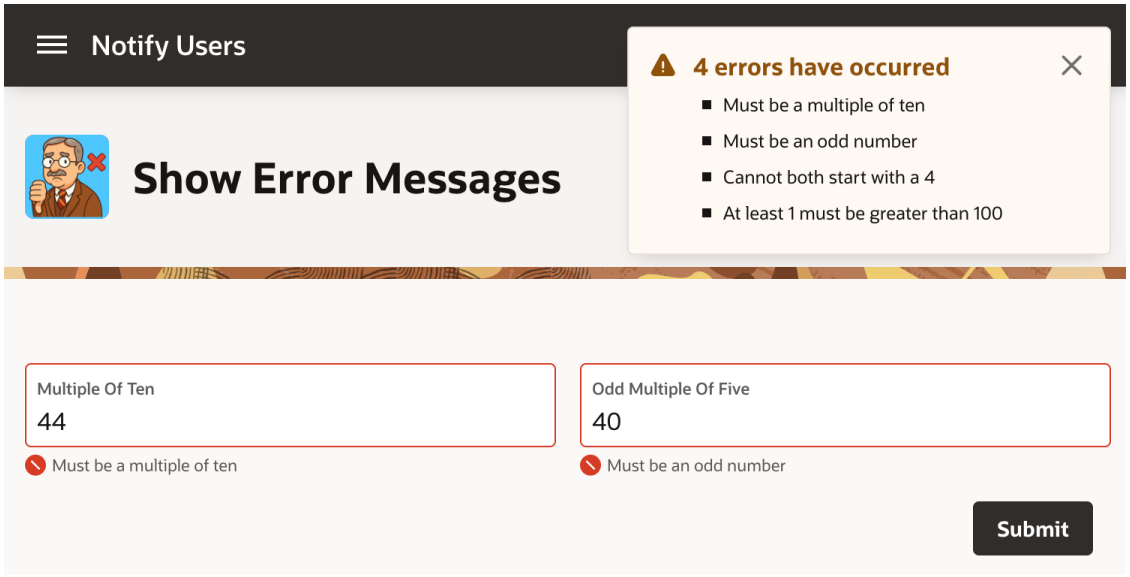
All failed validations automatically display as shown below. You can associate field-specific validations with the relevant page item, and configure them to show inline with the field. Validation error messages with no specific field show in the notification area. The figure shows a page with two number input fields **Multiple of Ten** and **Odd Multiple of Five**. The user has entered `44` in the **Multiple of Ten** field, and `40` in the **Odd Multiple of Five** one and clicked

Copyright © 2026, Oracle and/or its affiliates.

(**Submit**). A field-specific validation error message shows just below each field. Four error messages display in the error notification area as a bulleted list:

- *Must be a multiple of ten*
- *Must be an odd number*
- *Cannot both start with a 4*
- *At least 1 must be greater than 100*

Figure 20-4 All Failed Validation Error Messages Display on Submit



- Validating By Returning True/False Result
  Validate input with a PL/SQL expression that returns `true` for valid data and `false` for invalid data.
- Validating By Returning Error Text
  Validate input with a function body that returns null for success or error text for failure.

20.1.2.1 Validating By Returning True/False Result

Validate input with a PL/SQL expression that returns `true` for valid data and `false` for invalid data.

The *Error Message* you configure on a PL/SQL *Expression*-type validation shows if the expression evaluates to `false`. A validation named *Can't both start with a 4* uses the expression below. You can read it:

- the data is *valid* if it's "**not** the case that both page items' first character is a 4"

```
not(    substr(:P3_MULTIPLE_OF_TEN,1,1)      = '4'
    and substr(:P3_ODD_MULTIPLE_OF_FIVE,1,1) = '4')
```


Copyright © 2026, Oracle and/or its affiliates.

Tip

Logically, this is identical to the alternative expression below that you can read:

- the data is valid "if either one has a first character that's *not equal to* 4":

```
   substr(:P3_MULTIPLE_OF_TEN,1,1)      != '4'
or substr(:P3_ODD_MULTIPLE_OF_FIVE,1,1) != '4'
```

The translatable *Text Message* named `CANNOT_BOTH_START` has a single placeholder named `first_digit`:

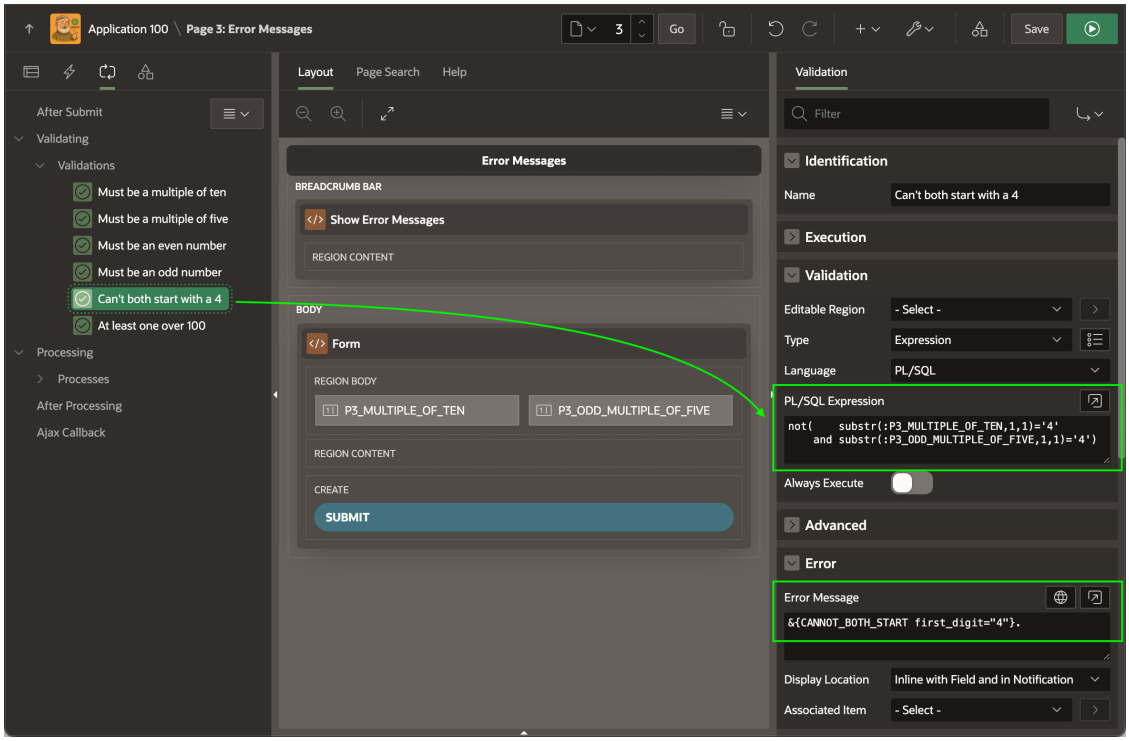
```
Cannot both start with a %first_digit
```

The validation *Error Message* references it by name, and provides the value `4` for the named placeholder:

```
&{CANNOT_BOTH_START first_digit="4"}.
```

The figure shows the **Can't both start with a 4** validation selected in the processing tree and highlights the **PL/SQL Expression** it uses to test validity as well as the **Error Message** configured to display when the validation fails by evaluating to `false`.

Figure 20-5 Error Message Displays When Expression Evaluates to False



Copyright © 2026, Oracle and/or its affiliates.

20.1.2.2 Validating By Returning Error Text

Validate input with a function body that returns null for success or error text for failure.

A validation can also use type *Function Body (returning Error Text)*. If it returns null, then the validation is a success. Any other return value is used as the error message text. It can use `APEX_LANG.GET_MESSAGE` as shown below to return a translatable message with placeholders. The validation named *At least one over 100* uses this approach.

The translatable *Text Message* named `AT_LEAST_N_ABOVE_M` has two placeholders named `mincount` and `limit`:

```
At least %mincount must be greater than %limit
```

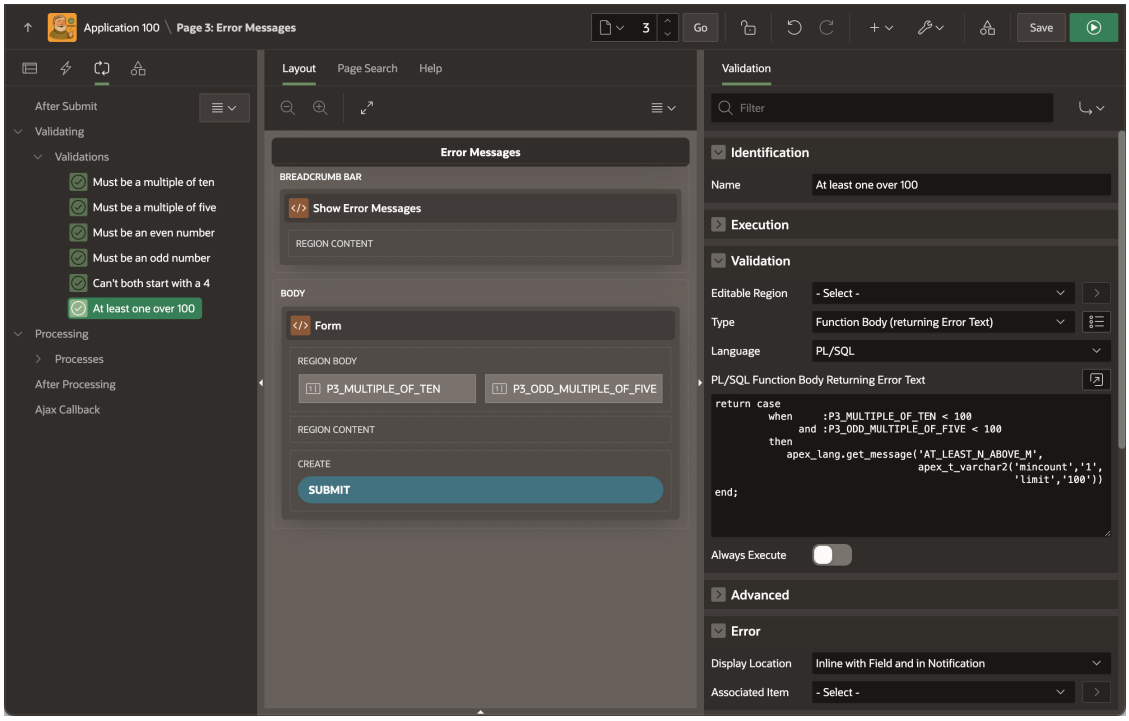
The validation Function Body references it by name, and provides values for the two placeholders using an `APEX_T_VARCHAR2` string list of key/value pairs:

```
return case
         when     :P3_MULTIPLE_OF_TEN < 100
              and :P3_ODD_MULTIPLE_OF_FIVE < 100
         then
            apex_lang.get_message('AT_LEAST_N_ABOVE_M',
                                  apex_t_varchar2('mincount','1',
                                                  'limit'   ,'100'))
end;
```

> **Tip**
>
> A `CASE` expression with no `ELSE` clause returns `NULL` if no `WHEN` condition is met.

Copyright © 2026, Oracle and/or its affiliates.

Figure 20-6 Validation Signals Failure by Returning the Error Text



20.1.3 Auto-Dismissing Success Messages

Configure success messages to dismiss automatically, with care for users who may need more time.

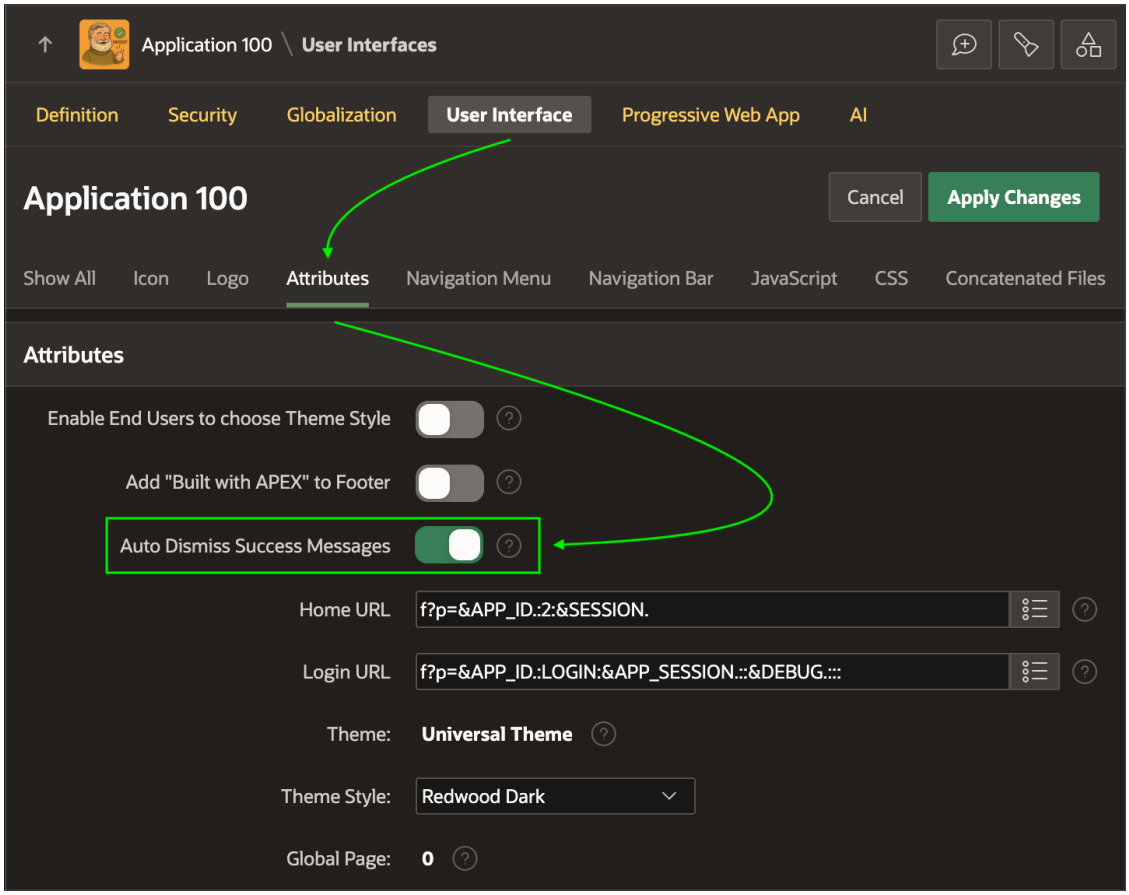
By default, success and error messages stay until the user dismisses them. However, you can opt for *success* messages to disappear after five seconds. On your app's *User Interface Attributes* page, just enable the *Auto Dismiss Success Messages* switch.

> ⚠ **Caution**
>
> Auto-dismiss can undermine accessibility. Some users may not notice a success message before it disappears. If you enable auto-dismiss, consider giving users a way to opt in or out with the `apex.message.setDismissPreferences` JavaScript API.

The figure highlights the **Auto Dismiss Success Messages** switch on the **Attributes** tab of the Shared Components **User Interface** settings page.

Copyright © 2026, Oracle and/or its affiliates.

Figure 20-7 Enabling Auto-Dismissal of Success Messages After Five Seconds



20.1.4 Showing Messages with Dynamic Actions

Show success or error messages from dynamic actions, including eager validation messages returned by server-side checks.

Use the native *Show Success Message* and *Show Error Message* dynamic actions to show a message to the user in reaction to an event in your page. When necessary, set a Client-side condition to only run the action under certain conditions. For example, suppose you have the following `CHECK_EMPNO` function in a `MESSAGES_APP` package that validates an employee id. If the `p_empno` passed in is valid, it returns `null`; otherwise, it returns an appropriate error message.

```
-- In package messages_app
function check_empno(
    p_empno in number)
    return     varchar2
is
begin
    for j in (select null
                from emp
               where empno = p_empno)
    loop
        return null; -- valid
    end loop;
```


Copyright © 2026, Oracle and/or its affiliates.

```
    return apex_lang.get_message('INVALID_EMPNO'); -- invalid
end;
```

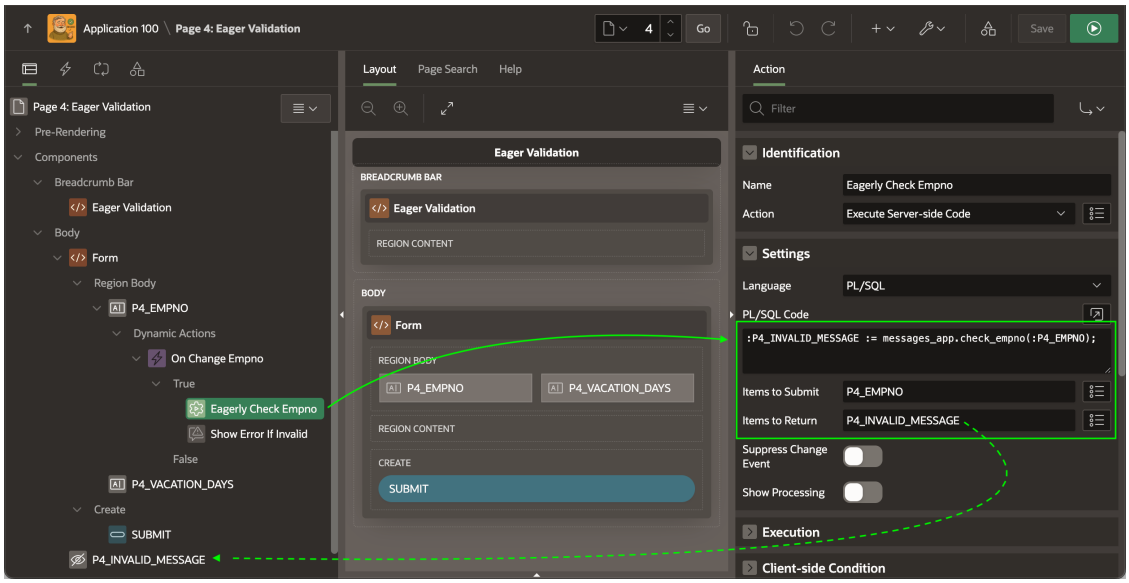
A *Change* event handler on the `P4_EMPNO` field shown below can eagerly validate the user's entry without submitting the page. If the value is invalid, it uses the *Show Error Message* dynamic action step to display the message in the page.

A hidden error message field `P4_INVALID_MESSAGE` with *Value Protected* disabled is part of the recipe. First an *Eagerly Check Empno* dynamic action of type *Execute Server-side Code* runs the following line of PL/SQL to invoke `CHECK_EMPNO`:

```
:P4_INVALID_MESSAGE := messages_app.check_empno(:P4_EMPNO);
```

Notice below that it configures the *Items to Submit* and *Items to Return* to send the latest value of `P4_EMPNO` over to the server and return the value of `P4_INVALID_MESSAGE` assigned there.

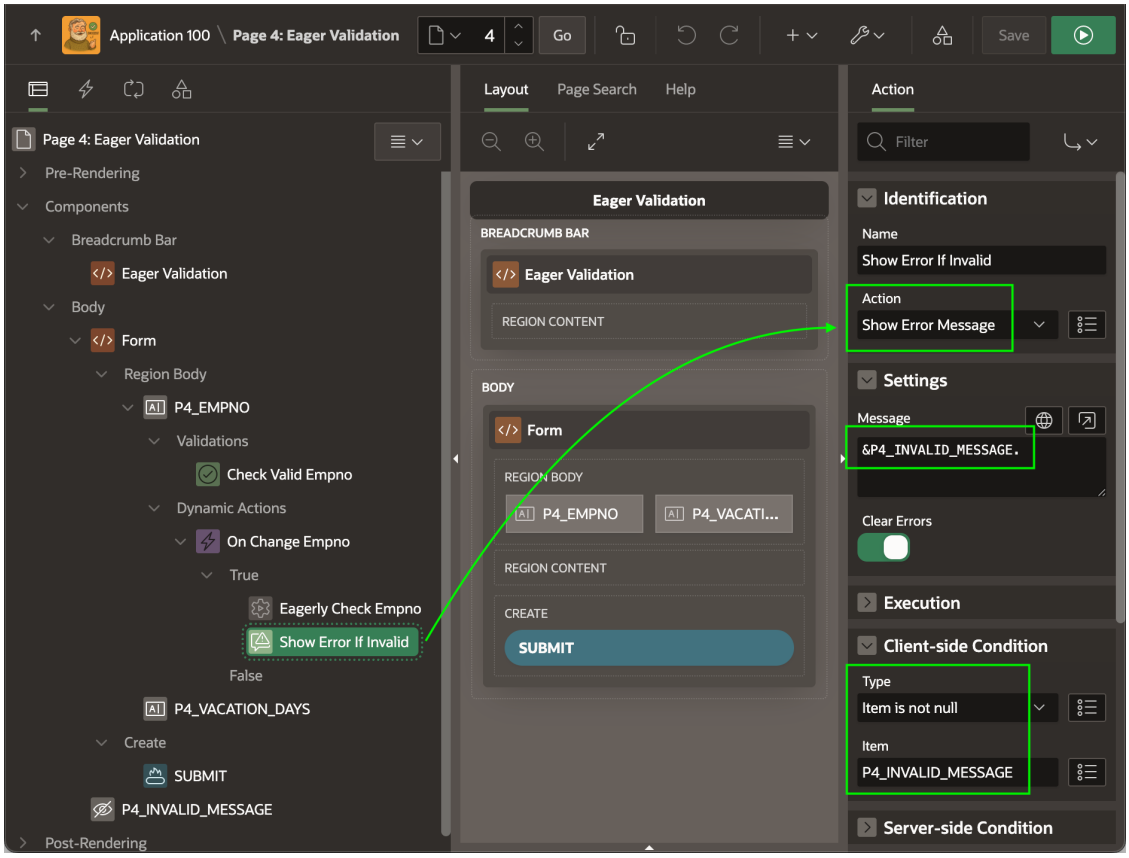
Figure 20-8 Dynamic Action Eagerly Calls CHECK_EMPNO Validation Function



As shown below, the *Show Error If Invalid* dynamic action of type *Show Error Message* follows the eager employee ID check. It configures a client-side condition to only execute if `P4_INVALID_MESSAGE` hidden page item is not null. The message it shows comes from the same page item using the `&P4_INVALID_MESSAGE.` substitution syntax.

Copyright © 2026, Oracle and/or its affiliates.

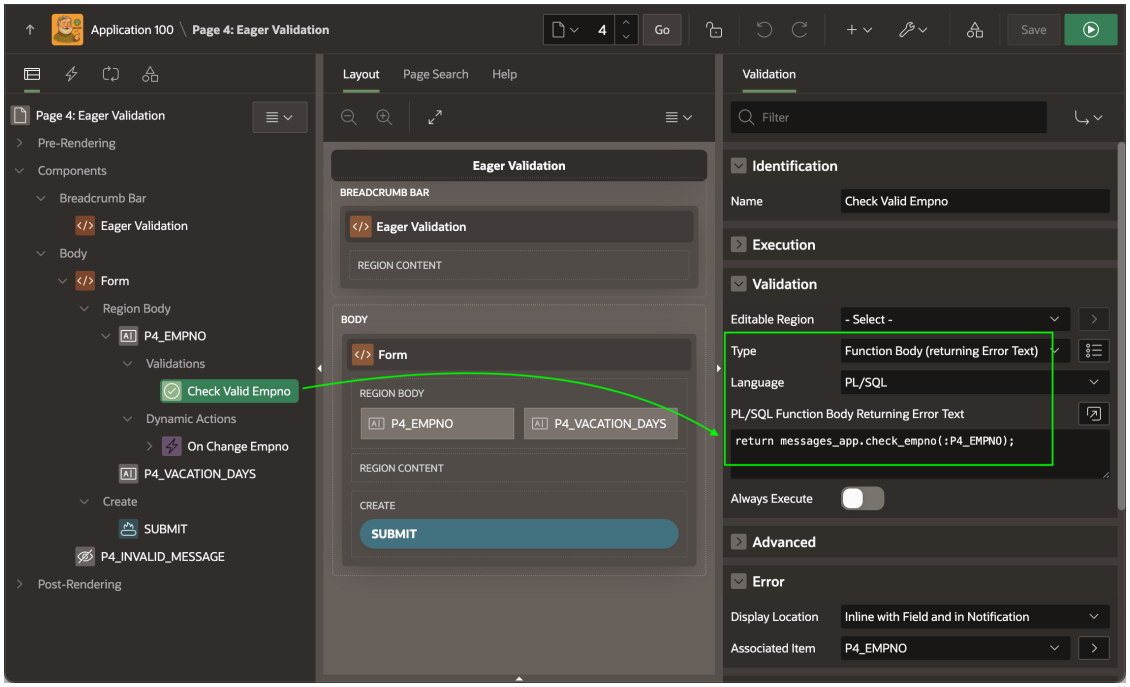
Figure 20-9 Conditionally Displaying an Error Message if One Was Returned



As shown below, the *Check Valid Empno* validation of type *Function Body (returning Error Text)* calls the same `CHECK_EMPNO` function to verify the data at page submit time as well.

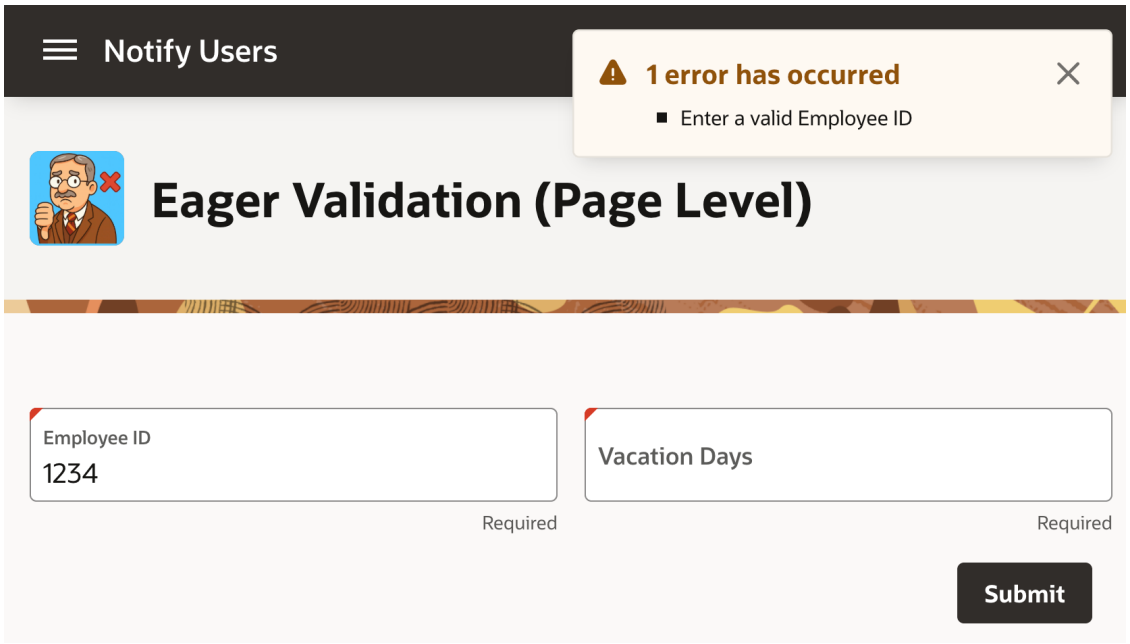
Copyright © 2026, Oracle and/or its affiliates.

Figure 20-10 Validation Performs Same Check on Page Submit



This combination yields the behavior shown below. The user sees an error message about an invalid employee ID as soon as they tab out of the *Employee ID* field.

Figure 20-11 Eagerly Displaying a Page-Level Error Message Before User Submits



20.1.5 Showing Messages Programmatically

Show and clear success or error messages from PL/SQL or JavaScript code.

Copyright © 2026, Oracle and/or its affiliates.

Use `APEX_ERROR.ADD_ERROR` to signal an error in server-side code with a message the user sees in the notification area. In your page, you can react to events and call JavaScript APIs to show success messages. You can also use JavaScript to both show *and* clear error messages.

- Adding an Error Message in PL/SQL
  Use `APEX_ERROR.ADD_ERROR` to add an error message from server-side processing.
- Displaying Messages with JavaScript
  Show and clear page or field-level messages from JavaScript, including errors returned by eager server-side checks.

20.1.5.1 Adding an Error Message in PL/SQL

Use `APEX_ERROR.ADD_ERROR` to add an error message from server-side processing.

It halts further processing, abandons data changes, and shows the error in the notification area.

- Enforcing Rules with Read-Consistency
  Use a page process after the data-saving processes to validate pending changes before APEX commits them.
- Applying Aggregate Constraints with SQL
  Use SQL aggregation in a page process after the data-saving processes to reject changes that violate cross-row rules.

20.1.5.1.1 Enforcing Rules with Read-Consistency

Use a page process after the data-saving processes to validate pending changes before APEX commits them.

When a user submits a page, page processes save any changes in the database. However, the APEX engine only *commits* the pending changes after all page submission processing succeeds. If any page process signals an error, this "vetoes" all pending changes so nothing is saved.

This gives you a powerful way to validate pending changes. Just add a page process *after* the ones that save user-submitted data changes. It can perform complex data validation using simple SQL statements. Oracle's read consistency model lets any queries it performs "see" the pending-but-not-yet-committed changes along with other unmodified data. Any aggregations like `SUM` or `COUNT` you perform always get an accurate answer. You can then *reject* the changes if an aggregate condition is invalid, just by signaling an error with an actionable message that helps the user fix the situation.

20.1.5.1.2 Applying Aggregate Constraints with SQL

Use SQL aggregation in a page process after the data-saving processes to reject changes that violate cross-row rules.

Consider a constraint that the total monthly compensation for all employees in a department cannot exceed a limit of 12000. The following SQL query using `SUM`, `GROUP BY`, and `HAVING` returns any departments that violate the rule. The `COALESCE` function returns its first non-null argument. So a `NULL` salary or commission is treated as a zero (`0`) while computing the total compensation by department.

```
-- Return any departments over monthly compensation limit
select deptno,
       sum(coalesce(sal,0) + coalesce(comm,0)) as total_comp
```


Copyright © 2026, Oracle and/or its affiliates.

```
  from emp
 group by deptno
having sum(coalesce(sal,0) + coalesce(comm,0)) > 12000
```

The following `CHECK_DEPARTMENTS_OVERBUDGET` procedure uses this query in a `FOR` loop to check for offending departments. For any it finds, it adds their department number and total compensation amount to the `l_overbudget` string list using `APEX_STRING.PUSH`.

After the loop, if `l_overbudget` is not null, then some department was over budget. If so, it calls `APEX_ERROR.ADD_ERROR` to signal an error. This *vetoes* the transaction with a user-friendly error message. The `p_message` it passes is a translatable text message named `OVERBUDGET_DEPARTMENTS` whose source text looks like:

```
Departments exceed the %amount budget: %department_list
```

It passes values for the two message placeholders `amount` and `department_list` when calling `APEX_LANG.GET_MESSAGE` to retrieve the text.

```
-- In package messages_app
procedure check_departments_overbudget
is
    l_overbudget apex_t_varchar2;
begin
    for j in (select deptno,
                     sum(coalesce(sal,0) + coalesce(comm,0)) as total_comp
                from emp
               group by deptno
              having sum(coalesce(sal,0) + coalesce(comm,0)) > 12000
               order by deptno)
    loop
        apex_string.push(
           l_overbudget,
           apex_string.format('%s (%s)',j.deptno,j.total_comp));
    end loop;
    if l_overbudget is not null then
        apex_error.add_error (
                p_message => apex_lang.get_message(
                                'OVERBUDGET_DEPARTMENTS',
                                apex_t_varchar2(

'department_list',apex_string.join(l_overbudget,', '),
                                    'amount'         ,'12000')),
                p_display_location => apex_error.c_on_error_page);
    end if;
end check_departments_overbudget;
```

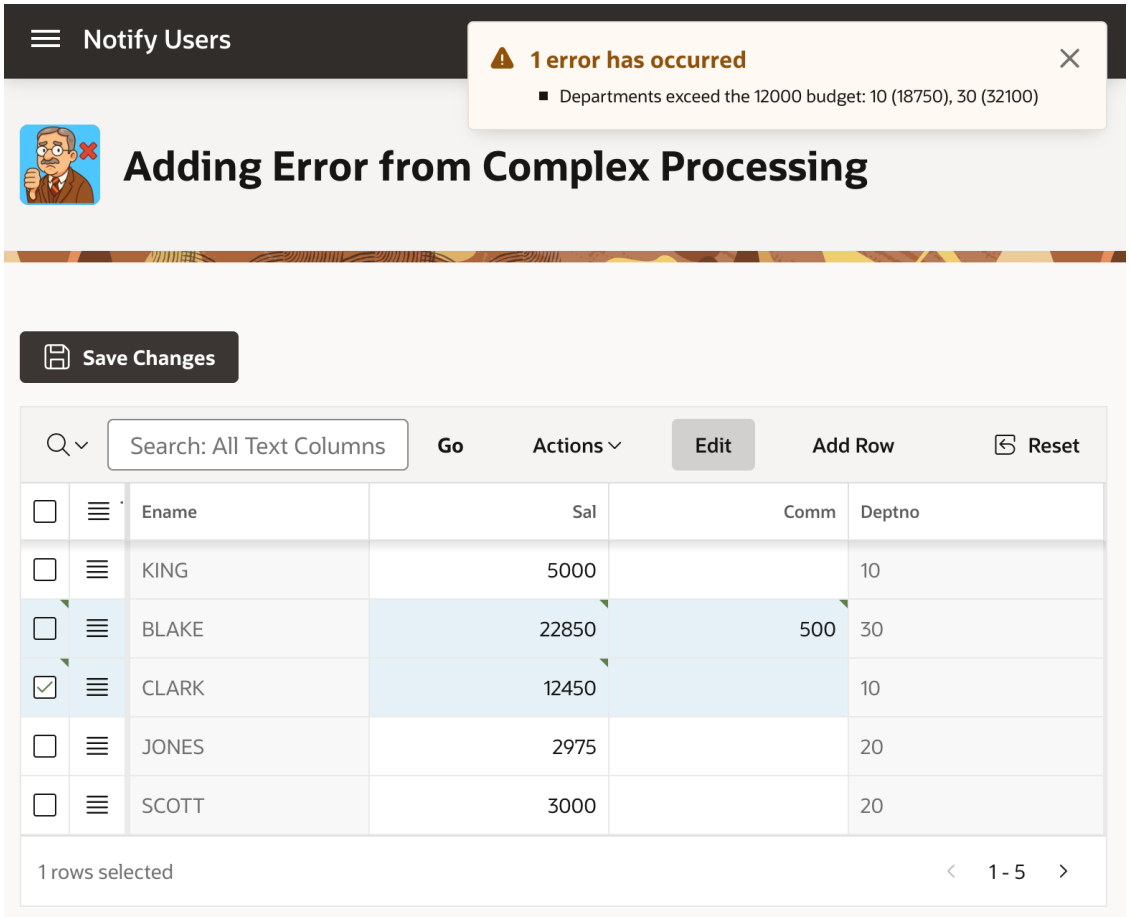
The result looks like the figure below at runtime.

Copyright © 2026, Oracle and/or its affiliates.

> **Tip**
>
> This logic can't be done as a *Function Body (returning Error Text)* validation, because the APEX engine runs validations *before* the *Processing* phase where data is saved. Since this aggregate query needs to "see" the pending changes, it must happen in the *Processing* section *after* the ones that save the user's changes. The effect is the same, however: changes get rejected with a user-friendly error message.

In the figure, a user editing an interactive grid of salaries and commissions clicks *(Save Changes)*, and any offending departments with total compensation exceeding the 12000 monthly limit appear in the error message. No changes are saved. The user can correct their data entry and try again. The error notification error shows one error message with the text: *"Departments exceed the 12000 budget: 10 (18750), 30 (32100)"*

Figure 20-12 Aggregate SQL Checks Can Veto the Transaction Using ADD_ERROR



20.1.5.2 Displaying Messages with JavaScript

Show and clear page or field-level messages from JavaScript, including errors returned by eager server-side checks.

Use the `apex.message.showPageSuccess` JavaScript function to show a success message as part of custom client-side logic. `apex.message.showErrors` lets you display one or more errors

Copyright © 2026, Oracle and/or its affiliates.

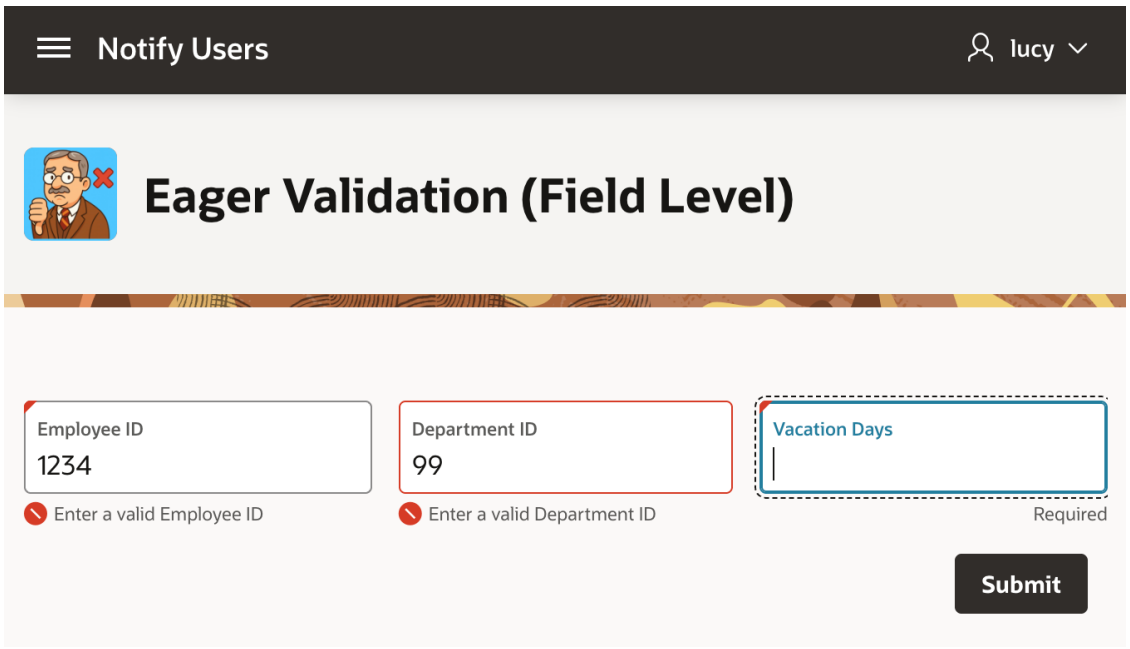
inline on a particular page item or at page-level. Use the companion `apex.message.clearErrors` to clear all errors, or just the error for a specific page item. To explore an example, combine `clearErrors` and `showErrors` with an *Execute Server-side Code* dynamic action to eagerly validate user data entry and show field-specific error messages *before* the user submits the page.

Start by adding the following `CHECK_DEPTNO` function to complement the `CHECK_EMPNO` from Showing Messages with Dynamic Actions. If `p_deptno` is a valid department ID, it returns `null`; otherwise it returns an appropriate error message.

```
-- In package messages_app
function check_deptno(
    p_deptno in number)
    return     varchar2
is
begin
    for j in (select null
                from dept
               where deptno = p_deptno)
    loop
        return null; -- valid
    end loop;
    return apex_lang.get_message('INVALID_DEPTNO'); -- invalid
end;
```

The goal is eagerly validating separate `P5_EMPNO` and `P5_DEPTNO` form fields with error messages showing *inline* with the relevant field as shown below. The figure shows a page with three fields. The user has entered `1234` in **Employee ID** and 99 in **Department ID**, then tabbed into **Vacation Days**. Below **Employee ID**, a red field-level error appears with a small red circular icon with a white diagonal slash: *"Enter a valid Employee ID"*. Below **Department ID**, a similar error appears: *"Enter a valid Department ID"*.

Figure 20-13 Showing Eager Field-Level Error Messages Before Submitting the Page

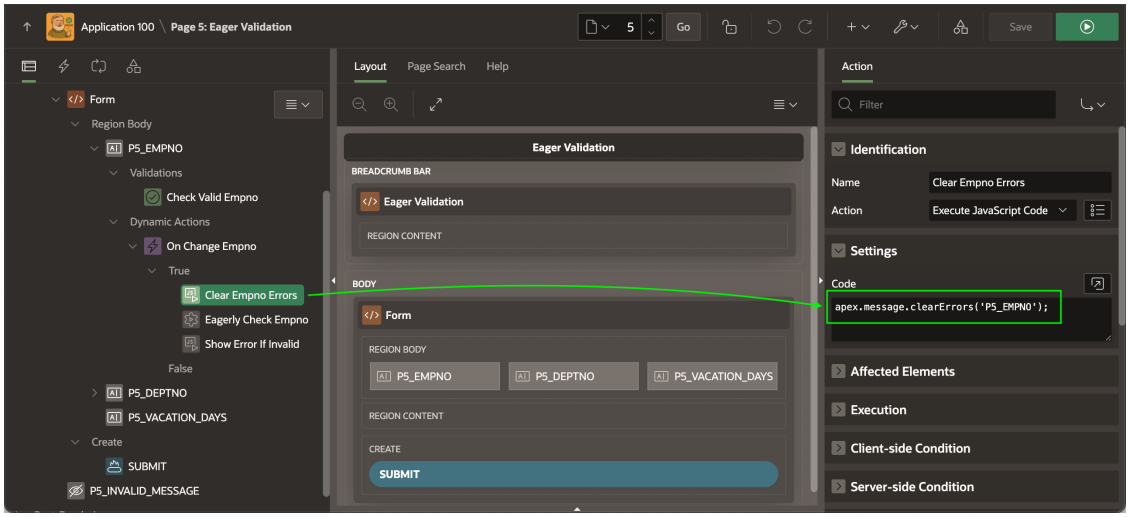


Copyright © 2026, Oracle and/or its affiliates.

As you saw in Showing Messages with Dynamic Actions, a *Check Valid Empno* validation of type *Function Body (returning Error Text)* passes `:P5_EMPNO` to the `CHECK_EMPNO` function. Recall that it returns an error message if the employee ID is invalid. As shown below, a dynamic action event handler on the `P5_EMPNO` item's *Change* event starts by using a *Clear Empno Errors* action of type *Execute JavaScript Code* to clear the errors on that field with the following line of JavaScript:

```
apex.message.clearErrors('P5_EMPNO');
```

Figure 20-14 Clearing Field-Level Errors Before Checking Changed Employee ID

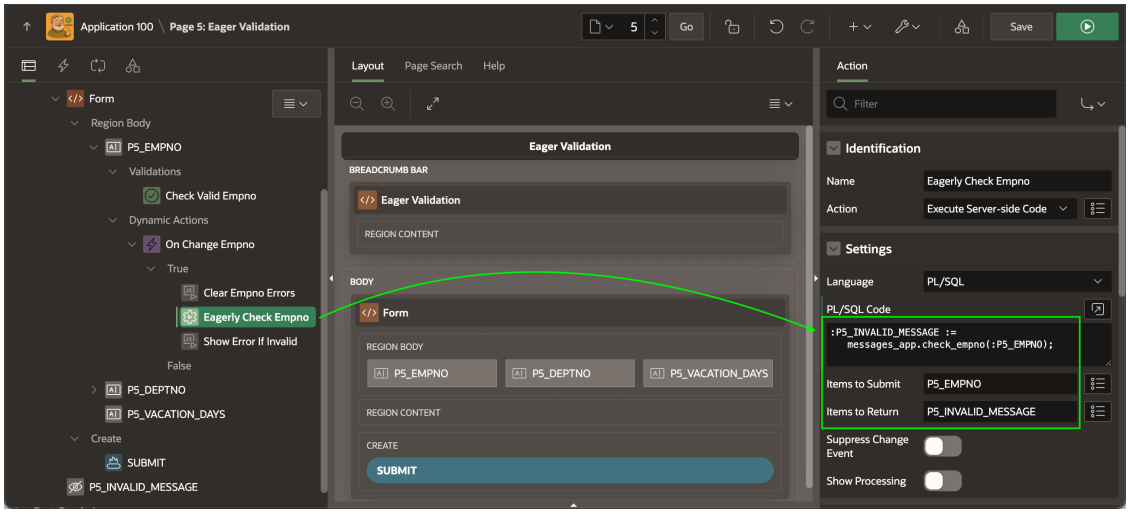


Then, the *Eagerly Check Empno* dynamic action of type *Execute Server-side Code* calls the `CHECK_EMPNO` function, passing in `P5_EMPNO` and assigns its returned error message to the hidden `P5_INVALID_MESSAGE` page item. Notice below how the *Items to Submit* and *Items to Return* ensure your code gets the latest value of `P5_EMPNO` from the browser and returns a server-side value back into `P5_INVALID_MESSAGE`. The action runs this one line of PL/SQL to get the job done:

```
:P5_INVALID_MESSAGE := messages_app.check_empno(:P5_EMPNO);
```


Copyright © 2026, Oracle and/or its affiliates.

Figure 20-15 Validating the Employee ID on the Server



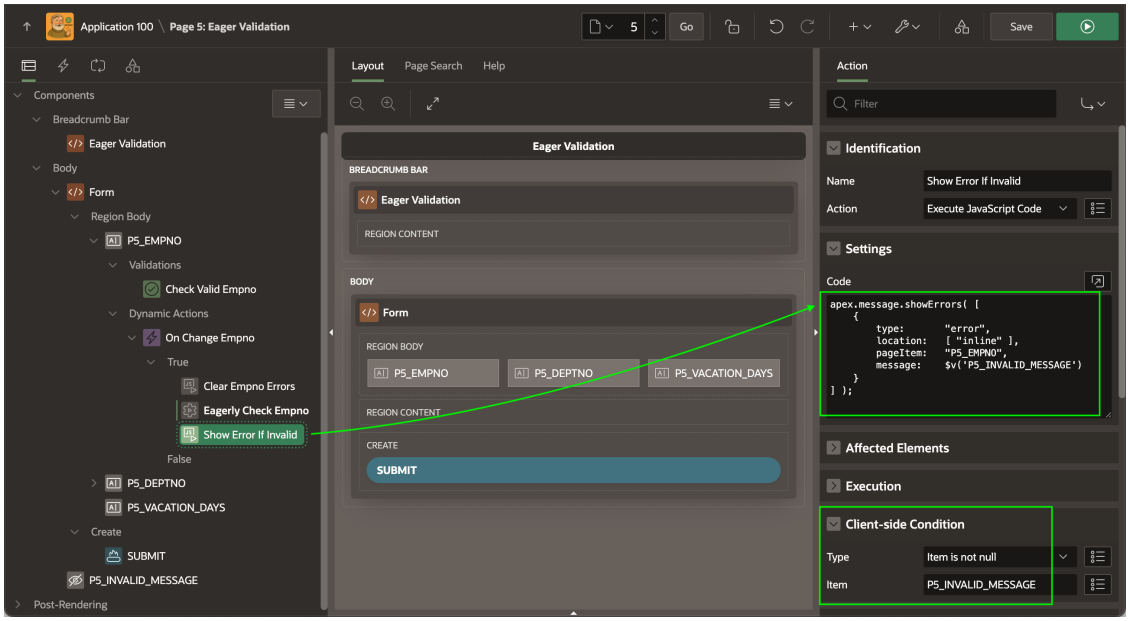
Finally, the *Show Error If Invalid* action of type *Execute JavaScript Code* finishes the recipe by conditionally showing the field-level error with the `showErrors` call below. It passes the error message returned by the server-side check using `v$('P5_INVALID_MESSAGE')` and indicates the error is `inline` on the `P5_EMPNO` item. Notice its Client-side condition is set to execute only if `P5_INVALID_MESSAGE` is not null.

> **Tip**
>
> If your error message text contains HTML markup, pass the additional `unsafe: false` property to `showErrors` to avoid "escaping" reserved characters like angle brackets.

```
apex.message.showErrors( [
    {
        type:       "error",
        location:   [ "inline" ],
        pageItem:   "P5_EMPNO",
        message:    $v('P5_INVALID_MESSAGE')
    }
] );
```

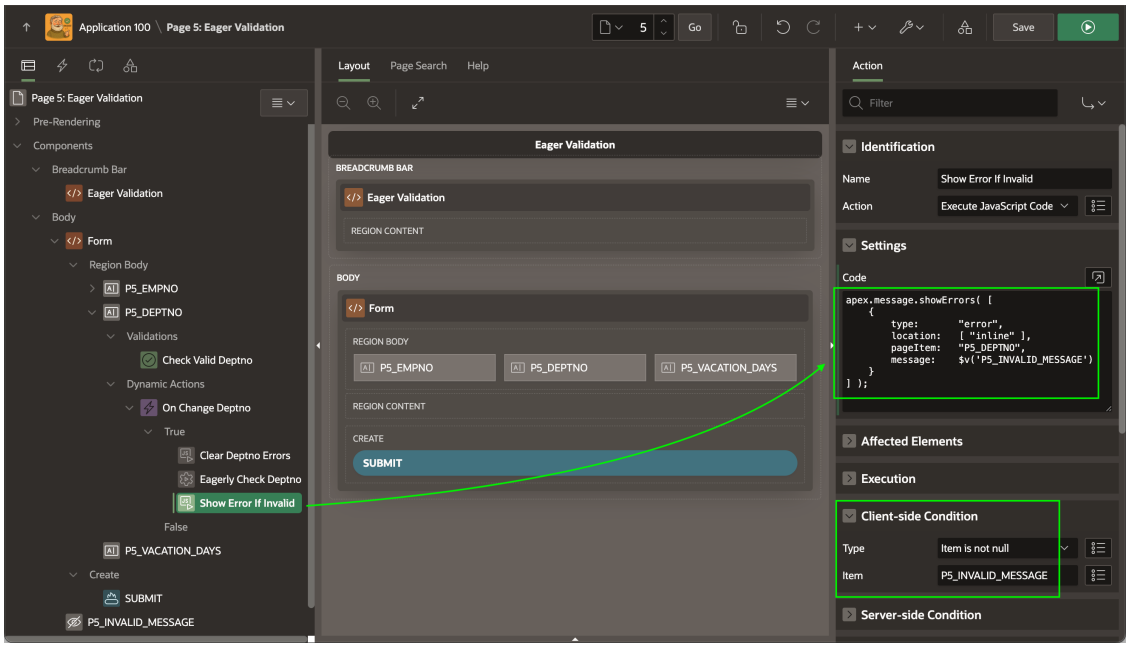

Copyright © 2026, Oracle and/or its affiliates.

Figure 20-16 Conditionally Showing Error Message Returned by Server-side Check



To repeat the same field-level validation on the *second* field `P5_DEPTNO`, just reuse the same recipe. Add a *Check Valid Deptno* validation of type *Function Body (returning Error Text)* that calls `CHECK_DEPTNO`. In the *Eagerly Check Deptno* server-side code, pass `P5_DEPTNO` to the `CHECK_DEPTNO` function instead, then show the error on `P5_DEPTNO` field. You can reuse the same hidden `P5_INVALID_MESSAGE` field.

Figure 20-17 Reusing the Same Recipe to Eagerly Validate Department ID



Copyright © 2026, Oracle and/or its affiliates.

20.1.6 Customizing Error Message Display

Use an *Error Handling Function* to customize how users see errors raised from database constraints, trigger exceptions, or the APEX engine.

- Studying Default Error Message Behavior
  Review default error messages to see why an Error Handling Function can improve what users see.
- Specifying an Error Handling Function
  Define an Error Handling Function to rewrite server-side errors before APEX reports them.
- Examining Error Handling Function Code
  Examine an Error Handling Function that logs internal errors and replaces constraint errors with friendlier messages.
- Logging Error Info for Troubleshooting
  Log error details in a separate transaction so troubleshooting data survives the rollback.
- Experiencing Custom Error Handling
  Test custom error handling with constraint violations, database exceptions, and logged internal errors.

20.1.6.1 Studying Default Error Message Behavior

Review default error messages to see why an Error Handling Function can improve what users see.

To understand why an *Error Handling Function* may be useful, consider the following examples. Suppose you add the constraint below to the `EMP` table to enforce the rule that an employee's commission can never be more than *twice* their salary:

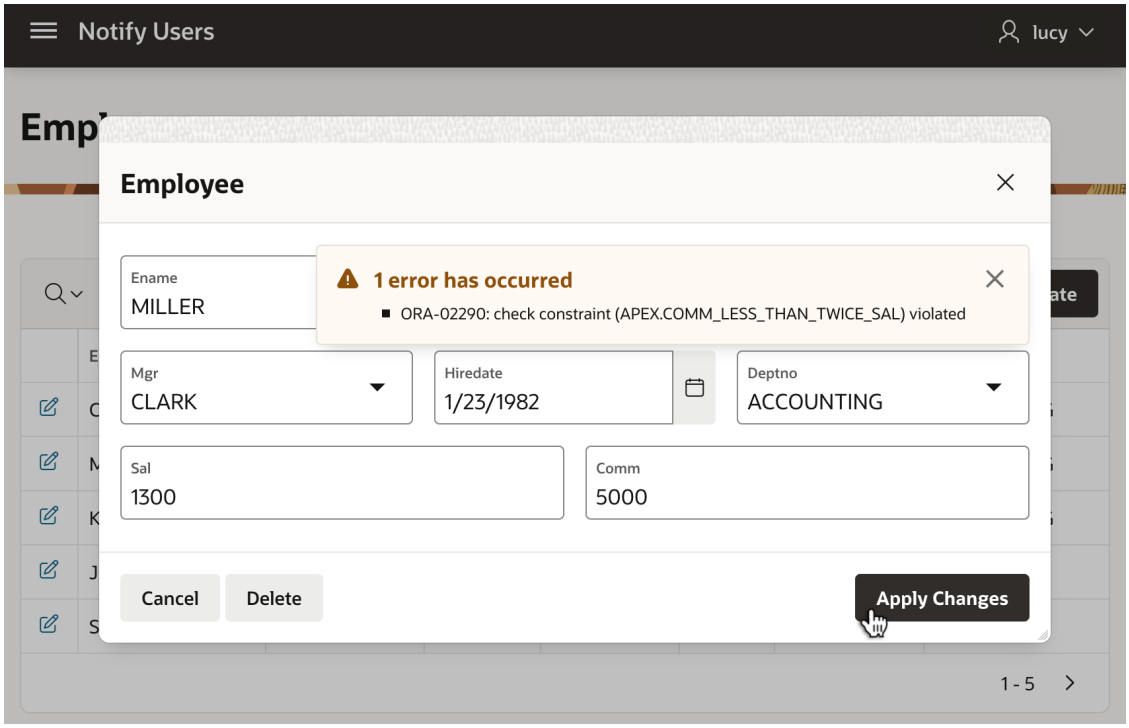
```
alter table emp
add constraint comm_less_than_twice_sal
check (comm < 2 * sal);
```

By default, when a user submits changes that violate the constraint, they see the error message:

```
ORA-02290: check constraint (SCHEMA.COMM_LESS_THAN_TWICE_SAL) violated
```


Copyright © 2026, Oracle and/or its affiliates.

Figure 20-18 By Default, Users See Constraint Violation Error Name

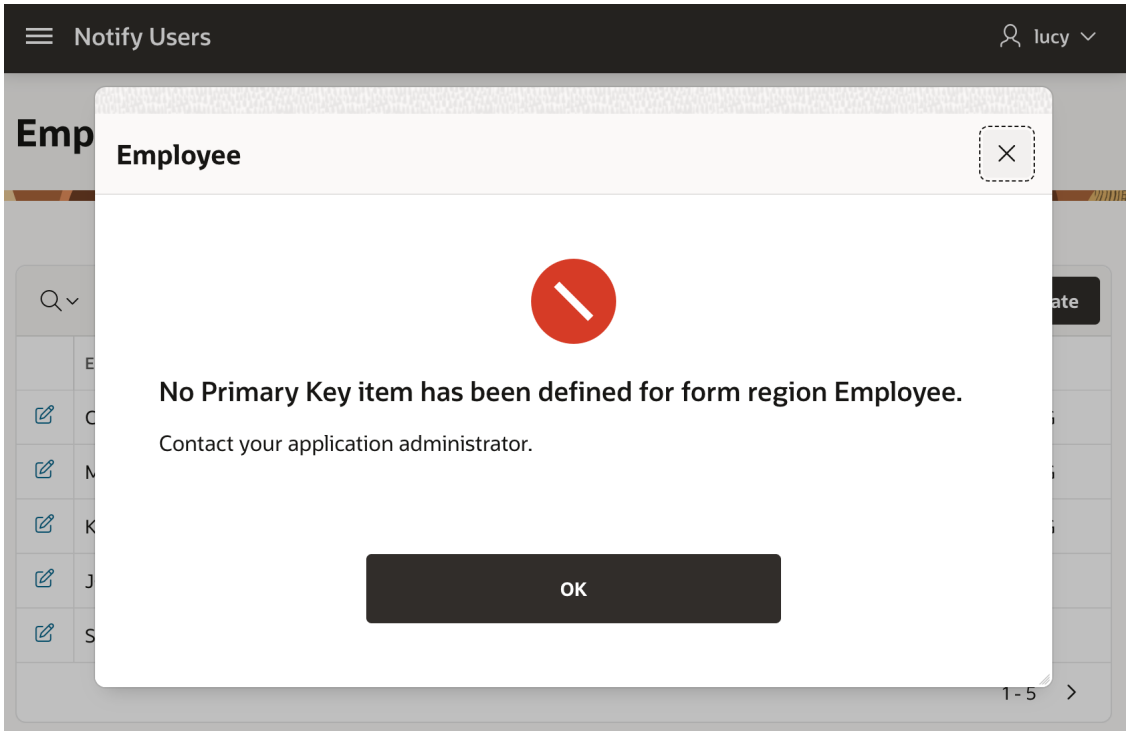


Similarly, when the APEX engine raises internal errors, by default the user sees the error message and an invitation to contact their application administrator:

```
No Primary Key item has been defined for form region Employee
```


Copyright © 2026, Oracle and/or its affiliates.

Figure 20-19 By Default, Users See Internal Errors



20.1.6.2 Specifying an Error Handling Function

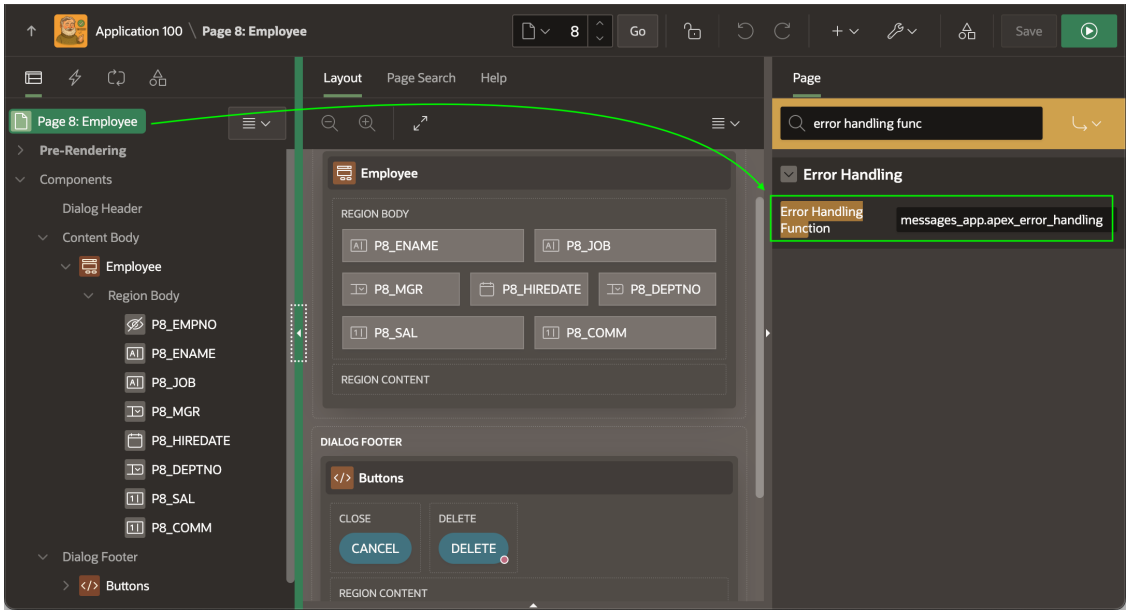
Define an Error Handling Function to rewrite server-side errors before APEX reports them.

The APEX engine calls your error handling function before reporting any server-side error. When appropriate, your function can change the error message to something more user-friendly. The function can have any *name*, but it must have the single parameter and return type as shown below. The APEX engine passes it a `T_ERROR` record, and it returns a `T_ERROR_RESULT` one. Optionally, define the function in a package like `MESSAGES_APP`. Then, configure a page to *use* it by setting the *Error Handling Function* property of the page to the function's name as shown below.

```
-- In package messages_app
function apex_error_handling (
    p_error in apex_error.t_error )
    return     apex_error.t_error_result;
```


Copyright © 2026, Oracle and/or its affiliates.

Figure 20-20 Configuring a Page's Error Handling Function



20.1.6.3 Examining Error Handling Function Code

Examine an Error Handling Function that logs internal errors and replaces constraint errors with friendlier messages.

The following `APEX_ERROR_HANDLING` function initializes the error result, then checks to see if the error is an APEX internal error. For uncommon internal errors, it calls the `ADD_ERROR_LOG` helper procedure to log the error. It then sets user-facing error message by assigning the alternate text to the `message` field in the `l_result` return record. It uses a generic message with the name `UNEXPECTED_INTERNAL_ERROR`.

For other errors, it checks `p_error.ora_sqlcode` for a database constraint code. If it finds one, it calls `EXTRACT_CONSTRAINT_NAME` in the `APEX_ERROR` package. Using that constraint name, say *XXX*, it calls `APEX_LANG.GET_MESSAGE` with message name `CONSTRAINT_`*XXX*. If that message name exists, it uses the the corresponding translatable text. Otherwise, `GET_MESSAGE` returns the *original* message name itself. So if the returned message is *different* from the message name passed in, it sets that text as the error message to show the user.

If no other path changed the error message yet, it simplifies the error by calling `GET_FIRST_ORA_ERROR_TEXT` in the `APEX_ERROR` package. Then, finally, it returns the `l_result` to the APEX engine.

```
-- In package messages_app
function apex_error_handling (
    p_error in apex_error.t_error )
    return apex_error.t_error_result
is
    l_result          apex_error.t_error_result;
    l_constraint_name varchar2(255);
    l_error_message   varchar2(255);
begin
    -- Always start by initializing the error result
    l_result := apex_error.init_error_result(p_error);
```


Copyright © 2026, Oracle and/or its affiliates.

```
    -- For uncommon internal errors, log and show a generic message instead
    if p_error.is_internal_error then
        -- only log errors that are not common ones
        if not p_error.is_common_runtime_error then
            add_error_log( p_error );
            -- Change the error message to a generic one
            l_result.message        :=
apex_lang.get_message('UNEXPECTED_INTERNAL_ERROR');
            l_result.additional_info := null;
        end if;
    else
        -- Always show the error as inline error
        l_result.display_location := apex_error.c_inline_in_notification;

        -- If it's a constraint violation like:
        --
        --   ORA-00001: unique constraint violated
        --   ORA-02091: transaction rolled back (-> can hide a deferred
constraint)
        --   ORA-02290: check constraint violated
        --   ORA-02291: integrity constraint violated - parent key not found
        --   ORA-02292: integrity constraint violated - child record found
        --
        -- try to get a friendly error message from our constraint lookup
table.
        -- If the constraint in our lookup table is not found, fallback to
        -- the original ORA error message.
        if p_error.ora_sqlcode in (-1, -2091, -2290, -2291, -2292) then
            l_constraint_name := apex_error.extract_constraint_name (
                                    p_error => p_error );

            begin
                -- Try to find text message CONSTRAINT_XXX for constraint
name XXX
                l_error_message := apex_lang.get_message('CONSTRAINT_'||
l_constraint_name);
                -- If we found a text message (message will NOT just be the
name itself)
                if l_error_message != 'CONSTRAINT_'||l_constraint_name then
                    l_result.message := l_error_message;
                end if;
            exception
                -- not every constraint has to be in our lookup table
or
                when no_data_found then null;
            end;
        end if;

        -- If its an error that we didn't find a message for yet, simplify it
        if     p_error.ora_sqlcode is not null
           and l_result.message = p_error.message
        then
            l_result.message := apex_error.get_first_ora_error_text (
                                    p_error => p_error );
        end if;
    end if;
```


Copyright © 2026, Oracle and/or its affiliates.

```
    return l_result;
end apex_error_handling;
```

20.1.6.4 Logging Error Info for Troubleshooting

Log error details in a separate transaction so troubleshooting data survives the rollback.

The `ADD_ERROR_LOG` helper procedure receives the `T_ERROR` record the APEX engine passed your error handling function. It writes useful information this record contains to the following `MESSAGE_APP_ERRORS` table.

```
create table messages_app_errors
(
    id number generated by default on null as identity not null,
    err_time        timestamp(6) default systimestamp,
    app_id          number,         /* Application ID           */
    app_page_id     number,         /* Page ID                  */
    app_user        varchar2(512),  /* Application User         */
    user_agent      varchar2(4000), /* Browser User Agent       */
    ip_address      varchar2(512),  /* Browser IP Address       */
    ip_address2     varchar2(512),  /* ORDS IP Address          */
    message         varchar2(4000), /* Error Message            */
    page_item_name  varchar2(255),  /* Page Item Name           */
    region_id       number,         /* Region ID                */
    column_alias    varchar2(255),  /* Column Alias             */
    row_num         number,         /* Row Number               */
    apex_error_code varchar2(255),  /* APEX Error Code          */
    ora_sqlcode     number,         /* Oracle SQL Code          */
    ora_sqlerrm     varchar2(4000), /* Oracle SQL Error Message */
    error_backtrace varchar2(4000), /* Error Backtrace          */
    procedure_name  varchar2(1000), /* Procedure Name           */
    error_text      varchar2(4000), /* Error Text               */
    constraint message_app_errors_id_pk primary key (id)
);
```

The procedure operates in a separate transaction by including the following in its declare section.

```
pragma autonomous_transaction;
```

This requires also that it perform an explicit `commit`, which it does at the end. This combination ensures the logged row is saved even though the APEX engine performs a rollback to abandon pending data changes when encountering an error.

The procedure starts by deleting log messages older than three weeks. Then it inserts a new `MESSAGES_APP_ERRORS` row. Some of the values comes from APEX substitution strings like:

- `APP_ID` – application id
- `APP_PAGE_ID` – page id
- `APP_USER` – currently logged-in user name

The remaining values are from the `p_error` record that includes, among others, details like:

- `p_error.page_item_name` – page item name

Copyright © 2026, Oracle and/or its affiliates.

- `p_error.message` – error message
- `p_error.region_id` – region id
- `p_error.apex_error_code` – APEX error code
- `p_error.ora_sqlcode` – ORA error code
- `p_error.ora_sqlerrm` – ORA error message
- `p_error.error_backtrace` – PL/SQL call stack

> **Tip**
>
> See the reference documentation for the `APEX_ERROR` package for *complete* details on the fields of the `T_ERROR` record.

```
-- In package messages_app
procedure add_error_log (
    p_error           in apex_error.t_error,
    p_procedure_name  in varchar2 default null,
    p_error_text      in varchar2 default null)
is
pragma autonomous_transaction;
begin
    -- Remove old errors
    delete from messages_app_errors where err_time <= current_timestamp - 21;
    -- Log the error.
    insert into messages_app_errors (
        app_id,
        app_page_id,
        app_user,
        user_agent,
        ip_address,
        ip_address2,
        message,
        page_item_name,
        region_id,
        column_alias,
        row_num,
        apex_error_code,
        ora_sqlcode,
        ora_sqlerrm,
        error_backtrace,
        procedure_name,
        error_text)
    select v('APP_ID'),
        v('APP_PAGE_ID'),
        v('APP_USER'),
        owa_util.get_cgi_env('HTTP_USER_AGENT'),
        owa_util.get_cgi_env('REMOTE_ADDR'),
        sys_context('USERENV', 'IP_ADDRESS'),
        substr(p_error.message,0,4000),
        p_error.page_item_name,
        p_error.region_id,
        p_error.column_alias,
```


Copyright © 2026, Oracle and/or its affiliates.

```
            p_error.row_num,
            p_error.apex_error_code,
            p_error.ora_sqlcode,
            substr(p_error.ora_sqlerrm,0,4000),
            substr(p_error.error_backtrace,0,4000),
            p_procedure_name,
            p_error_text
        from dual;
        commit;
end add_error_log;
```

20.1.6.5 Experiencing Custom Error Handling

Test custom error handling with constraint violations, database exceptions, and logged internal errors.

With the error handling function in place, suppose you add the following trigger on the `EMP` table to simulate two *additional* kinds of database errors. One comes from an unexpected `ZERO_DIVIDE` error the trigger encounters if the user sets `COMM` to the value `1234`. The second error is a custom exception from `RAISE_APPLICATION_ERROR`. This happens when a user sets `COMM` to `1235`.

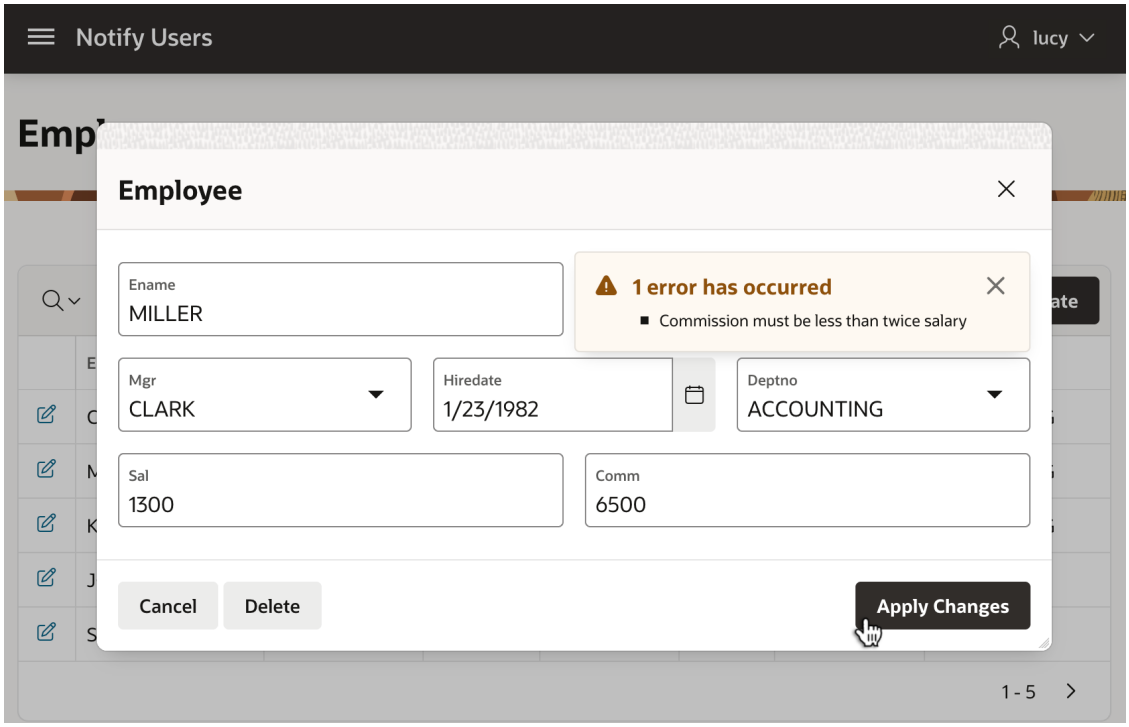
```
create or replace trigger emp_conditional_error_test
before insert or update on emp
for each row
begin
    -- Intentionally cause a divide-by-zero error for educational purposes
    if :new.comm = 1234 then
        if :new.comm / 0 > 1 then
            null;
        end if;
    -- Raise a custom exception message from the database
    elsif :new.comm = 1235 then
        raise_application_error(-20020,'Some error from the database');
    end if;
end;
```

Now, if the user enters a commission greater than twice the employee's salary, they see the custom error message provided by the translatable text message named **`COMMISSION`**`_COMM_LESS_THAN_TWICE_SAL`.

```
Commission must be less than twice salary
```


Copyright © 2026, Oracle and/or its affiliates.

Figure 20-21 Error Handling Function Shows Custom Error for Constraint Violation

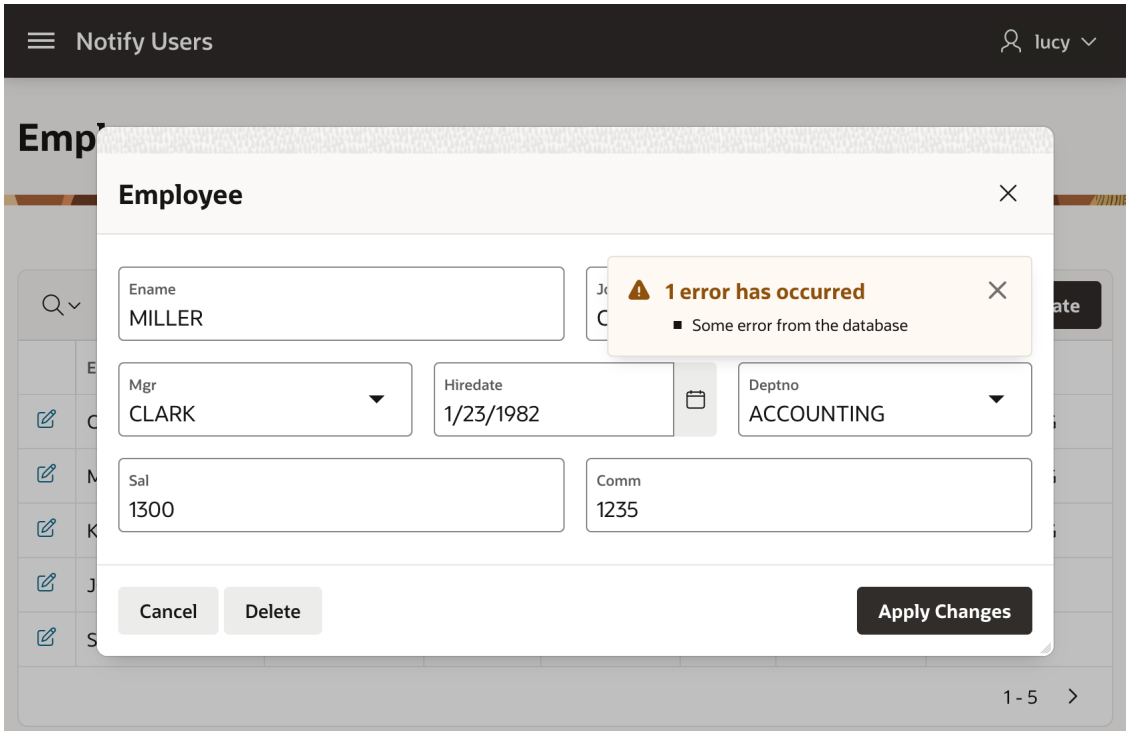


If the user enters a commission value of 1235, they see a simplified version of the error message from `RAISE_APPLICATION_ERROR`:

```
Some error from the database
```


Copyright © 2026, Oracle and/or its affiliates.

Figure 20-22 Error Handling Function Shows Simplified Database Exception Message



Similarly, if they enter 1234 to cause the divide by zero error in the trigger, they see a simplified error message:

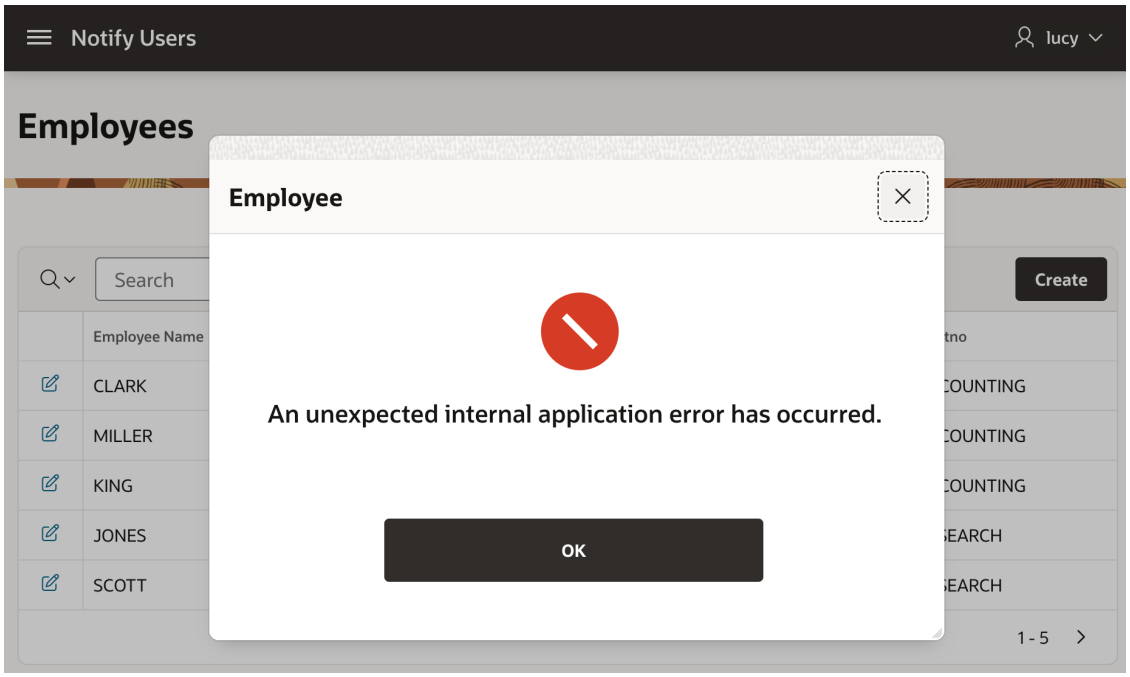
```
divisor is equal to zero
```

Suppose in a hurry, while doing an emergency fix for your app, you inadvertently **unset** the *Primary Key* switch on the `P8_EMPNO` page item that identifies the unique identifier for a form region. When you deploy the emergency fix to production, users suddenly start seeing this custom message for the uncommon internal APEX engine error:

```
An unexpected internal application error has occurred.
```

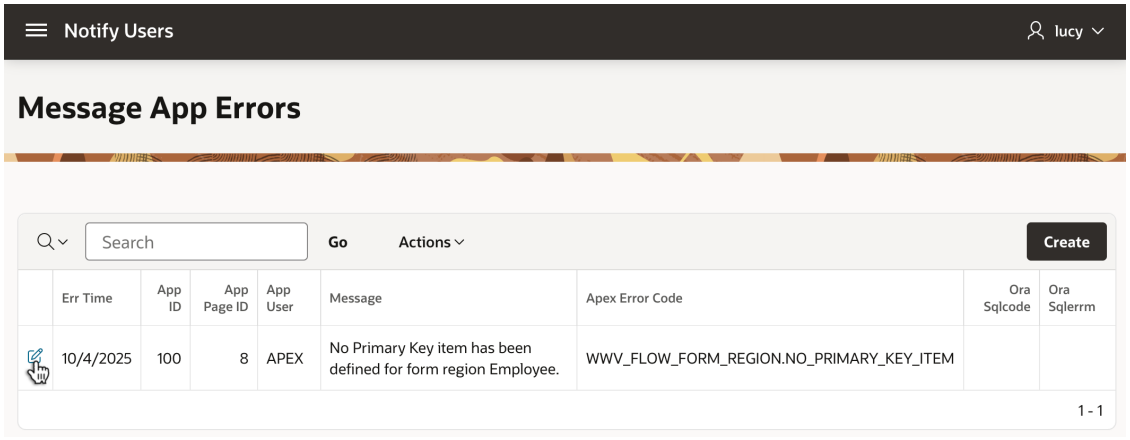

Copyright © 2026, Oracle and/or its affiliates.

Figure 20-23 Error Handling Functions Show Custom Message for Uncommon Internal Error



When users report the problem, you check a page only application administrators can see that shows the logged error information in the `MESSAGES_APP_ERRORS` table. As shown below, you see immediately that the problem relates to a primary key field setting in page 8 of application 100.

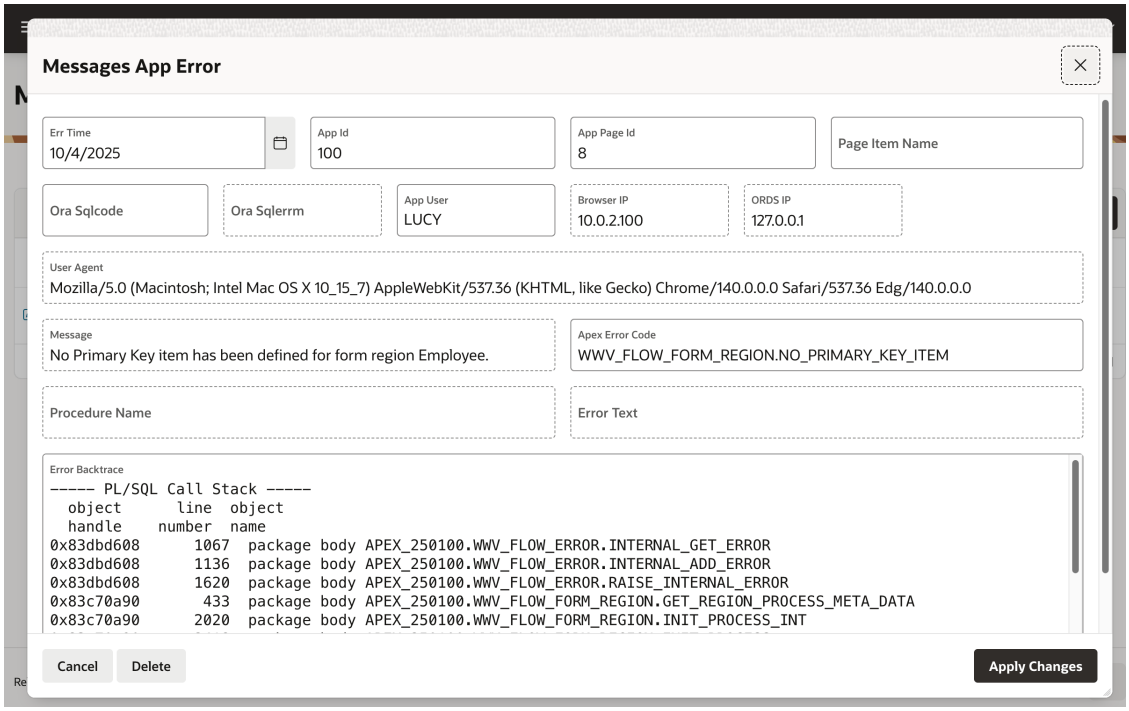
Figure 20-24 Error Handling Function Logged Uncommon Internal Error for Review



Clicking the details link, you see additional error info your error handling function logged to the table. Using these logged details, you quickly apply the fix, deploy a new version to production, and get users back working happily again.

Copyright © 2026, Oracle and/or its affiliates.

Figure 20-25 Viewing Logged Error Details for Uncommon Internal Error



20.2 Sending Emails with Templates

Send email from a native page process or workflow activity.

Use the native *Send E-Mail* page process or workflow activity to deliver a message. The native action type in task definitions and automations works the same way. When you need to send a mail from business logic, use the `SEND` function or procedure in the `APEX_MAIL` package. To monitor the queue of outgoing emails, use the `APEX_MAIL_QUEUE` view, and to view recently sent email consult the `APEX_MAIL_LOG` view.

- Setting Up Default From Email Address
  Configure the default From address your app uses for native email features.
- Sending Email Without a Template
  Send emails with inline plain text or HTML and substitution strings.
- Defining Email Template with Directives
  Define an email template with placeholders and directives for conditional email content.
- Using Email Template and Placeholders
- Sending Emails from Business Logic
  Send templated email from PL/SQL business logic with `APEX_MAIL.SEND`.

20.2.1 Setting Up Default From Email Address

Configure the default From address your app uses for native email features.

The native *Send E-Mail* facilities default the sender in the *From* property using the `&APP_MAIL.` substitution. Configure its value in the *Application Email From Address* field on the *Properties* section of your app definition.

Copyright © 2026, Oracle and/or its affiliates.

> **Tip**
>
> Your APEX instance administrator configures Email *server* settings for use by all workspaces, so you don't need to configure it yourself.

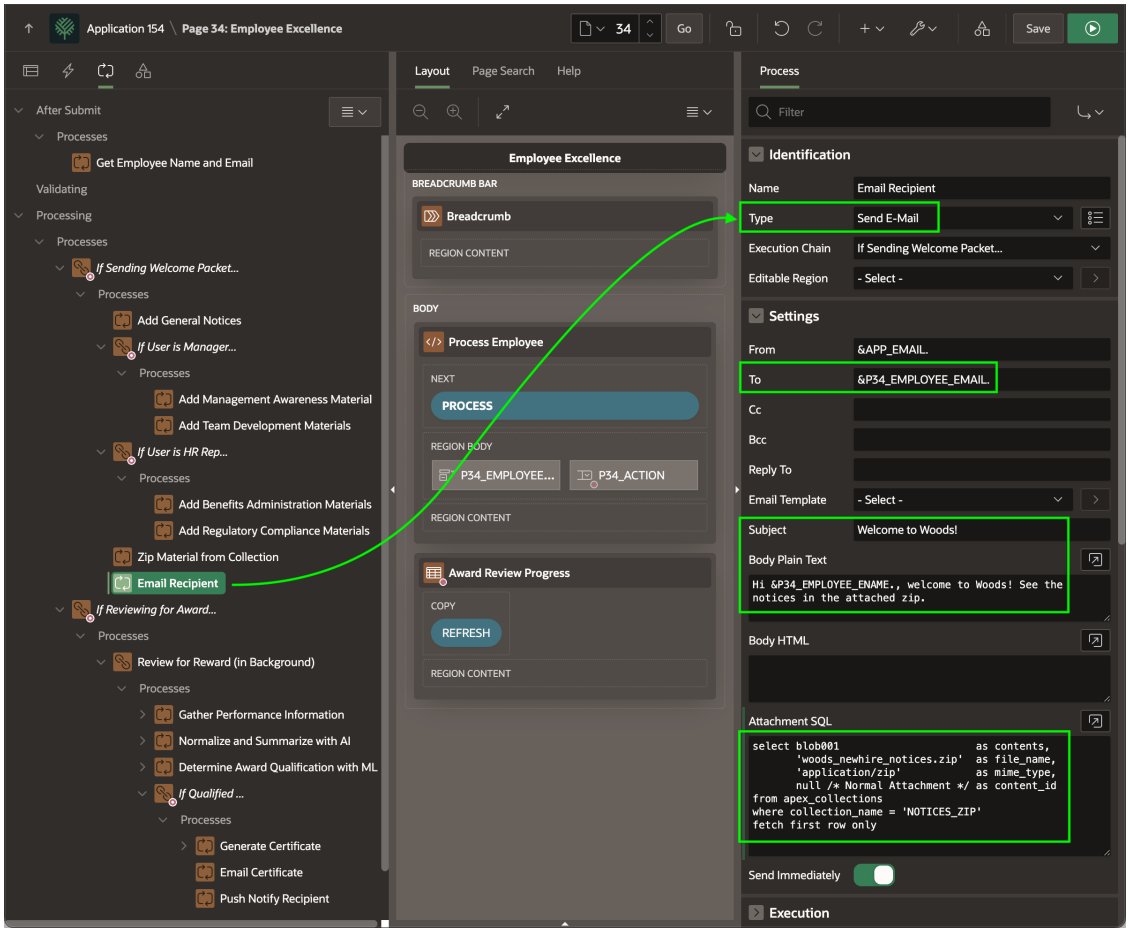
20.2.2 Sending Email Without a Template

Send emails with inline plain text or HTML and substitution strings.

You can send one-off emails using inline text and HTML with substitution strings for the email body. As shown below, the *Welcome Packet* email from Choosing Descriptive Process Names used this approach. The figure shows the Email Recipient page process selected in the processing tree and highlights:

- Its **Type** is **Send E-Mail**
- A **To** field value of `&P34_EMPLOYEE_EMAIL.`
- **Subject** and **Body Plain Text** are configured
- The SQL statement it uses to attach a ZIP file of welcome packet documents.

Figure 20-26 Sending Email with a Native Page Process Without a Template



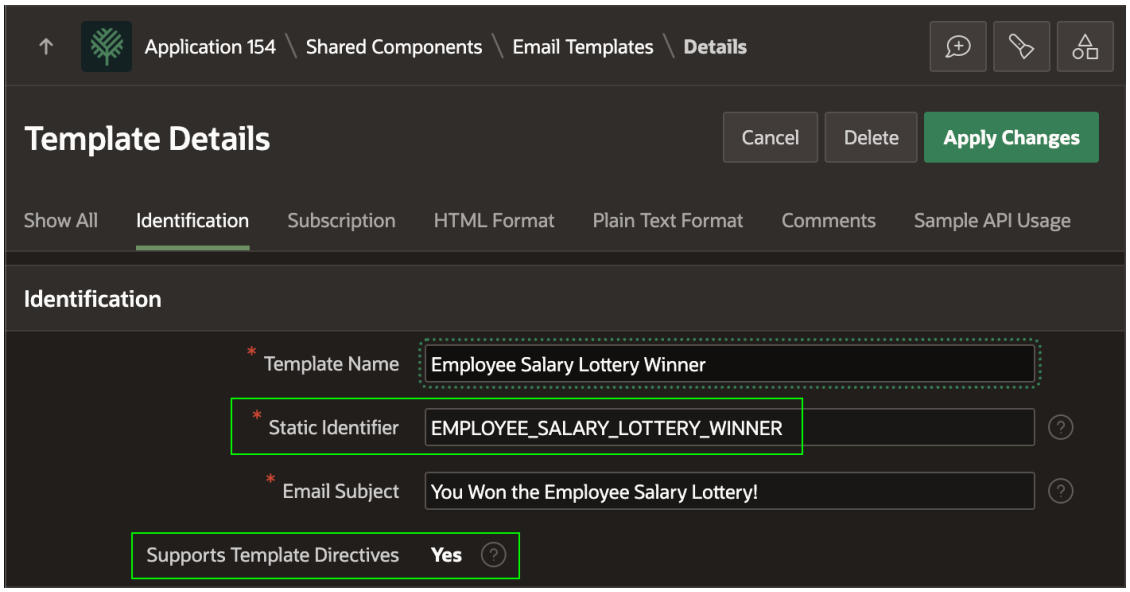
Copyright © 2026, Oracle and/or its affiliates.

20.2.3 Defining Email Template with Directives

Define an email template with placeholders and directives for conditional email content.

Using an email *template* simplifies the job by separating the formatting from the data and attachments that change with each use. When you send a mail, you can reference an existing template, set values for any placeholders it contains, and include any optional attachments required. The *Employee Salary Lottery Winner* email template from the example in Defining Automation Actions in Sequence appears below. Notice its *Static Identifier* that's used when using the `APEX_MAIL.SEND` API, and that it supports HTML Template Directives for conditional formatting in the email body text and HTML markup.

Figure 20-27 Template Has Static ID for API Use and Can Use Template Directives

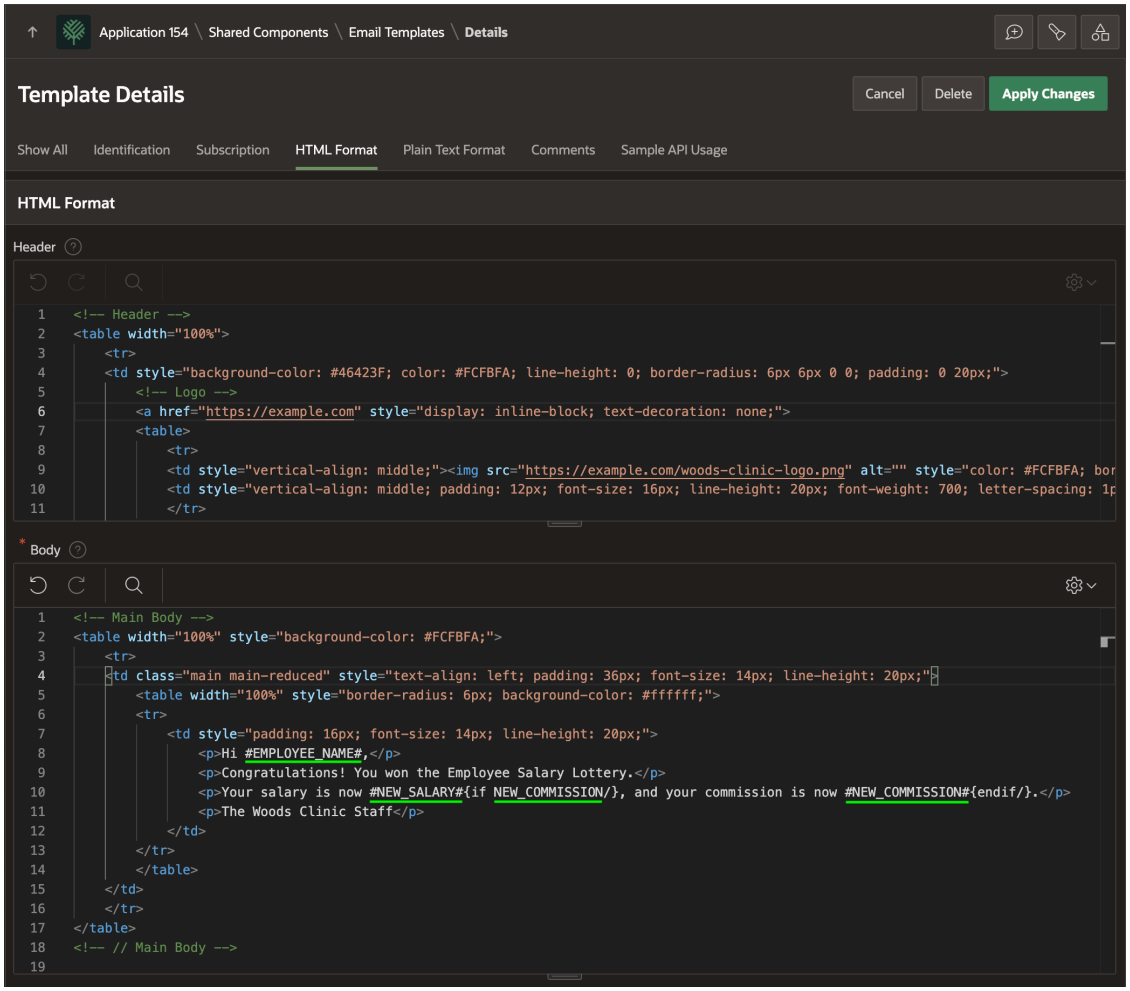


This email template specifies the HTML markup for the *Header* and *Body* sections, and includes placeholders `EMPLOYEE_NAME`, `NEW_SALARY`, and `NEW_COMMISSION` underlined for emphasis in the template *Body*. Notice the body uses the `{if/}`…`{endif/}` template directive to conditionally include the phrase about a new commission only if `NEW_COMMISSION` has a value.

```
{if NEW_COMMISSION/}, and your commission is now #NEW_COMMISSION#{endif/}
```


Copyright © 2026, Oracle and/or its affiliates.

Figure 20-28 Defining an Email Template with Placeholders



20.2.4 Using Email Template and Placeholders

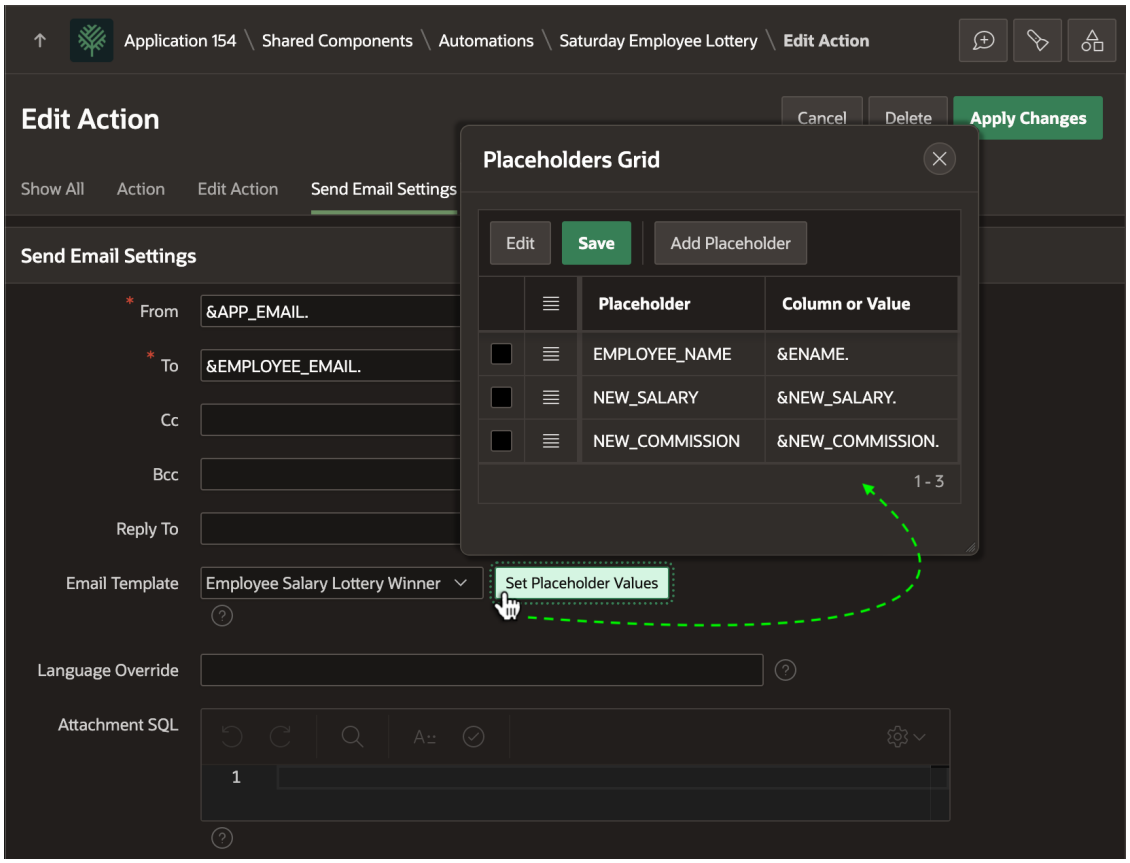
When a native *Send E-Mail* action, process, or activity uses the email template, as shown below, you set placeholder values for the current message to send using substitution syntax. You can reference the value of columns, page items, workflow variables, application items, or other system substitutions, depending on the context.

The figure shows the **Placeholders Grid** dialog, opened by clicking the (**Set Placeholder Values**) button in the **Edit Action** page of an automation definition in App Builder. The automation's Send Email action uses the *Employee Salary Lottery Winner* email template. Its three placeholder names appear in the grid. Each's value is configured to use the column value of the current automation row being processed using substitution notation:

- EMPLOYEE_NAME → **&ENAME.**
- NEW_SALARY → **&NEW_SALARY.**
- NEW_COMMISSION → **&NEW_COMMISSION.**

Copyright © 2026, Oracle and/or its affiliates.

Figure 20-29 Configuring Email Template and Placeholder Values for Send-Email Action



20.2.5 Sending Emails from Business Logic

Send templated email from PL/SQL business logic with `APEX_MAIL.SEND`.

The equivalent code to send the same email from an *Execute Code* action looks like this. No `p_from` parameter is required because it defaults to using the value of `APP_EMAIL`. Notice the p_placeholders parameter expects a JSON document with the placeholder names and values. The `PLIST_TO_JSON_CLOB` function in the `APEX_STRING` package is handy to produce it. Just pass the name/value pairs as an `APEX_T_VARCHAR2` string list. The values in the odd positions are the JSON property names and the ones in even positions are the corresponding values.

```
apex_mail.send(
   p_to                => :EMPLOYEE_EMAIL,
   p_template_static_id => 'EMPLOYEE_SALARY_LOTTERY_WINNER',
   p_placeholders      => apex_string.plist_to_json_clob(apex_t_varchar2(
                             'EMPLOYEE_NAME' , :ENAME,
                             'NEW_SALARY'    , :NEW_SALARY,
                             'NEW_COMMISSION', :NEW_COMMISSION)));
```

20.3 Delivering Push Notifications

Send push notifications from native processes or PL/SQL to users who opted in.

Copyright © 2026, Oracle and/or its affiliates.

Use the native *Send Push Notification* page process or workflow activity to send a push notification to your app users. APEX automatically delivers them *only* to users who have opted-in to receive them. The native action type in task definitions and automations works identically. To send a push notification from business logic, use the `SEND_PUSH_NOTIFICATION` procedure in the `APEX_PWA` package. To monitor the queue of outgoing push notifications, use the `APEX_PUSH_NOTIFICATIONS_QUEUE` view.

- Enabling Push Notifications in an App
  Enable PWA and push notification support so users can install your app and opt in to notifications.
- Generating Key Pairs for Push Notifications
  Generate a key pair credential to encrypt and sign push notifications.
- Letting Users Manage Notification Settings
  Add notification settings pages so users can manage push notification opt-in.
- Opting-In to Receive Push Notifications
  Users opt in to push notifications from the user settings menu on each device.
- Pushing a Smile to a Colleague
  Send a push notification to an opted-in user from a native page process.
- Overriding Notification Target Link
  Configure a push notification target link to open a specific app page.
- Pushing Notification from Business Logic
  Send push notifications from PL/SQL business logic with `APEX_PWA.SEND_PUSH_NOTIFICATION`.

20.3.1 Enabling Push Notifications in an App

Enable PWA and push notification support so users can install your app and opt in to notifications.

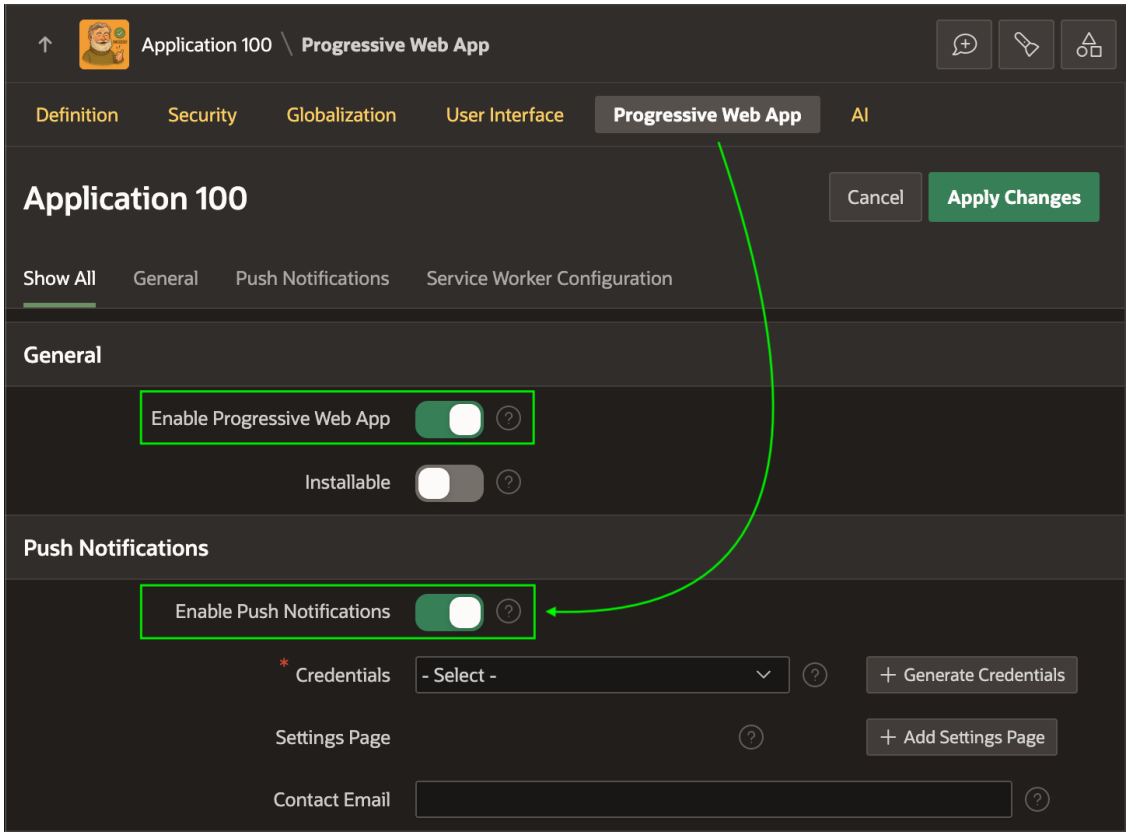
To enable push notification support, visit the *Progressive Web App* section in *Shared Components* to switch on both *Enable Progressive Web App* and *Enable Push Notifications* as shown below.

> ⚠ **Caution**
>
> Some mobile platforms require push notifications to be associated with an *installed* Progressive Web App (PWA), so consider enabling the *Installable* switch as well. This lets users install your app in a click or tap. After that, it looks and launches like a native app on their desktop or mobile device.

The figure shows the **Progressive Web App** tab of the **Application Definition** page in *Shared Components* in App Builder. It highlights the two switches you need to enable to send push notifications from your application.

Copyright © 2026, Oracle and/or its affiliates.

Figure 20-30 Enabling Push Notifications in Your Application



20.3.2 Generating Key Pairs for Push Notifications

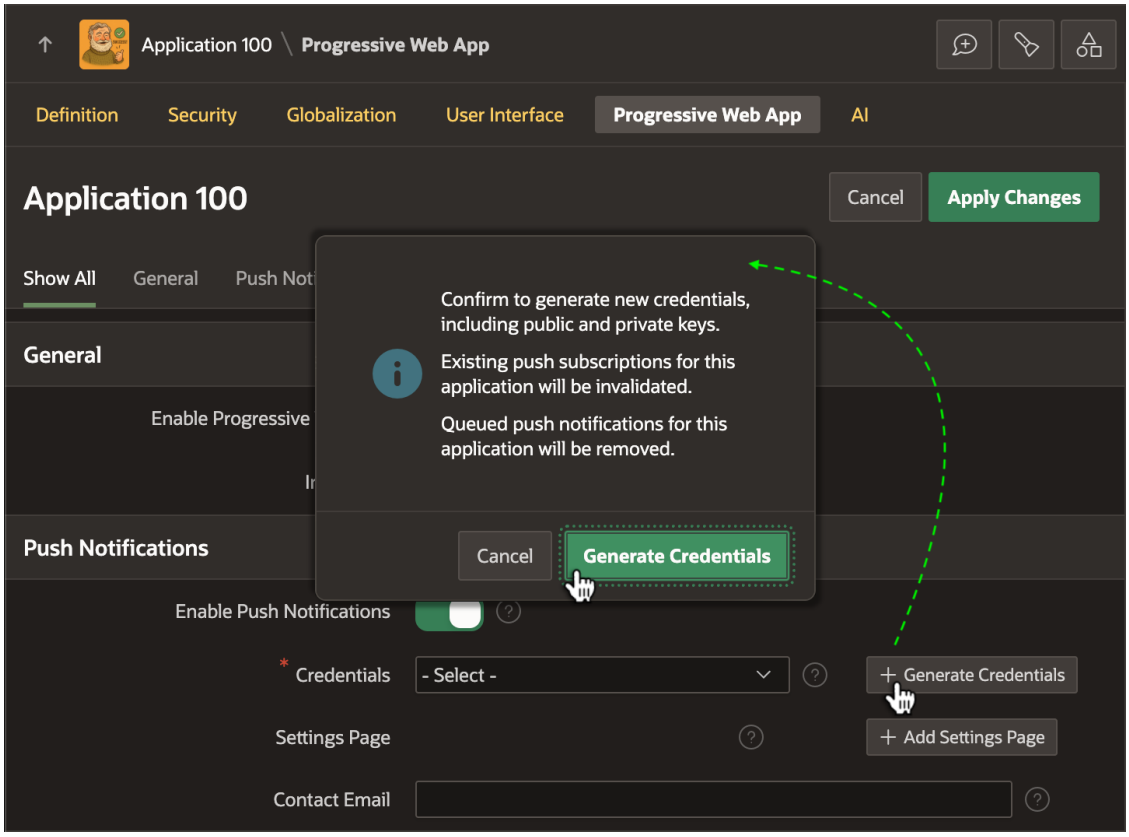
Generate a key pair credential to encrypt and sign push notifications.

Push notifications are encrypted and signed using a public/private key pair credential. Click the *(Generate Credentials)* button to generate a new *Key Pair* credential and select it as the *Credentials* entry for push notifications.

> **Tip**
>
> Once you generate the credentials, the button changes to *(Regenerate Credentials)*. You need to use this when deploying your app to a new environment, or when you want to change the existing push notification signing keys.

Copyright © 2026, Oracle and/or its affiliates.

Figure 20-31 Generating Key Pair Credential for Signing Push Notifications



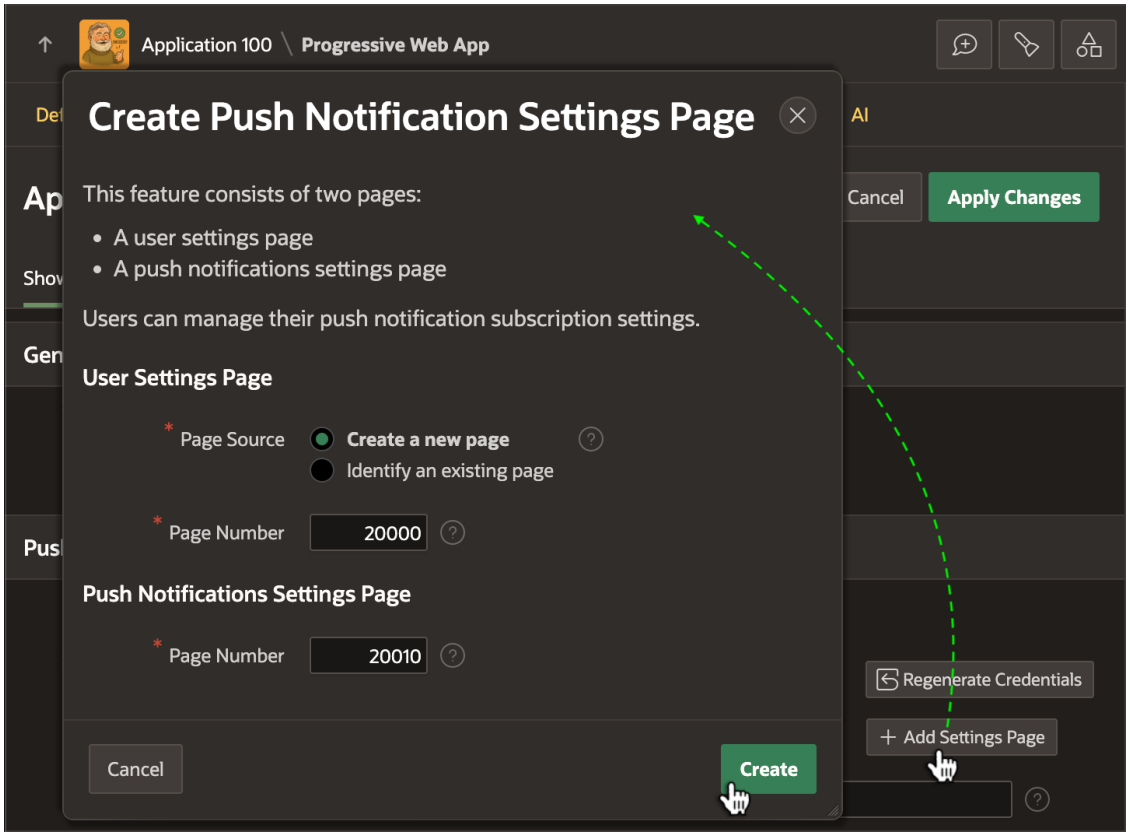
20.3.3 Letting Users Manage Notification Settings

Add notification settings pages so users can manage push notification opt-in.

Push notifications only deliver to users who opt-in. As shown below, to add pages and menu entries to your app that let users manage their push notification preferences, click the *(Add Settings Page)* button in the **Push Notifications** section of the **Progressive Web App** app settings page.

Copyright © 2026, Oracle and/or its affiliates.

Figure 20-32 Adding Push Notification Settings Pages to Your App



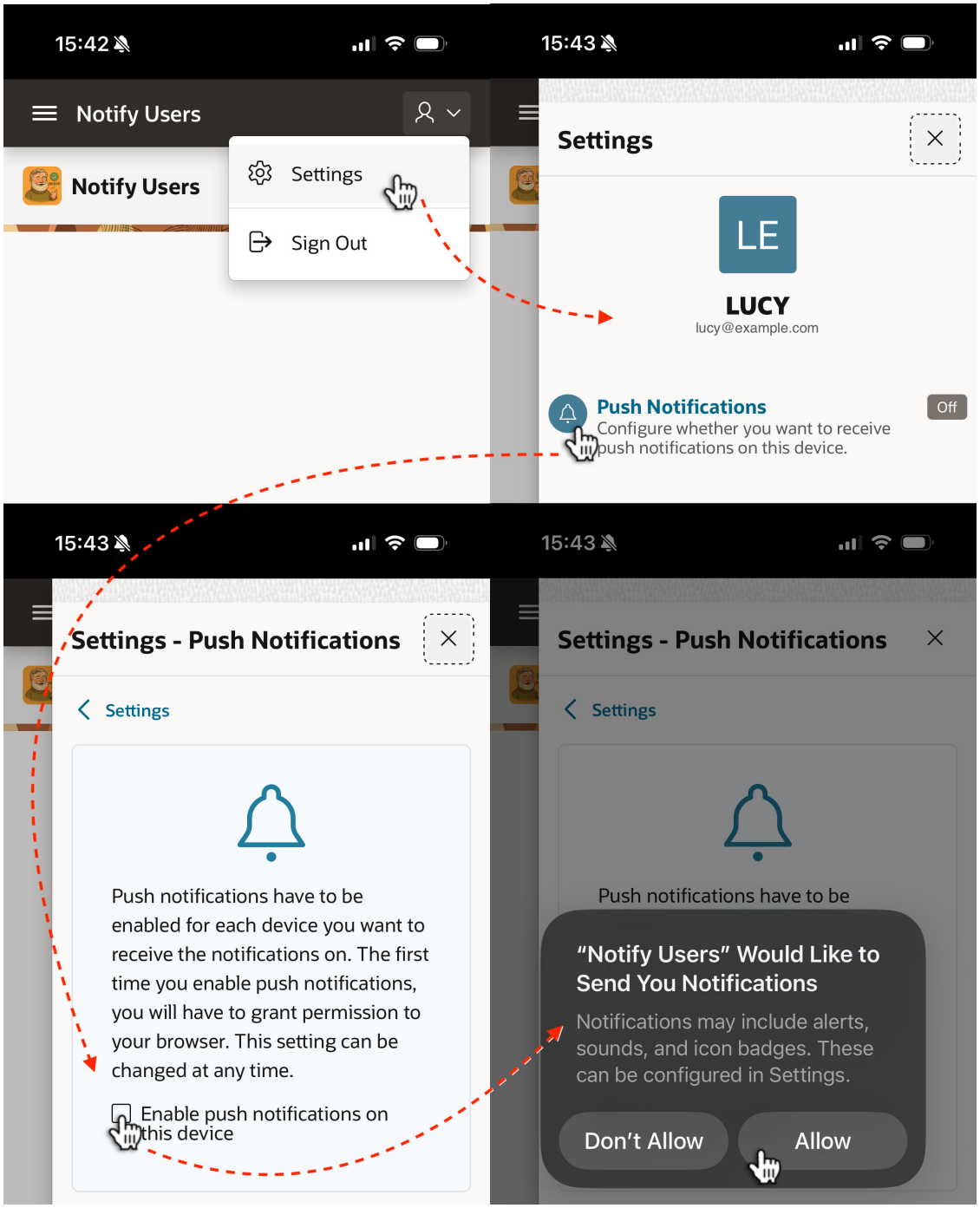
20.3.4 Opting-In to Receive Push Notifications

Users opt in to push notifications from the user settings menu on each device.

An application user like LUCY manages her settings under the logged-in user drop down menu. Lucy opts-in to receive push notifications by following the sequence shown below. She access the *Settings* menu, taps on *Push Notifications*, taps to *Enable push notifications on this device*, and taps to *Allow* notifications. The user preference persists until Lucy disables it. However, if you ever change the *Key Pair* credential, users would need to opt-in again.

Copyright © 2026, Oracle and/or its affiliates.

Figure 20-33 Lucy Uses App's Settings to Opt-In to Notifications



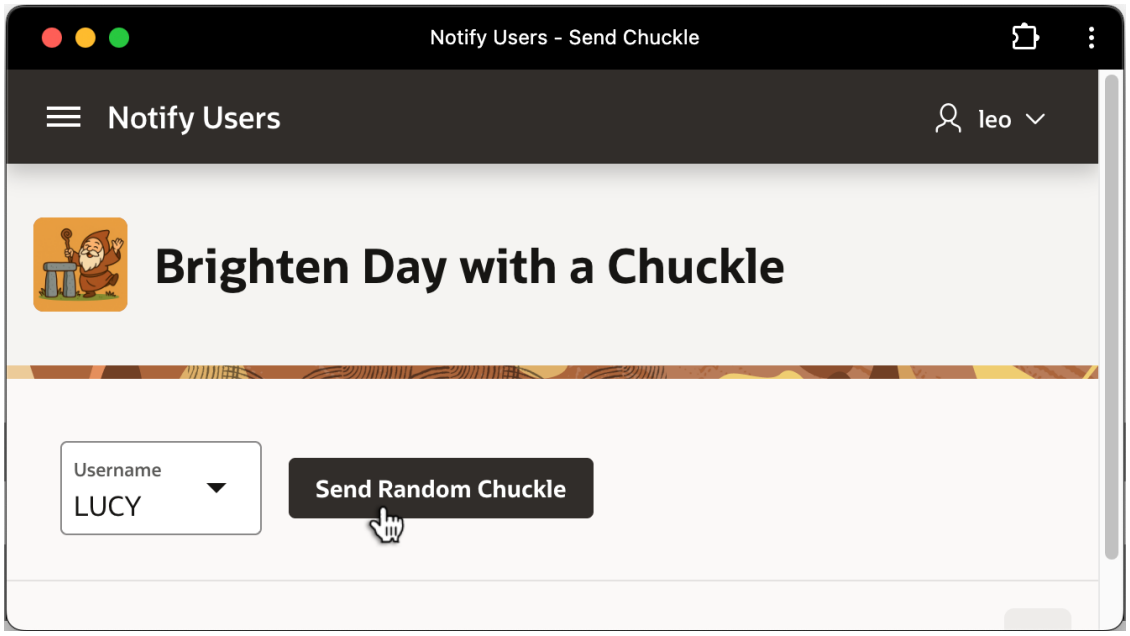
20.3.5 Pushing a Smile to a Colleague

Send a push notification to an opted-in user from a native page process.

Push notifications usually flag something important; they can also deliver a laugh. In a *Send Chuckle* page like the one below, users can "gift" a colleague a random line that might make them smile.

Copyright © 2026, Oracle and/or its affiliates.

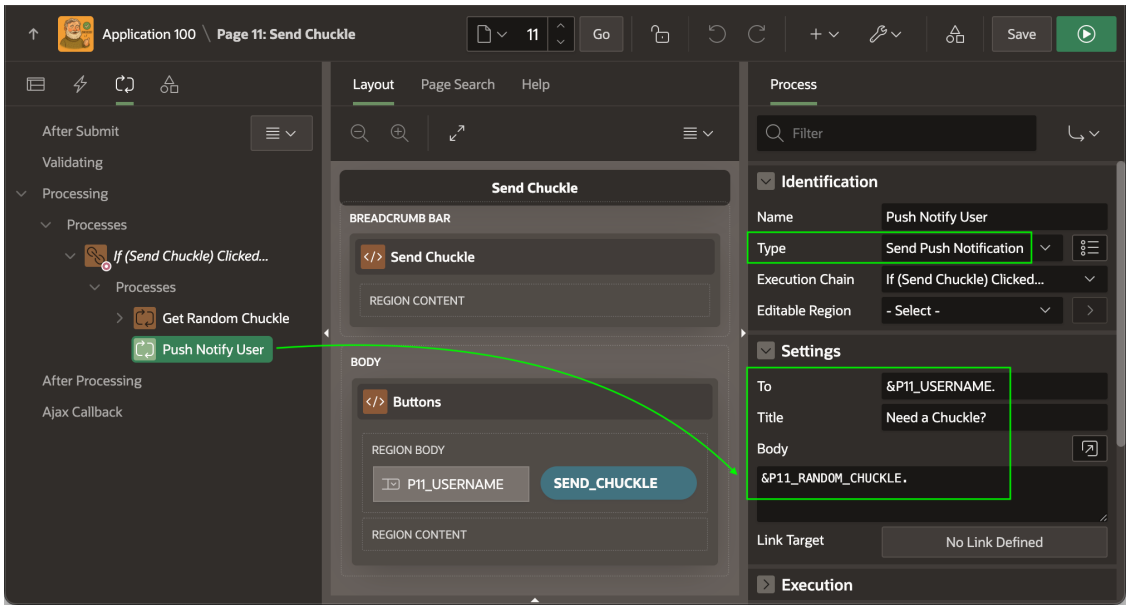
Figure 20-34 Users Send Push Notification with Random Laughable Line



The page uses a *Get Random Chuckle* page process of type *Invoke API* to call the `RANDOM_CHUCKLE` function. It configures the hidden `P11_RANDOM_CHUCKLE` page item to receive the result. Then, it uses the *Send Push Notification* page process shown below to deliver the push notification to the username selected in the `P11_USERNAME` select list.

APEX optimizes delivery to alert only opted-in users, and automatically sends to all their subscribed devices. If you push a notification to a user who has *not* opted in, no problem. APEX simply ignores it for that user. This saves you from having to keep track of subscribed users yourself.

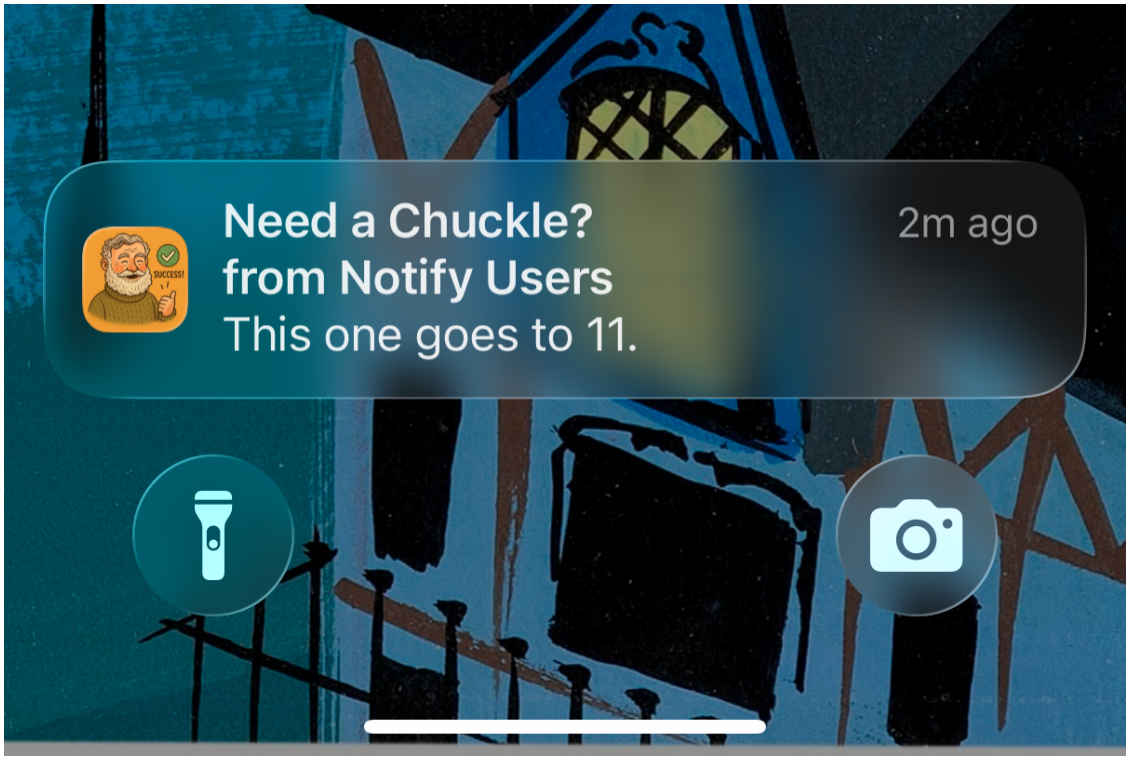
Figure 20-35 Delivering a Push Notification with Native Page Process



Copyright © 2026, Oracle and/or its affiliates.

Since Lucy has opted-in to push notification, she receives the line in a notification on her smartphone and paired watch and smiles thinking about one of her favorite movies.

Figure 20-36 Lucy Receives the Notification and Laughs



20.3.6 Overriding Notification Target Link

Configure a push notification target link to open a specific app page.

By default, clicking or tapping on the notification opens your application's home page. However, you can optionally configure the *Target Link* to take the user to a *specific* page in your application. For this to work as expected, follow these steps:

- Use an *absolute* URL, and
- Enable *Deep Linking* in the *Session Management* section of the app's *Security* settings.

20.3.7 Pushing Notification from Business Logic

Send push notifications from PL/SQL business logic with `APEX_PWA.SEND_PUSH_NOTIFICATION`.

The equivalent code to send the same push notification from an *Execute Code* page process looks like this. Use optional parameters `p_target_url` and `p_icon_url` to override the defaults.

```
apex_pwa.send_push_notification(
   p_user_name => :P11_USERNAME,
   p_title     => 'Need a Chuckle?',
   p_body      => :P11_RANDOM_CHUCKLE));
```


Copyright © 2026, Oracle and/or its affiliates.

21

Producing Pixel-Perfect Reports

Use a *Report Query* with an associated *Report Layout* to produce a pixel-perfect PDF report.

The report sources in the *Report Query* definition identify the data the report requires, while the *Report Layout* document template formats it appropriately.

The APEX engine sends the layout file and the data in JSON format to a remote printing service. The report server combines the layout with the data and returns a formatted report. Your *Report Query* definition can include multiple data sources and your layout can use any formatting the template document allows. The result is a flexible system to produce whatever pixel-perfect reports your application may need.

- Setting Your App's Remote Print Server
  Use instance print server settings to generate reports with the configured remote printing service.
- Printing Report with an Excel Template
  Generate a PDF report by merging application data into an Excel template.
- Customizing Reports with Word Template
  Use a *Report Layout* to customize the PDF download format for an Interactive Report or Classic Report region.

21.1 Setting Your App's Remote Print Server

Use instance print server settings to generate reports with the configured remote printing service.

While you can choose from *multiple* remote printing solutions, here you explore using Oracle Cloud Infrastructure Document Generator Function. It defines a set of "tags" you use to reference the data inside your layout document template. In this section's examples, you encounter a few simple tags to include data. Consult the Document Generator's reference guide for a complete tag list.

OCI Document Generator supports:

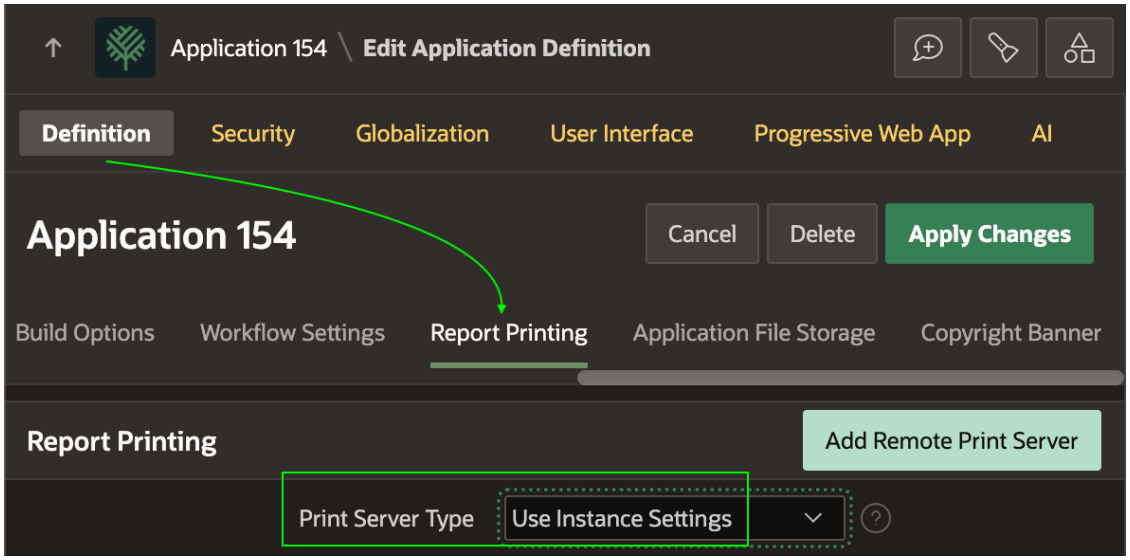
- using Microsoft Word or Excel files as layouts, and
- producing PDF, Word, or Excel files as the report output.

> **Tip**
>
> Other remote printing solutions may support additional layout file types and output formats. They may also use a different syntax for the tags to include data.

The only setup step required in your own apps is to set the *Print Server Type* as shown below to *Use Instance Settings*. This is the simplest path to pixel-perfect printing.

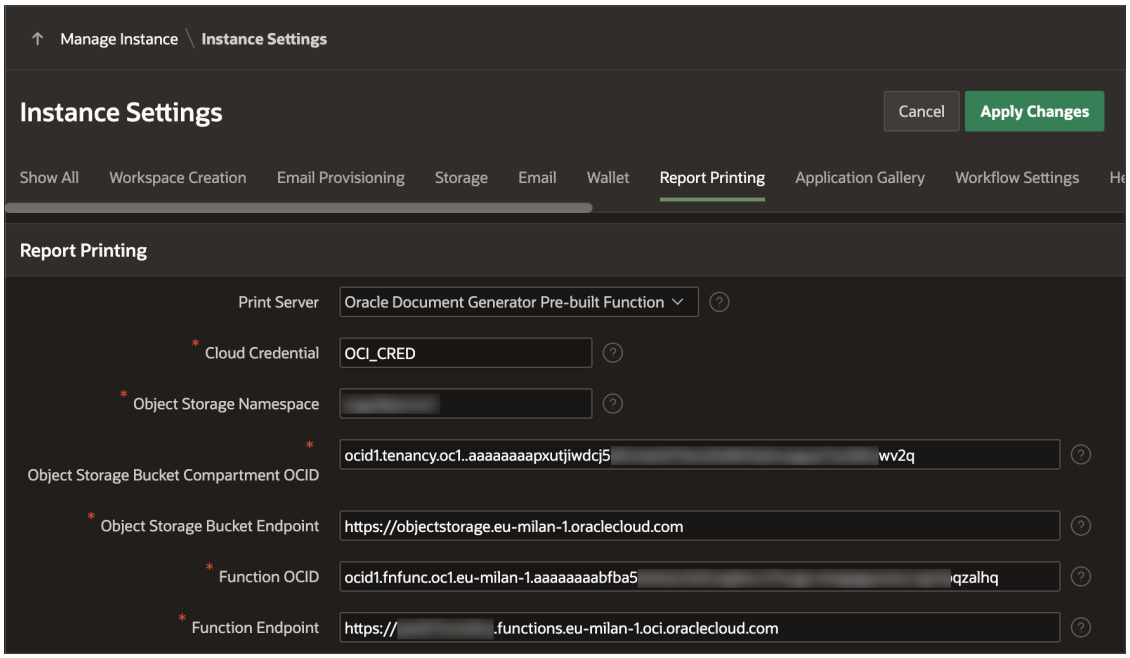
Copyright © 2026, Oracle and/or its affiliates.

Figure 21-1 Configuring App to Use Instance Settings for Remote Printing



Your APEX instance administrator should have already setup a remote print server for use by all workspaces. When they setup the instance to use Oracle Document Generator as the Print Server, as shown below, they configure the details that let APEX automate reporting for your applications. Details include a database cloud credential for an OCI user with access to an object bucket and the Oracle Document Generator Function. They also provide OCIDs and URLs to access them. However, once they have setup the remote print server, all app developers can use it without being aware of these details. The figure shows the APEX instance administrator's view of the **Report Printing** tab of the **Instance Settings** page.

Figure 21-2 Glimpse of the Oracle Document Generator Instance Settings



Copyright © 2026, Oracle and/or its affiliates.

21.2 Printing Report with an Excel Template

Generate a PDF report by merging application data into an Excel template.

HR Representatives at Woods Clinic asked for an easy way to generate an employee tenure report by department.

- Formatting Data in Excel with Tags
  Format an Excel report template with tags that map data sources and fields into the layout.
- Creating Layout for Document Template
  Create a *Report Layout* shared component to hold the document template.
- Defining Report Query Sources
  Create a *Report Query* definition to supply the report data.
- Assigning Sources a Data Loop Name
  Set each report source's **Data Loop Name** to match the short name used in the template tags.
- Using SQL Features in Report Sources
  Use SQL features and custom functions in report sources to shape data for a report.
- Inspecting the Report Query JSON
  Inspect the generated Report Query JSON to see how data loop names and source rows are structured.
- Printing Report from a Page
  Print and download a report from a page using a button, application item, and *Print Report* process.
- Experiencing Report Generation as a User
  Follow the user flow from choosing report data to downloading the generated PDF.

21.2.1 Formatting Data in Excel with Tags

Format an Excel report template with tags that map data sources and fields into the layout.

Use Microsoft Word or Excel to create a template layout for the report. For this one, you choose a spreadsheet. When thinking of the data your report requires, consider its source and the fields from it you need to incorporate. Assign each data source a short name.

For the *Tenure Report* you need data like:

- Department info (short name `dep`)
  - with fields: `dname`, `deptno`, `loc`
- Employee info ordered by tenure (short name `emp_t`)
  - with fields: `ename`, `empno`, `tenure`, `job`
- Employee info ordered by name (short name `emp_n`)
  - with fields `ename`, `hiredate`, `job`

In your document template, reference the short name of your data source using Document Generator tags {`#shortname`} and {`/shortname`} to surround {`fieldname`} tags that reference the data source field values by name.

Copyright © 2026, Oracle and/or its affiliates.

As shown below, the Excel sheet formats the three fields of the Department data source using its short name and field names like this:

```
{#dep}Department {dname} ({deptno}) in {loc}{/dep}
```

In a similar way, under the *Employees by Tenure* heading, it references the fields of the Employee data source ordered by tenure using the following tags. A vertical bar below represents a cell boundary. Document Generator repeats the *row* of cells between `{#emp_t}` and `{/emp_t}` for each row in the data source. When data source row repetition produces new *spreadsheet* rows, it's known as *horizontal* repetition of cells.

```
{#emp_t}{ename}({empno}) | {tenure} | {job}{/emp_t}
```

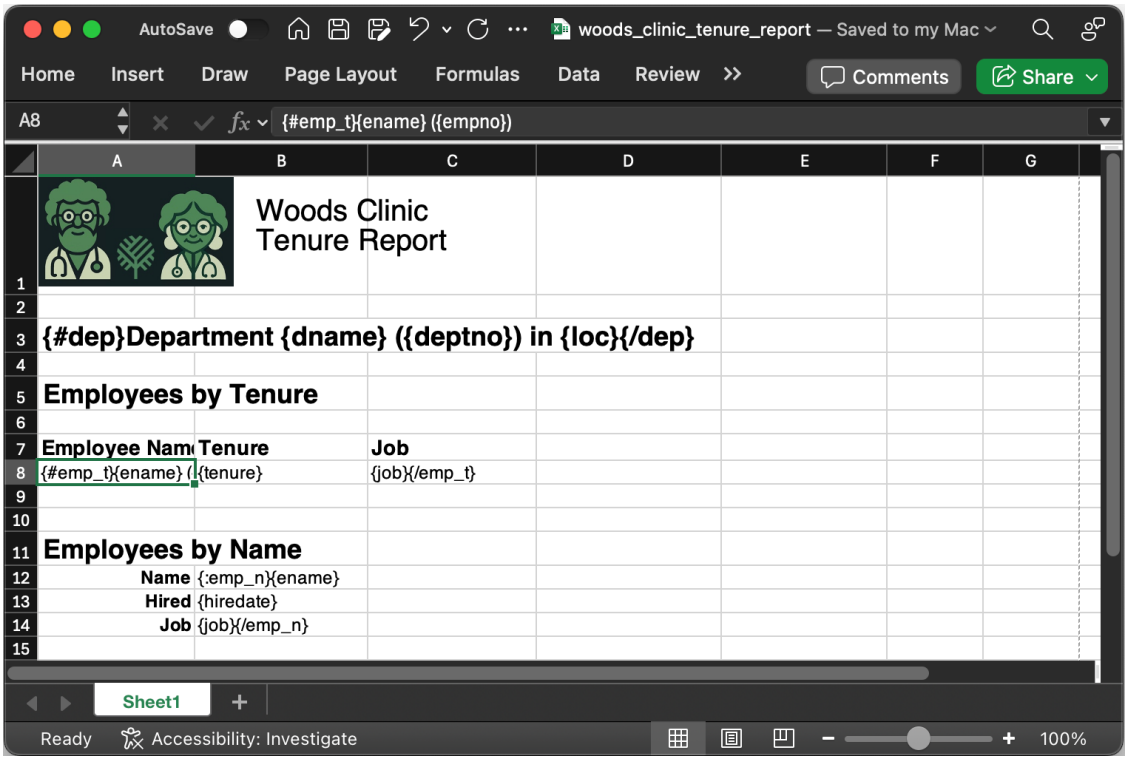
Under the *Employees by Name* heading, it uses slightly different tags to repeat the Employees ordered by name *vertically* instead. It references the fields of the Employee data source ordered by name using the following tags. A horizontal bar represents a cell boundary. Document Generator repeats the *column* of cells between `{:emp_n}` and `{/emp_n}` for each row in the data source. When data source row repetition produces new *spreadsheet* columns, it's known as *vertical* repetition of cells.

```
{:emp_n}{ename}
–
{hiredate}
–
{job}{/emp_n}
```

The figure shows the *Woods Clinic Tenure Report* in Excel, with a graphic featuring two seasoned physicians in the upper left corner, the report title to its right, and Document Generator tags and text in the cells below it to format department and employee information.

Copyright © 2026, Oracle and/or its affiliates.

Figure 21-3 Using Microsoft Excel to Create Report Layout

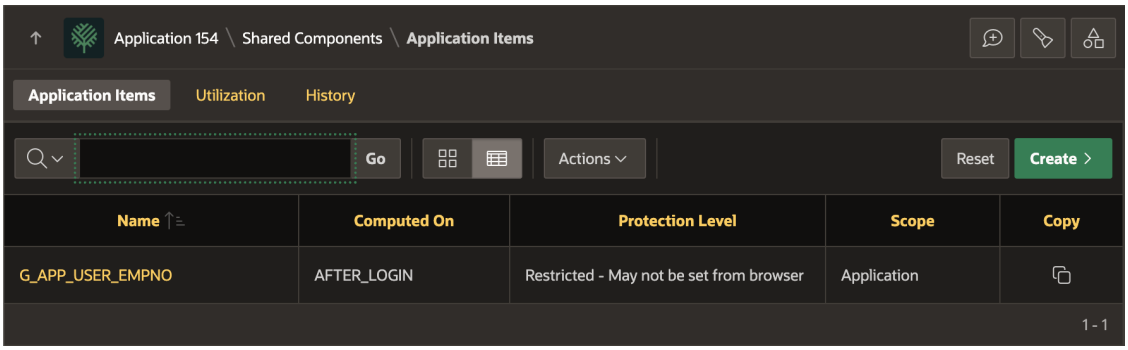


21.2.2 Creating Layout for Document Template

Create a *Report Layout* shared component to hold the document template.

As shown below, you give it a name like *Tenure Report* and assign a *Static ID* used when calling the `GENERATE_DOCUMENT` function in the `APEX_PRINT` package to produce a PDF report in your business logic. To add the `woods_clinic_tenure_report.xlsx` spreadsheet, either drag and drop it onto the *File* drop zone, or click the area and pick it from the system file dialog.

Figure 21-4 Defining Report Layout for Excel Document Template



21.2.3 Defining Report Query Sources

Create a *Report Query* definition to supply the report data.

Copyright © 2026, Oracle and/or its affiliates.

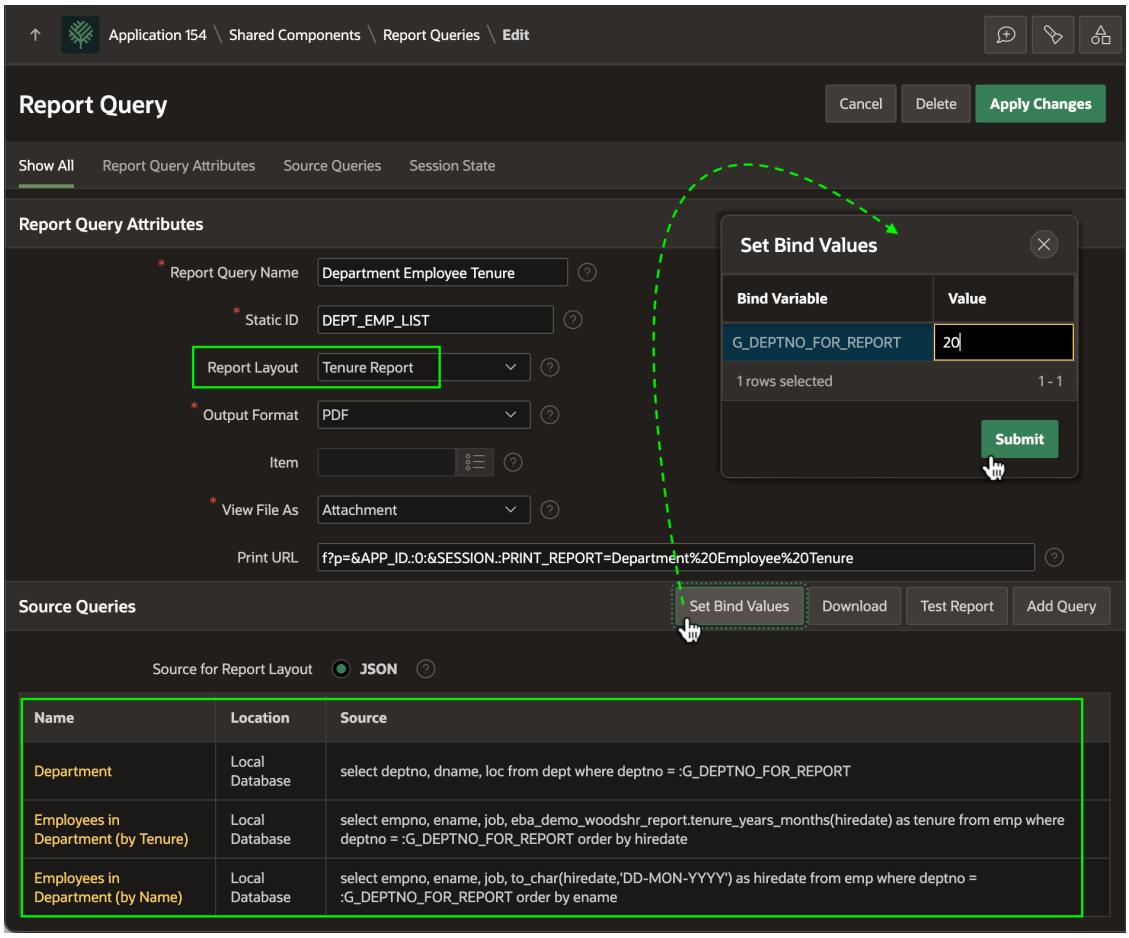
As shown below, this *Department Employee Tenure* definition includes three report sources and references the *Tenure Report* layout. Each source you include can be from any supported data source type including local or remote queries, tables, and views, REST Data Sources, and more. Here all three sources use a local database SQL query.

For example, the *Department* source uses the SQL query below that references the `G_DEPTNO_FOR_REPORT` application item as a bind variable.

```
select deptno, dname, loc
  from dept
 where deptno = :G_DEPTNO_FOR_REPORT
```

Notice the *(Set Bind Values)* button. It opens a dialog in which you can set example values for any bind variables your report source queries might reference. These example values are used when you click the *(Download)* button to produce a sample JSON file representing the data for your report. If you click *(Test Report)* the same JSON file is sent along with the layout to the remote print server so you can validate your report output.

Figure 21-5 Configuring Report Query Data for Tenure Report



21.2.4 Assigning Sources a Data Loop Name

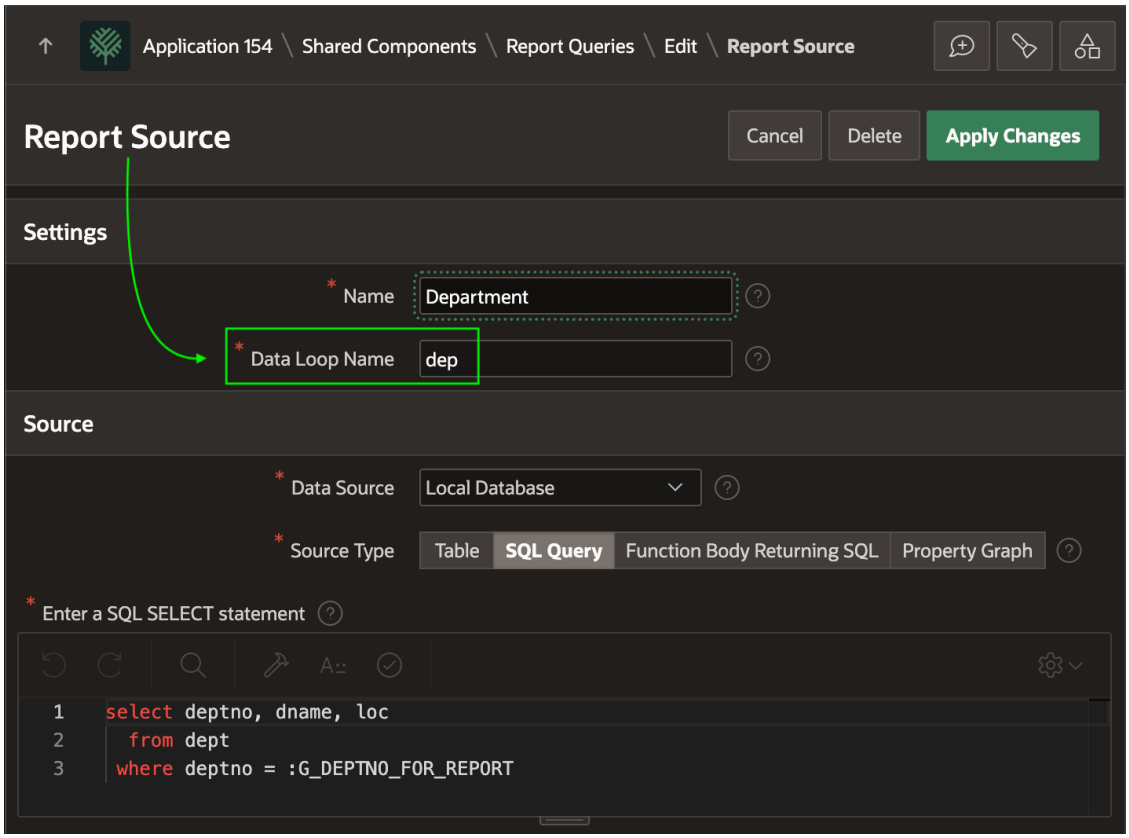
Set each report source's **Data Loop Name** to match the short name used in the template tags.

Copyright © 2026, Oracle and/or its affiliates.

In each report source, use the *Data Loop Name* setting to assign the short name you used in your layout document tags. The setting name echoes its purpose. It's the name you use in a tag to "loop over" each row of data in the report source. For example, it's set to `dep` for the *Department* report source below. Recall that the Excel spreadsheet referenced this *Data Loop* short name and lowercase column names from this query with the syntax:

```
{#dep}Department {dname} ({deptno}) in {loc}{/dep}
```

Figure 21-6 Configuring Report Source Data Loop to Match Layout Tags



21.2.5 Using SQL Features in Report Sources

Use SQL features and custom functions in report sources to shape data for a report.

Your report source queries can use any SQL functionality, including custom function calls, to retrieve the data your report needs. The *Employees in Department (by Tenure)* report source with data loop name `emp_t` has the following query. It calls the `TENURE_YEARS_MONTHS` function in the `EBA_DEMO_WOODSHR_REPORT` package to format the employee's tenure as a phrase like "1 year, 5 months".

```
select empno,
       ename,
       job,
       eba_demo_woodshr_report.tenure_years_months(hiredate) as tenure
  from emp
```


Copyright © 2026, Oracle and/or its affiliates.

```
 where deptno = :G_DEPTNO_FOR_REPORT
 order by hiredate
```

The source for this helper function appears below.

```
-- In package eba_demo_woodshr_report
function tenure_years_months (p_date in date)
  return varchar2
is
  l_months      number;
  l_years       number;
  l_rem_months  number;
begin
  if p_date is null then
    return null;
  end if;

  -- Whole months completed between the dates
  l_months := floor(months_between(trunc(sysdate), trunc(p_date)));

  -- If p_date is in the future, clamp to 0
  if l_months < 0 then
    l_months := 0;
  end if;

  l_years      := trunc(l_months / 12);
  l_rem_months := mod(l_months, 12);

  return  l_years || ' ' ||
          case when l_years = 1 then 'year' else 'years' end || ', ' ||
          l_rem_months || ' ' ||
          case when l_rem_months = 1 then 'month' else 'months' end;
end;
```

21.2.6 Inspecting the Report Query JSON

Inspect the generated Report Query JSON to see how data loop names and source rows are structured.

The *(Download)* button in the *Report Query* definition page lets you inspect the contents of the JSON file your report uses .It has the following format (edited slightly for brevity). Each *Data Loop Name* is a top-level JSON object property whose value is an array of JSON objects representing the source's rows. Lowercase JSON property names match the names of each report source column.

```
{
    "dep": [
        {
            "deptno": 20,
            "dname": "RESEARCH",
            "loc": "DALLAS"
        }
    ],
    "emp_t": [
        {
```


Copyright © 2026, Oracle and/or its affiliates.

```
                "empno": 7566,
                "ename": "JONES",
                "job": "MANAGER",
                "tenure": "44 years, 6 months"
            },
            ▒
            {
                "empno": 7788,
                "ename": "SCOTT",
                "job": "ANALYST",
                "tenure": "1 year, 2 months"
            }
        ],
        "emp_n": [
            {
                "empno": 7876,
                "ename": "ADAMS",
                "job": "CLERK",
                "hiredate": "12-JAN-1983"
            },
            ▒
            {
                "empno": 7788,
                "ename": "SCOTT",
                "job": "ANALYST",
                "hiredate": "14-JUL-2024"
            }
        ]
}
```

21.2.7 Printing Report from a Page

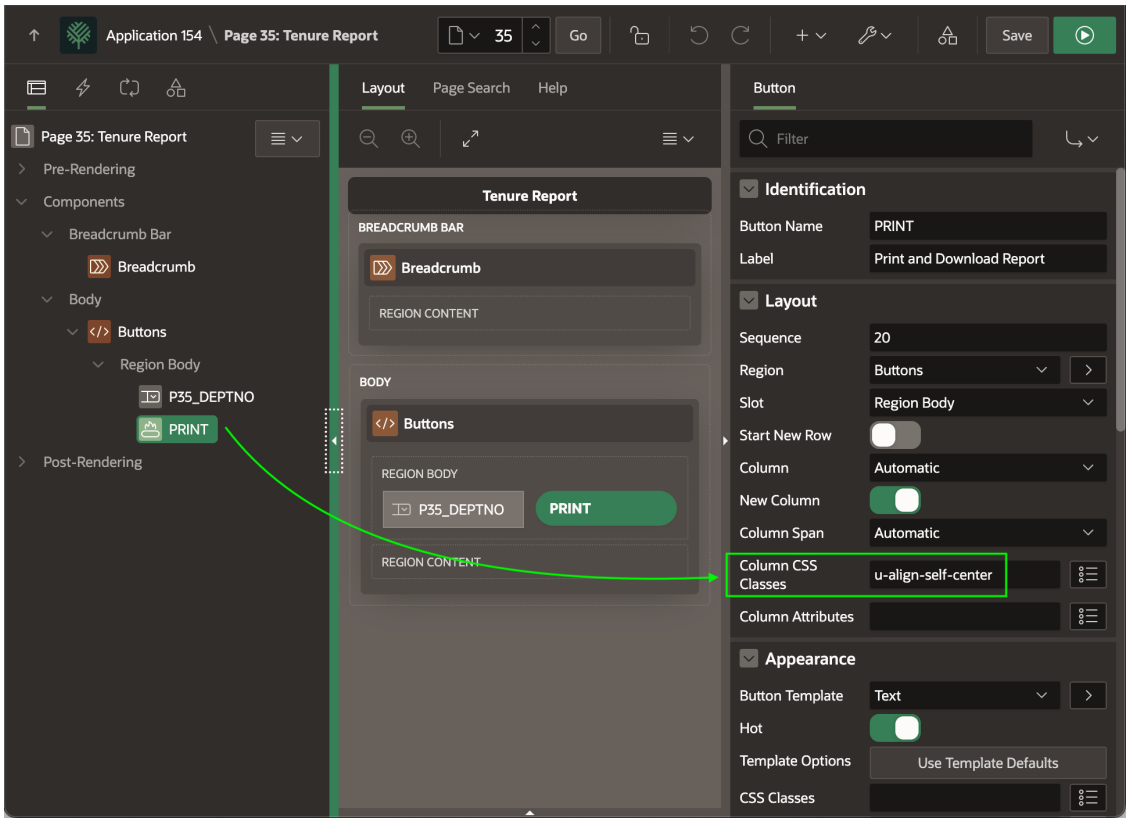
Print and download a report from a page using a button, application item, and *Print Report* process.

To print the report from a page, just configure a button to trigger its generation and download. The `PRINT` button below appears in the same row as the `P35_DEPTNO` select list HR Reps use to choose the desired department.

> **Tip**
>
> To avoid having the page item and button stretched across the full width of the page, the *Buttons* static content region containing them uses the *Buttons Container* template. To keep the styling simple, its *Style* template option is set to *Remove UI Decoration*. The `PRINT` button uses the native Universal Theme CSS class `u-align-self-center` in its *Column CSS Classes* to vertically center the button with the select list.

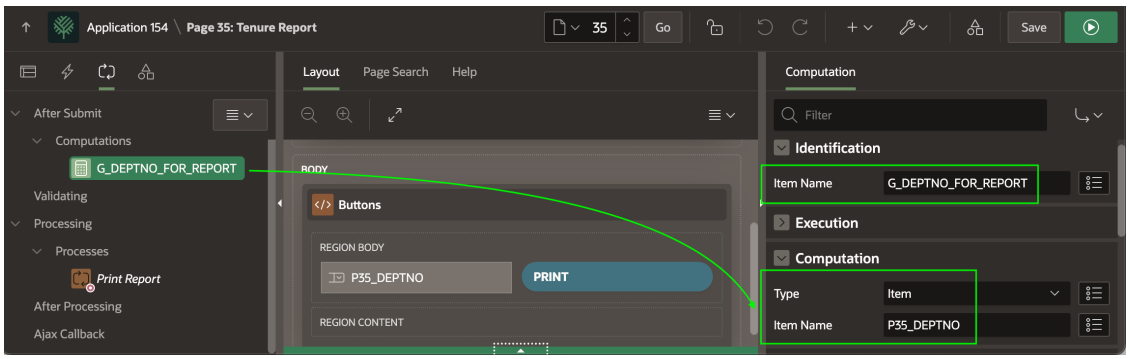
Copyright © 2026, Oracle and/or its affiliates.

Figure 21-7 Using CSS Class to Vertically Center Button with Page Items in the Same Row



When an HR Rep selects a department and clicks the *(Print and Download Report)* button, it submits the page. As shown below, an *After Submit* computation sets the value of the `G_DEPTNO_FOR_REPORT` application item to the value of the `P35_DEPTNO` select list containing the user's selected department number. This ensures the report source queries have the right value when they reference this application item as a bind variable.

Figure 21-8 Setting Application Item to Selected Department Number



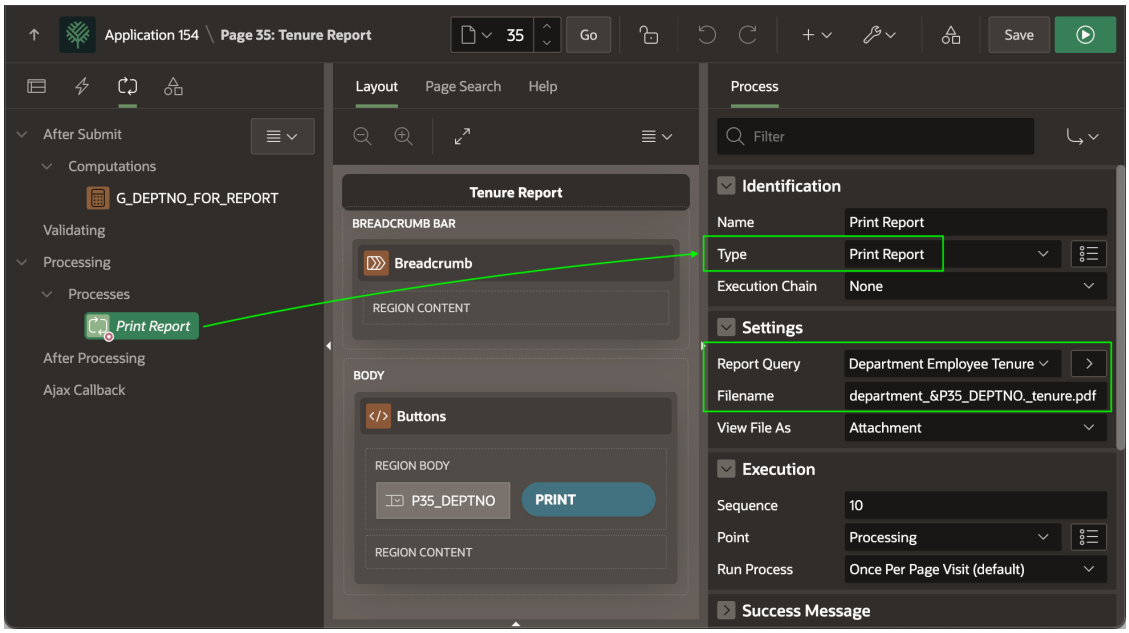
With the application item value assigned, the *Print Report* page process generates and downloads the pixel-perfect PDF using the native *Print Report* process. It configures its *Report Query* property to use the *Department Employee Tenure* definition. Notice the *Filename* uses a substitution `&P35_DEPTNO.` as part of the PDF file name to download. A similar native

Copyright © 2026, Oracle and/or its affiliates.

Print Report dynamic action is also available. It does the same job, just triggered from a page *event* instead of a page submit.

> **Tip**
>
> Use a *Print Report* page process either in the *Pre-Rendering* section, or the *Processing* section of your page. In the latter case, also set your page's *Reload on Submit* property to *Always*.

Figure 21-9 Printing Tenure Report Using Report Query



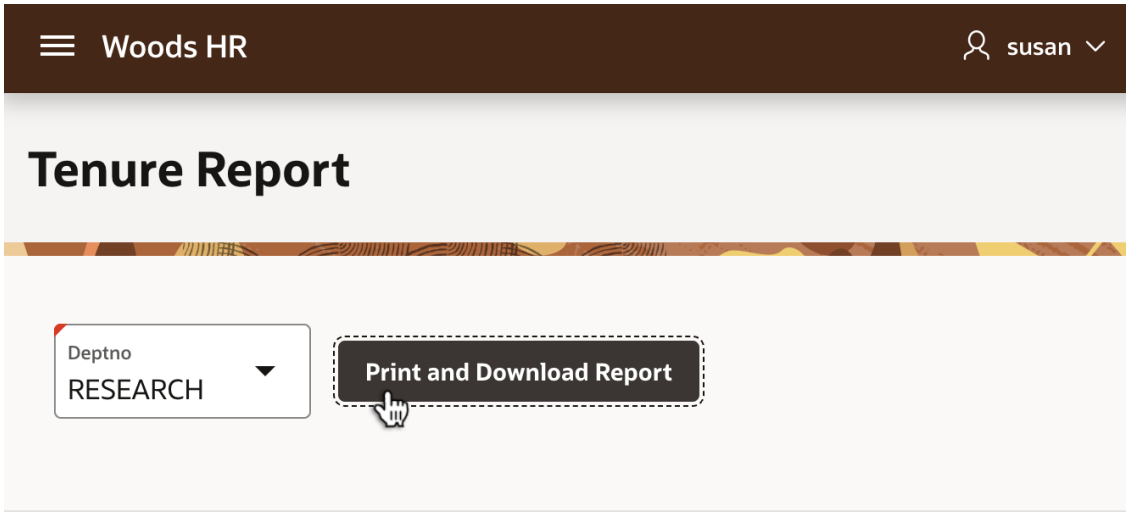
21.2.8 Experiencing Report Generation as a User

Follow the user flow from choosing report data to downloading the generated PDF.

Susan visits the *Tenure Report* page visible only to HR Representatives, selects the *RESEARCH* department in the list, and clicks *(Print and Download Report)*.

Copyright © 2026, Oracle and/or its affiliates.

Figure 21-10 HR Rep Susan Prints Tenure Report for Research Department



The APEX engine:

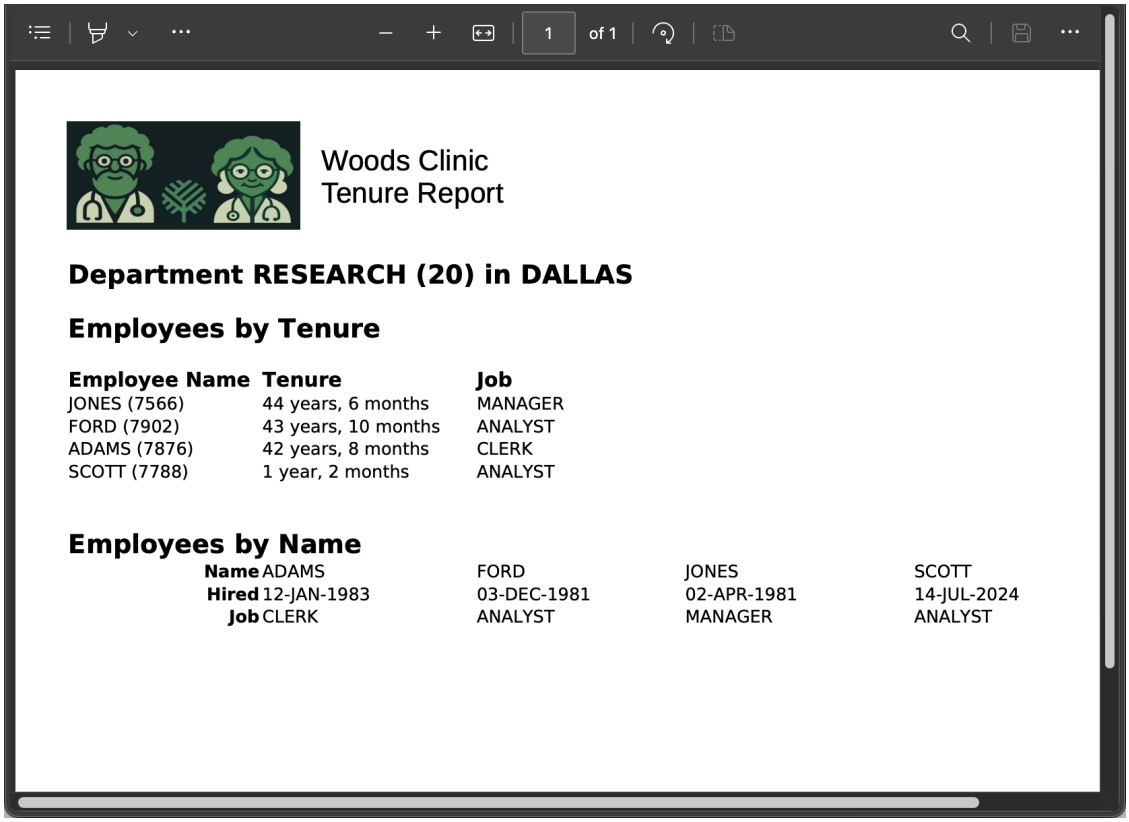
- executes all the report source queries in the report query definition,
- formats their results in a JSON file,
- sends it along with the associated layout to the remote print server, and
- downloads the resulting PDF file.

Susan opens the PDF to view the finished report shown below.

The figure shows the PDF *Woods Clinic Tenure Report* for the RESEARCH department that Susan requested. It includes an *Employees by Tenure* section with employees listed in a tabular style, one per row. It also includes an *Employees by Name* section, with employee information one per column across the page.

Copyright © 2026, Oracle and/or its affiliates.

Figure 21-11 Tenure Report PDF Result for Research Department



21.3 Customizing Reports with Word Template

Use a *Report Layout* to customize the PDF download format for an Interactive Report or Classic Report region.

- Formatting Data in Word with Tags
  Format a Word report template with tags that map a report region's data into the layout.
- Setting Layout Data Loop Name
  Set the layout's Data Loop Name to match the short name used in the report template.
- Overriding PDF Layout of Report Regions
  Override a report region's default PDF output by assigning a custom Report Layout.
- Downloading Customized Report PDF
  Download a customized PDF that reflects the Interactive Report's current filters and sort order.

21.3.1 Formatting Data in Word with Tags

Format a Word report template with tags that map a report region's data into the layout.

Use Microsoft Word or Excel to customize the PDF download for the Interactive Report on the *Employee Directory* page from Shaping Experience with Rules and Roles. For this report, you choose a Word document.

Copyright © 2026, Oracle and/or its affiliates.

Here, you use Oracle Document Generator tags in a Microsoft Word layout document template, just as you did in an Excel Spreadsheet. For the *Employee Directory* layout, you need data like the following, to which you associate a convenient short name:

- Employee directory info (short name `dir`)
  - with fields: `ename`, `job`, `mgr`, `deptno`

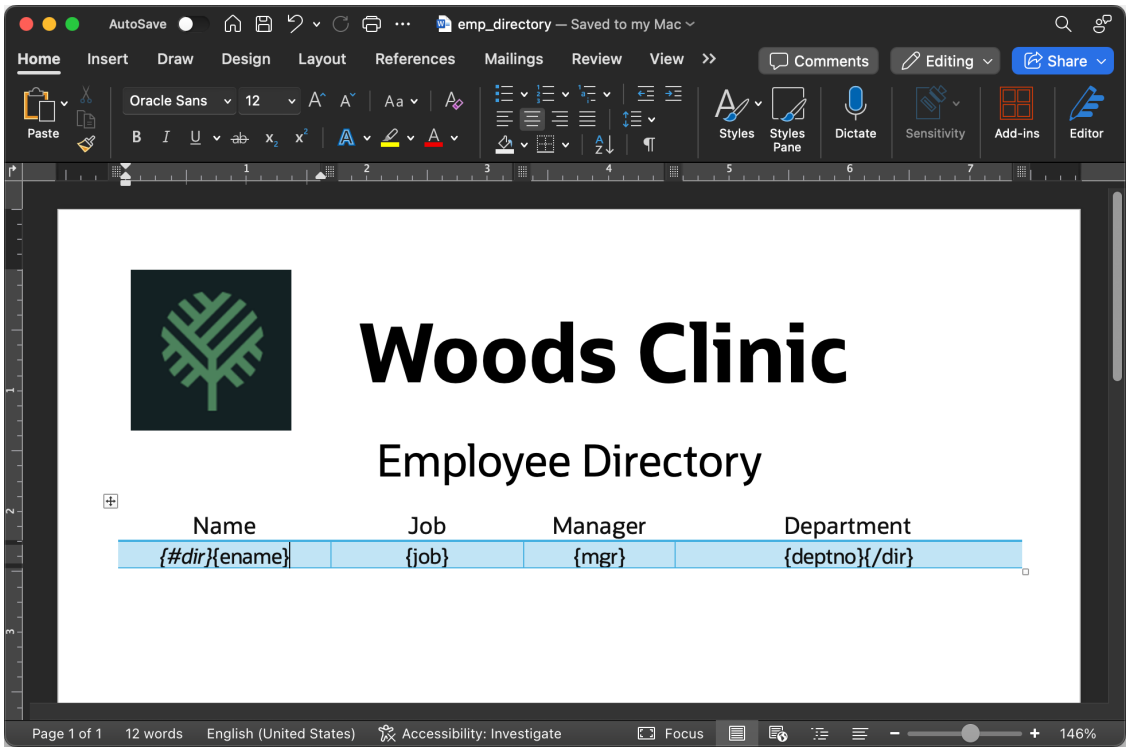
In your document template, reference the short name of your data source using Document Generator tags {*#shortname*} and {*/shortname*} to surround {*fieldname*} tags that reference the data source field values by name.

> **Tip**
>
> Field tag names are the lowercase column names of the report region associated with the layout.

As shown below, the Word document includes a formatted table, with a single row containing the four fields of the associated region's data source. It references the `dir` short name and field names using the following tags. The vertical bars below represent table cell boundaries.

`{#dir}{ename}  |  {job}  |  {mgr}  |  {deptno}{/dir}`

Figure 21-12 Microsoft Word Layout Document for Employee Directory



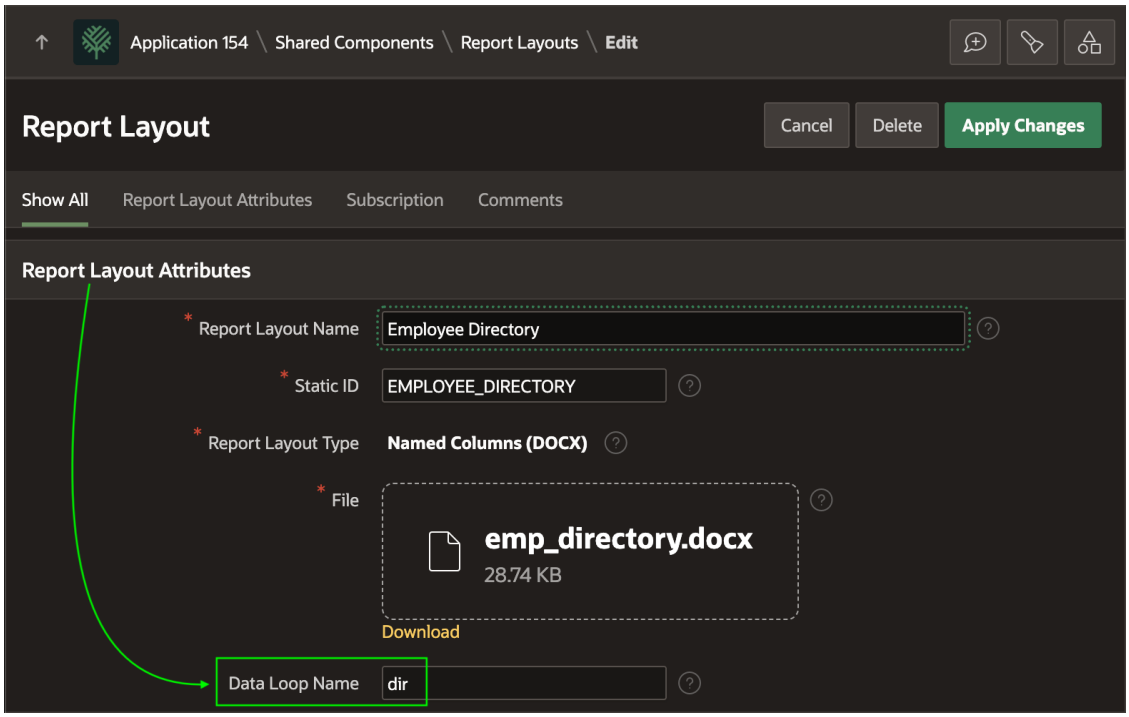
21.3.2 Setting Layout Data Loop Name

Set the layout's Data Loop Name to match the short name used in the report template.

Copyright © 2026, Oracle and/or its affiliates.

While each *Report Query* source specifies its own *Data Loop Name*, when pairing a layout with a report region, set its short name in the layout definition itself. As shown below, the *Employee Directory* layout specifies `dir` as its *Data Loop Name* to match the tags used in the template. It also includes the `emp_directory.docx` Word document as the template file.

Figure 21-13 Specifying Layout Data Loop Name to Match Layout Tags



21.3.3 Overriding PDF Layout of Report Regions

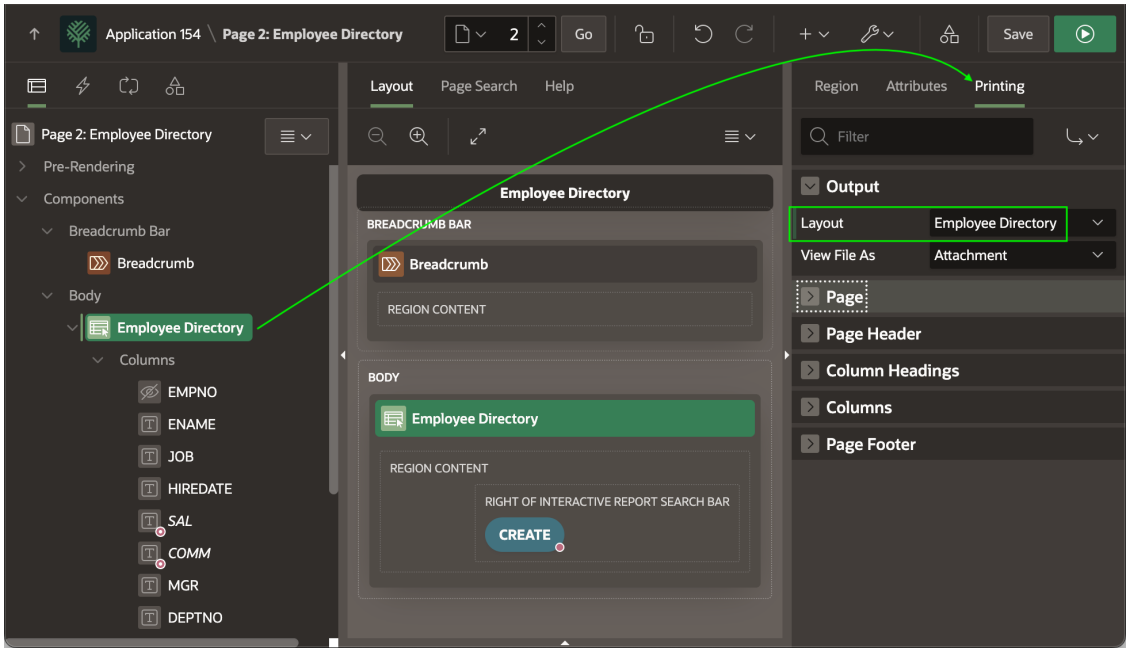
Override a report region's default PDF output by assigning a custom Report Layout.

In the *Employee Directory* page, select the Interactive Report region and the *Printing* tab in the Property Editor. There, as shown below, you can select the *Employee Directory* layout to override the default report PDF formatting.

> **Tip**
>
> To associate a Report Layout with a Classic Report, ensure the region's *Printing* switch is enabled on the *Attributes* tab in the Property Editor. Then on the *Printing* tab that appears, set the *Layout*.

Copyright © 2026, Oracle and/or its affiliates.

Figure 21-14 Configuring Interactive Report PDF Printing to Use Custom Layout



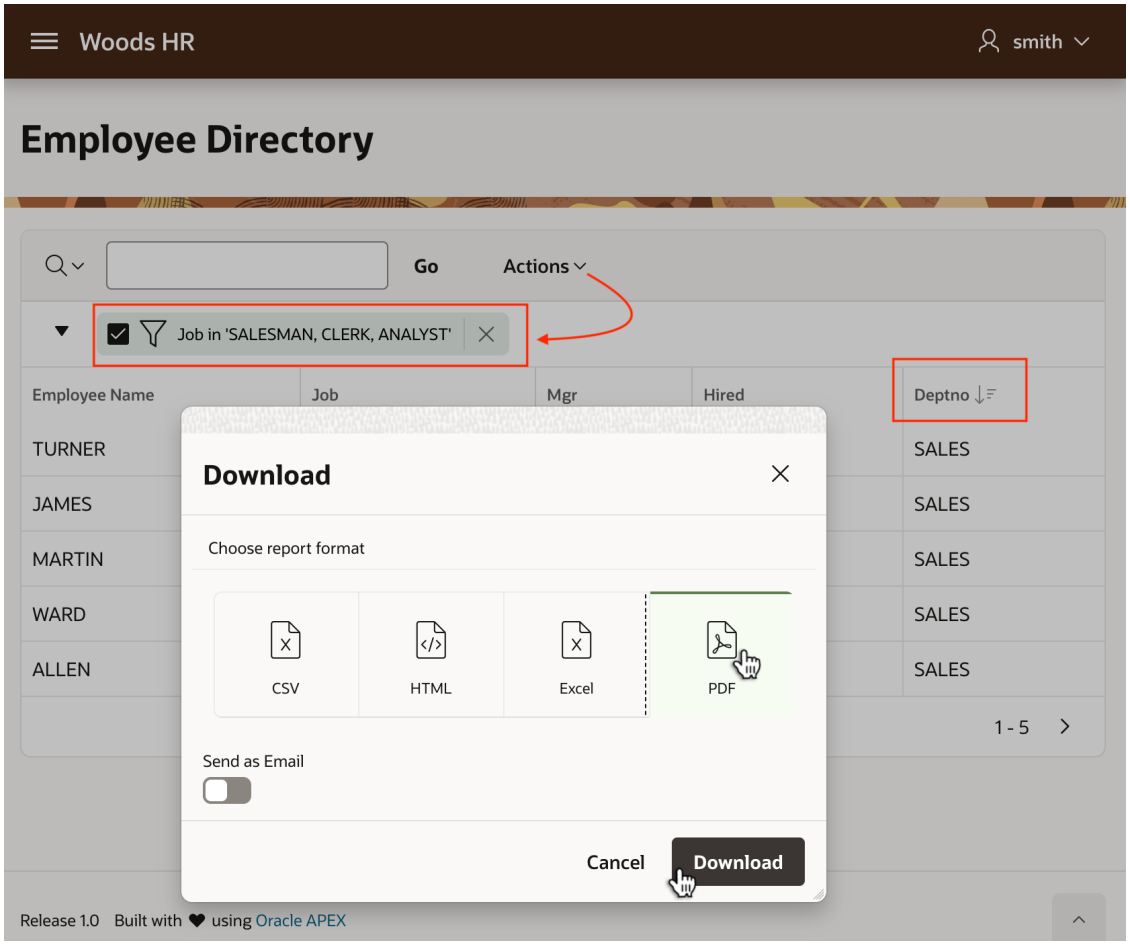
21.3.4 Downloading Customized Report PDF

Download a customized PDF that reflects the Interactive Report's current filters and sort order.

Now when a user chooses *Download* from the Interactive Report's *Actions* menu, if they choose the PDF report format they get a pixel-perfect Employee Directory. Notice below that the user uses features like filtering on the *Job* column and sorting on the *Deptno* column before performing the PDF download. This affects *which* rows appear and *how* they are sorted in the downloaded report, too.

Copyright © 2026, Oracle and/or its affiliates.

Figure 21-15 Employee Directory Interactive Report PDF Print Uses Custom Layout

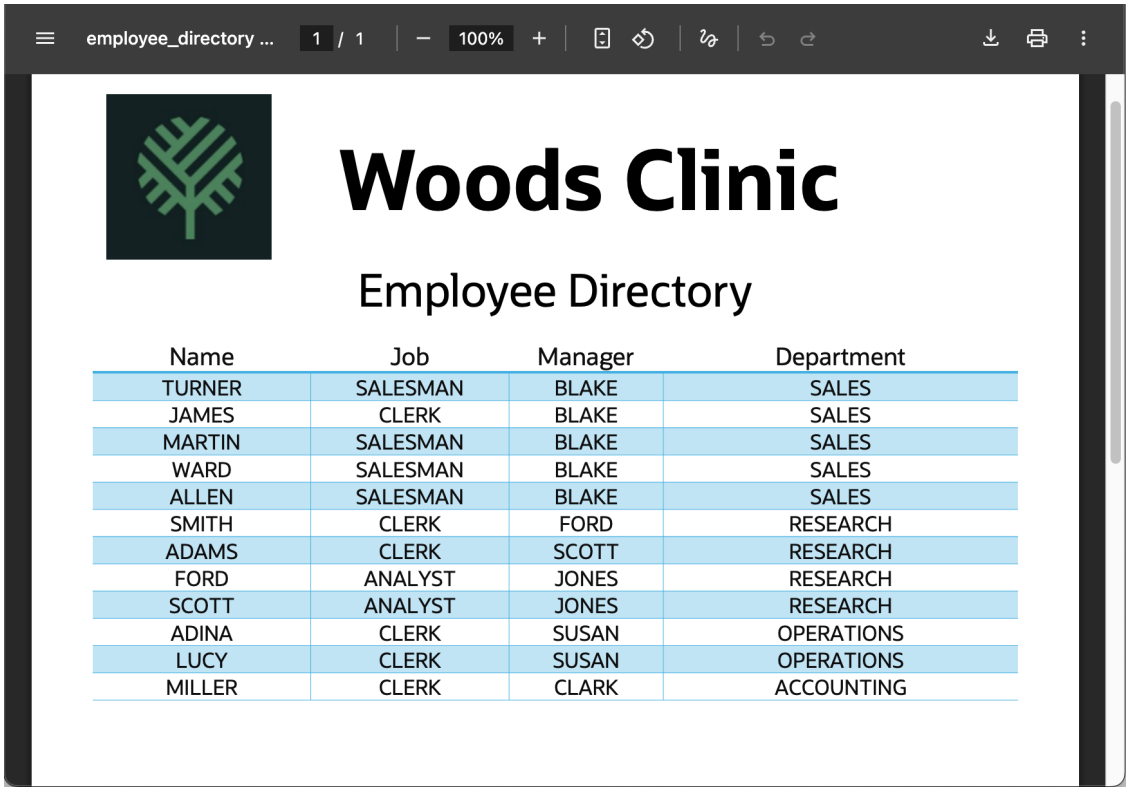


When the user opens the PDF file, they see the filtered, sorted *Employee Directory* contents presented now in your pixel-perfect layout.

> **⚠ Caution**
>
> If a user removes a column from the Interactive Report, that column is also omitted from the JSON data the report layout receives. Make sure users understand not to remove columns that the PDF layout depends on, or some areas of the generated report may appear blank.

Copyright © 2026, Oracle and/or its affiliates.

Figure 21-16 PDF Download from Employee Directory Interactive Report



Copyright © 2026, Oracle and/or its affiliates.

22

Translating Your Application

Translate your app with Text Messages, Application Translations, bulk exports, and App Builder editing tools.

Use *Text Messages* and *Application Translations* to deliver your app in multiple languages. You can synchronize app translations at any time to replace inline text with references to Text Message components. While building pages, you can use existing messages easily in Page Designer with the *Text Messages Picker*. When ready, translate strings by exporting and importing them in bulk using standard formats. You can also edit translations in App Builder directly in a language-filterable grid.

- Exploring Woods HR Pages to Translate
  Review the Woods HR pages you will translate from the app's primary language.
- Enabling the First Translation Language
  Enable text message-based translation and add the first translated language for your app.
- Understanding Initial Text Message Sync
  Understand how the first translation sync replaces inline text with Text Message references.
- Exporting and Importing Translations
  Export and import XLIFF files to translate Text Messages in bulk.
- Determining User's Preferred Language
  Choose how your app determines each user's preferred language.
- Experiencing App in Another Language
  View the translated app experience driven by the user's language preference.
- Translating Text as Your App Evolves
  Add new text directly or with Text Messages, then sync before the next bulk translation cycle.
- Synchronizing Text Messages Anytime
  Sync Text Messages to prepare newly added UI text for the next translation update.
- Editing Translations in a Filterable Grid
  Edit translations directly in a filterable grid for small in-house updates.
- Centralizing Reusable UI Text
  Centralize reusable UI text by converting it to Text Messages manually or in bulk.

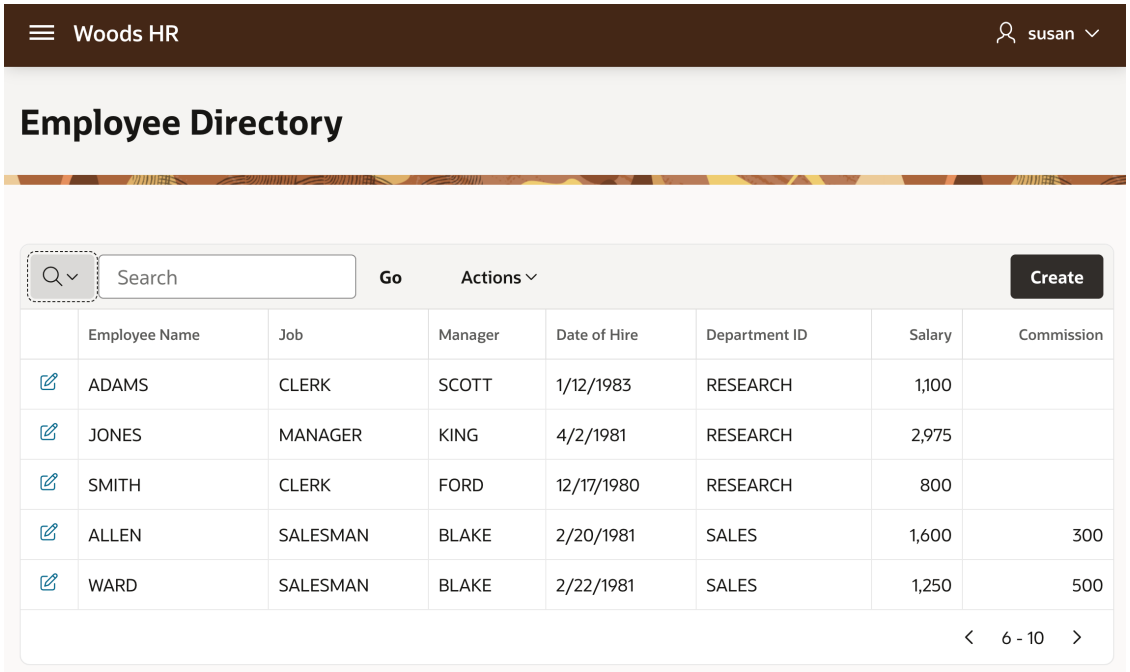
22.1 Exploring Woods HR Pages to Translate

Review the Woods HR pages you will translate from the app's primary language.

You create an app in a *primary* language of your choice. The *Woods HR* app uses *English*. For simplicity, it contains only the *Employee Directory* page shown below with an Interactive Report region and related form page. However, translations work the same for apps of any size.

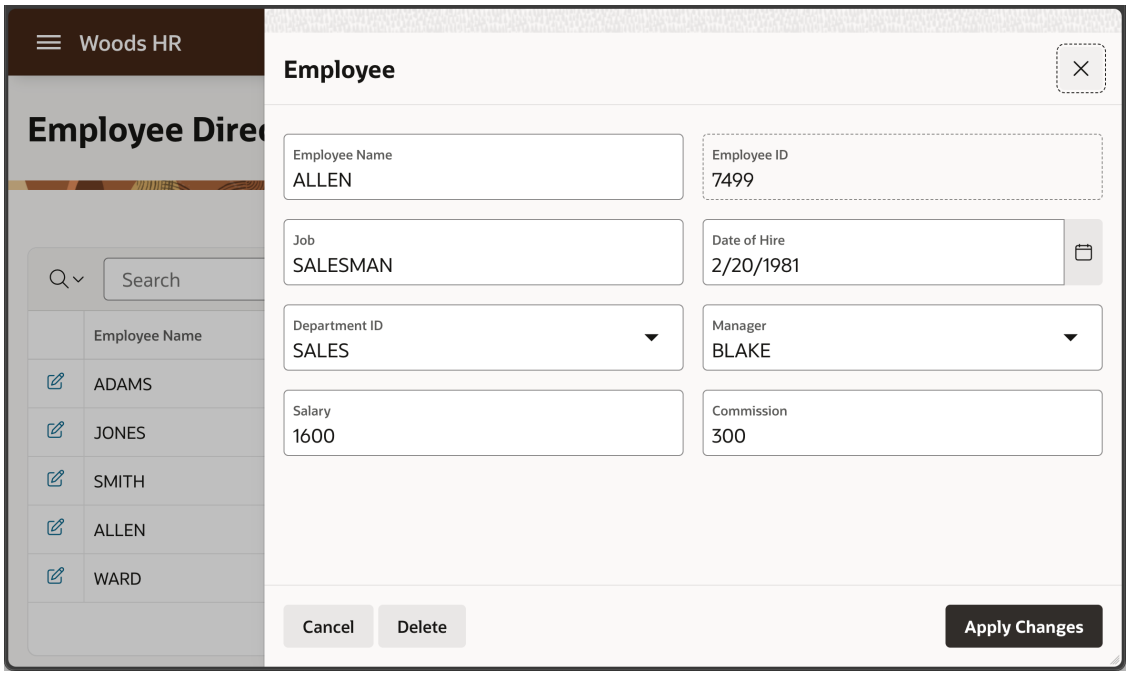
Copyright © 2026, Oracle and/or its affiliates.

Figure 22-1 Employee Directory Page with an Interactive Report in Primary Language



The companion *Employee* modal drawer page appears below. HR reps access it from the directory page to view and edit employee details.

Figure 22-2 Employee Details Form Page for HR Representatives Only in Primary Language



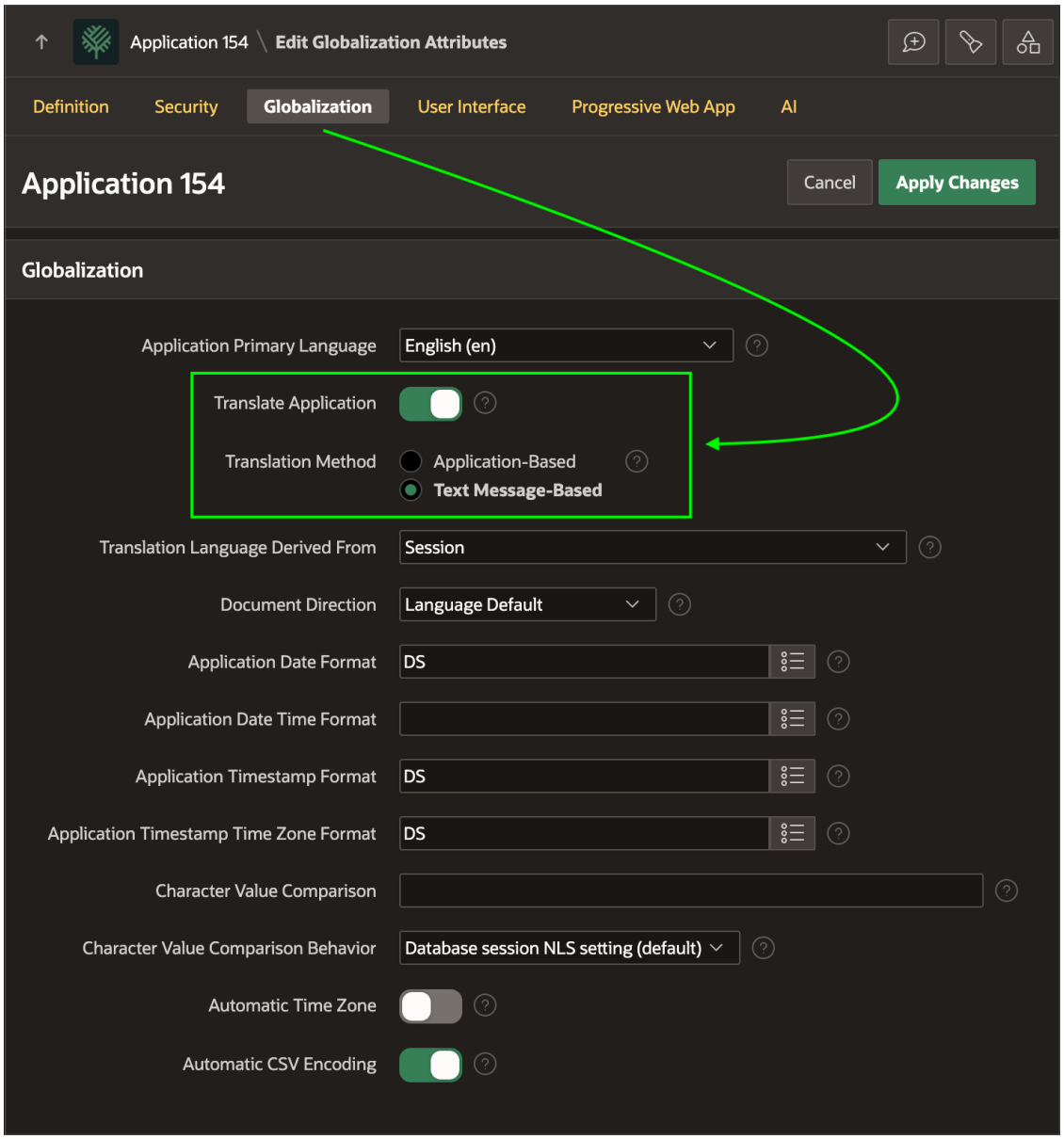
Copyright © 2026, Oracle and/or its affiliates.

22.2 Enabling the First Translation Language

Enable text message-based translation and add the first translated language for your app.

When you decide to add the first *translated* language, go to *Shared Components* > *Globalization Attributes* and enable the *Translate Application* switch as shown below. Use the *Text Message-Based* option. It's the simplest approach with best support in App Builder and Page Designer.

Figure 22-3 Enabling Application Translation in Globalization Settings



Next, go to *Shared Components* > *Application Translations* and click *(Add Language)*. For example, if some of your end users need to use your app in Italian, you can add the *Italian (Italy) (it)* language as shown below.

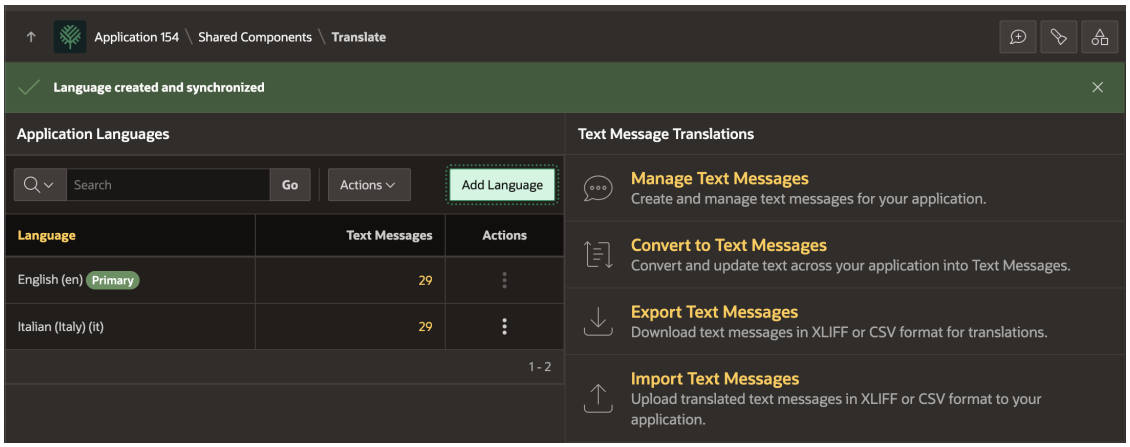
Copyright © 2026, Oracle and/or its affiliates.

Figure 22-4 Adding the First Translation Language



The new language appears along with your application's primary language in the list of language translations your app now supports.

Figure 22-5 Primary Language and First Translated Language Created and Synchronized



22.3 Understanding Initial Text Message Sync

Understand how the first translation sync replaces inline text with Text Message references.

When adding the *first* app translation language, App Builder automatically creates a Text Message component for each unique translatable string it finds in all your pages. It also *updates* every place it found translatable text *inline* to instead reference a new corresponding text message. Lastly, it creates a mirrored entry for every text message in the new target language. This process is known as *synchronizing* your application's translatable text. It happens *automatically* when you add a language. You can also trigger it on-demand later when needed.

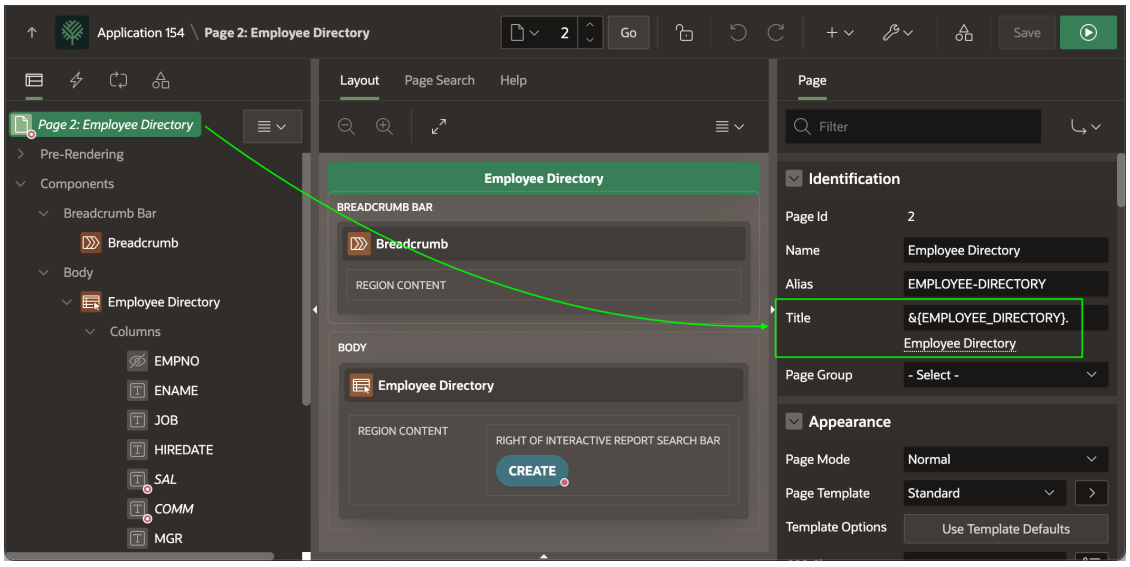
For example, the *Employee Directory* page's *Title* property started as the simple, inline text "Employee Directory". As shown below, after adding the first translatable language the *Title*

Copyright © 2026, Oracle and/or its affiliates.

property now references a new `EMPLOYEE_DIRECTORY` text message name using the syntax below. Page Designer shows the translated text "Employee Directory" just below in the Property Editor.

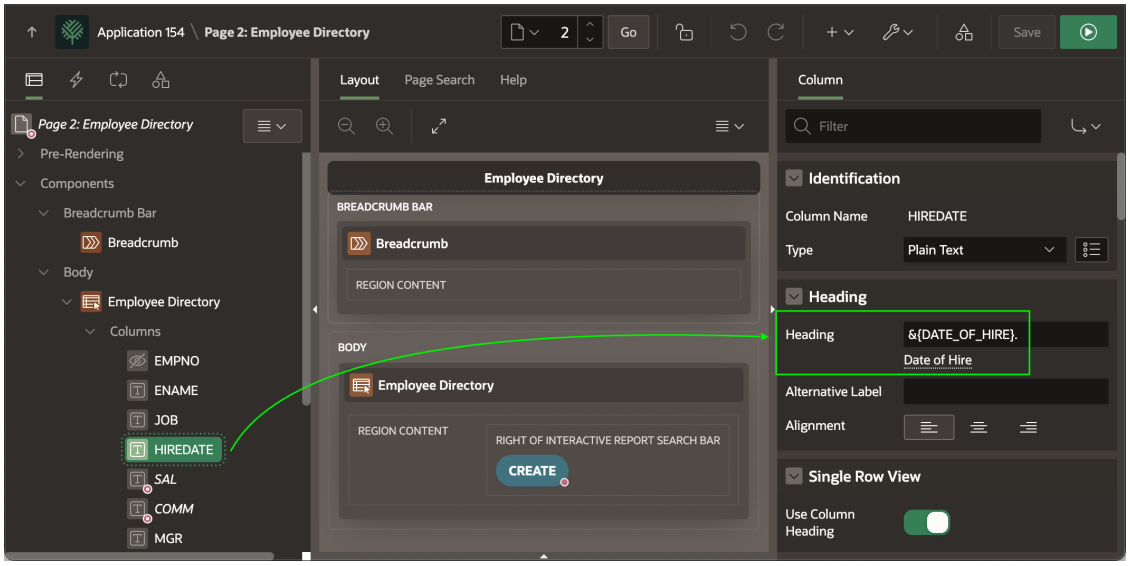
```
&{EMPLOYEE_DIRECTORY}.
```

Figure 22-6 Page Title Changed to Reference Translatable Text Message by Name



Similarly, column headings on this page like `HIREDATE`'s undergo the same transformation.

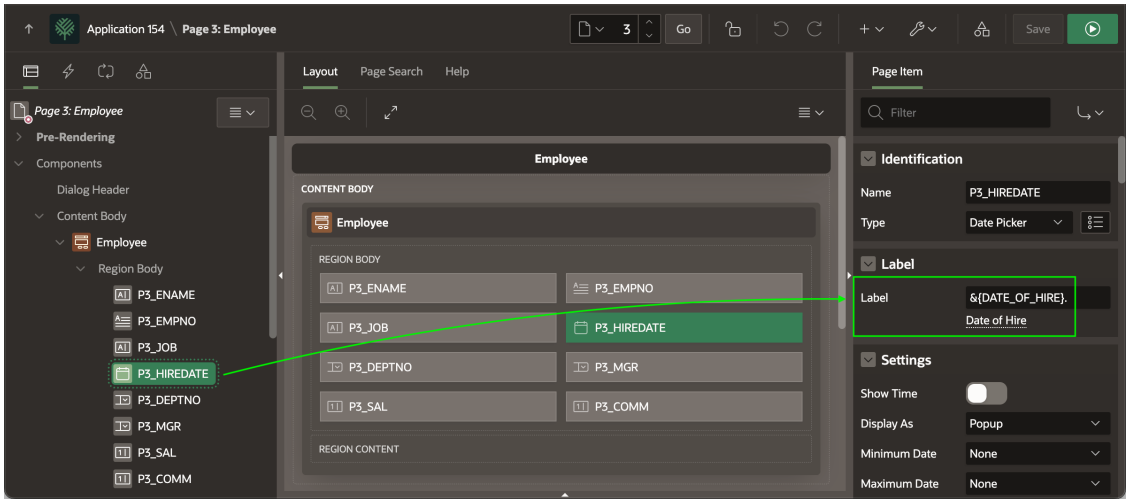
Figure 22-7 All Translatable Text Undergoes Same Transformation



If the exact same text appears in *multiple* pages, all usages get updated to reference the *same* Text Message component. For example, the `P3_HIREDATE` page item on the *Employee* page now also uses the same `DATE_OF_HIRE` text message as shown below.

Copyright © 2026, Oracle and/or its affiliates.

Figure 22-8 Identical Text References the Same Translatable Text Message

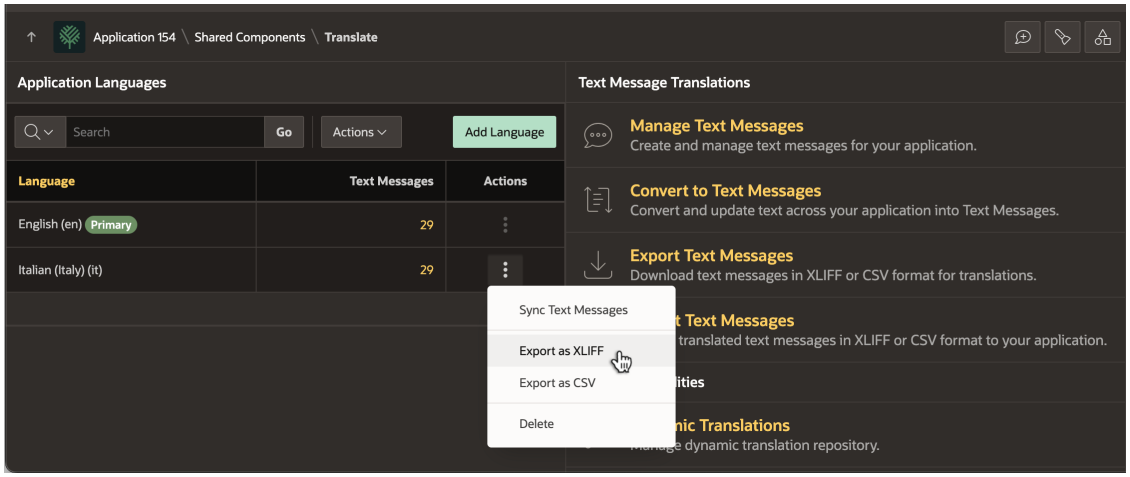


22.4 Exporting and Importing Translations

Export and import XLIFF files to translate Text Messages in bulk.

You *could* edit the Italian version of each message one by one to translate it. However, usually translation happens in bulk using industry-standard XML Localization Interchange Format (XLIFF) files. To export the Italian messages for translation in XLIFF format, use the *Export as XLIFF* option on the three vertical dots menu for that application language on the *Application Translations* page.

Figure 22-9 Exporting all Text Messages for a Language to an XLIFF File



The name of the XLIFF file contains the application id, the source language and the target language. For example, for the *Woods HR* app with ID 154, the file for the English source and Italian target languages is named `f154_en_it.xlf`. It contains an XML format like the following

Copyright © 2026, Oracle and/or its affiliates.

with source and target strings. Before translation, each target language string has the same value as its corresponding source.

```
<?xml version="1.0"?>
<xliff version="1.0">
  <file original="f154_en_it.xlf"
        source-language="en"
        target-language="it"
        datatype="html">
    <trans-unit id="EMPLOYEE_DIRECTORY">
      <source>Employee Directory</source>
      <target>Employee Directory</target>
    </trans-unit>
    <trans-unit id="DATE_OF_HIRE">
      <source>Date of Hire</source>
      <target>Date of Hire</target>
    </trans-unit>
     ▒
  </file>
</xliff>
```

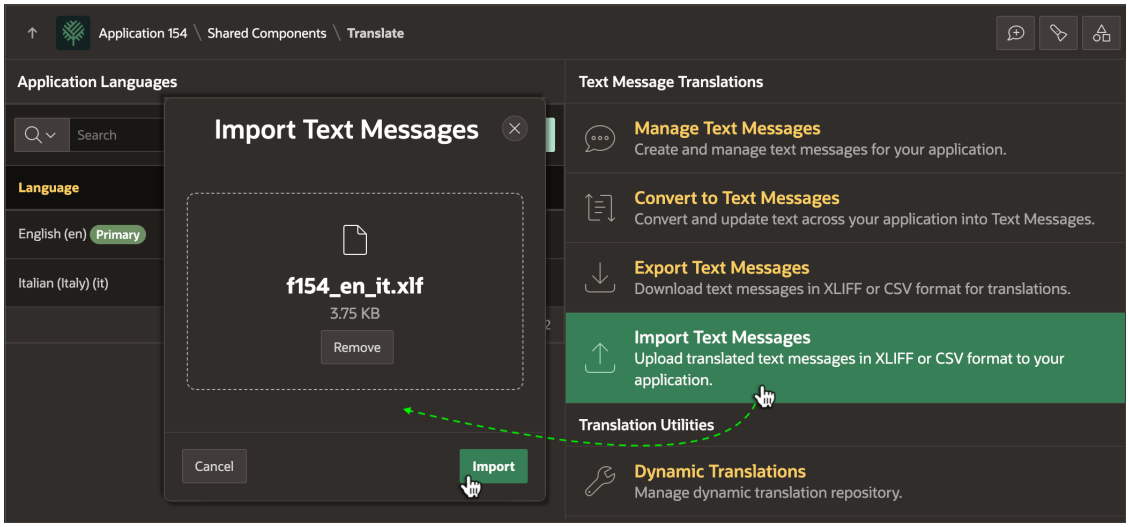
You would typically provide this file to a translation service. They return a file with the `<target>` element text updated with Italian language translations like this:

```
<?xml version="1.0"?>
<xliff version="1.0">
  <file original="f154_en_it.xlf"
        source-language="en"
        target-language="it"
        datatype="html">
    <trans-unit id="EMPLOYEE_DIRECTORY">
      <source>Employee Directory</source>
      <target>Elenco dipendenti</target>
    </trans-unit>
    <trans-unit id="DATE_OF_HIRE">
      <source>Date of Hire</source>
      <target>Data di assunzione</target>
    </trans-unit>
     ▒
  </file>
</xliff>
```

To apply the Italian language translation to your app, go to the *Application Translations* page. As shown below, click *Import Text Messages* and supply the XLIFF file name to import.

Copyright © 2026, Oracle and/or its affiliates.

Figure 22-10 Importing Translated Strings from XLIFF File



22.5 Determining User's Preferred Language

Choose how your app determines each user's preferred language.

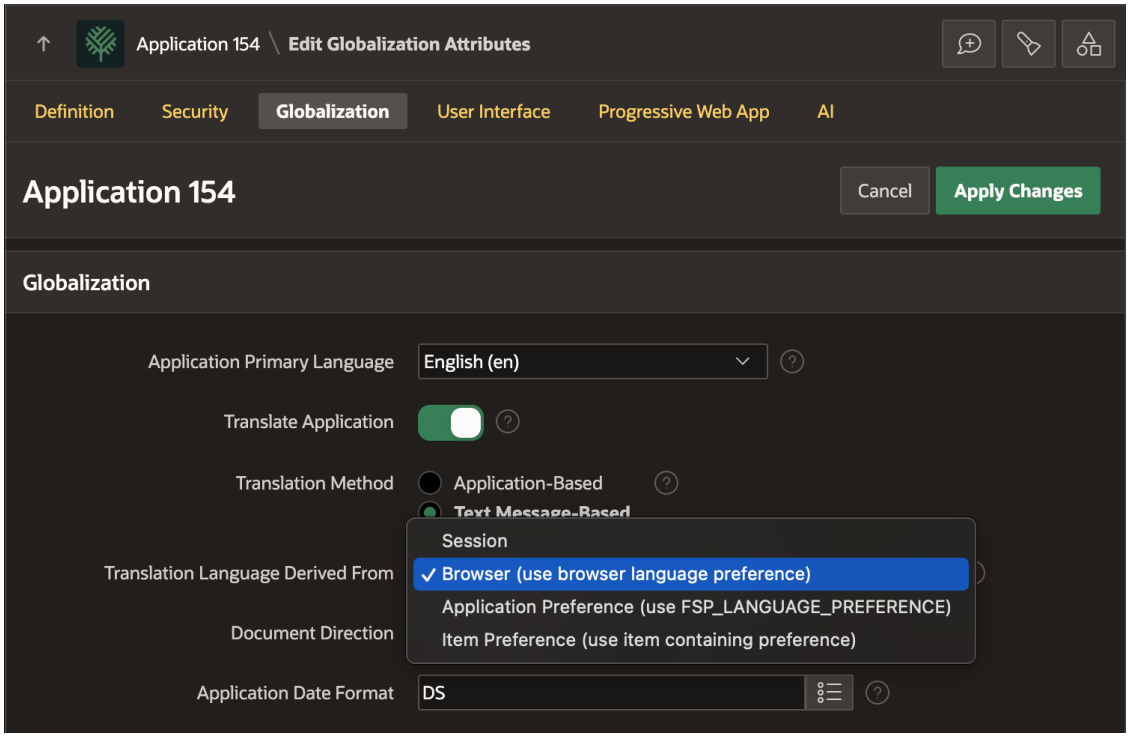
In your application's *Globalization Attributes*, choose how it determines the current user's language. There are four options:

- *Session* language set by `APEX_UTIL.SET_SESSION_LANG` or `P_LANG` app URL parameter
- Web *browser* language preference
- User *preference* `FSP_LANGUAGE_PREFERENCE` set with `APEX_UTIL.SET_PREFERENCE`
- *Application item* `FSP_LANGUAGE_PREFERENCE`

For example, as shown below, selecting *Browser* infers the language to use based on the user's browser language settings.

Copyright © 2026, Oracle and/or its affiliates.

Figure 22-11 Configuring How Your App Determines the Current User's Language



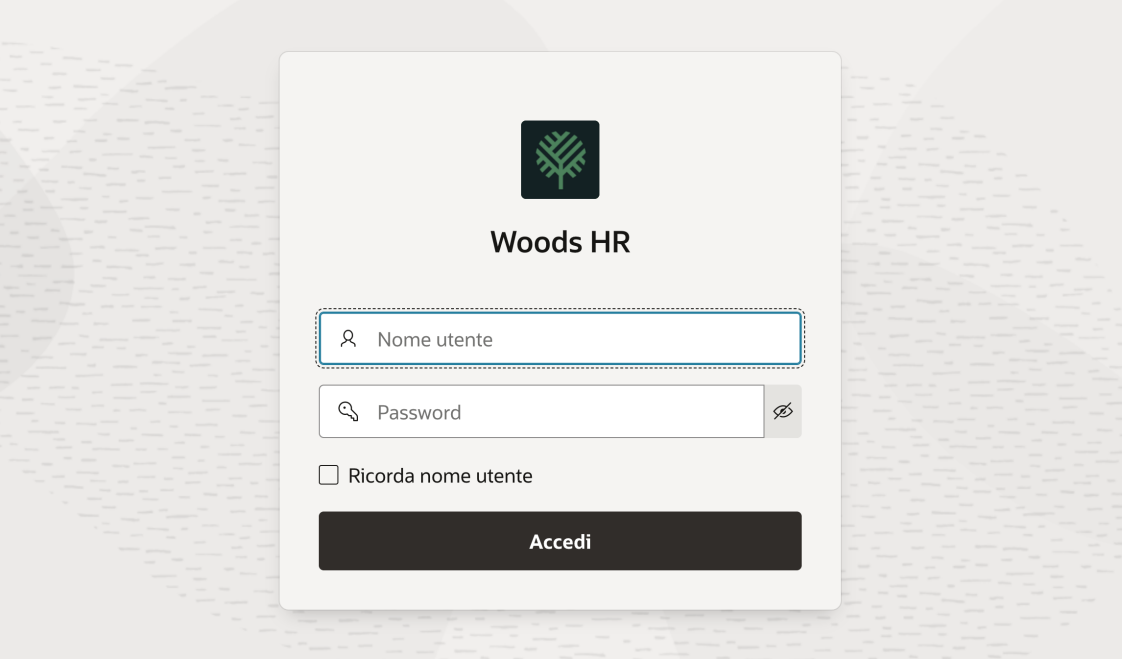
22.6 Experiencing App in Another Language

View the translated app experience driven by the user's language preference.

If Susan the HR Rep logs into the app with her browser set to prefer Italian, she sees the login page in Italian.

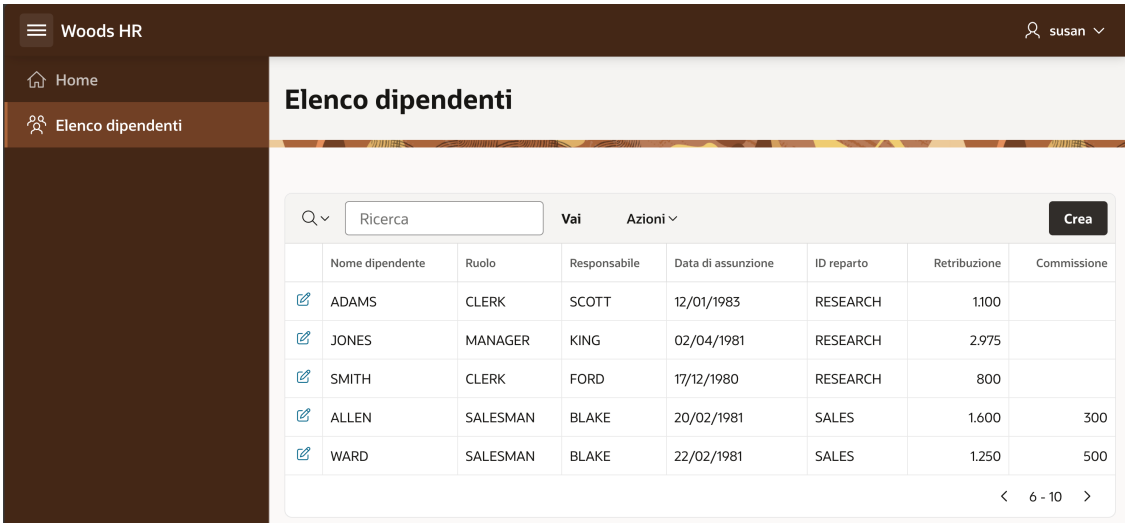
Copyright © 2026, Oracle and/or its affiliates.

Figure 22-12 User's Italian Browser Language Preference Drives App Language



When she accesses the navigation menu and visits the *Employee Directory* page, everything is now in Italian as well.

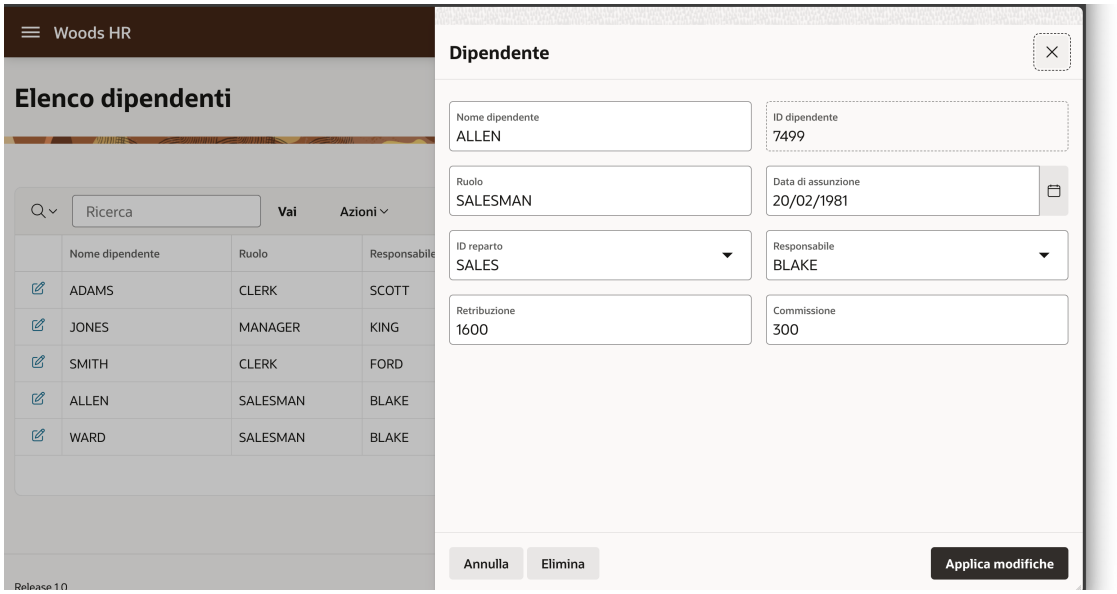
Figure 22-13 Navigation Menu and Employee Directory Use Italian Translations



When she edits an employee like *ALLEN* she also sees the *Employee* details page in Italian.

Copyright © 2026, Oracle and/or its affiliates.

Figure 22-14 Employee Details Modal Drawer Uses Consistent Italian Translations



22.7 Translating Text as Your App Evolves

Add new text directly or with Text Messages, then sync before the next bulk translation cycle.

As you continue evolving your app, you have two equally valid options for creating new translatable text:

- Create a Text Message component, then reference it in a page, or
- Enter text inline as you add new pages or page elements.

Then, at any time, you can *Sync Text Messages* to automatically prepare all new text for translation.

- Adding a New Text Message
  Create the Text Message first to choose a meaningful key for new translatable UI text.
- Using the Text Messages Picker
  Pick a Text Message reference for translatable page properties.
- Entering New Text Directly Inline
  Enter new UI text inline and let the next sync create its Text Message automatically.

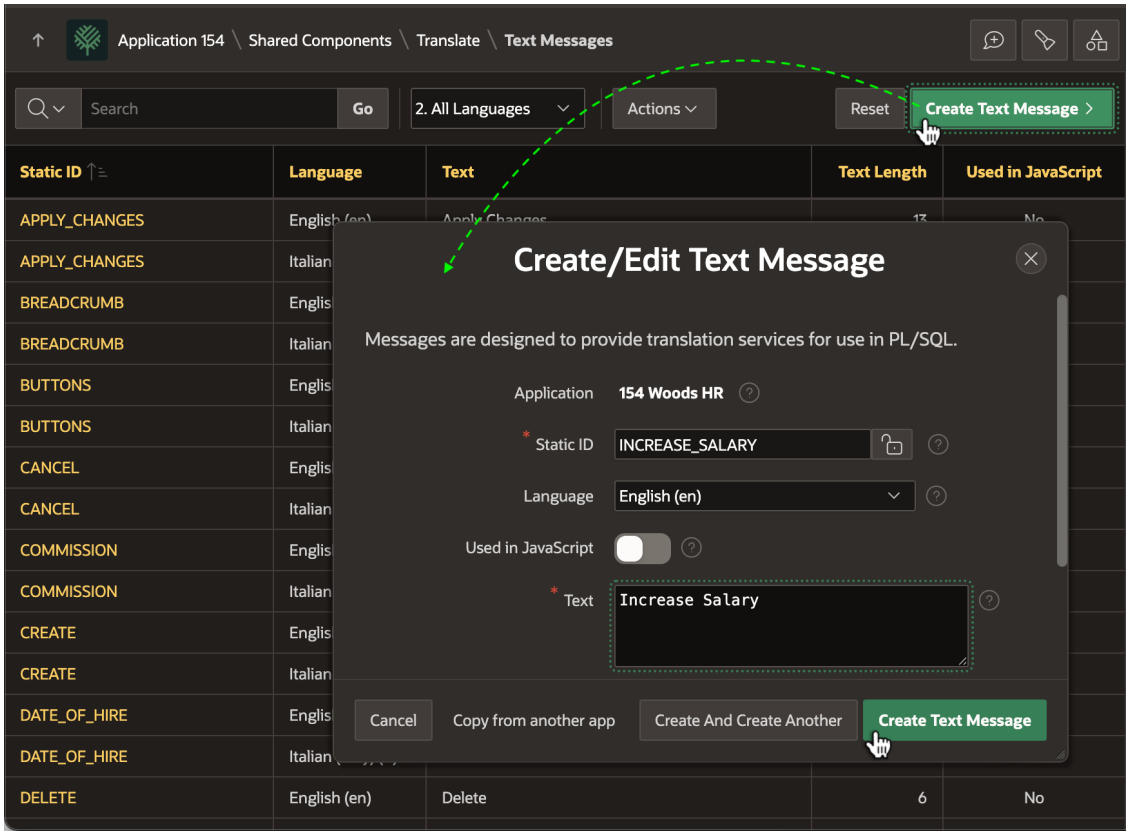
22.7.1 Adding a New Text Message

Create the Text Message first to choose a meaningful key for new translatable UI text.

Imagine you are asked to add a new *(Increase Salary)* button to the *Employee* page. You can start by creating a new Text Message `INCREASE_SALARY` as shown below with the text *Increase Salary* in the primary language.

Copyright © 2026, Oracle and/or its affiliates.

Figure 22-15 Creating New Text Message in Primary Language



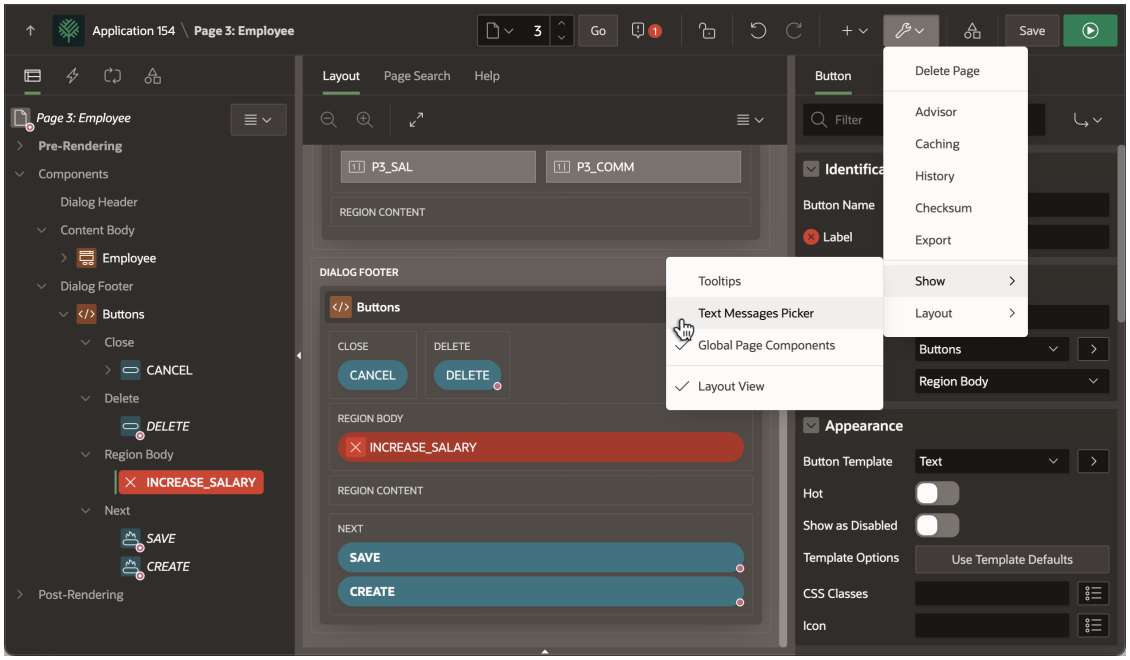
22.7.2 Using the Text Messages Picker

Pick a Text Message reference for translatable page properties.

You can use the `INCREASE_SALARY` text message as a new button's label in the *Employee* page. Start by ensuring the *Text Messages Picker* is enabled. It's a button with a globe icon that appears next to every property containing translatable text. If you *don't* see the globe buttons, as shown below, enable them using the wrench toolbar dropdown menu option *Show > Text Messages Picker*.

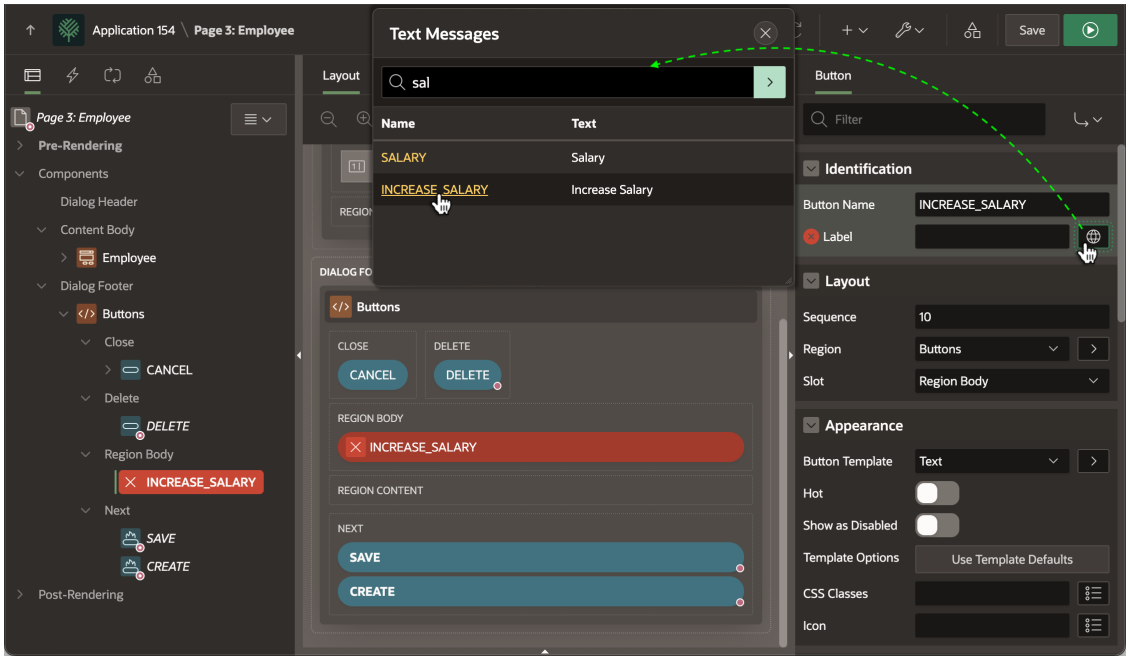
Copyright © 2026, Oracle and/or its affiliates.

Figure 22-16 Enabling Text Messages Picker



Once enabled, you can click the globe icon as shown below to search for and select the text message you want to use.

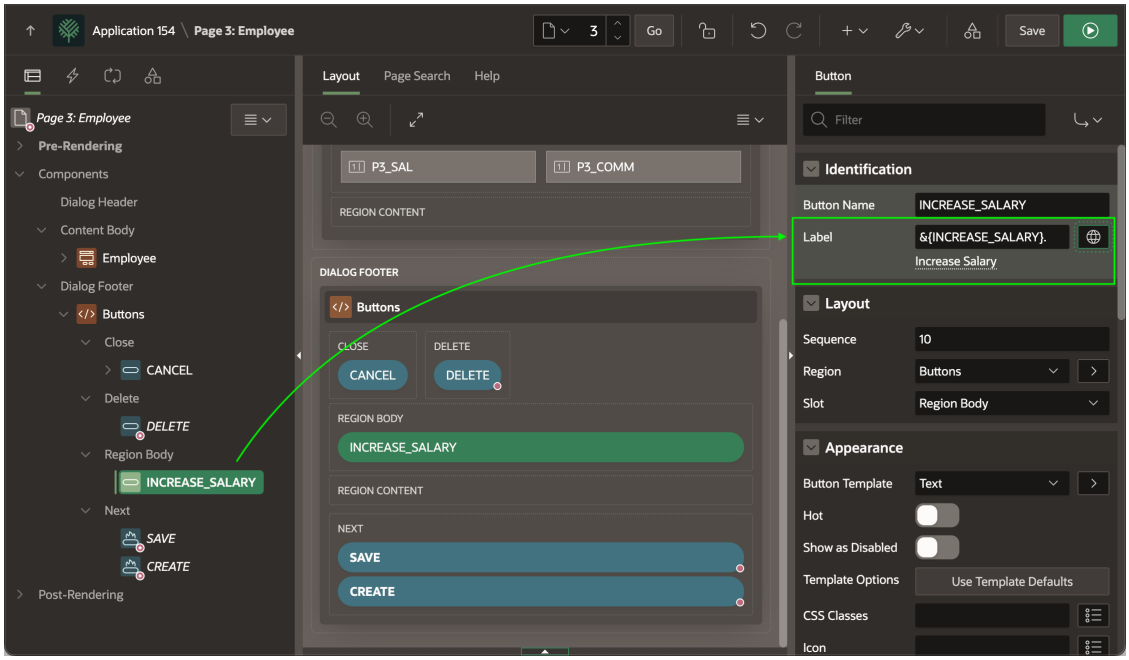
Figure 22-17 Choosing Message in Text Message Picker Opened with the Globe Button



The *Text Messages Picker* automatically includes the selected message using the appropriate substitution syntax as shown below and the translated string appears just below: *Increase Salary*.

Copyright © 2026, Oracle and/or its affiliates.

Figure 22-18 Picked Text Message Included with Proper Substitution Syntax



Tip

Suppose the `INCREASE_SALARY` text message had included placeholders like `%percent` and `%starting`:

```
Your salary increase is %pct percent, and starts %date
```

In that case, the *Text Messages Picker* includes the placeholder names so you can enter appropriate static or dynamic values. If you need to use a substitution modifier like `!RAW`, it goes between the closing curly brace and the final dot:

- `&{INCREASE_SALARY}.`
- `&{INCREASE_SALARY pct="5" date="&P3_SOME_ITEM."}.`
- `&{INCREASE_SALARY pct="5" date="&P3_SOME_ITEM."}!RAW.`

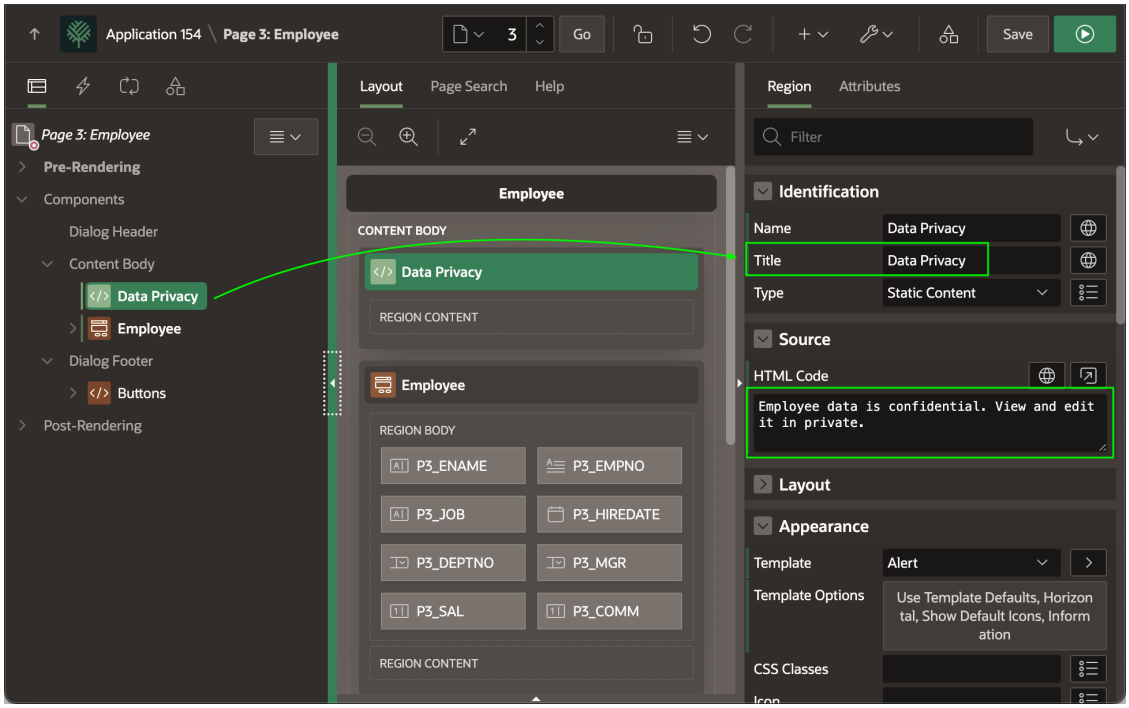
22.7.3 Entering New Text Directly Inline

Enter new UI text inline and let the next sync create its Text Message automatically.

You can also continue entering literal text into your page. For example, imagine a requirement emerges to add an info box about data privacy to the top of the *Employee* form as shown below. You can enter the *Title* and *HTML Code* directly into the page as shown below. The figure shows the **Data Privacy** static content region selected in the component tree and highlights that both its **Title** and **HTML Code** contain literal text the developer entered in the application's base language.

Copyright © 2026, Oracle and/or its affiliates.

Figure 22-19 Adding New Translatable Text Inline on a Page



22.8 Synchronizing Text Messages Anytime

Sync Text Messages to prepare newly added UI text for the next translation update.

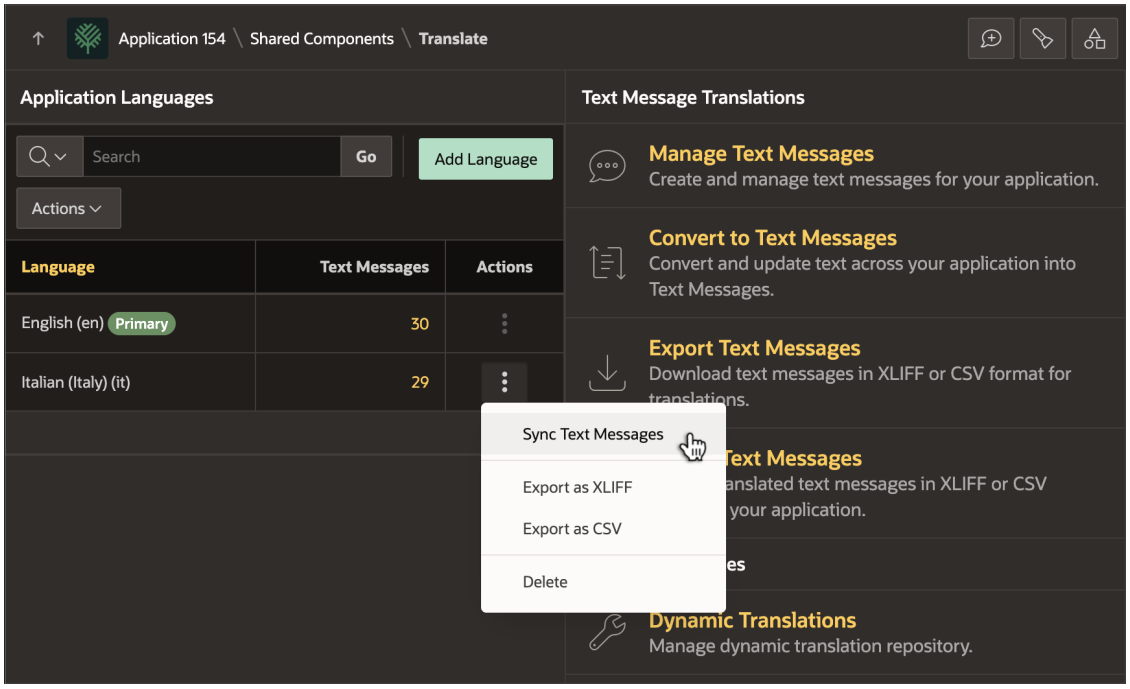
When you are ready to translate the newly added text, as shown below, use the *Sync Text Messages* option on the three vertical dots menu for the *Italian* language. Then click *(Synchronize)* in the dialog that appears. This action:

- Adds the missing *Italian* version of `INCREASE_SALARY` text message,
- Creates new primary language text messages `DATA_PRIVACY` and `PLUG_SOURCE`
- References the new messages in the *Employee* page with substitution syntax, and
- Copies the two new text messages to *Italian* entries, initially with primary language text.

> **Note**
>
> To convert literal text in your application into Text Messages without synchronizing the base language and the target language, use the **Convert to Text Messages** option in this page. When doing a sync on a target language, App Builder always performs this step *implicitly* before synchronizing the base language and target language.

Copyright © 2026, Oracle and/or its affiliates.

Figure 22-20 Synchronizing Text Messages Creates All New Messages Required

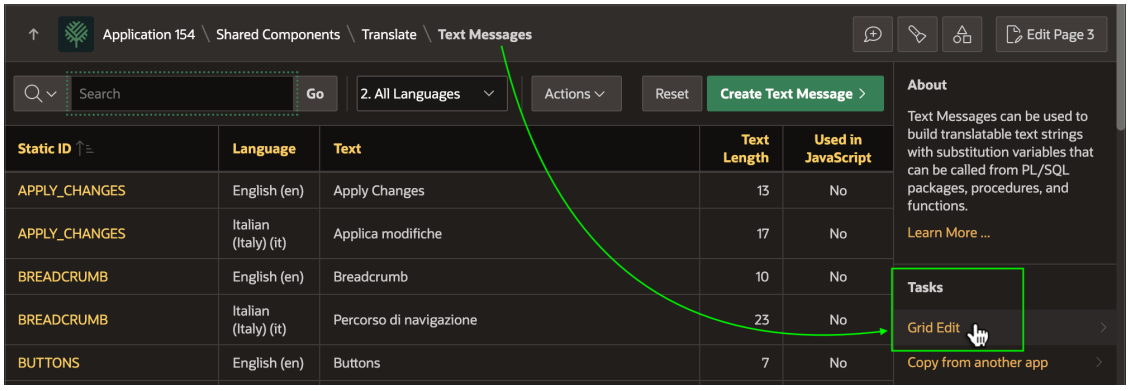


22.9 Editing Translations in a Filterable Grid

Edit translations directly in a filterable grid for small in-house updates.

You can export the XLIFF any time to have an external translation service provide the missing Italian translations. However, the *Grid Edit* task in the *Text Messages* page is handy for in-house translation. You can edit the translated text of multiple messages at once.

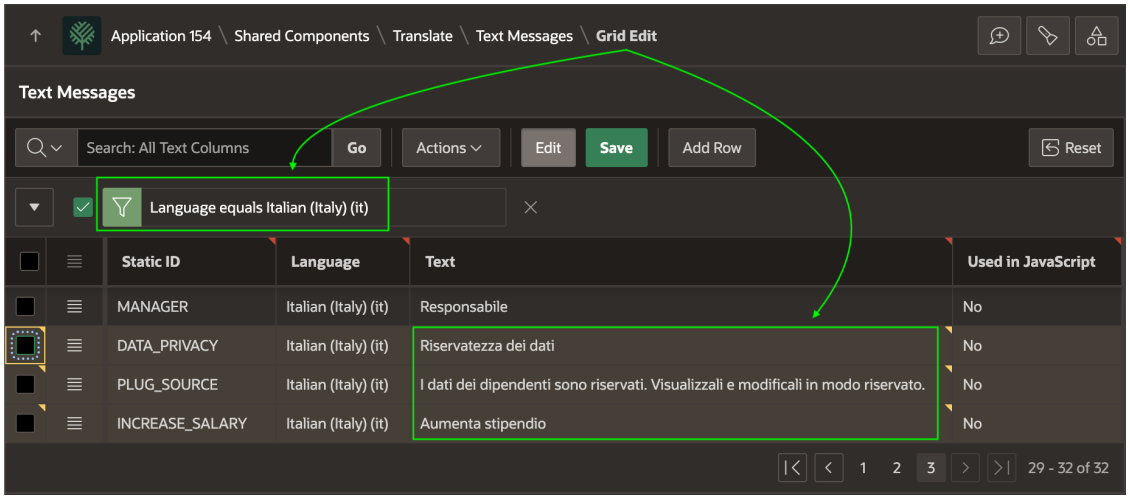
Figure 22-21 Accessing the Text Messages Grid Edit for Bulk Changes



After filtering on language, as shown below, you can enter the Italian translations for the three new text messages and click *(Save)* to apply the changes.

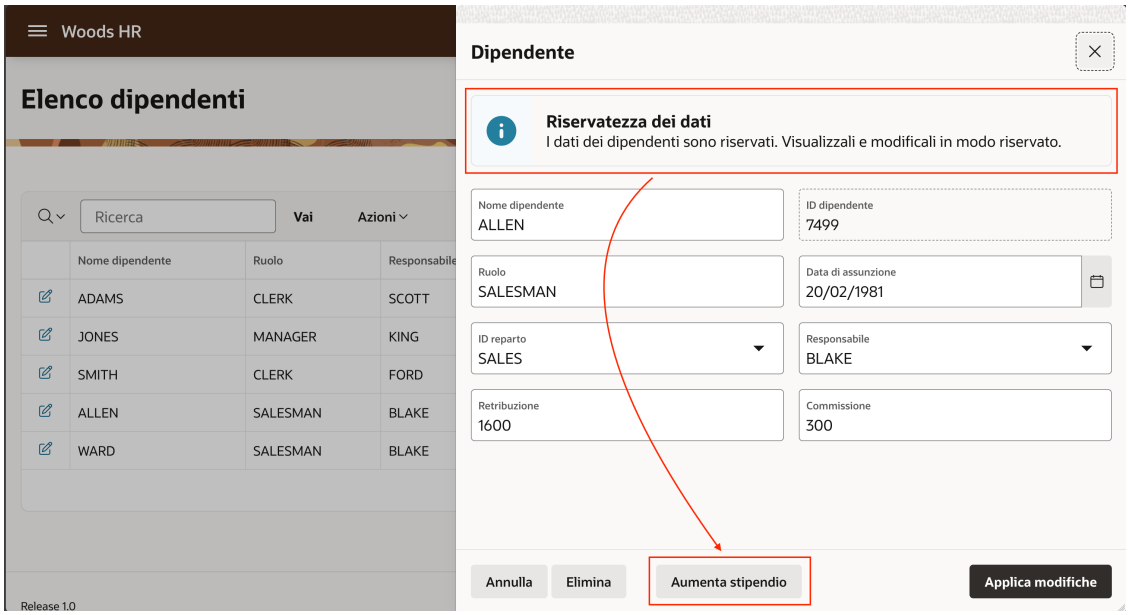
Copyright © 2026, Oracle and/or its affiliates.

Figure 22-22 Filtered Grid Edit is Handy for Bulk Translation



After saving the Italian translations for the latest text you added, a user like Susan running in Italian sees the updates to the page in her preferred language.

Figure 22-23 Manually Entered and Synchronized Messages Show for Italian Users



22.10 Centralizing Reusable UI Text

Centralize reusable UI text by converting it to Text Messages manually or in bulk.

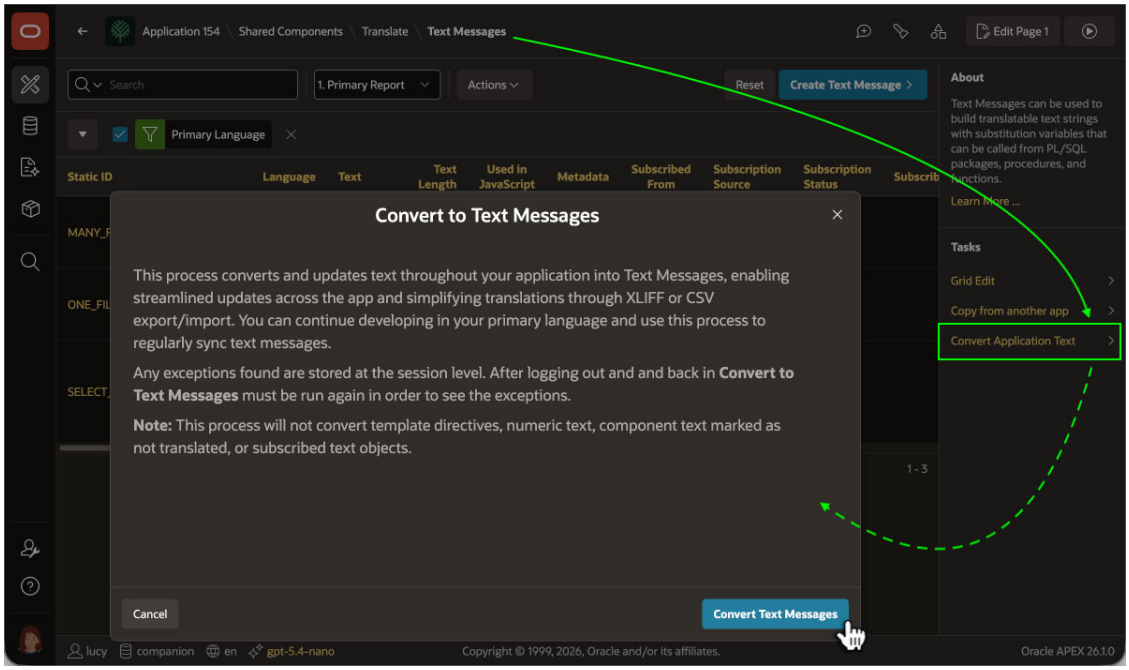
Using Text Messages to centralize reusable UI text is useful, even if your application does not require translation. Convert each reusable string as needed to a Text Message *SOMENAME*. Then use `&{SOMENAME}.` to replace the original text. In Page Designer, you can also click the globe button. For more information, see Using the Text Messages Picker.

Copyright © 2026, Oracle and/or its affiliates.

Alternatively, you can automatically convert *all* text values of translatable properties to Text Messages in bulk. On the **Text Messages** page in Shared Components, click the **Convert Application Text** link in the **Tasks** section as shown below. When you click (**Convert Text Messages**) in the **Convert to Text Messages** dialog, App Builder creates any new Text Messages required and replaces each original text with the appropriate substitution.

> **Tip**
>
> You can repeat **Convert Application Text** any time to convert new strings added since the last time you ran it.

Figure 22-24 Converting Application Text to Text Messages



Copyright © 2026, Oracle and/or its affiliates.

23

Understanding Application Lifecycle

Learn how teams maintain and enhance an Oracle APEX application over time using proven lifecycle practices.

As users work with your app, they'll report bugs and suggest improvements. Every development team strives to make steady progress through a prioritized list of fixes and features. The development *lifecycle* is the rhythm of selecting work, dividing tasks, testing results, and releasing updates to users. The goal is simple: deliver small sets of high-quality changes on a regular cadence.

Effective application lifecycle management (ALM) means understanding where your APEX app's definitions live, how to move them between environments, and how to track changes over time. This section explains APEX application artifacts and how to export and import apps, set up multiple environments, isolate changes with working copies, and use source control to preserve history.

- Identifying Application Artifacts
  Understand APEX application artifacts as exportable metadata snapshots you can compare, version, and restore.
- Exporting and Importing Applications
  Export and import APEX apps as portable snapshots.
- Working with Multiple Environments
  Use separate environments with distinct banners to develop, test, and release APEX changes safely.
- Isolating Changes with Working Copies
  Use working copies to build, test, and merge isolated APEX changes safely.
- Tracking Changes with Source Control
  Use Git to preserve APEX export history, compare changes, and support team collaboration.
- Summarizing Lifecycle Recommendations
  Combining **SQLcl** and **Git** creates a simple, effective application lifecycle pipeline.

23.1 Identifying Application Artifacts

Understand APEX application artifacts as exportable metadata snapshots you can compare, version, and restore.

An APEX application is a collection of definitions stored in Oracle Database. Unlike file-based apps, its pages, components, and logic exist as metadata that the APEX engine reads to render pages on demand. An APEX workspace provides the context for one or more applications and links to one or more schemas. Within a workspace, an application consists of many components: pages, reports, forms, navigation menus, LOVs (lists of values), and processes. All of these together form the application definition.

Because APEX applications are stored as metadata, the data dictionary always shows only their *current* state. There's no built-in version history. When you change a page or component, the old definition is replaced. Preserving history is up to you. To track changes over time, you must export the application as text files. APEX makes this easy. It can generate a SQL script,

Copyright © 2026, Oracle and/or its affiliates.

or a set of scripts, representing the entire app in its current form. These exports are called *application artifacts*: human-readable snapshots of your app's metadata that you can store, compare, and version.

The export is a portable snapshot of your app definition at a moment in time. When you export an APEX application, by default you get one SQL file named `f<app-id>.sql` containing all the application's pages, components, and settings. You can also choose a *split* export. This variant creates one file per component, organized in folders for pages, shared components, and so on. This format works better with source control because each file reflects a single page or component. If you change one report page, only that file differs in the next export. This makes identifying what's changed much simpler.

These text file artifacts are your safety net and historical record. By saving each meaningful version of the application, you can recreate any past version on any instance by re-importing those files. Over time, these exports become the source of truth for your app, in effect, checkpoints in its life. In production, the database holds the version users see; in development, you shape the next version. Managing these artifacts well lets you move between versions, compare changes, and recover earlier states when needed.

Tip

While SQL exports remain the default, evaluate whether developer-friendly *APEXlang* is right for your project's source control and external tooling needs. It's easy to read, fully editable, can be strictly validated without a database connection using SQLcl's `apex validate` command, and works well with AI coding agents. For more info, see Working with APEXlang.

23.2 Exporting and Importing Applications

Export and import APEX apps as portable snapshots.

Exporting an app extracts its definitions from the database into files. You can do this in two ways: through the APEX web interface or with the Oracle SQL Command Line (SQLcl) utility. In App Builder, open the application you want to export and choose **Export/Import > Export** from the application actions menu. APEX then generates a SQL script for you. You can export it as a single file or as a ZIP archive containing separate component files (by enabling *Split Export*). The UI prompts you to download the result. This quick, interactive method is ideal for one-off backups or GUI-based migrations.

SQLcl is Oracle's modern command-line tool for working with the database, with built-in commands for APEX. It's ideal for automating or scripting exports. Make sure SQLcl and Java are installed, then connect to the database using your workspace's parsing schema. Run a command such as:

```
sql> apex export -applicationid 12345
```

This exports the application with ID 12345 into the current directory, creating one file named `f12345.sql` by default.

In practice, include a few options for best results. The most useful is `-split`, which creates an organized folder of component files. Other handy options include `-skipExportDate`, which omits timestamps to avoid false diffs in source control, and `-expOriginalIds`, which preserves internal IDs across environments for easier comparison and refresh. If your app uses Supporting Objects, add `-expSupportingObjects` to export those as well.

Copyright © 2026, Oracle and/or its affiliates.

SQLcl also allows specifying the *format* of export using `-expType`. The default is `APPLICATION_SOURCE` (the normal SQL scripts), but you can also request `APEXLANG` for a more readable and editable representation, or `EMBEDDED_CODE` to extract all SQL, PL/SQL, and JavaScript code separately. You can combine formats. For example, to export both the standard SQL and APEXlang version in one go, use a command line this all on a single line with a space between options:

```
sql> apex export -applicationid 12345
                 -split
                 -skipExportDate
                 -expOriginalIds
                 -expSupportingObjects Y
                 -expType APPLICATION_SOURCE,APEXLANG
```

This command exports app 12345, splits it into component files, skips timestamps, preserves IDs, includes supporting object scripts, and produces both SQL and APEXlang formats. After it runs, you'll have a directory *`appalias/`* containing the APEXlang `.apx` files as well as a `f12345/` directory with the SQL Format artifacts. Together, these files represent your application in text form.

Importing an application is the reverse: it loads a previously exported file or set of files into an APEX instance, creating or replacing the app in a workspace. You can import it in two ways: through the APEX web interface or with the Oracle SQL Command Line (SQLcl) utility.

In App Builder, open the target workspace and use the **Import** function. Upload the export file (*.sql* or *.zip*), and APEX guides you through the steps. You may be asked to assign a new application ID or confirm the workspace. For example, if you exported app 100 from development and want to import it into test without overwriting another app, assign a new ID during import. APEX then recreates all pages, components, and shared logic defined in the file. If the app already exists, APEX installs a new copy or updates the existing one, depending on your choices.

You can also import a SQL format application export from the command line by running the SQL export script. Before doing so, set the context with APEX APIs if you need to target a different workspace or change the application ID. Use the `APEX_APPLICATION_INSTALL` package for this. For example, first run the PL/SQL block below:

```
begin
   apex_application_install.set_workspace('MyWorkspace');
   apex_application_install.set_application_id(56789);
end;
```

Then run `f12345.sql` or `f12345/install.sql` to import the app with new id `56789` in that workspace. To have APEX install any *Supporting Objects* automatically, add a call to `set_auto_install_sup_obj(true)` to the setup block above.

> **Tip**
>
> See Using SQLcl with APEXlang for details on importing an app in APEXlang format.

If you import into the same APEX instance but a different workspace you must use a different application ID, since IDs are unique within an instance. For example, you might do this when moving an app from *dev* to *test* on the same server. The `APEX_APPLICATION_INSTALL` API can generate a new ID and adjust the application alias and parsing schema for you. When moving

Copyright © 2026, Oracle and/or its affiliates.

to a separate APEX ***instance*** you can usually keep the same ID and alias if there's no conflict. This could happen when moving from on-premises development to a cloud environment. In either case, the import process faithfully rebuilds the application from the export. Afterward, run the app in its new environment to verify it works and complete any post-install steps, such as setting environment-specific settings or running data scripts.

> **Tip**
>
> Exporting and importing covers only the APEX application definition. Any supporting database objects must be migrated separately, typically with SQL scripts or schema migration tools. Examples include tables, views, and PL/SQL packages. This section focuses on the application artifact itself. Managing database schema changes is a separate but essential part of the lifecycle. The SQLcl *Projects* feature can simplify this work by coordinating application exports with related database objects. For more information, see Database Application CI/CD in *Oracle SQLcl User's Guide*.

23.3 Working with Multiple Environments

Use separate environments with distinct banners to develop, test, and release APEX changes safely.

Most teams use at least three environments in an application's lifecycle: development (DEV) for making changes, testing (TEST or QA) for validation, and production (PROD) for end users. Multiple environments let you develop and test safely without risking the stability of the live system. Separate environments let developers work without disrupting end users and let testers experiment without touching real data. It's far better to find a bug in TEST than have end users stumble on it in PROD. Developers can also try ideas freely in DEV, confident they're not altering the production app.

There are two ways to organize dev/test/prod setups in APEX: separate APEX instances, and separate workspaces on one instance.

For larger or mission-critical apps, the ideal setup is to use separate APEX instances for each environment. For example, you might run one database (or pluggable database) with APEX for development, another for test, and another for production. Each environment is fully isolated. Changes in DEV have no effect on PROD until you explicitly export and import the app. A key benefit is that you can reuse the same application ID, alias, and workspace name across environments without conflict, since each has its own APEX installation. Your DEV workspace *MyApp* can host application 100, and the TEST instance can host the same app ID and workspace name. This one-to-one setup simplifies tracking. Many Oracle teams follow this model, provisioning separate APEX services for DEV, TEST, and PROD.

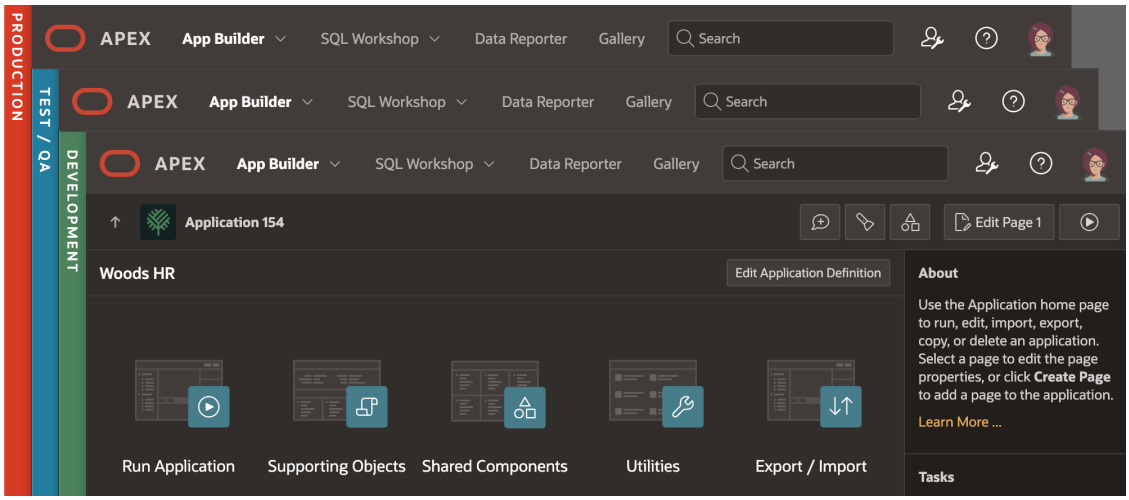
If running multiple APEX instances isn't practical, you can still separate environments logically using multiple workspaces in a single instance. For example, a *MYAPP_DEV* workspace can handle development, *MYAPP_TEST* for testing, and *MYAPP_PROD* for production, each mapped to its own schema. The apps remain distinct even though they share the same APEX installation. You export from the DEV workspace and import into the test workspace, then on to prod when QA gives the thumbs up. The trade-off is that application IDs and aliases must be unique across the instance, so you may need different IDs. For example, app 100 in DEV might become app 1100 in TEST, and 2100 in PROD. APEX lets you assign new IDs, new aliases, and remap schemas during import. This approach achieves isolation, though it requires more coordination since everything runs in one database.

Whatever setup you choose, label your environments clearly. APEX offers an Environment Banner that lets you display a colored stripe and text at the top or along the side of the screen.

Copyright © 2026, Oracle and/or its affiliates.

For example, you might show a bright red *PRODUCTION* banner in production, a blue *TEST* banner in test, and a green *DEV* banner in development. These cues appear in the App Builder, helping prevent mistakes like editing production by accident. A distinct banner for each environment is a simple but powerful safeguard. The figure shows three App Builder in three different workspaces, each with a different color banner clearly representing **PRODUCTION**, **TEST / QA**, and **DEVELOPMENT** environments.

Figure 23-1 Using Environment Banners for DEV, TEST, and PROD



With multiple environments in place, a typical release workflow looks like this: Developers build and test changes in DEV. When a set of updates is ready, the team exports the APEX application and any required data scripts from DEV and imports them into TEST. In TEST, QA or stakeholders verify everything works as expected. If issues appear, fixes are made in DEV and re-tested. Once the app passes testing, the same export file is imported into PROD. This ensures the code in production matches what was tested. Teams often increment the application's *Version Number* setting with each release so versions are easy to track. DEV might be on 2.0 while PROD uses at 1.9. Moving changes through stages keeps environments isolated, reduces risk, and builds confidence that production runs only well-tested code.

23.4 Isolating Changes with Working Copies

Use working copies to build, test, and merge isolated APEX changes safely.

When several developers work on the same APEX app, coordination matters. APEX is multi-user by design: team members can build or edit different parts of an app at once. To prevent conflicts, APEX offers page locking. For example, if you're editing Page 5, you can lock it so others can't change it until you're done. But for larger or riskier changes, or when developing a new feature in isolation, *Working Copies* provide a safer path.

A *Working Copy* is a clone of the main application within the same workspace, a personal sandbox for each developer. At any time, a team member can create one to experiment, fix a bug, or build a new feature without affecting the main app. It's like creating a temporary branch, similar in spirit to branching code in Git, but entirely within APEX. The working copy starts identical to the main application. From there, you make changes in your copy while others continue working on the main app or their own.

Working copies enable parallel development without conflict. For example, Alice creates a working copy to redesign report pages while Bob fixes bugs in the main app. Alice can make

Copyright © 2026, Oracle and/or its affiliates.

extensive changes safely in her copy. APEX lets you compare your working copy with the main app, or another working copy, to see what's different. It's an effective way to review changes in isolation. If others update the main app while you're working, you can refresh your copy to pull in those changes and keep your sandbox current.

When you're satisfied with your changes, you can either merge your working copy into the main app or discard it. Merging applies all changes to the main definition, typically when a feature or fix is complete and tested. If the work was experimental, simply delete the working copy with no impact on the main app. It's a safe playground for development.

Working copies merge or refresh at the page and component level. If two copies modify different pages, they can merge cleanly. But if both change the same page, a conflict may occur. To prevent this, lock any page you're modifying in a working copy. For example, before Alice edits Page 42 in her working copy, she locks Page 42 in the main app with a note like "Working on new report UI." The lock signals to others not to touch that page until she's done, and *prevents* them from doing so in the main application. After merging her changes, she can unlock it. Together, working copies and page locks let teams develop in parallel, isolated when needed, yet merging back into a single shared application.

In addition to using App Builder, developers can also work on a working copy using APEXlang. Since it is editable using external tools, it is ideal for use with code editors, AI coding agents, and diff/merge tools. A developer can export her working copy to APEXlang, work on it externally, then import the APEXlang working copy app back into App Builder. The final merge back into the main application of new or changed pages and components can use normal working copy functionality.

If working copy conflicts prevent a merge in App Builder, export the main application and working copy to APEXlang. Resolve the conflict manually using your favorite external tools. Then, validate the updated main application using `apex validate`. When it passes validation, import the merged APEXlang result to overwrite the main application. Finally, delete the conflicted working copy.

Working copies let teams build features or fixes in isolation without *requiring* separate environments per developer or complex branching. The approach suits small teams and fast iterations, keeping all work within the familiar App Builder while providing useful separation. Many developers use it to sandbox big changes, prototype freely, review differences, and merge into the main app only when ready.

For more information, see Creating a Working Copy to Merge, Refresh, or Compare in *Oracle APEX App Builder User's Guide*.

Tip

A working copy has a different application id from the main application, but APEX knows they are related. If application logic depends the value of the main application ID, use `MAIN_APP_ID` instead of `APP_ID` as a substitution or bind variable. It always evaluates to the ID of the main application.

23.5 Tracking Changes with Source Control

Use Git to preserve APEX export history, compare changes, and support team collaboration.

Even with multiple environments and good team practices, one question remains: how do you keep a history of changes? APEX doesn't track version history. Once you update an app, the old version is gone unless you exported it. Relying on memory or notes is risky. That's where source control comes in, and for most teams today, that means Git.

Copyright © 2026, Oracle and/or its affiliates.

Git is a popular open-source version control system and the de facto standard for development teams worldwide. In simple terms, it manages a repository of your files, such as exported APEX applications and database scripts. This repository acts as a time machine for your project. Each time you make changes, you *commit* them with a message describing what changed. Git tracks who made each change, when it was made, what was added or removed, and why. Later, you can review the history to answer questions like: When was this bug introduced? Who wrote this code? What changed between versions 1.4 and 1.5?

A typical Git setup uses a remote repository on GitHub, GitLab, Bitbucket, or an internal server. Each team has at least one local copy of the "repo". To keep things simple, many teams nominate one teammate to handle Git on behalf of the group. That person periodically exports the latest version of the APEX app, then commits and pushes those files to the shared repository with a short, descriptive message such as "Week 34 fixes and improvements." Each commit acts as a checkpoint, and over time the repository builds a chronological record of versions. If a page worked two weeks ago but doesn't now, you can compare exports in Git to see exactly what changed. That is an invaluable aid for debugging and auditing.

> **Tip**
>
> Working copies hold in-progress changes that aren't yet ready to be part of the main application. A team using working copies typically only exports the *main application* to Git.

Getting started with Git doesn't require deep expertise. Create a free private repository on GitHub or GitLab and use it as your central store. Begin with an initial export of your APEX app as the first commit, then make new commits at regular milestones: after key changes, before testing, or on a set schedule. Write short, meaningful messages that describe what changed; over time they tell the story of your project.

Using Git gives you more than backups: it lets you compare versions easily. Git's diff tools show exactly which lines in APEX application artifact files changed between commits. If an unintended edit slips in, you'll see it. When someone asks, "What changed in this release?" you can simply point them to the commit history instead of listing every change.

A major advantage of using Git is traceability: the who, what, when, and why behind every change. This information is gold for maintenance. Six months later, when you wonder "Why was this validation logic removed?", you can check the commit history, see who made the change, and read the message explaining it. For example, it might reference a bug number or business rule update. Without source control, that knowledge is easily lost.

Using Git also opens the door to automation. You might integrate with continuous integration (CI) tools such as **Jenkins** or **GitHub Actions** to deploy the latest commit to a test environment and run checks automatically. Even without full CI, a lightweight approach works well: many teams script both directions: exporting from APEX to Git and importing from Git back into an environment for deployment. These scripts, often built with SQLcl and simple shell or batch commands, provide control and transparency without the complexity of larger frameworks. With just a few basic commands you can automate backup and versioning in a way that's easy to maintain. APEX gives you the flexibility to adopt as much DevOps automation as suits your team's workflow and pace.

23.6 Summarizing Lifecycle Recommendations

Combining **SQLcl** and **Git** creates a simple, effective application lifecycle pipeline.

This **SQLcl**/**Git** pipeline:

Copyright © 2026, Oracle and/or its affiliates.

- Use SQLcl to regularly export your APEX app (and related database scripts) into files.
- Use Git to commit those exports, capturing each change with context.
- Import the exported files into test and production environments.

This approach needs no special tools beyond what you already have: APEX, SQLcl, and a Git host. This makes it practical for teams of any size. It follows a lightweight automation philosophy: script the repetitive parts, keep control, and scale up only when needed. As your team or app grows, you can evolve these scripts or integrate with CI tools, but the core practices of exporting, versioning, and using multiple environments will serve you well.

By understanding your application artifacts, moving cleanly between DEV, TEST, and PROD, isolating work with working copies, and tracking everything in Git, you'll manage your app's lifecycle with confidence. No change is lost, every version is reproducible, and production updates happen with care and traceability.

Tip

Larger development teams can also use APEX for a feature-centric development approach. Each developer uses a private Git clone of the repository and works in an isolated "branch instance" where they are the only developer making changes. This approach requires more coordination, but lets larger teams use popular code review tools to separately give teammates feedback on the changes required to implement each bug fix or each new feature.

Copyright © 2026, Oracle and/or its affiliates.

24

Working with APEXlang

APEXlang is a practical text representation of your application that works both inside App Builder and with external tools. Its declarative definitions let you move through your development flow with confidence. APEX skills for popular AI coding agents enable rich, new ways of collaborating with AI coding agents for app development.

With App Builder, you create Oracle APEX apps right in your browser. You configure smart data components on your pages by choosing a table, view, or query and adjusting declarative options. Then you add business logic only you could write.

APEXlang gives you another way to work with the same application definition in external tools. You can review, edit, validate, version control, export, and import an application's APEXlang definitions using external tools like SQLcl, the SQL Developer extension for Visual Studio Code, and other development tools.

SQLcl, ORDS, and SQL Developer share a common APEXlang compiler driven by the rules that Page Designer has used for years. This makes external editing feel natural and safe, with strict validation, code completion, and integrated import and export support.

- Why APEXlang Matters
  APEXlang lets you decide when to work in the browser with App Builder and when to use external tools.
- Using APEXlang with AI Coding Agents
  Use APEXlang skills, SQLcl validation, and blueprints to collaborate with AI coding agents.
- Exporting APEXlang from App Builder
  When you export an application from App Builder, you can choose the APEXlang format.
- Importing APEXlang in App Builder
  To import an APEXlang application, use the Import wizard.
- Reading APEXlang Syntax
  APEXlang's clean, modern syntax makes it easy for developers and AI coding assistants to read and write.
- Using App Builder and External Tools Together
  Export your application to APEXlang to work on it externally and import it to reflect the changes in App Builder. Teams using working copies to isolate a developer's changes for a feature or fix can also embrace external tools.
- Editing APEXlang in Visual Studio Code
  Using the SQL Developer extension for Visual Studio Code, you can export, validate, import, and edit APEXlang applications with context-sensitive assistance and proactive highlighting of errors.
- Using SQLcl with APEXlang
  Use the SQLcl command line tool to export, import, and validate APEXlang applications.
- APEXlang Inside App Builder
  View any page in APEXlang from Page Designer, and efficiently review working copy diffs interactively.

Copyright © 2026, Oracle and/or its affiliates.

24.1 Why APEXlang Matters

APEXlang lets you decide when to work in the browser with App Builder and when to use external tools.

Before APEXlang, external editing of APEX applications was not supported. SQL exports worked well for source control, but application settings were represented as internal API calls and numerical IDs. Readable YAML simplified reviews and diffs, but it was not an editable import format.

APEXlang provides the best of both worlds. You can continue to use App Builder for visual and declarative development, and use external tools when they fit the task. SQLcl can export, import, and validate applications from the command line or in scripts. In Visual Studio Code, the SQL Developer extension can help you export, edit, validate, and import applications while showing validation problems proactively.

> **Tip**
>
> For more information, see the.APEXlang Language Reference and the APEXlang Atlas interactive learning tool.

24.2 Using APEXlang with AI Coding Agents

Use APEXlang skills, SQLcl validation, and blueprints to collaborate with AI coding agents.

APEXlang is well-structured and easy to understand, so you can also work productively with AI coding agents like Claude, Codex, and others. With Oracle's APEXlang skills from GitHub, you can describe what you need and let an AI collaborator help review an application, suggest improvements, generate pages and components, make incremental changes, and find and fix problems using SQLcl's `apex validate` command.

This workflow is most useful when you treat validation as part of the editing loop: change the APEXlang files, validate the application, review the results, and import only after the source validates cleanly.

> **Tip**
>
> To install or update the APEXlang skills, use the `skills sync` command in SQLcl.

If you work with stakeholders to capture and agree on specifications before starting a project, APEX blueprints help teams refine requirements before costly surprises appear. Starting from a draft functional spec and data model description, an AI coding agent uses Oracle's detailed blueprint generation prompt to produce a structured Markdown plan for pages, navigation, reports, forms, charts, filters, actions, and initial behavior. App Builder imports this blueprint and deterministically generates a scaffolded app from tested APEXlang building blocks, letting stakeholders try real pages and give early feedback. You can revise the specification and regenerate until everyone agrees on the app's shape. After sign-off, the final scaffold becomes the foundation you and your team evolve to add the details real systems require using a combination of App Builder and conversational development with AI coding agents using APEXlang skills.

Copyright © 2026, Oracle and/or its affiliates.

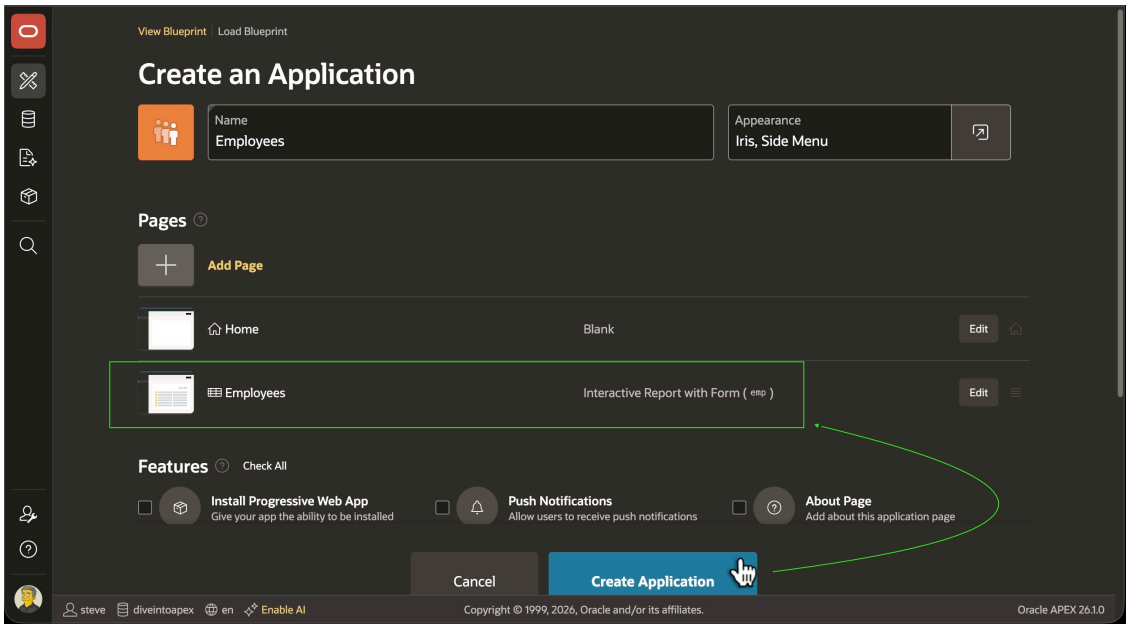
24.3 Exporting APEXlang from App Builder

When you export an application from App Builder, you can choose the APEXlang format.

Creating an Example Application to Study

To see what APEXlang looks like, start with a simple APEX application built by the Create App Wizard. The example application has an Interactive Report page and a modal Form page on the familiar `EMP` table.

Figure 24-1 Creating a Small Employees Application Using the Create App Wizard

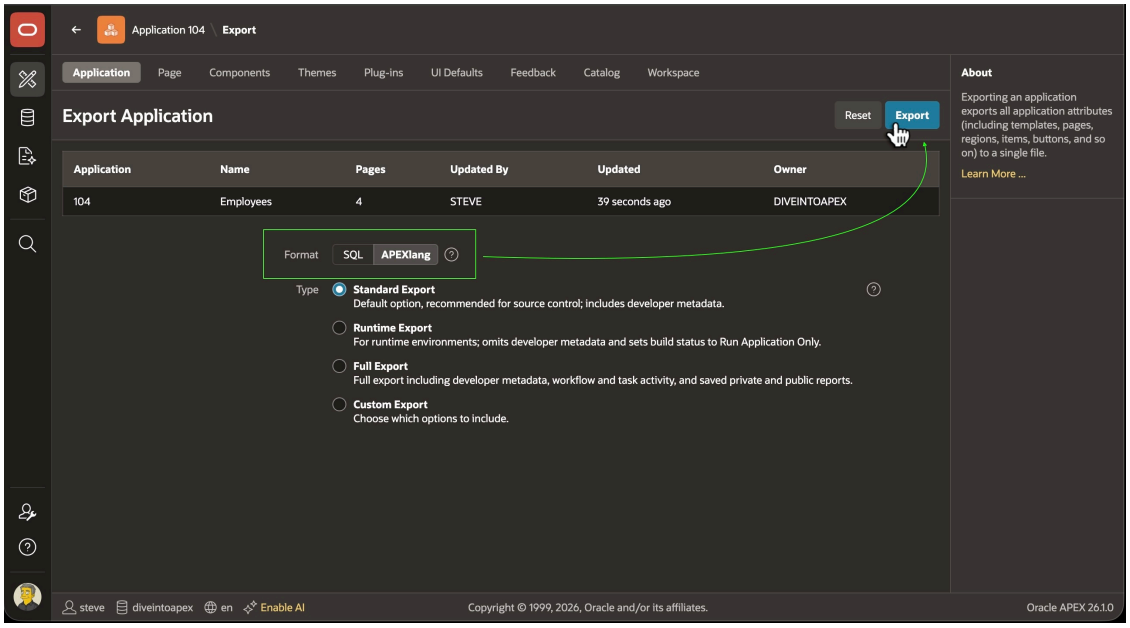


Exporting the Application in APEXlang

To export the app in APEXlang, just choose that format on the Export Application page as shown below.

Copyright © 2026, Oracle and/or its affiliates.

Figure 24-2 Choosing APEXlang Format for Your Application Export in App Builder



This downloads an `employees.zip` file. As shown below, it contains a file for each page and component. The `application.apx` file has app-level settings. The `.apx` files contain APEXlang.

Contents of an APEXlang Application Export

The `deployments`, `pages`, `shared-components` subdirectories organize app components at the root level of the zip. Notice filenames for pages use both number and alias for additional clarity, and the compiler enforces the page alias matches the page number and alias in the file name. Static Application Files are included as-is along with a list of their names and metadata in `static-files.apx`.

```
employees.zip
├── application.apx
├── page-groups.apx
├── deployments
│   └── default.json
├── pages
│   ├── p00000-global-page.apx
│   ├── p00001-home.apx
│   ├── p00002-employees.apx
│   ├── p00003-employee.apx
│   └── p09999-login.apx
├── shared-components
│   ├── static-files
│   │   └── icons
│   │       ├── app-icon-144-rounded.png
│   │       ├── app-icon-192.png
│   │       ├── app-icon-256-rounded.png
│   │       ├── app-icon-32.png
│   │       └── app-icon-512.png
│   ├── themes
```


Copyright © 2026, Oracle and/or its affiliates.

```
│   │   └── universal-theme
│   │       └── theme.apx
│   ├── authentications.apx
│   ├── authorizations.apx
│   ├── breadcrumbs.apx
│   ├── build-options.apx
│   ├── component-settings.apx
│   ├── lists.apx
│   ├── lovs.apx
│   └── static-files.apx
└── .apex
    └── apexlang.json
```

Application IDs and Deployment Settings

In APEXlang, the app id is stored in the `defaults.json` file in the `deployments` subdirectory. This lets the downloaded file `employees.zip` use a descriptive name based on the `employees` app alias. This separation also gives you the option to personalize each deployment environment with an appropriate set of environment-specific settings.

```
{
  "app" : {
    "id" : 104,
    "runtime" : {
      "debugging" : true
    }
  }
}
```

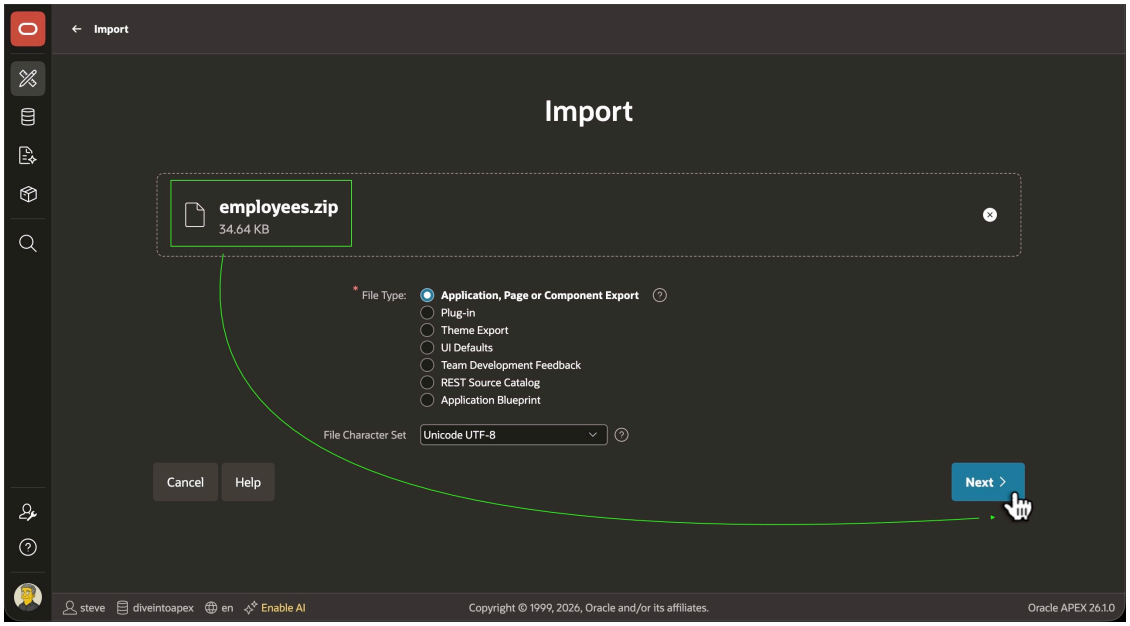
24.4 Importing APEXlang in App Builder

To import an APEXlang application, use the Import wizard.

App Builder can import an application in APEXlang format. Just drag or select its zip file on the Import page and continue through the wizard.

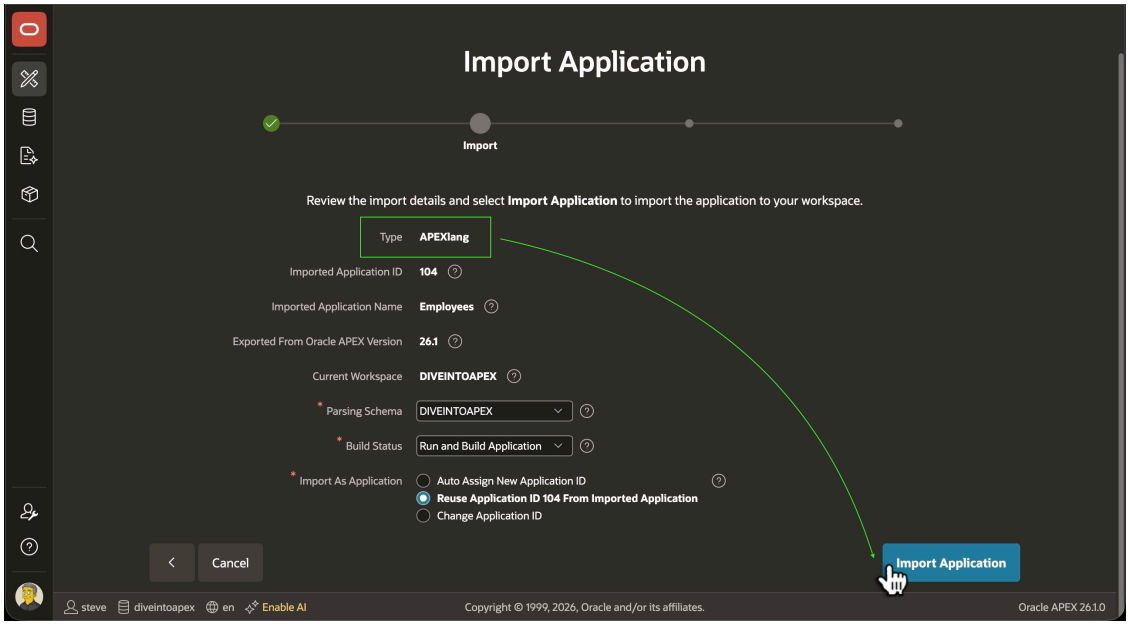
Copyright © 2026, Oracle and/or its affiliates.

Figure 24-3 Importing an Application in APEXlang Format in App Builder



The Import wizard confirms the Type is APEXlang and lets you choose the application ID. Just click (Import Application) to proceed. App Builder validates and compiles the APEXlang source. If no errors are raised, the application imports successfully.

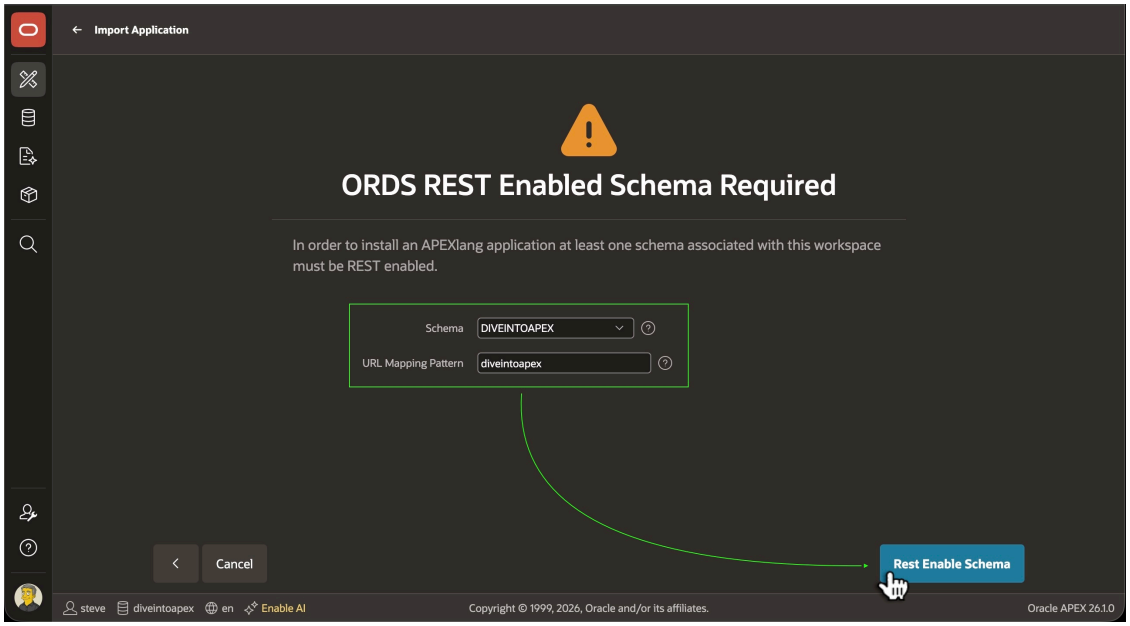
Figure 24-4 Previewing the Application to Import and Choosing an Application ID



App Builder uses a dedicated ORDS endpoint with Builder authentication to compile APEXlang artifacts. So, the first time you import an APEXlang format app in a workspace, you may be prompted to enable your workspace's parsing schema for ORDS REST services. Just click the **REST Enable Schema** button and the wizard proceeds to the next step to continue the import.

Copyright © 2026, Oracle and/or its affiliates.

Figure 24-5 Enabling Workspace Parsing Schema for ORDS REST Services if Needed



24.5 Reading APEXlang Syntax

APEXlang's clean, modern syntax makes it easy for developers and AI coding assistants to read and write.

Studying APEXlang Syntax of a Page

When you open an app folder with APEXlang files in your favorite editor, you can browse, search, and edit them like any project. For example, page 3 is an Employee form page with alias `employee`. Its filename incorporates both the page number and the alias. Opening `p00003-employee.apx`, you see the wizard created a `modalDialog` page using the `drawer` template.

```
page 3 (
    name: Employee
    alias: EMPLOYEE
    title: Employee
    appearance {
        pageMode: modalDialog
        dialogTemplate: @/drawer
        templateOptions: [
            #DEFAULT#
            js-dialog-class-t-Drawer--pullOutEnd
        ]
    }
    dialog {
        chained: false
    }
  ▒
)
```


Copyright © 2026, Oracle and/or its affiliates.

In the example above, it's easy to see this `page` component's `pageMode` property – in the `appearance` group – has the value `modalDialog`. For brevity, you can say `appearance.pageMode` is `modalDialog`.

APEXlang avoids unnecessary delimiters. Every component (e.g. `page`) contains its properties, grouped by function, as name : value pairs. Properties like `name`, `alias`, and `title` in the `identification` group can optionally appear with no group as shown below. Required properties must have a value, but files stay concise since any property with its default value can be omitted. Leave optional properties out until you need to assign them a value.

Values don't require quotes in most cases, and a new line separates properties and array elements. For example, the value of `appearance.templateOptions` is an array with two entries. In this case, they are CSS class names the template defines with the special `#DEFAULT#` placeholder for "any default classes" the template specifies.

```
page 3 (
    name: Employee
      ▒
    appearance {
      ▒
        templateOptions: [
            #DEFAULT#
            js-dialog-class-t-Drawer--pullOutEnd
        ]
    }
  ▒
)>
```

Component Identifiers and References

Each component type's metadata nominates an identifier property. Most of the time, `staticId` plays this role, but sometimes it's the `name` or `alias`. All components have a mandatory human-readable static ID so APEXlang files never need to reference internal ID numbers. APEX automatically assigns sensible static IDs to any components missing one on import or upgrade.

The APEXlang compiler distinguishes each component within a given scope using its identifier property value. It enforces component uniqueness, depending on the type, across the app, a page, or just within a parent. For example:

- an app can have only one page `3`
- page `2` and `3` can both have a `region` with identifier `employees`
- page `3` can have `employees` and `departments` regions each with a `DEPTNO` column
- the `departments` region on page `3` cannot have two `DEPTNO` columns.

A component's identifier value follows its type, just before the opening parenthesis of its property list. In the example below, `3` is the `page` component's identifier, and `employee` identifies the `region`.

```
page 3 (
 ▒
   region employee (
     ▒
    )
 ▒
)
```


Copyright © 2026, Oracle and/or its affiliates.

Component References in APEXlang

Each `@`-prefixed value is a reference to the identifier property value of another component. Later in `p00003-employee.apx` you find the page contains a Form region named `Employee` identified by `employee`. Notice its `appearance.template` property references the `blank-with-attributes` template.

```
page 3 (
 ▒
   region employee (
        name: Employee
        type: form
        source {
            location: localDatabase
            tableName: EMP
        }
        layout {
            sequence: 10
            slot: contentBody
        }
        appearance {
            template: @/blank-with-attributes
            templateOptions: #DEFAULT#
        }
        edit {
            enabled: true
            allowedOperations: [
                add
                update
                delete
            ]
        }
    )
 ▒
)
```

Each page item appears in the page's `.apx` file as well. Notice the hidden, primary key `P3_EMPNO` item based on the `EMPNO` column. It references the `employee` region as the value of both its `layout.region` and `source.formRegion` properties.

When a `pageItem` references an `employee` region on the same page, it uses `@employee`.

```
page 3 (
 ▒
    pageItem P3_EMPNO (
        type: hidden
        layout {
            sequence: 10
            region: @employee
            slot: regionBody
        }
        source {
            formRegion: @employee
            column: EMPNO
            dataType: number
```


Copyright © 2026, Oracle and/or its affiliates.

```
            queryOnly: true
            primaryKey: true
        }
        security {
            sessionStateProtection: checksumRequiredSessionLevel
        }
    )
 ▒
)
```

Similarly, when a pageItem references a shared component like an LOV in the same app, it uses `@`. Notice the `P3_DEPTNO` item below is a `selectList` referencing the `sharedComponent` LOV using `@dept-dname`.

```
page 3 (
 ▒
    pageItem P3_DEPTNO (
        type: selectList
        label {
            label: Deptno
            alignment: left
        }
        lov {
            type: sharedComponent
            lov: @dept-dname
        }
        layout {
            sequence: 80
            region: @employee
            slot: regionBody
            alignment: left
        }
        appearance {
            template: @/optional-floating
            templateOptions: #DEFAULT#
        }
        source {
            formRegion: @employee
            column: DEPTNO
            dataType: number
        }
    )
 ▒
)
```

Two exceptions to the `@`-prefix rule use an `@/`-prefix instead. They are the components that are either:

- on the global page, or
- part of the standard Universal Theme.

For example, in a regular page, reference the `my-global-page-header` region from the global page using `@/my-global-page-header`.

Copyright © 2026, Oracle and/or its affiliates.

When a template comes from the standard Universal Theme set, use references like `@/drawer` or `@/blank-with-attributes` or `@/optional-floating`. In contrast, when a theme template is defined locally within the app, use `@my-slideshow`.

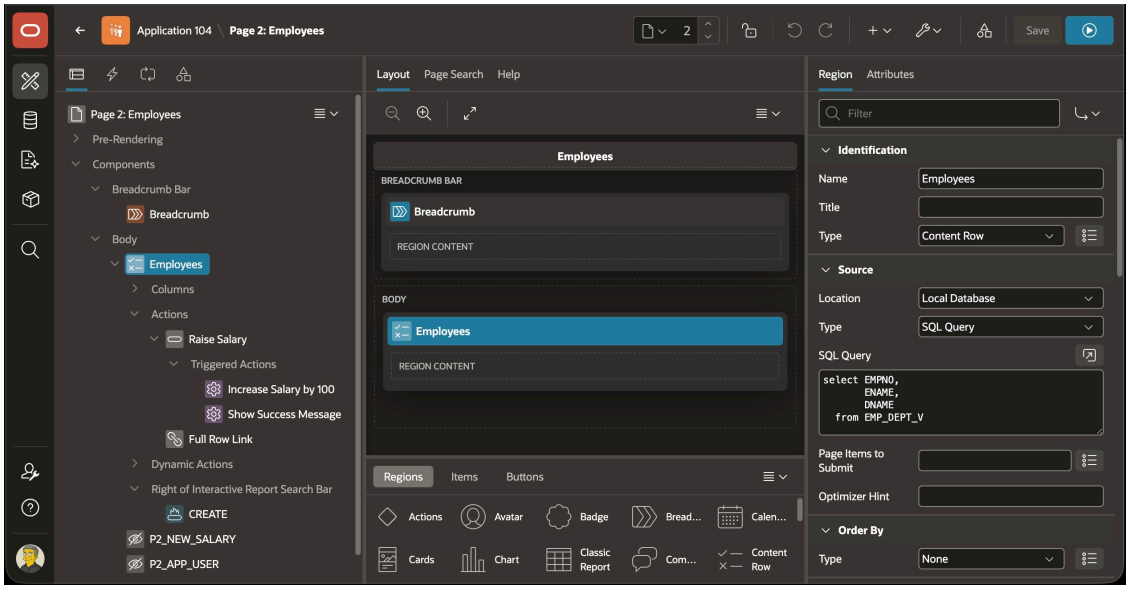
Adding Additional Features to Example App in App Builder

To explore how SQL, PL/SQL, JavaScript, HTML, and CSS appear in APEXlang files, suppose you use Page Designer in App Builder to evolve the initial Employees page to make the following enhancements:

- switch the Interactive Report region to be a Content Row
- change it to be based on a SQL query,
- update the page help text with some HTML markup,
- define an inline CSS rule to increase the font size by 20%
- add a Raise Salary action button to run PL/SQL business logic, and
- use JavaScript on the Page Load event to welcome the user back to your app.

The figure below shows your page after these modifications. You could export the modified application again from App Builder to study how these new features look in the `.apx` files. However, since you haven't made changes yet in VS Code, you can simply export the app to APEXlang directly in VS Code to reflect the latest App Builder changes in the project files.

Figure 24-6 Modified Employees Page in Page Designer



SQL, PL/SQL, JavaScript, HTML, and CSS in APEXlang

APEXlang represents multi-line code using fenced code blocks with a language indicator. The following example shows a SQL query in the source of the Employees region.

```
page 2 (
    name: Employees
▒
    region employees (
        name: Employees
```


Copyright © 2026, Oracle and/or its affiliates.

```
        type: themeTemplateComponent/contentRow
        source {
            location: localDatabase
            type: sqlQuery
            sqlQuery:
                ```sql
                select EMPNO,
                       ENAME,
                       DNAME
                  from EMP_DEPT_V
                ```
        }
)
)
```

PL/SQL in the application appears in a block with the `plsql` language indicator. In this example, the logic is a single `update` statement that returns the adjusted salary into a hidden page item.

```
page 2 (
    name: Employees
    region employees (
        name: Employees
        action raise-salary (
            position: primaryActions
            template: button
            label: Raise Salary
            layout {
                sequence: 10
            }
            behavior {
                type: triggerAction
            }
            triggerAction increase-salary-by-100 (
                name: Increase Salary by 100
                action: executeServerSideCode
                settings {
                    plsqlCode:
                        ```plsql
                        update emp
                           set sal = nvl(sal,0) + 100
                         where empno = :EMPNO
                         returning sal into :P2_NEW_SALARY;
                        ```
                    itemsToSubmit: EMPNO
                    itemsToReturn: P2_NEW_SALARY
                }
                execution {
                    sequence: 10
                }
)
```

Copyright © 2026, Oracle and/or its affiliates.

```
        )
    )
)
```

Page-load JavaScript runs in the browser, so it has a `javascript-browser` language indicator. Any server-side JavaScript would use `javascript-mle` instead.

```
page 2 (
    name: Employees
    dynamicAction on-page-load (
        name: On Page Load
        execution {
            sequence: 20
        }
        when {
            event: ready
        }
        action native-javascript-code (
            action: executeJsCode
            settings {
                jsCode:
                    ```javascript-browser
                    if ( !sessionStorage.getItem( "p2Welcomed" ) ) {
                        const username =
apex.item( "P2_APP_USER" ).getValue();
                        apex.message.showPageSuccess( `Welcome back, $
{username}!` );
                        sessionStorage.setItem( "p2Welcomed", "true" );
                    }
                    ```
            }
            execution {
                sequence: 10
            }
)
)
)
```

HTML and CSS use the same fenced block pattern with `html` and `css` language indicators.

```
page 2 (
    name: Employees
    css {
        inline:
            ```css
            body {
                font-size: 1.2rem;
            }
            ```
    }
```

Copyright © 2026, Oracle and/or its affiliates.

```
)
```

And page help HTML markup looks similar.

```
page 2 (
    name: Employees
    help {
        helpText:
            ```html
            <p>
              Select an employee to edit it,
              or click (Raise Salary) to increase
              an employee's salary by 100.
            </p>
            ```
    }
)
```

Tip

For more information, see the.APEXlang Language Reference and the APEXlang Atlas interactive learning tool.

24.6 Using App Builder and External Tools Together

Export your application to APEXlang to work on it externally and import it to reflect the changes in App Builder. Teams using working copies to isolate a developer's changes for a feature or fix can also embrace external tools.

Application is the Unit of Export and Import

You can use App Builder as you normally would, but be mindful that the unit of APEXlang export is the application. If you have made changes to your app outside App Builder, importing its APEXlang replaces App Builder's current version to reflect the latest external changes. Conversely, if you make changes in App Builder, external tools see them when you next export the app to APEXlang.

Using APEXlang for Source Control

Maintaining your app's development history over time by source controlling its external artifacts in a system like Git continues to be a best practice. APEXlang files work great for this purpose, too.

Resolving Conflicts Using APEXlang

If you realize your external APEXlang and App Builder changes are out of sync, since APEXlang files are editable, validatable, and importable, you can resolve the situation by:

- Exporting the App Builder app version to APEXlang
- Unzipping it into a temporary folder

Copyright © 2026, Oracle and/or its affiliates.

- Unifying versions, resolving any conflicts, with your favorite diff/merge tool
- Importing the unified APEXlang back into App Builder when appropriate.

The app only imports when it validates cleanly, so if it fails just review the errors, address them, and try again.

Using APEXlang with a Working Copy

If you create a working copy of your main application to isolate work on a feature or fix, you can export the working copy to APEXlang, use external tools to modify it, and import it again. APEX preserves the link between the working copy and its main application. When you are ready to merge your changes, the process works the same as if you had made them entirely in App Builder.

For more information, see Creating a Working Copy to Merge, Refresh, or Compare in *Oracle APEX App Builder User's Guide*.

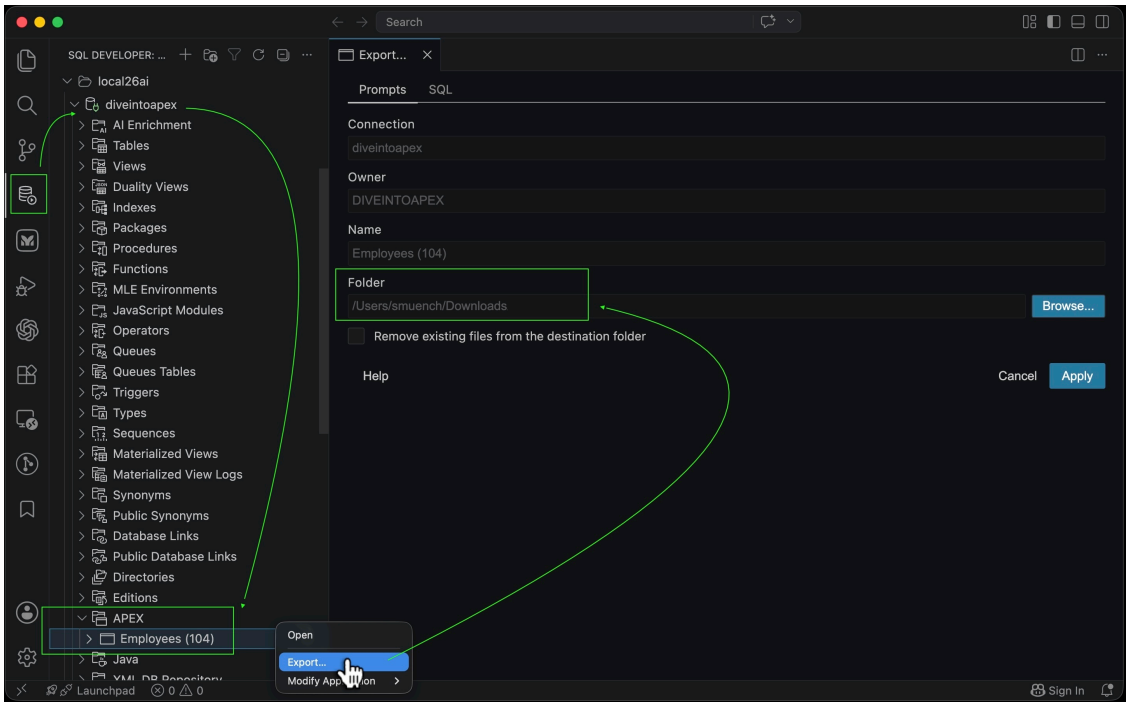
24.7 Editing APEXlang in Visual Studio Code

Using the SQL Developer extension for Visual Studio Code, you can export, validate, import, and edit APEXlang applications with context-sensitive assistance and proactive highlighting of errors.

Exporting APEXlang with SQL Developer

In VS Code, SQL Developer shows APEX apps in a connection's APEX folder. As an alternative to exporting from App Builder, choose **Export…** on an app's context menu under the connection. Then pick a parent folder for the app's APEXlang artifacts and click **Apply**. If necessary, for our example here, SQL Developer creates the `employees` subdirectory in that parent folder and exports the app in APEXlang format into it.

Figure 24-7 Exporting Employees App (104) in APEXlang from SQL Developer

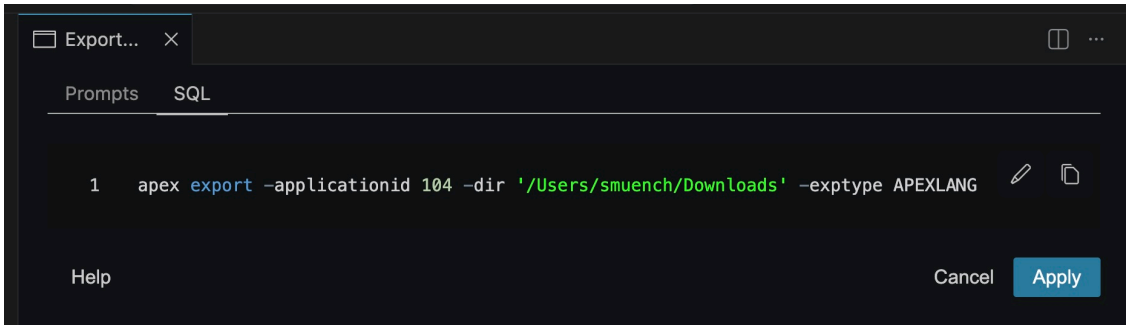


Copyright © 2026, Oracle and/or its affiliates.

If you're curious, the SQL tab in the Export… panel shows the SQLcl command to do this APEXlang app export. The `-dir` parameter is optional in practice. If omitted, SQLcl exports the app into a folder named after the app alias to the current directory.

```
apex export -applicationid 104
            -dir '/Users/smuench/Downloads'
            -exptype APEXLANG
```

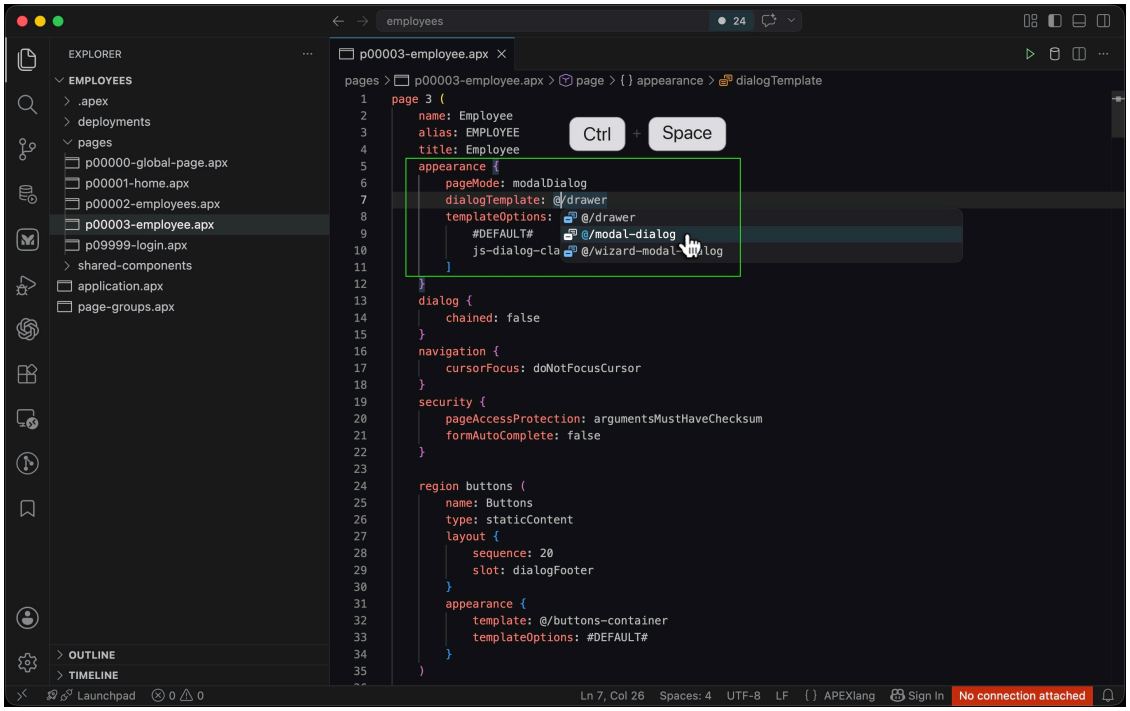
Figure 24-8 Previewing SQLcl Command to Export App 104 in APEXlang Format to a Parent Directory



Editing APEXlang in Visual Studio Code

The SQL Developer extension adds APEXlang assistance for `.apx` files. For example, in the figure below when the cursor is in the value of the `appearance.dialogTemplate` property of `page 3`, pressing (Ctrl) + (Space) shows the valid templates available for this page whose `pageMode` is `modalDialog`. You can switch the page from a drawer template to a modal dialog one by choosing `@/modal-dialog` from this list.

Figure 24-9 Using APEXlang Code Completion While Editing an .apx File

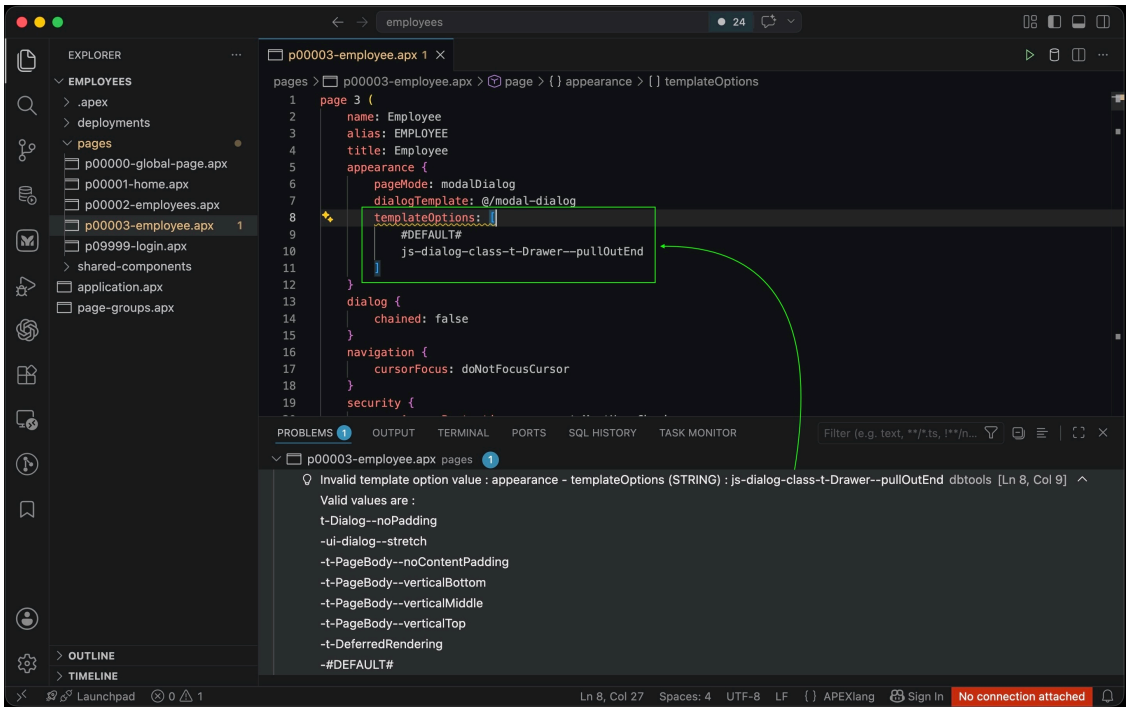


Copyright © 2026, Oracle and/or its affiliates.

Problems Panel Proactively Presents Errors

After changing the page to use the `modal-dialog` template, `templateOptions` gets a squiggly underline and the **Problems** panel shows “Invalid template option value…” This validation error indicates one of the existing values is no longer applicable to a modal dialog.

Figure 24-10 VS Code Problems Panel Shows APEXlang Validation Errors Proactively



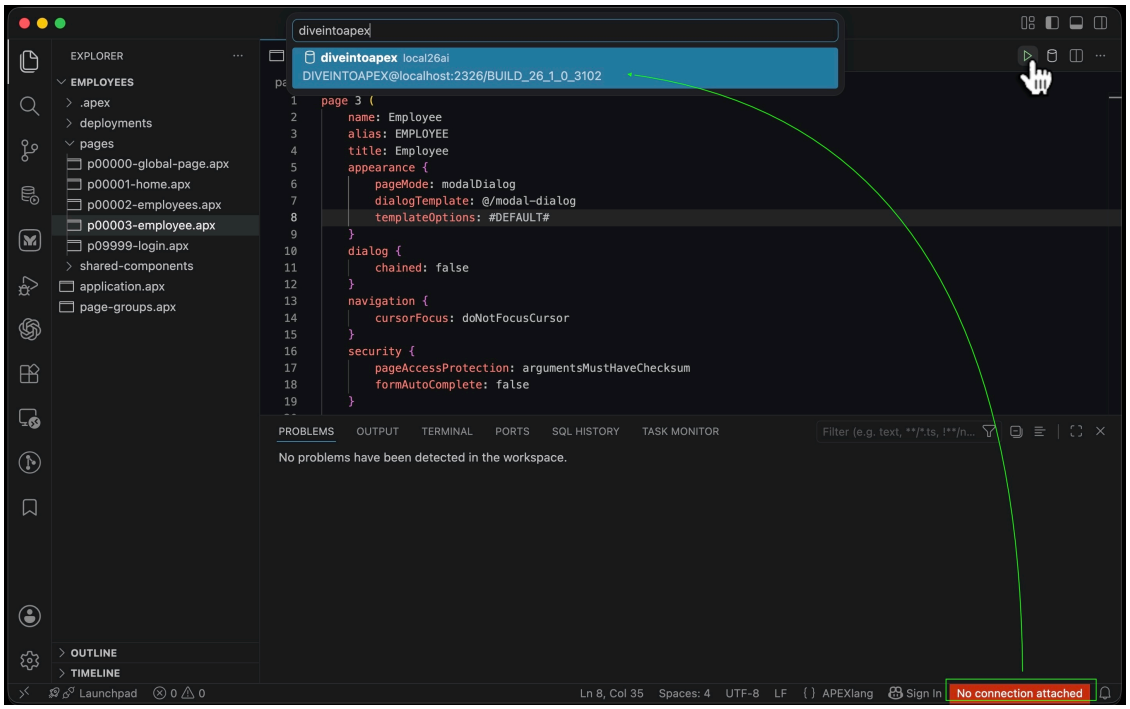
Importing APEXlang from VS Code into App Builder

After correcting the `templateOptions` value, optionally removing the square brackets for a one-element array, we’re ready to import the changed application into App Builder. As shown below, just click the “play” button in toolbar in the upper right of the editor. Since importing an APEX app requires a connection to your APEX workspace’s parsing schema, SQL Developer prompts you to choose one if needed.

As always, importing an app overwrites the current version in App Builder so ensure all App Builder changes are already reflected in the APEXlang you’re about to import.

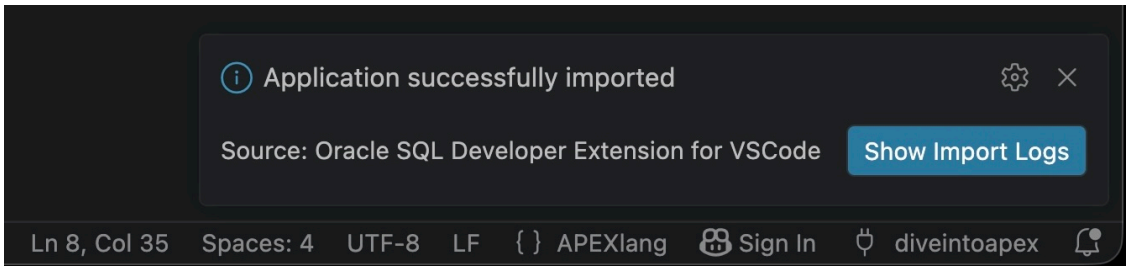
Copyright © 2026, Oracle and/or its affiliates.

Figure 24-11 Clicking the "Play" Button to Import Your APEXlang Application to App Builder



If the app validates successfully, then the import succeeds and a confirmation appears.

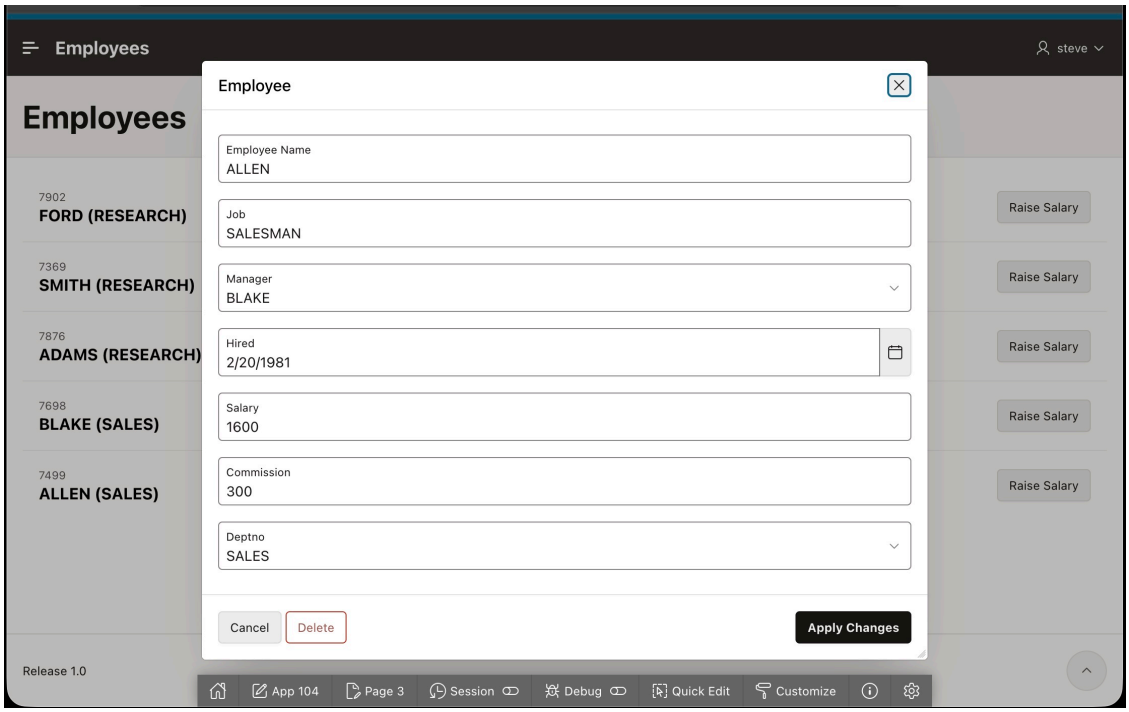
Figure 24-12 Confirmation of a Successful APEXlang Application Import in VS Code



Refreshing the application page in the browser, you see the Employee page has changed from a modal drawer to a modal dialog.

Copyright © 2026, Oracle and/or its affiliates.

Figure 24-13 Verifying the Results of Importing the Latest Application Version from VS Code



24.8 Using SQLcl with APEXlang

Use the SQLcl command line tool to export, import, and validate APEXlang applications.

Exporting and Importing APEXlang Apps with SQLcl

You can also export, import, and validate your APEX apps in APEXlang using the SQLcl command line.

To export an app in APEXlang format, connect to your workspace's parsing schema, use the `apex export` command, and indicate the application id and export type of `apexlang`.

```
SQL> apex export -applicationid 104 -exptype apexlang
```

To import an APEXlang application with SQLcl, connect to your workspace's parsing schema, use `apex import`, and indicate the input to load as a path to a directory or zip file:

```
SQL> apex import -input /Users/smuench/Downloads/employees
```

Importing an APEXlang app always validates it first, so any errors or warnings appear as with `apex validate` below. If no errors are detected, SQLcl proceeds. If the import is successful, you'll see:

```
SQL> apex import -input /Users/smuench/Downloads/employees

Importing application ID: 104 into workspace: DIVEINTOAPEX
```


Copyright © 2026, Oracle and/or its affiliates.

```
Import successful.
```

Validating an APEXlang Application with SQLcl

To validate an APEXlang application in SQLcl without importing it, connect to your workspace's parsing schema, use `apex validate`, and indicate the input to verify as a path to a directory or zip file:

```
SQL> apex validate -input /Users/smuench/Downloads/employees
```

You'll see `Validation successful`, otherwise any errors or warnings appear in the SQLcl console.

For example, suppose you edit the `pageItem P3_EMPNO` section of the `p00003-employee.apx` page to introduce three typos as shown below and save the file:

1. `@employee` -> `@employe` (remove the trailing `e`)
2. `number` -> `numbe` (remove the trailing `r`)
3. `primaryKey` -> `primaryKe` (remove the trailing `y`)

```
page 3 (
  ▒
   pageItem P3_EMPNO (
        type: hidden
        layout {
            sequence: 10
            region: @employe      <-- 1 --
            slot: regionBody
        }
        source {
            formRegion: @employee
            column: EMPNO
            dataType: numbe       <-- 2 --
            queryOnly: true
            primaryKe: true       <-- 3 --
        }
        security {
            sessionStateProtection: checksumRequiredSessionLeve
        }
    )
  ▒
)
```

The errors display immediately in the VS Code Problems panel, and `apex validate` shows them as follows in SQLcl:

```
SQL> apex validate -input /Users/smuench/Downloads/employees

APEXLang Compile Errors:

File: pages/p00003-employee.apx
Line: 123
Column: 12
```


Copyright © 2026, Oracle and/or its affiliates.

```
Type: LOV_NOT_FOUND
Error: Invalid LOV required parameter: source - dataType (string)
Valid parameters are:
-bfile
-blob
-boolean
-clob
-date
-intervalDayToSecond
-intervalYearToMonth
-number
-rowid
-sdoGeometry
-timestamp
-timestampWithLocalTimeZone
-timestampWithTimeZone
-varchar2

File: pages/p00003-employee.apx
Line: 117
Column: 12
Type: REFERENCE_NOT_FOUND
Error: Reference not found: @employe

File: pages/p00003-employee.apx
Line: 125
Column: 12
Type: INVALID_PROPERTY
Error: Invalid property: primaryKe
```

To collect any errors or warnings for later reference, use the SQLcl `spool` command to capture the console output into a file.

```
SQL> spool employees_errors.log

SQL> apex validate -input /Users/smuench/Downloads/employees

APEXLang Compile Errors:

File: pages/p00003-employee.apx
Line: 123
Column: 12
Type: LOV_NOT_FOUND
Error: Invalid LOV required parameter: source - dataType (string)

SQL> spool off
```

Metadata-Driven Validation

The APEXlang compiler knows which APEX components are available as well as what properties and child components each one has. It validates property values, nested components, and constraints on when each applies. For example, one type of `region` component like an `interactiveReport` has nested `column` definitions, but a `form` type region doesn't. Similarly, a `chart` type region has a `zoomAndScroll` property, but a `classicReport` does not.

Copyright © 2026, Oracle and/or its affiliates.

Page Designer has always reacted dynamically to show and hide appropriate properties and child components. Now the APEXlang compiler leverages the same details to strictly validate your app components and pages.

This information about APEX's app metadata is called meta-metadata (MMD). Over time, it will grow to reflect new APEX features, so each app stores the current APEX meta-metadata version used to export it. This lets the APEXlang compiler validate app files correctly. It also ensures SQL Developer offers accurate code completion in VS Code.

Your app's `.apex/apexlang.json` file records this MMD version. It's important this version remain unchanged until a future APEX version might export your app with a new value. Do not modify it by hand.

```
{
  "mmdVersion" : "26.1.0+3102"
}
```

Validating APEXlang Without a Connection

While *importing* an APEXlang application requires access to your workspace's parsing schema, you can validate an APEXlang application without a connection. For example, if you run SQLcl with the `/nolog` option to avoid a login, you can still run `apex validate` to compile your application and report errors and warning, if any. This can be useful to validate an application you have downloaded from an APEX instance without direct SQLNet connectivity, or useful for an AI coding agent to discover its own errors to fix them autonomously.

For example, if you are in the root directory of an APEXlang application you want to validate, you can do the following, using the dot (`.`) to represent the current directory as the input to validate.

```
$ sql /nolog
SQL> apex validate -input .
APEXLang Compile Errors:
File: pages/p10020-dashboard.apx
Line: 499
Column: 4
Type: DUPLICATED_COMPONENT
Error: Duplicated component within parent scope. For properties:
identification.buttonName: VIEW_ACTIVITY_BY_USER

File: pages/p10020-dashboard.apx
Line: 526
Column: 4
Type: DUPLICATED_COMPONENT
Error: Duplicated component within parent scope. For properties:
identification.buttonName: VIEW_ACTIVITY_BY_USER
```

24.9 APEXlang Inside App Builder

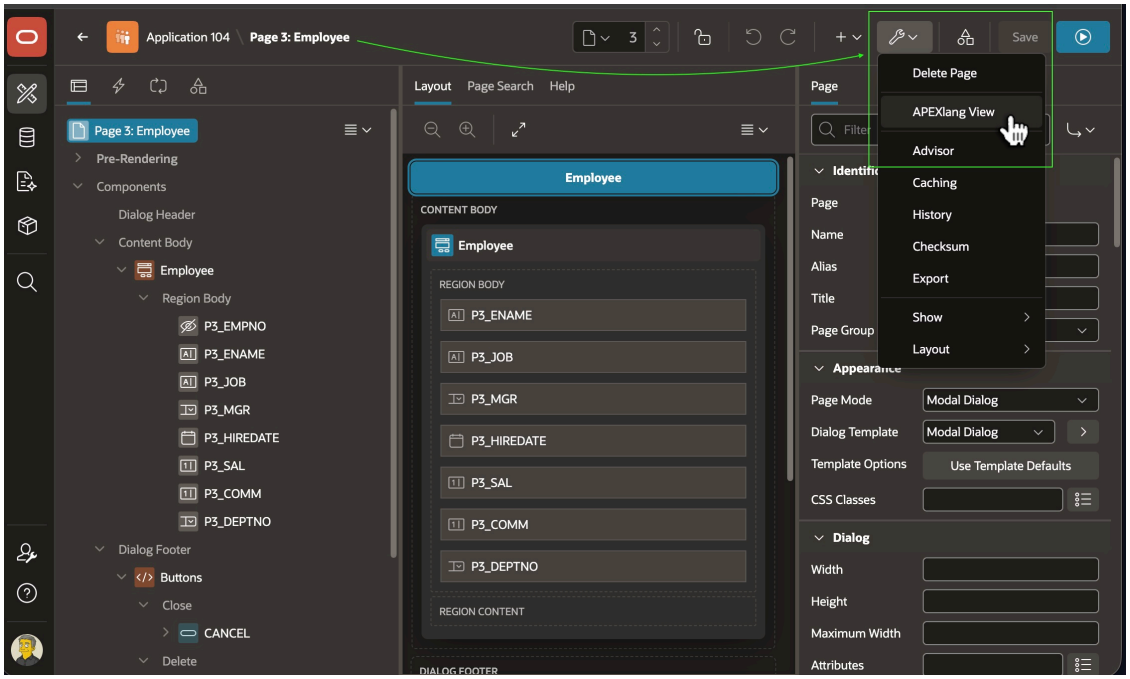
View any page in APEXlang from Page Designer, and efficiently review working copy diffs interactively.

Even if you continue doing all APEX development in App Builder, you still benefit from the readability of APEXlang: it replaces the readable YAML format when reviewing changes.

Copyright © 2026, Oracle and/or its affiliates.

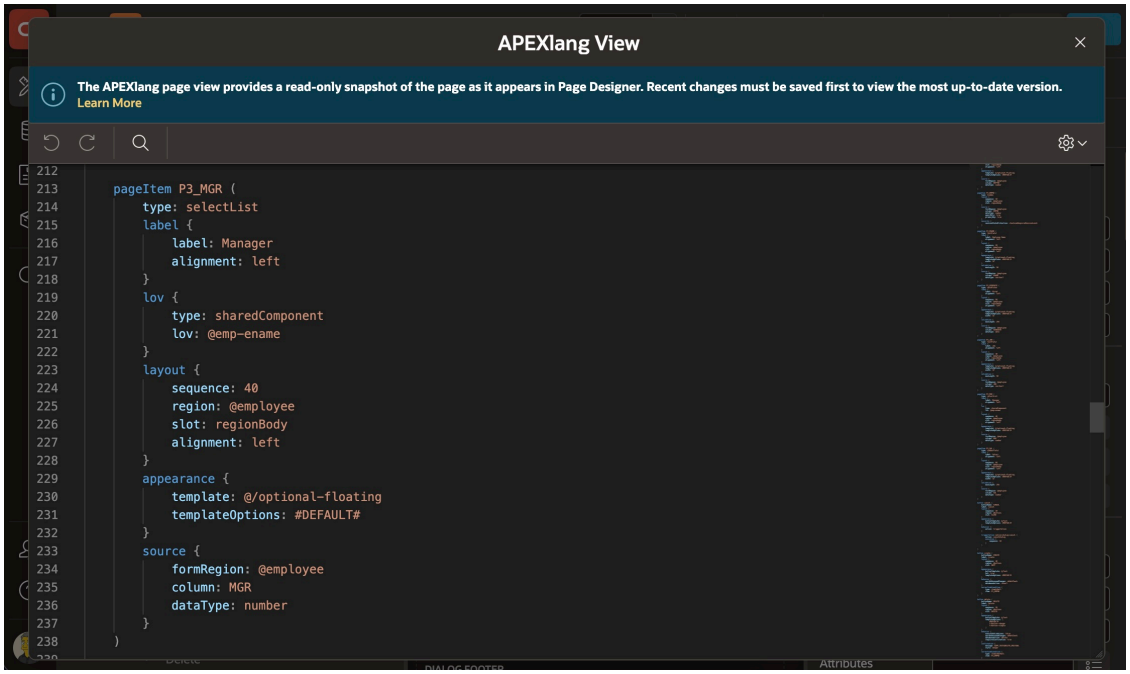
For example, while working on any page in Page Designer, see an APEXlang view of the page with the new APEXlang View menu option shown below.

Figure 24-14 Accessing an APEXlang View of the Current Page in Page Designer



This lets you review and search the page's APEXlang whenever useful.

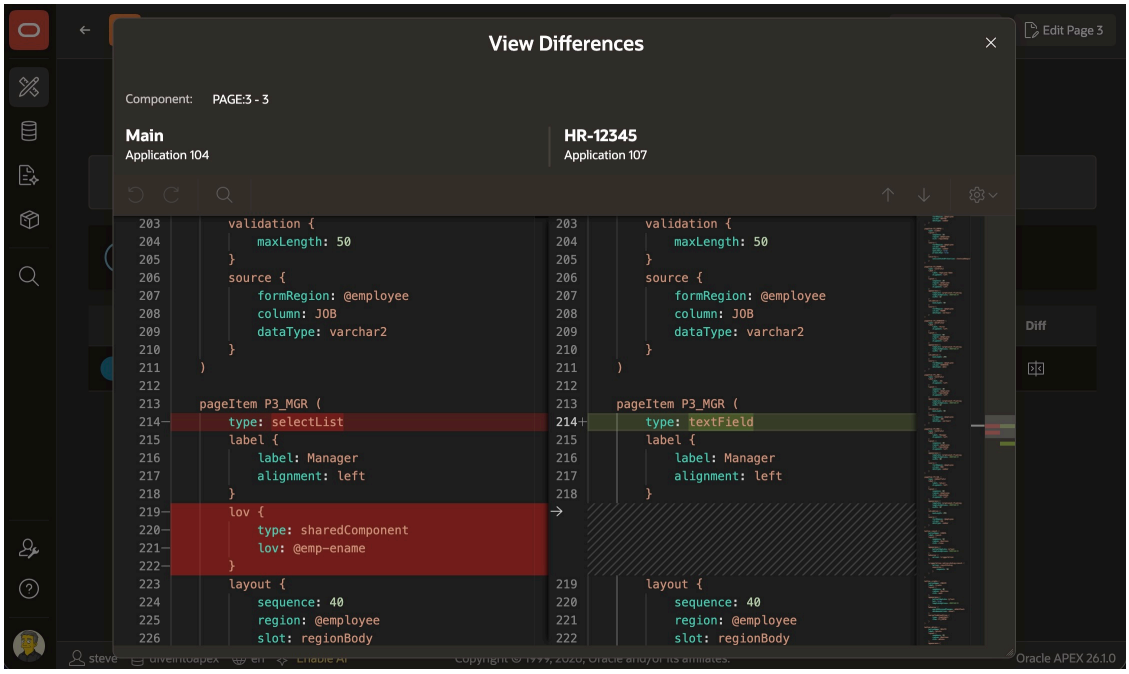
Figure 24-15 Reviewing the APEXlang Representation of the Current Page in Page Designer



Copyright © 2026, Oracle and/or its affiliates.

In addition, when reviewing changes between a working copy and the main application, you now see the diffs between APEXlang artifacts. For example, as shown below, it's easy to see that the `P3_MGR` page item changed from a `selectList` to a `textField`, losing its related `lov` in the process.

Figure 24-16 Reviewing APEXlang Diffs When Comparing Working Copy with Main App



Copyright © 2026, Oracle and/or its affiliates.

25

Automating Business Processes

Automate business processes with APEX Workflows to coordinate work and show progress clearly.

A business process is a sequence of steps your organization follows to get something done. It can involve the work of multiple colleagues, each responsible for an activity like completing a job or approving a request. The tasks may also include interacting with external applications, collecting data, researching historical information, notifying stakeholders, or waiting for an event to occur or a period of time to elapse.

When the process is manual, it can be hard for everyone involved to understand questions like:

- *What processes are in progress?*
- *Where are we in the process on this one?*
- *Who are we waiting on for that one?*

APEX Workflows let you *automate* business processes, improving efficiency for the people involved and enhancing visibility into the process for better decision-making.

- Automating a Process with Workflow
  Model business processes visually and show progress in your app as they run.
- Exploring Patient Onboarding Example
  Explore how APEX Workflow automates patient onboarding at the fictitous Woods Clinic.
- Automating Business Entity Lifecycle Logic
  Use a trigger and automation to wake workflow lifecycle logic on business entity status changes.
- Maintaining Workflow Processes
  Maintain workflows and task definitions through import, debugging, deployment, and cleanup.

25.1 Automating a Process with Workflow

Model business processes visually and show progress in your app as they run.

You can *automate* a business process of any complexity with an APEX Workflow. It runs in the background, can be short-lived or long-running, and can perform any kind of actions required, including useful highlights like:

- working with relational data, spatial data, graph data, JSON, XML, and more,
- adding a task to one or more user's task lists and waiting for its completion,
- predicting outcomes based on historical data using machine learning,
- leveraging AI models for semantic similarity searching using vector search,
- integrating with Generative AI services and other REST APIs,
- notifying users via email or push notification,
- generating pixel-perfect PDF reports, or

Copyright © 2026, Oracle and/or its affiliates.

- waiting for events to occur like data changes or a certain amount of time to elapse.

In App Builder, you define the sequence of activities in a visual diagram that anyone can understand at a glance. A workflow specifies the *blueprint* for a business process, and the APEX Engine manages each *concrete example* of every workflow. Using the Create Page Wizard, you can generate a workflow console page to show users "in flight" processes. The companion workflow detail page shows the status of a workflow instance complete with an *annotated* diagram so everyone involved stays aligned.

- Using the Diagram and Activities Palette
  Use the Workflow Designer to add activities, connect them, and edit their properties.
- Surveying the Native Activities
  Understand the extensible set of native workflow activities.
- Connecting and Routing Activities
  Learn the basic diagramming moves for creating, connecting, and routing workflow activities.
- Resolving Errors Before Saving a Workflow
  Resolve any Workflow Designer errors before saving your changes.
- Associating a Table with a Workflow
  Associate a workflow with a table or query so activities can reference related row data.
- Retrieving Data for an Activity
  Use activity-level Additional Data to query one row of values the current activity needs.
- Defining Parameters and Variables
  Use parameters and variables for values a workflow cannot query or must capture at a point in time.
- Understanding Background Execution
  Workflows run in a background session without page session state or a logged-in user.
- Getting Approval or User Input in a Workflow
  Wait for user approval or input by creating a task or sending a notification.
- Identifying Workflow Participants
  Assign workflow owners and administrators using case-sensitive usernames.
- Listing Workflows and Viewing Details
  Create Workflow Console pages to list the workflows a user owns, administers, or initiated.

25.1.1 Using the Diagram and Activities Palette

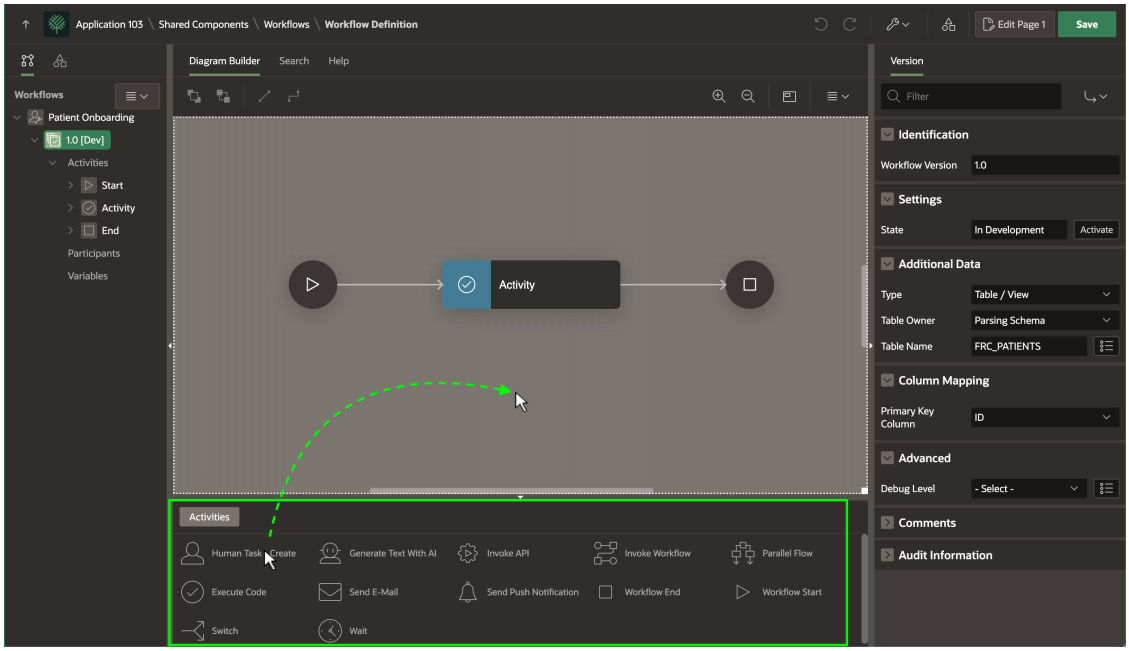
Use the Workflow Designer to add activities, connect them, and edit their properties.

The Diagram Builder surface occupies the bulk of the Workflow Designer layout. The *Activities* palette below contains actions you can drag onto the diagram as workflow steps. Connection arrows between activities define the sequence the APEX engine follows. The *Workflows* tab on the left presents diagram elements organized in a tree. The Property Editor on the right lets you inspect and change settings for activities and connections.

A workflow begins with the *Start* activity shown below as a circle with the "play" triangle icon. If the workflow contains *Switch* activities to make decisions, there are multiple "paths" from the start activity to a *Workflow End* activity that appears as a circle with the "stop" square icon. While there is always only one *Start* activity, to simplify diagram layout you can include as many *End* activities as needed.

Copyright © 2026, Oracle and/or its affiliates.

Figure 25-1 Activities Palette Shows Available Types You Can Drag onto Diagram



25.1.2 Surveying the Native Activities

Understand the extensible set of native workflow activities.

Your workflows can use the following native activities:

- *Invoke API* to call PL/SQL or REST APIs
- *Switch* to decide between alternative paths
- *Human Task - Create* to add an approval or action task for a user to complete before proceeding
- *Send E-Mail* to notify users for any reason, including relevant data
- *Wait* to pause until an event occurs or a period of time elapses
- *Execute Code* to run inline PL/SQL
- *Send Push Notification* to alert a user on smartphone, tablet, or desktop with a short message
- *Invoke Workflow* to call another workflow like a subroutine and wait for it to return
- *Parallel Flow* to run multiple sequences of activities in parallel and wait for all to complete
- *Generate Text with AI* to produce textual output based on a system prompt and custom data, and
- *Server Side Geocoding* to turn an address into coordinates to show on a map.

When none of the built-in activities meets your needs, you can install a community-developed workflow activity plug-in or create one yourself. Just create a process type plug-in with its *Workflow Activity* checkbox checked in the *Supported For* section.

Copyright © 2026, Oracle and/or its affiliates.

> **Tip**
>
> *Server Side* Geocoding is currently only available in Oracle APEX on Autonomous AI Database. In contrast, the *Geocode Address* dynamic action works in all APEX installations. It cooperates with a *Geocoded Address* page item type to geocode an address in a page.

25.1.3 Connecting and Routing Activities

Learn the basic diagramming moves for creating, connecting, and routing workflow activities.

Creating organized, readable diagrams is straightforward after learning and practicing the basics of creating, connecting and routing activities.

- Adding an Activity from the Palette
  Add workflow activities by dragging them from the palette onto the canvas.
- Naming a New Activity
  Name each new activity clearly in the Property Editor.
- Changing a Connection's Target Activity
  Reattach a connection by dragging its arrowhead or starting dot to another activity.
- Fine-Tuning Line Slope with Anchor Dots
  Adjust a connection line’s slope by moving its source or target anchor dots.
- Moving Connections with Grab Handles
  Move horizontal or vertical connection segments with their grab handles.
- Adding a Connection Between Activities
  Connect activities by dragging a plus sign from the source activity to the target activity.
- Bending a Connection with Elbow Dots
  Bend a connection by adding, moving, or removing elbow dots.
- Adding Next Activity from Existing One
  Add and connect the next activity directly from an existing activity.
- Configuring a Switch Connection
  Configure a Switch connection with a condition and name for its decision outcome.
- Inserting Activity Between Existing Ones
  Insert an activity into an existing connection.
- Moving a Group of Activities
  Move selected activities as a group while their connections follow.
- Bringing Connections in Front of Activities
  Ensure arrowheads stay visible over activities in the runtime workflow progress diagram.

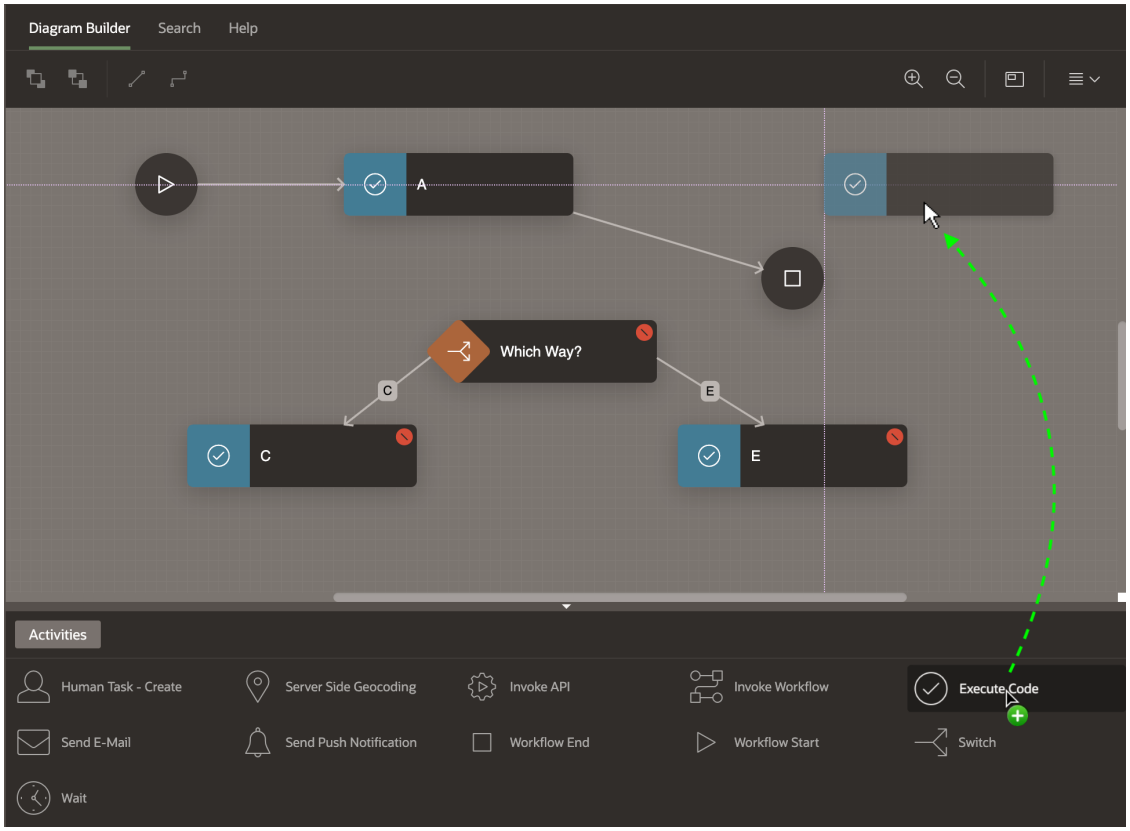
25.1.3.1 Adding an Activity from the Palette

Add workflow activities by dragging them from the palette onto the canvas.

To add a new activity to the workflow canvas, drag it from the *Activities* palette and drop it onto the canvas. As shown below, guidelines appear to help you align it to existing activities.

Copyright © 2026, Oracle and/or its affiliates.

Figure 25-2 Dragging an Activity from the Palette and Dropping onto the Canvas



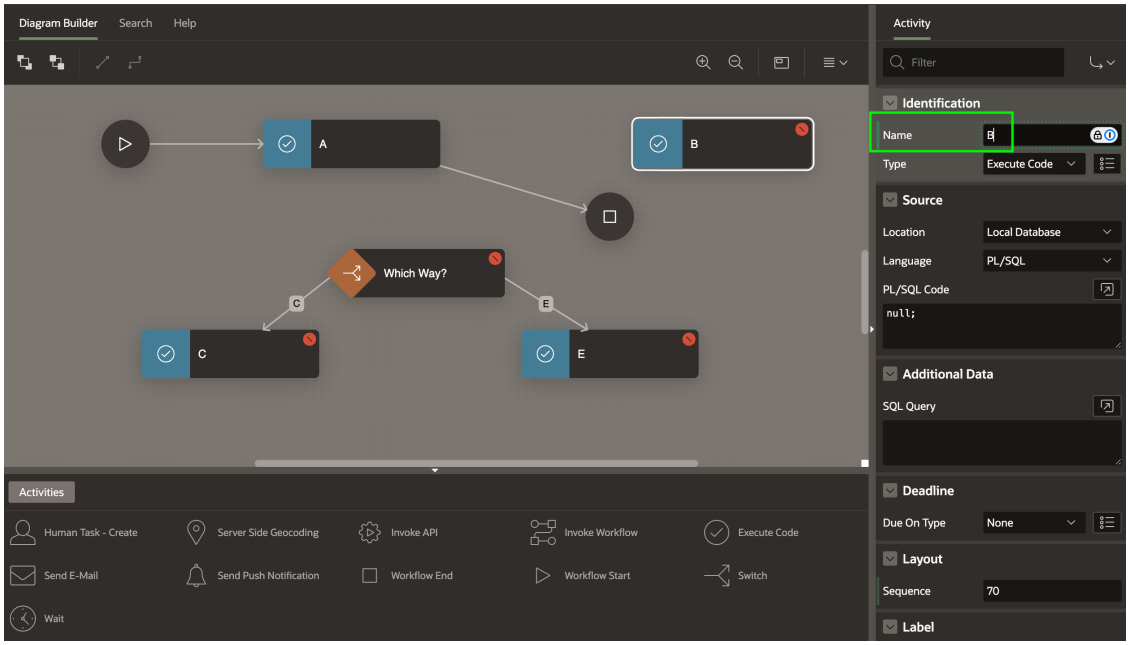
25.1.3.2 Naming a New Activity

Name each new activity clearly in the Property Editor.

By default, a new activity is named *New*, so use the *Property Editor* as shown below to give it a more meaningful name. For this exercise, you just name it *B*.

Copyright © 2026, Oracle and/or its affiliates.

Figure 25-3 Naming a New Activity in the Property Editor



25.1.3.3 Changing a Connection's Target Activity

Reattach a connection by dragging its arrowhead or starting dot to another activity.

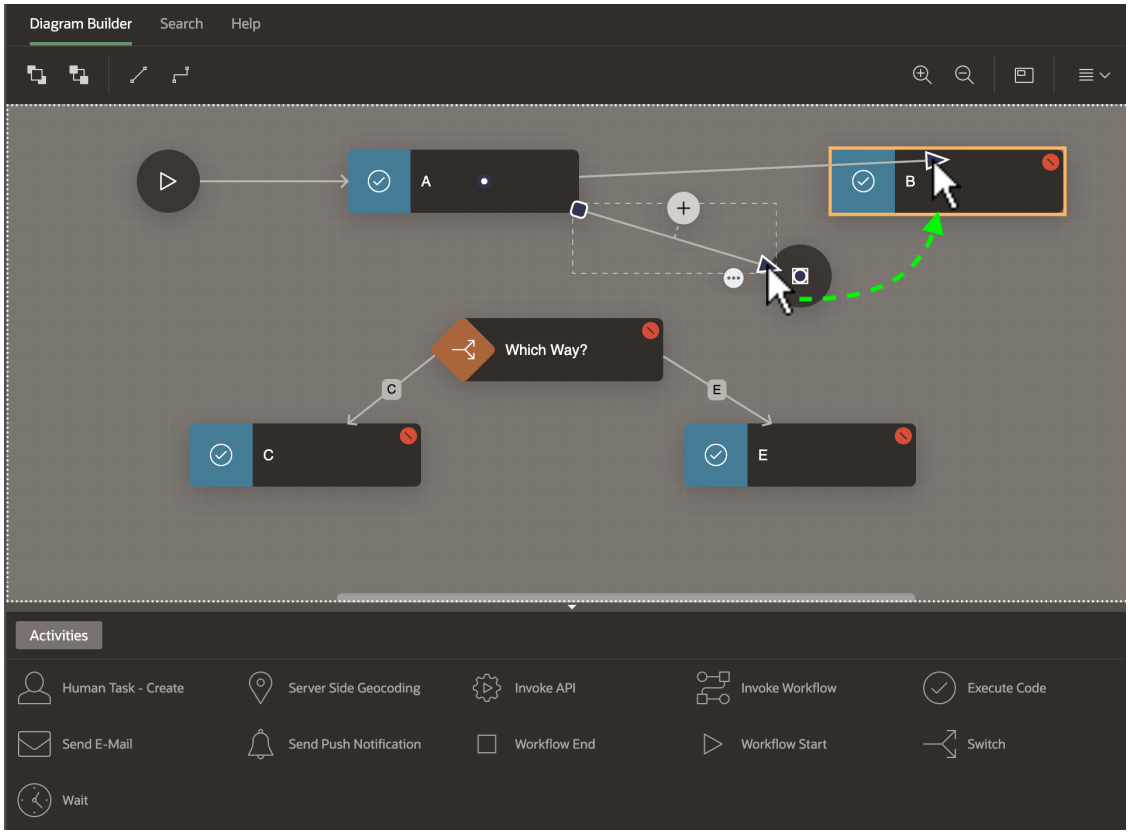
To reattach an existing connection line, as shown below, drag its arrowhead onto the new activity it should connect to. An orange outline helps you see when you are within the visual boundary of a new target activity. Release the mouse button to complete the operation. Now activity *A* continues to *B*.

The figure below shows a diagram with a start activity connected to an activity named *A*. *A* then connects to the End activity. A B activity is in the upper right corner, initially with no inbound or outbound connections. A *Which Way?* switch activity lies in the center with one connection to C and another connection to E.

You can change a connection's *origin* as well by dragging its *starting* dot onto a new *source* activity. In the figure, the mouse drags the arrowhead previously connecting *A* to the End activity to drop it onto *B*. The result is that *A* then connects to *B* instead.

Copyright © 2026, Oracle and/or its affiliates.

Figure 25-4 Reattach a Connection by Dragging its Arrowhead or Starting Dot



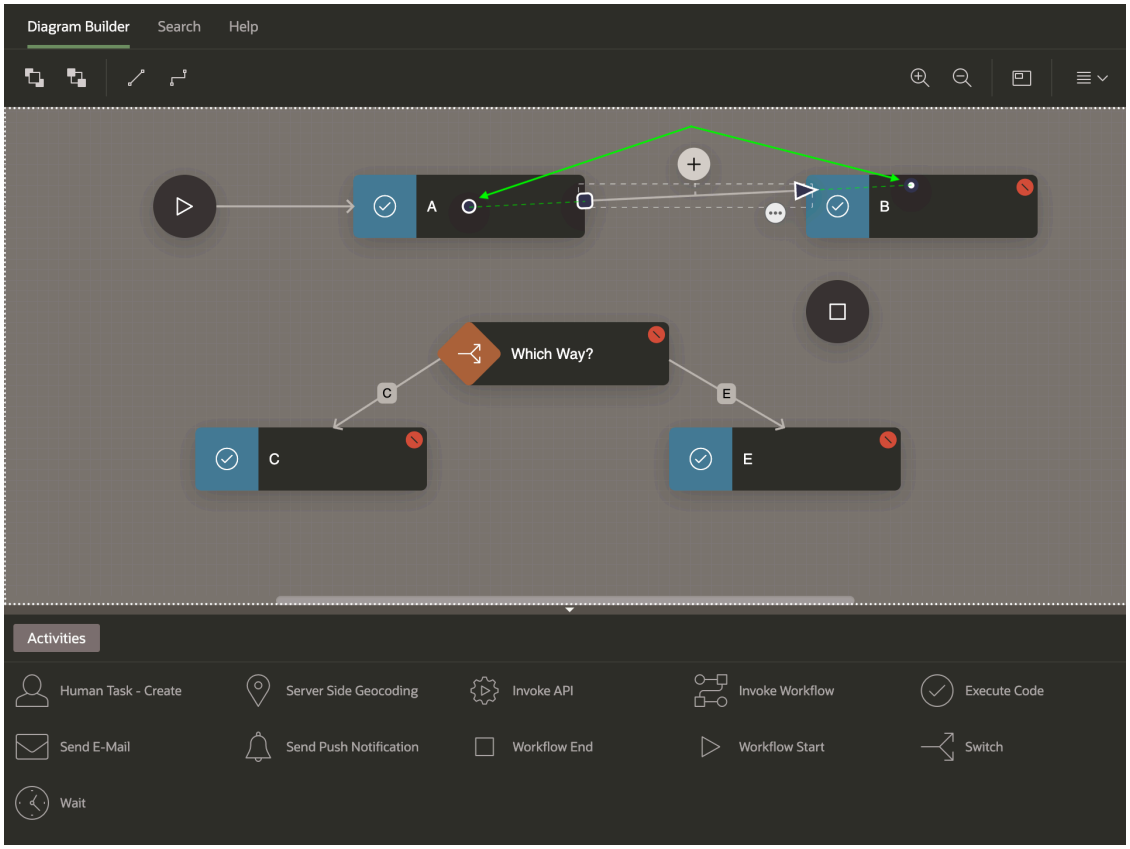
25.1.3.4 Fine-Tuning Line Slope with Anchor Dots

Adjust a connection line's slope by moving its source or target anchor dots.

When you select a connection line, anchor dots appear inside the boundaries of the source and target activities as shown below. The anchor dots determine the slope of the connection line between source and target. To help your eye see this concept, the figure below includes *additional* dotted line segments. Notice in the *other* connection lines that only the segment from the edge of the source activity to the edge of the target one is *actually* visible.

Copyright © 2026, Oracle and/or its affiliates.

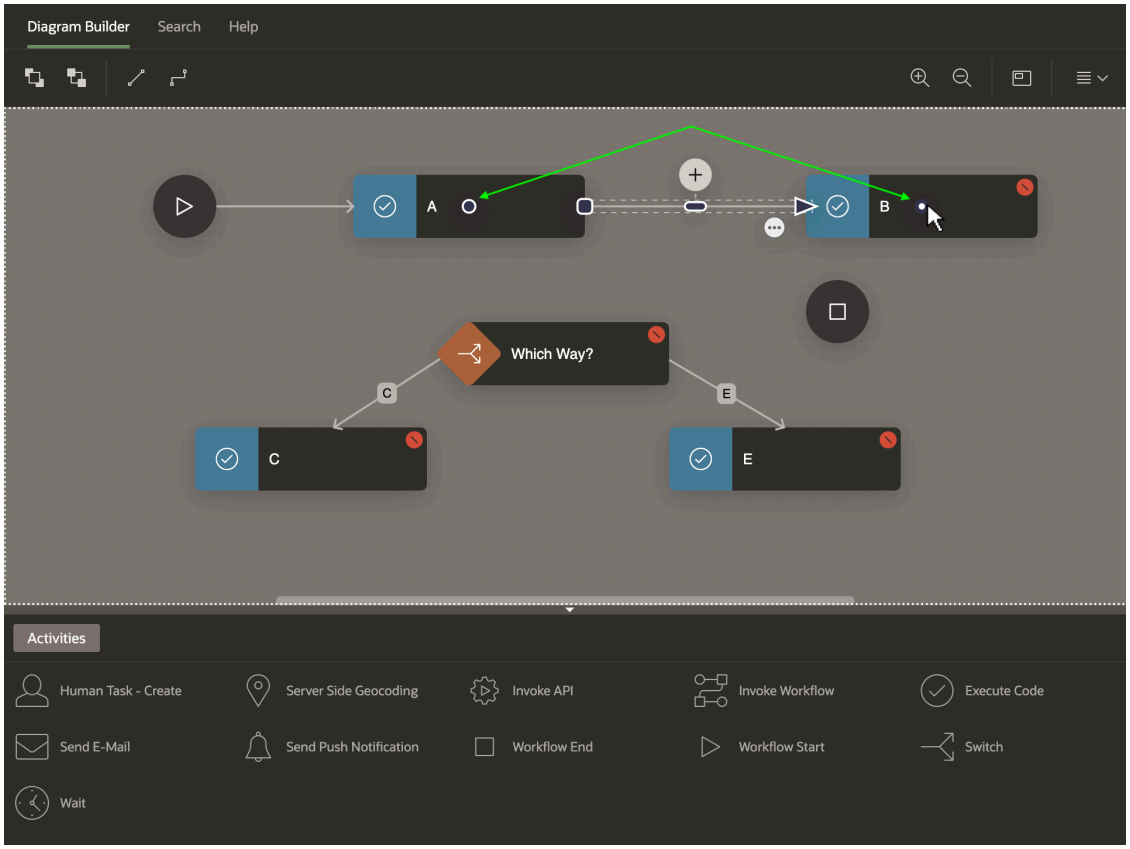
Figure 25-5 Connection Anchor Dots Determine Line Slope



You can move the anchor dot in the source or target activity anywhere within the boundary of the respective shape. For example, as shown below, by moving the anchor dot within the *B* activity's shape, you can change the slope of the connection line to be horizontal. Also notice the grab handle appears on the now-horizontal line.

Copyright © 2026, Oracle and/or its affiliates.

Figure 25-6 Reposition Anchor Dots in Source or Target Activities to Change Slope



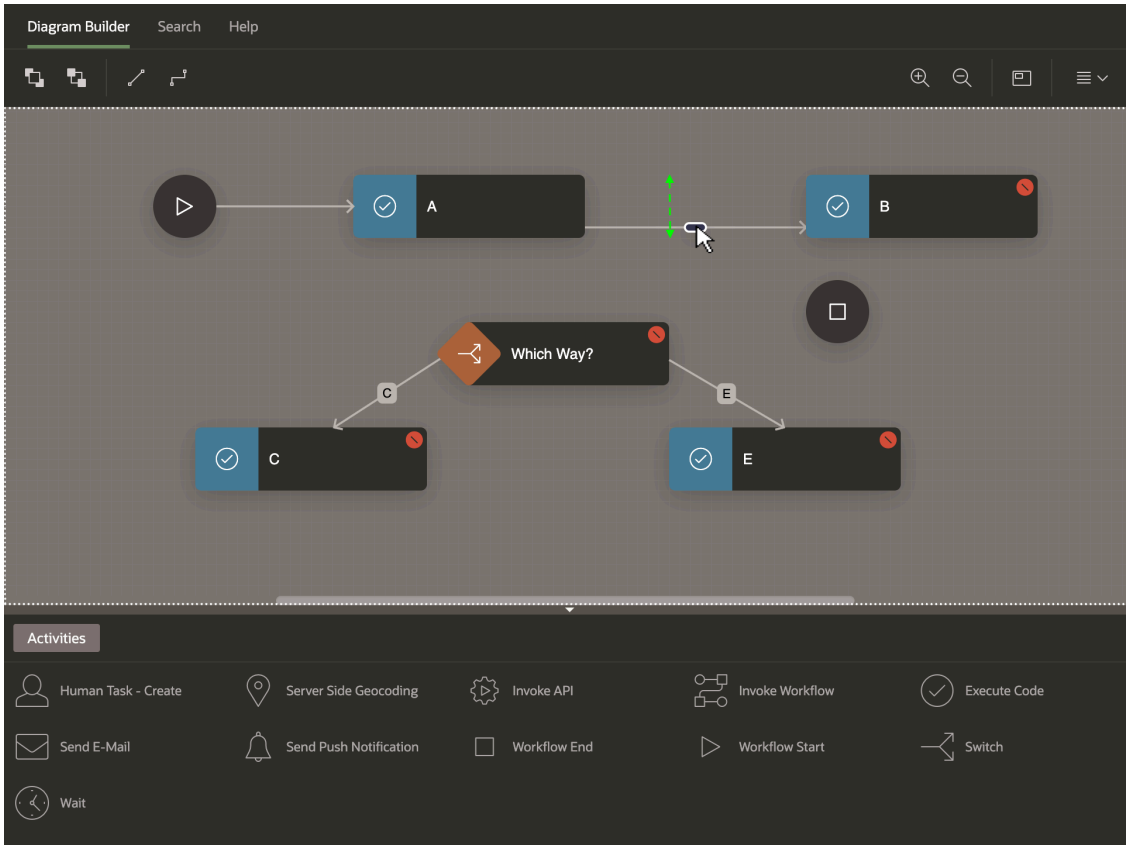
25.1.3.5 Moving Connections with Grab Handles

Move horizontal or vertical connection segments with their grab handles.

Grab handles appear on any segment of a connection line that is horizontal or vertical, and, as shown below, you can use the drag handle to move the line segment up/down. The same goes for using the grab handle to move a vertical segment left and right more easily.

Copyright © 2026, Oracle and/or its affiliates.

Figure 25-7 Drag Handles on Horizontal and Vertical Segments of Selected Connection



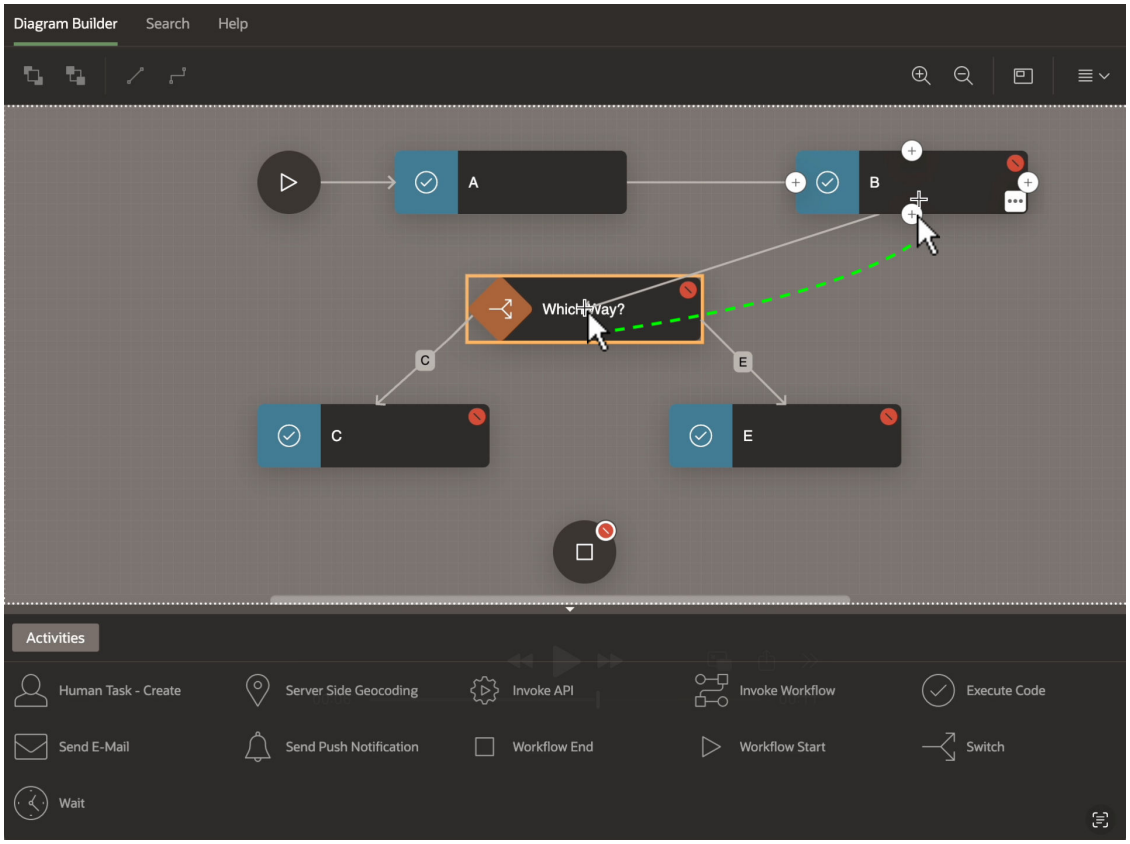
25.1.3.6 Adding a Connection Between Activities

Connect activities by dragging a plus sign from the source activity to the target activity.

To connect two activities, hover over the source activity until the four plus signs appear. Then, as shown below, drag one of these plus signs and drop it into target activity rectangle. The orange highlight confirms the target activity. Your pointer position *within* the target activity shape when you release the mouse button is meaningful. It determines the *initial* placement of the target's anchor dot. You can then adjust the position of a selected connection's anchor dots anytime.

Copyright © 2026, Oracle and/or its affiliates.

Figure 25-8 Adding a New Connection Between Activities



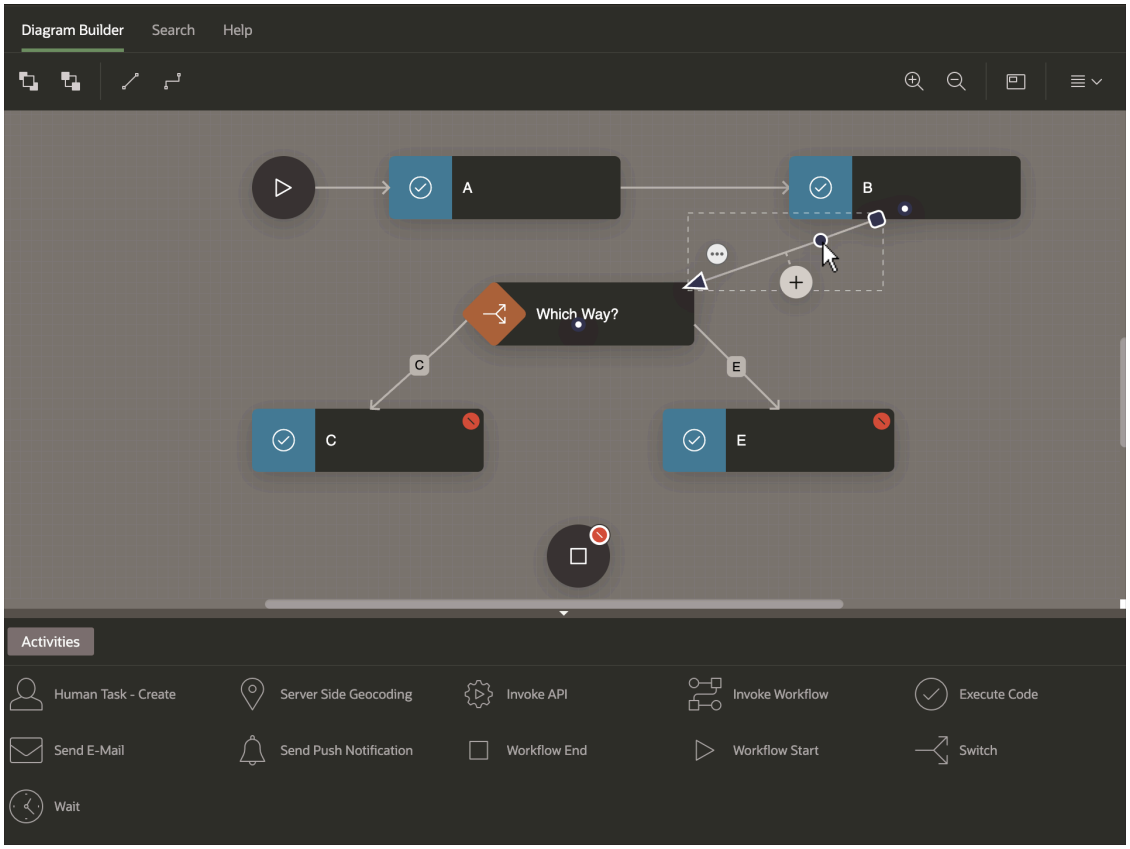
25.1.3.7 Bending a Connection with Elbow Dots

Bend a connection by adding, moving, or removing elbow dots.

To bend a line, start by selecting it. Then, as shown below, single-click on the line where you want to place the elbow dot.

Copyright © 2026, Oracle and/or its affiliates.

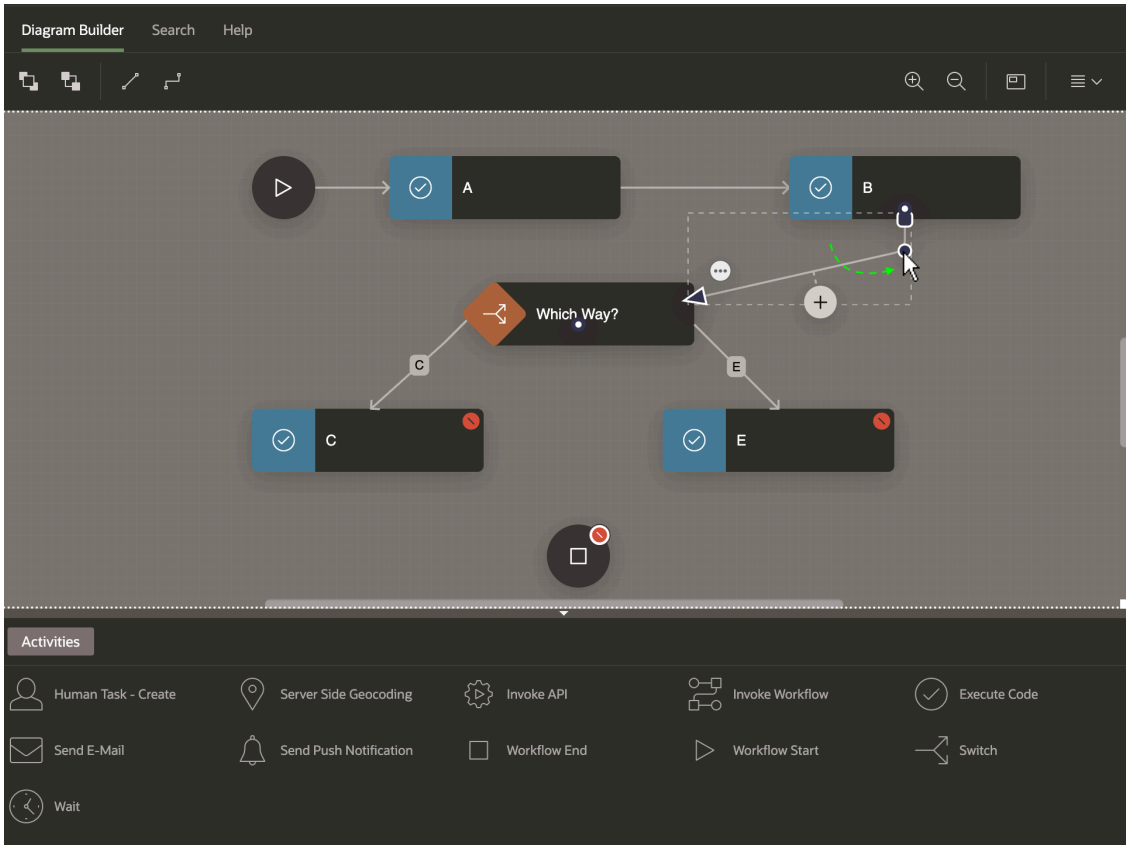
Figure 25-9 Select a Line Then Single Click to Add an Elbow Dot



After you've added an elbow dot, drag it into a new position as shown below.

Copyright © 2026, Oracle and/or its affiliates.

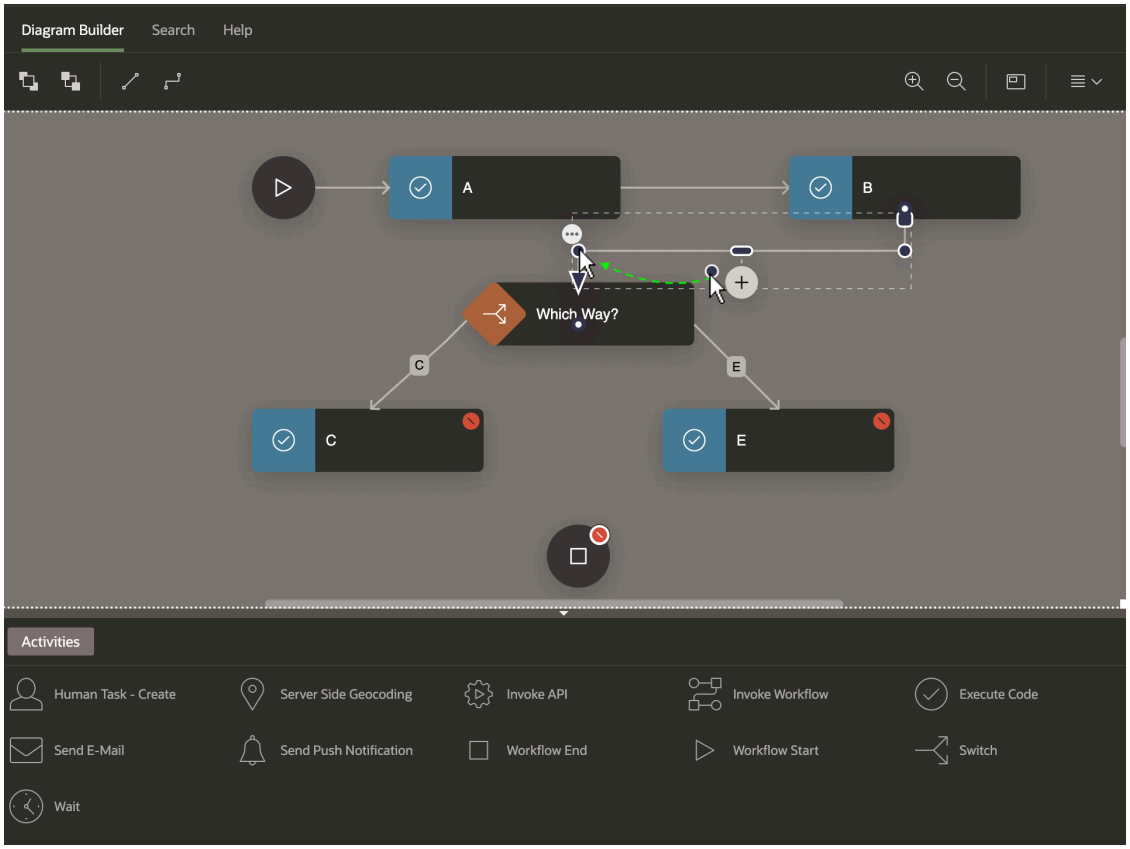
Figure 25-10 Repositioning an Elbow Dot with Drag and Drop



As shown below, you can repeat the process of adding an elbow dot and dragging it to a new position to achieve whatever clean line routing feels right. If necessary, you can reposition the source or target anchor dots to fine-tune the slope of the initial or terminal connection line segments. Notice when your connection contains horizontal or vertical segments, grab handles appear that you can use to adjust each one separately in a single movement. To *remove* an elbow dot, select the connection line and then *double-click* on the dot.

Copyright © 2026, Oracle and/or its affiliates.

Figure 25-11 Add Additional Elbow Dots and Reposition Them to Create Any Routing



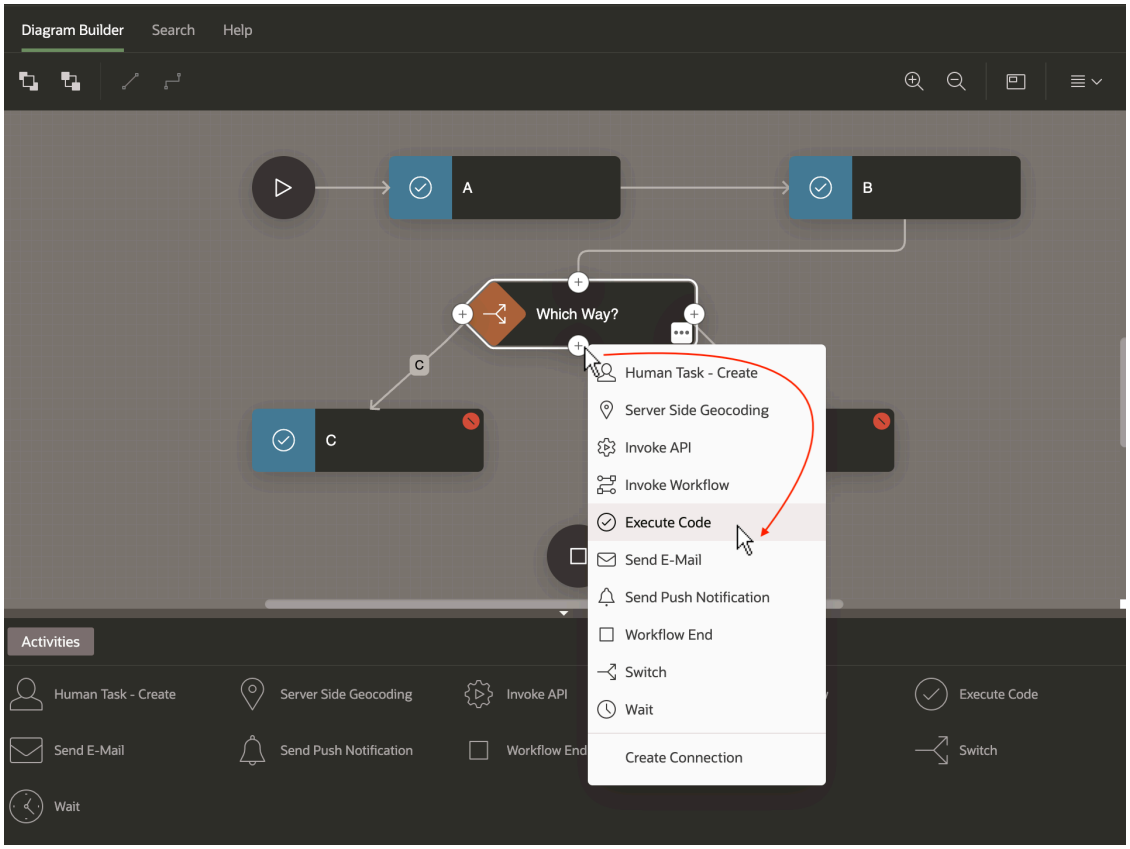
25.1.3.8 Adding Next Activity from Existing One

Add and connect the next activity directly from an existing activity.

Click on one of the plus icons on an existing activity to create an activity that happens *next* in the flow. As shown below, a context menu presents available activities to choose from.

Copyright © 2026, Oracle and/or its affiliates.

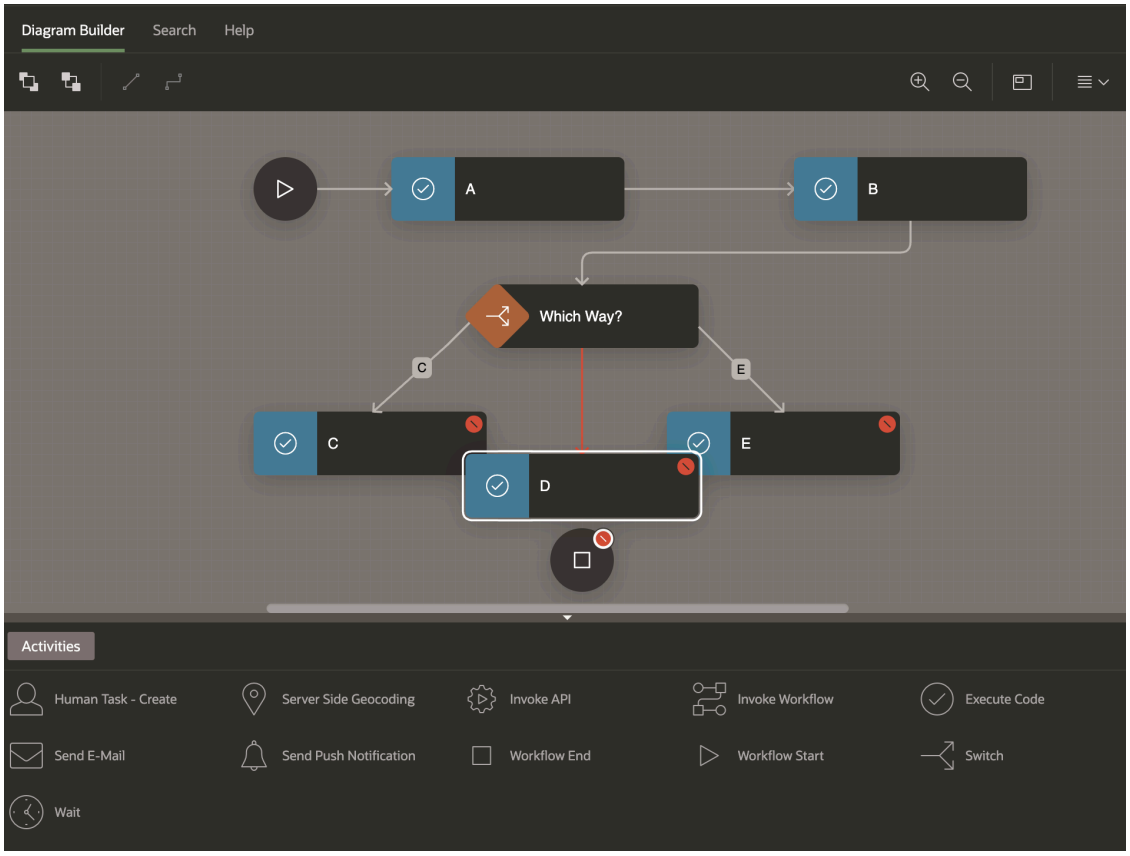
Figure 25-12 Adding a Next Activity From a Current One



Your chosen activity type appears on the diagram already connected with a connection from the source you clicked on to initiate the process. By default, it is named *New*, so use the Property Editor to change its name to something more meaningful. Here, you name it *D*. Since the source activity is a *Switch*, the new connection line is red to let you know some configuration adjustment is needed.

Copyright © 2026, Oracle and/or its affiliates.

Figure 25-13 Next Activity and Connection Appear on the Diagram



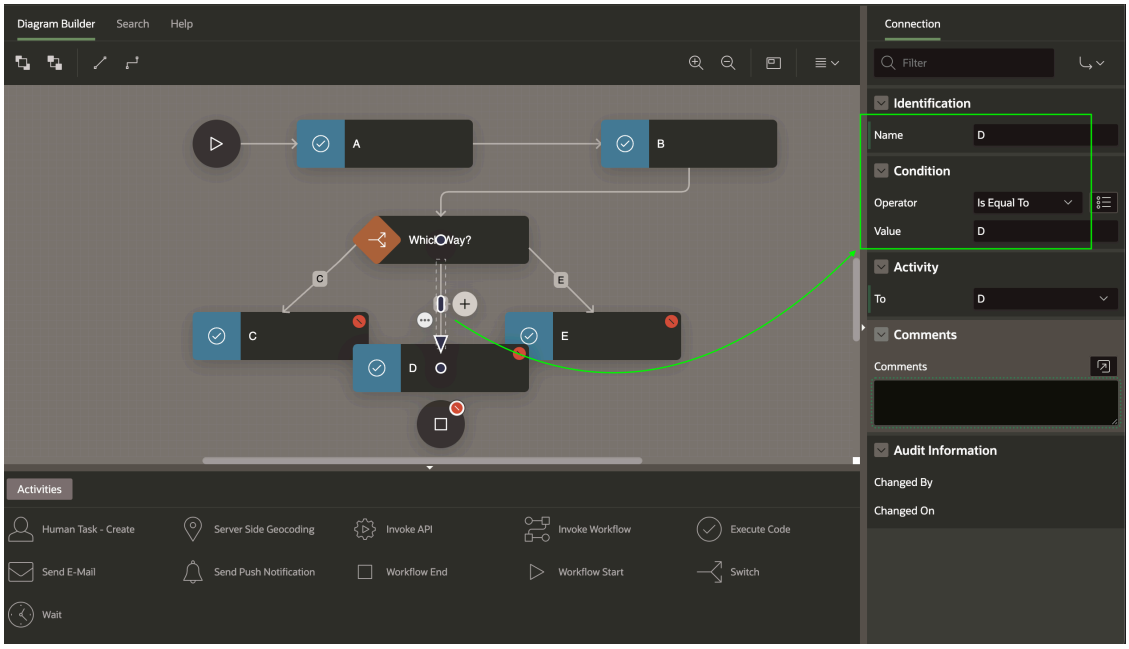
25.1.3.9 Configuring a Switch Connection

Configure a Switch connection with a condition and name for its decision outcome.

In addition to its descriptive name, a connection originating from a *Switch* activity also identifies which decision outcome it represents. A red connection line indicates this detail is missing. Select the connection line originating from the *Switch* and, as shown below, configure its condition and give it a meaningful name. The *Which Way?* switch activity in this example decides based on a workflow variable. This particular *D* connection line represents the case when the variable *Is Equal To* the value `D`. Here the connection name *and* the target activity are both named *D* for simplicity. Rest assured your workflows can use any connection and activity names that best communicate the business process flow to developers and end users.

Copyright © 2026, Oracle and/or its affiliates.

Figure 25-14 Configuring Properties on Connection Originating from a Switch



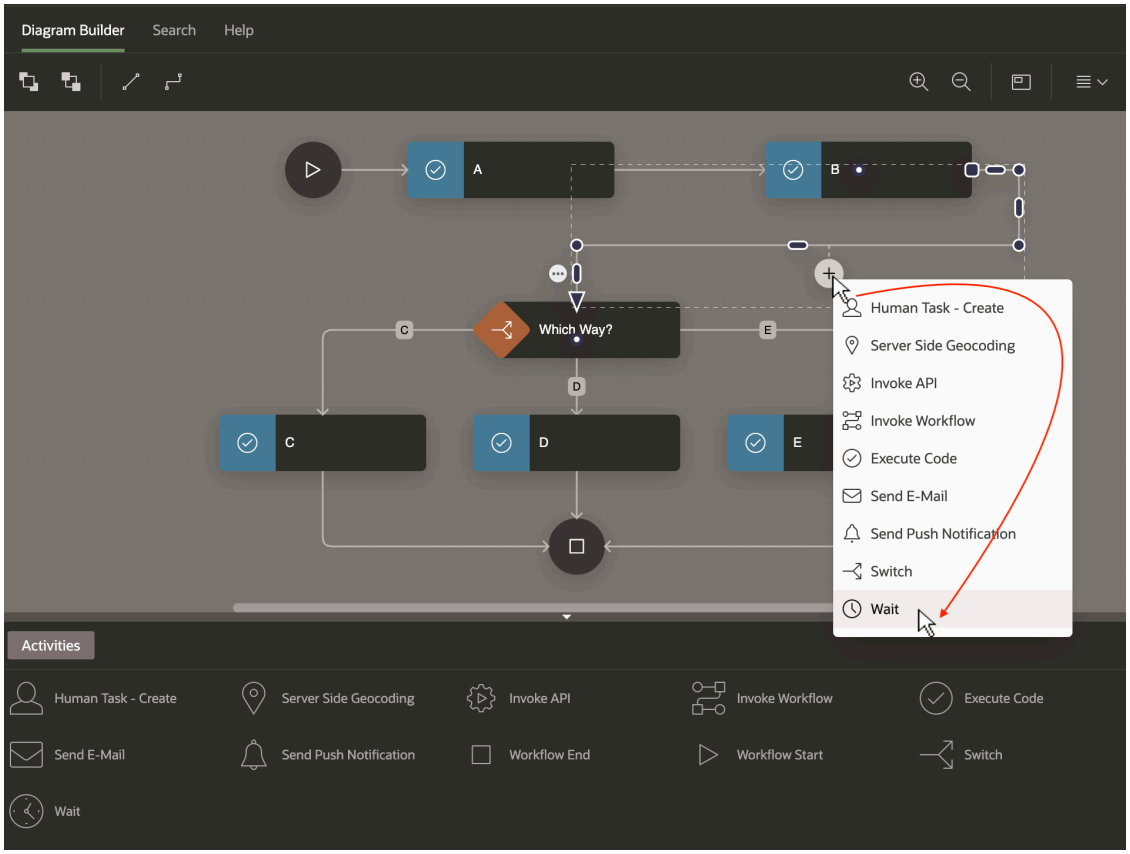
25.1.3.10 Inserting Activity Between Existing Ones

Insert an activity into an existing connection.

To add a new activity between two existing ones, select the connection line between them and click the plus in the middle of the line. As shown below, a context menu appears letting you choose the activity type to create.

Copyright © 2026, Oracle and/or its affiliates.

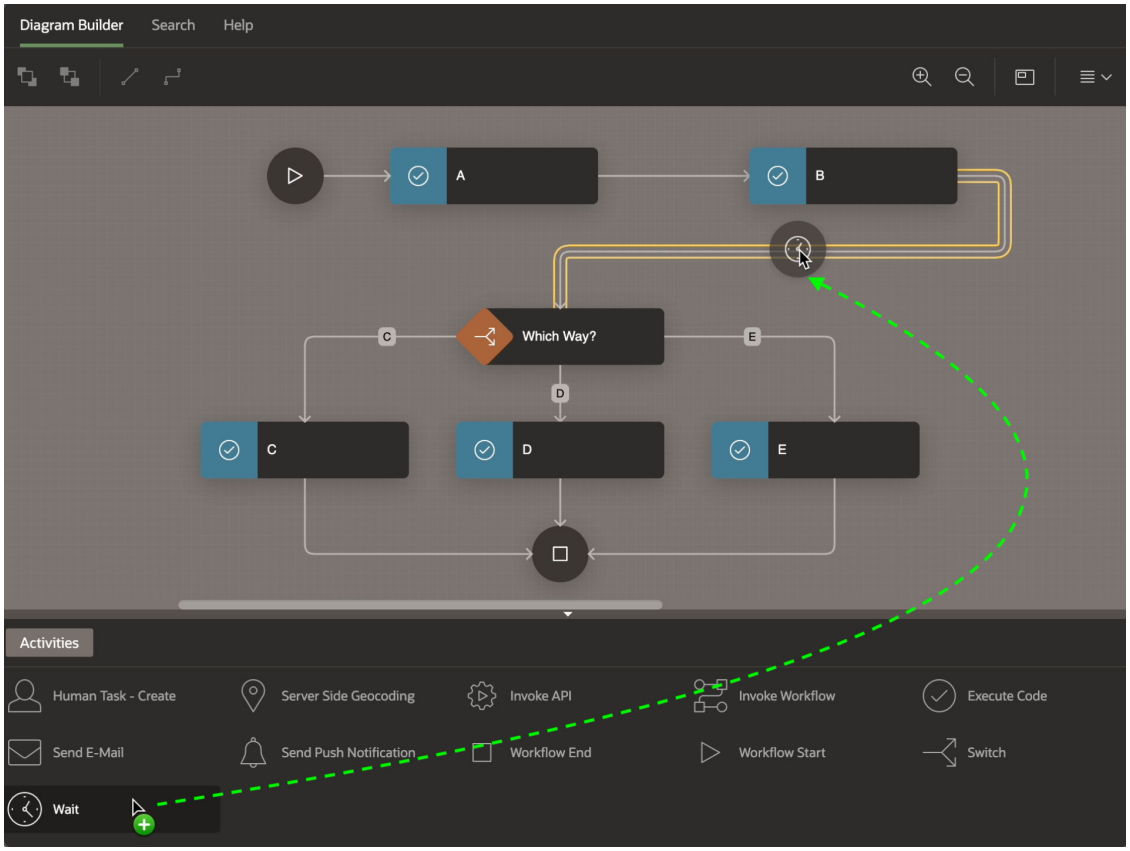
Figure 25-15 Using Content Menu to Add a New Activity Between Existing Ones



Alternatively, as shown below, you can drag an activity from the palette and drop it in the middle of the connection line between the existing activities.

Copyright © 2026, Oracle and/or its affiliates.

Figure 25-16 Dragging and Dropping a New Activity Between Existing Ones



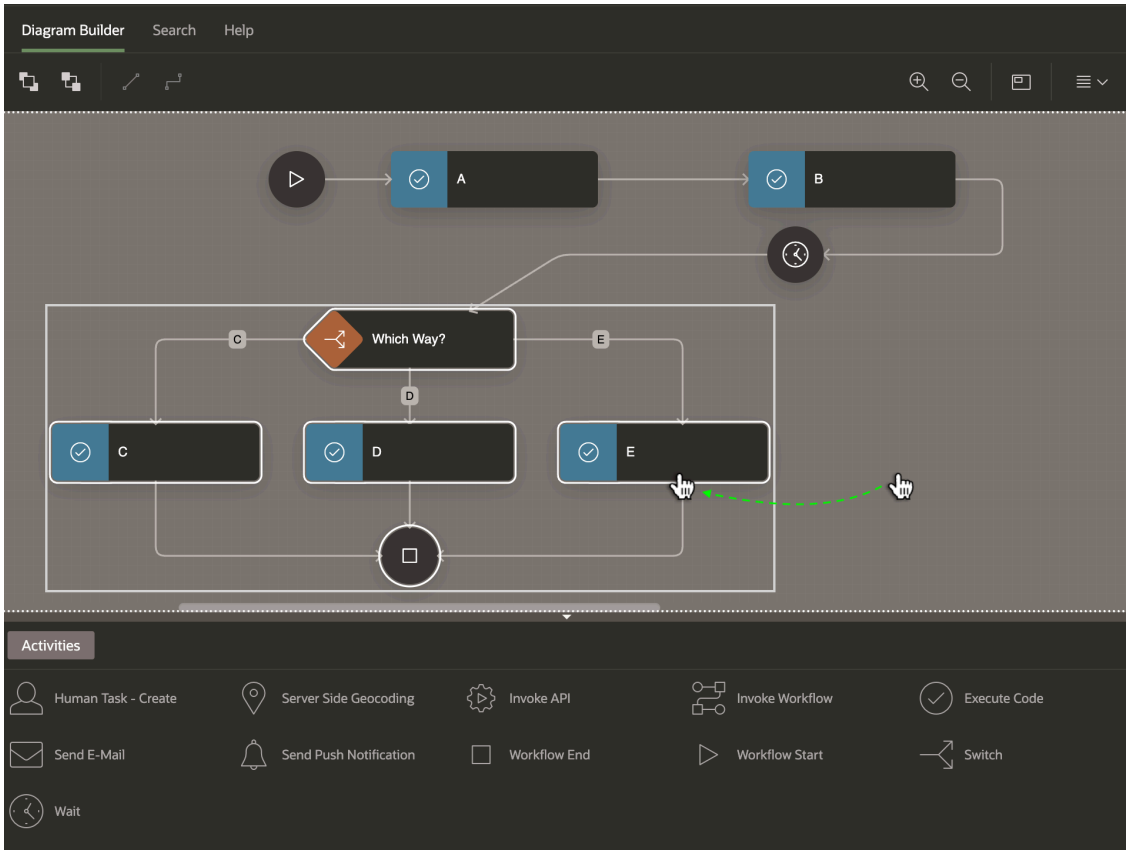
25.1.3.11 Moving a Group of Activities

Move selected activities as a group while their connections follow.

If you've already positioned a set of activities and connections but need to make space, you can select and move a group. Use [Control]-Click on Windows or [Command]-click on the Mac to select multiple activities as shown below. Then you can drag the group as a unit by clicking on a shape inside the selected area. This action maintains relative positions and alignment of the group elements. You only need to select *activities*, the connection lines move accordingly.

Copyright © 2026, Oracle and/or its affiliates.

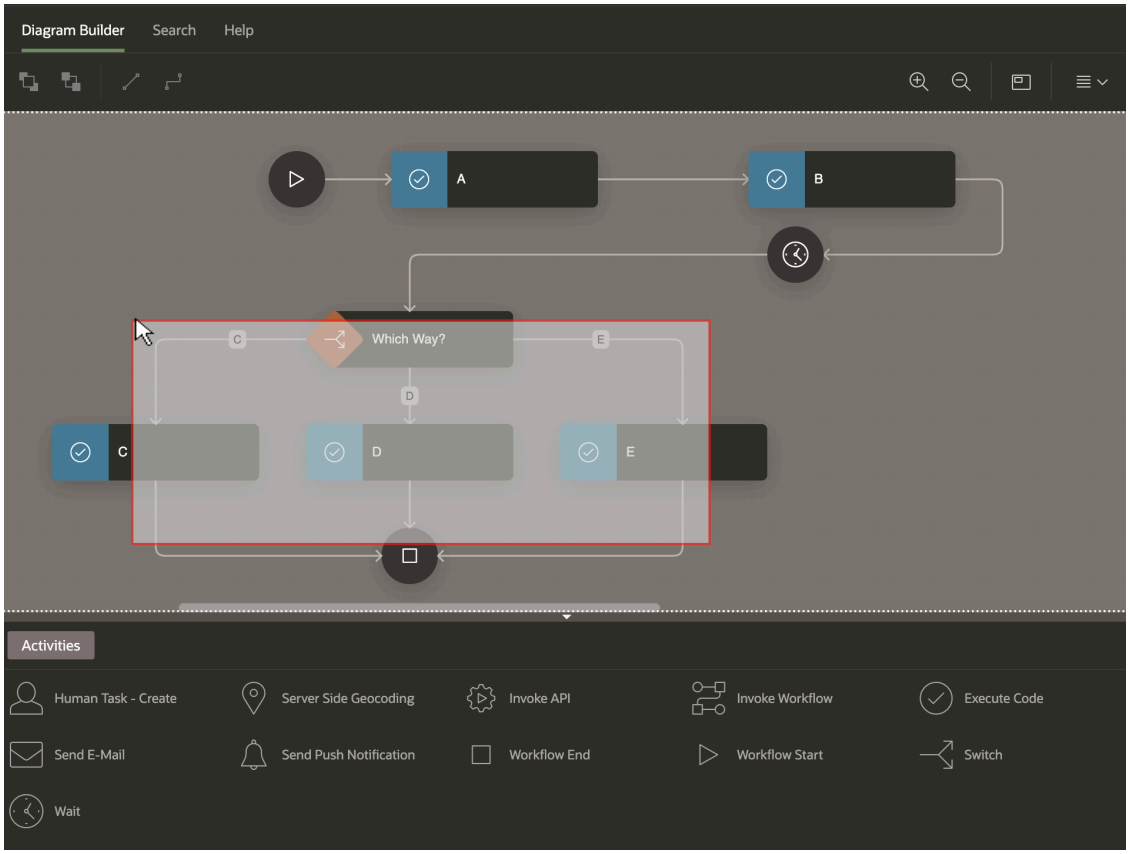
Figure 25-17 Selecting and Moving a Group of Activities



An even *faster* way to select multiple activities is to hold down the [Shift] key and drag out a rectangle as shown below. This gesture selects any activities the rectangular selection touches. Once selected, as before, you can move them as a group by clicking on a shape inside the selected area.

Copyright © 2026, Oracle and/or its affiliates.

Figure 25-18 Selecting a Set of Activities with a Shift+Drag Rectangle



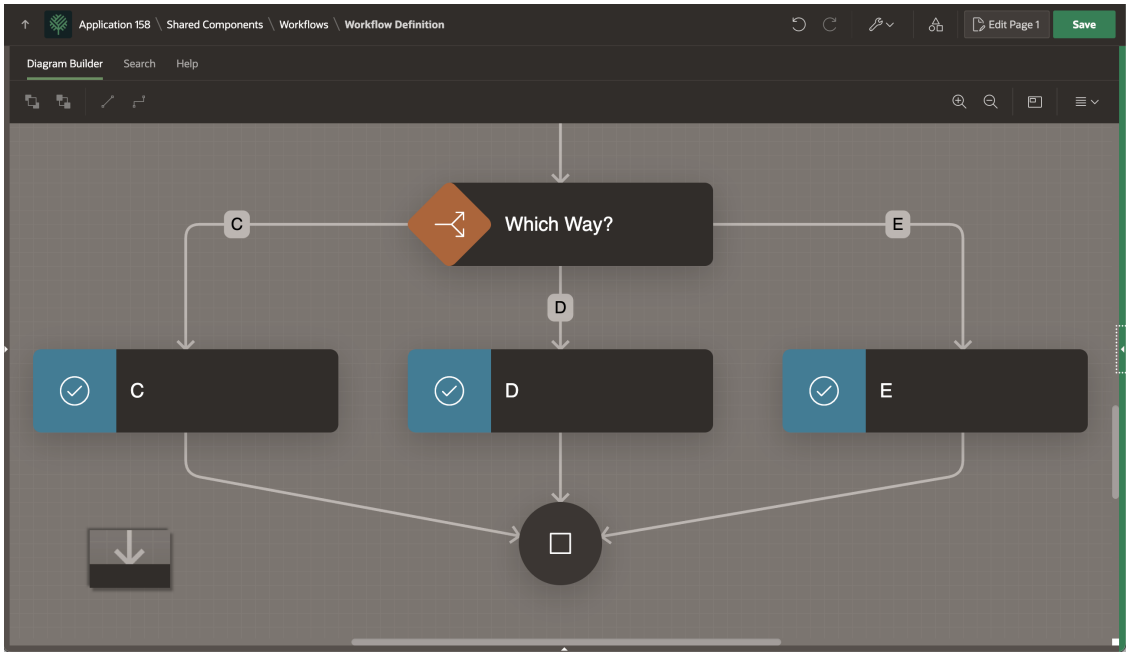
25.1.3.12 Bringing Connections in Front of Activities

Ensure arrowheads stay visible over activities in the runtime workflow progress diagram.

When viewing the runtime workflow diagram, you may notice some connection arrowheads appear to be "buried under" the activity shapes. While this effect does not confuse end users, it does look less than perfect. During the natural course of developing your workflow diagram, the "stacking order" of the shapes can result in activities stacking above connections. As shown below, this causes the tip of the connection arrow head to be obscured by the activity shape in a way that is even more noticeable in the read-only *Workflow Diagram* region.

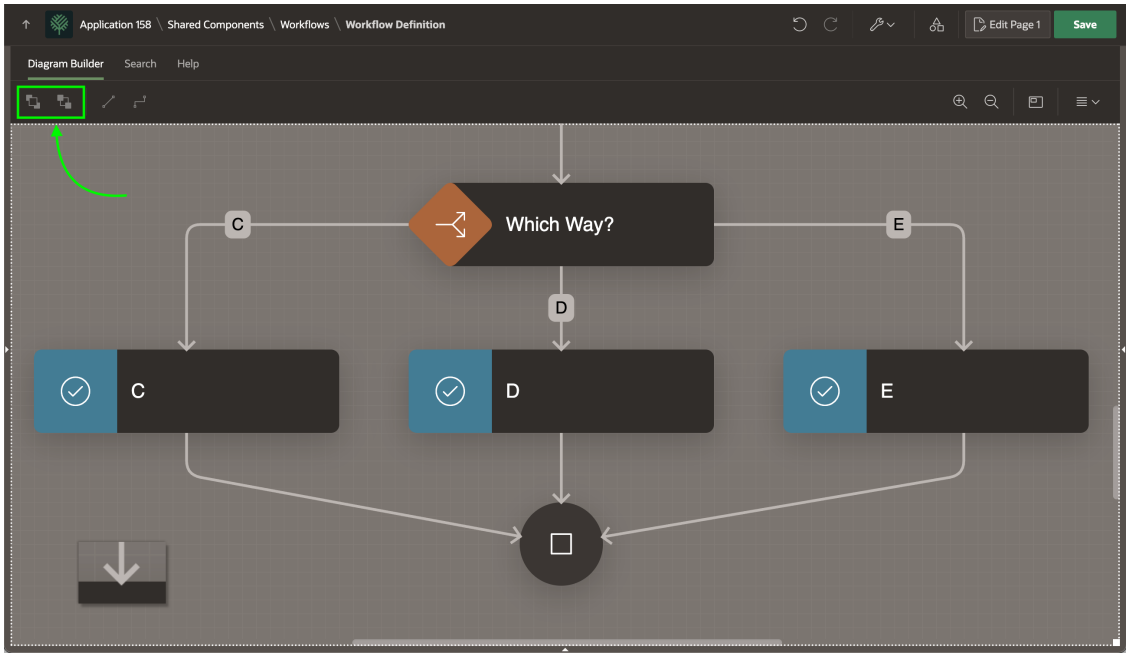
Copyright © 2026, Oracle and/or its affiliates.

Figure 25-19 Connection Arrow Tips Obscured by Activity Shapes



The solution is simple. As shown below, just use the "Bring to Front" tool in the toolbar to bring connections on top of activity shapes. Alternatively, use the "Send to Back" tool next to it to send the activity shapes behind the connections.

Figure 25-20 Use "Bring to Front" to Stack Connection Arrows On Top of Activity Shapes



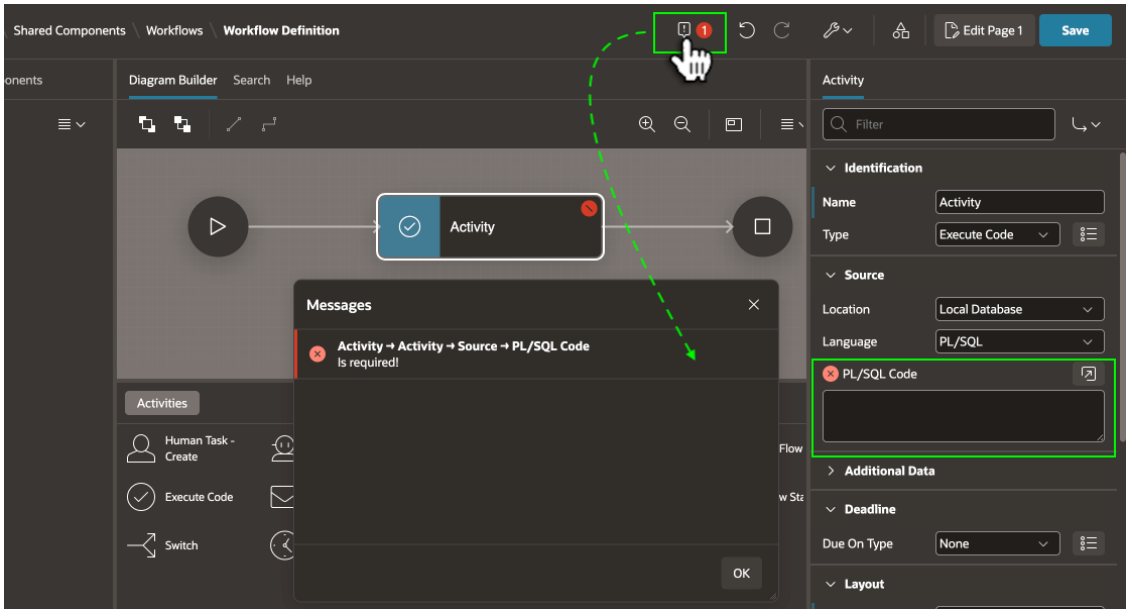
Copyright © 2026, Oracle and/or its affiliates.

25.1.4 Resolving Errors Before Saving a Workflow

Resolve any Workflow Designer errors before saving your changes.

When working in Workflow Designer, if your diagram contains any errors a toolbar indicator shows the number of problems in a red circle. Click this icon to open the **Messages** dialog to review the details. Click a message to navigate to the item that needs fixing. When the diagram has no errors, click (**Save**) to make your changes permanent.

Figure 25-21 Clicking Error Indicator to Review Pending Issues



25.1.5 Associating a Table with a Workflow

Associate a workflow with a table or query so activities can reference related row data.

A business process often involves a business entity in your application data model. A *workflow definition* models that process, and a *workflow instance* represents one example of it for a particular row in that business entity's table. Many workflow activities reference or modify data in this related business entity row. APEX simplifies this common usage pattern by letting you:

- configure the *Additional Data* table and primary key column in a workflow definition, and
- supply a *Details Primary Key* value of a corresponding row when creating a workflow instance.

Then, wherever your app *starts* a new workflow instance, remember to supply the primary key value of the business entity row to which the new flow relates.

This tiny investment of time pays big productivity dividends. Without writing any code or queries, you can reference the business entity row's details *anywhere* in the flow using *any* column name. Just use the syntax:

- `&NAME.` in text substitutions,
- `:NAME` in code and queries, and

Copyright © 2026, Oracle and/or its affiliates.

- *NAME* in activity properties requiring an item name.

For example if your *Additional Data* table's primary key column is `ID`, then reference `:ID` or `&ID.` or configure `ID` as the item name providing the value of an activity property.

> **Tip**
>
> For even more powerful automatic data access in your workflow, change the *Additional Data* > *Type* from *Table* to *SQL Query*. Your query typically includes a primary table and its primary key column, and can optionally join in any other related data. It ***must*** return exactly one row. For workflows accepting a *Details Primary Key*, your query's `WHERE` clause can reference its value using the `:APEX$WORKFLOW_DETAIL_PK` bind variable. Then throughout the workflow you can use *&NAME.* or *:NAME* notation to reference the value of any column in your *Additional Data* query's `SELECT` list.

25.1.6 Retrieving Data for an Activity

Use activity-level Additional Data to query one row of values the current activity needs.

In addition to the workflow-level *Additional Data* table or SQL query, every activity also supports an optional *Additional Data* query. Wherever it adds value, use it to declaratively retrieve a ***single row*** of relevant information the current activity requires to get its job done. Using bind variables, the query can reference column values from the workflow-level *Additional Data*, parameters, and version variables. It can join in data from any tables in the data model and engage functions in its `SELECT` list. This makes it convenient to perform on the fly computations whose results are useful to the current activity.

Once you've configured the optional *Additional Data* query on an *activity*, then in that specific activity you can reference the value of *any* of its `SELECT` list columns, using:

- *&NAME.* in text substitutions,
- *:NAME* in code and queries, and
- *NAME* in activity properties requiring an item name.

> **Tip**
>
> Activity-level *Additional Data* column values are only available to the activity whose *Additional Data* query retrieved them. *Subsequent* activities in the flow cannot reference these values. Retrieve the column in your workflow-level *Additional Details* SQL query to let *all* activities reference it. Or, if you need to store a point-in-time value that later activities can reference, use an *Execute Code* activity to assign it to a workflow variable.

25.1.7 Defining Parameters and Variables

Use parameters and variables for values a workflow cannot query or must capture at a point in time.

Workflow parameters and variables are a convenient way to persistently store values unique to a workflow *instance*. Parameters get their value at instance start time and are read-only during the flow. In contrast, your workflow activities can both read *and* change the value of variables.

Copyright © 2026, Oracle and/or its affiliates.

So one activity can store results in a variable that a later activity can reference or modify. The APEX engine manages their storage automatically.

You define parameters at the workflow *definition* level, while you specify variables on the workflow *version* level. While formally called *version variables*, often *workflow variable* or just *variable* are clear alternatives. While local *PL/SQL* variables are transient values available only within the scope you declare them, workflow variables are available from any activity for the life of the workflow.

Most workflows configure an *Additional Details* table or query at the workflow level. When combined with the *Details Primary Key* value passed in when starting the workflow instance, this makes the values of *Additional Data* columns available anywhere in the flow using `:NAME`, `&NAME.` or `NAME` syntax. Activities can also use the value of the additional data table's primary key column to query other necessary data the process requires by defining an activity-level *Additional Details* query.

However, if a workflow needs an input value that you cannot query from a table, you can define any number of parameters and variables. When considering whether to add a workflow parameter or variable, ask yourself:

- *Can I query the value from a table in my data model?*
  - *Yes*: retrieve it in the workflow *Additional Data* query if many activities need it, or in an activity *Additional Data* query otherwise.
  - *No*: it's a good candidate to accept as a parameter or to compute as a variable
- *Do I need to store a point in time value used for a calculation in this flow since it might change?*
  - *Yes*: it's a good candidate for a workflow variable
  - *No*: retrieve it in the workflow *Additional Data* query if many activities need it, or in an activity *Additional Data* query otherwise.

> **Tip**
>
> Both parameter and variables values appear for any instance in the workflow detail page you can generate from the Create Page Wizard. When necessary to fix a problem with a particular flow, that workflow instance's administrator can also *modify* variable values from this page.

- Creating a Workflow Parameter
  Create workflow parameters with meaningful static IDs for start-time values.
- Creating a Workflow Variable
  Create workflow variables with meaningful static IDs for values activities can change.

25.1.7.1 Creating a Workflow Parameter

Create workflow parameters with meaningful static IDs for start-time values.

Create a parameter using the *Create Parameter* option on the workflow's context menu in the *Workflows* tree. If a parameter already exists, the same option appears on the *Parameters* header in the tree. For each parameter you create, use the Property Editor to choose a data type and assign it a meaningful *Static ID*. While not required, using a parameter static ID that starts with a prefix like `P_` can help you distinguish parameter names from *Additional Data* column names and from workflow variable names. You use the parameter's static ID to reference its value as a bind variable `:P_YOUR_STATIC_ID` in code or queries,

Copyright © 2026, Oracle and/or its affiliates.

`&P_YOUR_STATIC_ID.` in substitutions, and `P_YOUR_STATIC_ID` in activity properties requiring an item name.

> **Tip**
>
> The parameter *Label* you define is the friendly description that appears in the workflow detail page when a user inspects a workflow instance.

25.1.7.2 Creating a Workflow Variable

Create workflow variables with meaningful static IDs for values activities can change.

Create a variable using the *Create Variable* option on the workflow version's context menu in the *Workflows* tree. The same option exists on the *Variables* heading in the tree. For each variable you create, use the Property Editor to choose a data type and assign it a meaningful *Static ID*. While not required, using a variable static ID that starts with a prefix like `V_` can help you distinguish variable names from *Additional Data* column names and from workflow parameter names. You use the variable's static ID to reference its value as a bind variable `:V_YOUR_STATIC_ID` in code or queries, `&V_YOUR_STATIC_ID.` in substitutions, and `V_YOUR_STATIC_ID` in activity properties requiring an item name. For example, when configuring a variable name to receive an "out" value from activities like *Invoke API*, *Invoke Workflow*, *Human Task - Create*, and others, use the `V_YOUR_STATIC_ID` variable name.

> **Tip**
>
> The variable *Label* you define is the friendly description that appears in the workflow detail page when a user inspects a workflow instance.

25.1.8 Understanding Background Execution

Workflows run in a background session without page session state or a logged-in user.

The APEX engine manages the progress of workflow instances asynchronously in the background. It processes each instance's next activity in a distinct engine session with no logged-in user. A workflow engine session is *also* different from the user session created for each end user of your application's pages.

By design, two consequences are:

- a workflow instance has *no access* to session state values from application pages, and
- evaluating `APP_USER` returns `nobody` in a workflow.

Therefore, any value your workflow depends on needs to be:

- A built-in APEX workflow substitution (e.g. `APEX$WORKFLOW_INITIATOR`),
- An *Additional Data* column,
- Saved in a table, so an activity can query it,
- Passed as a parameter value to the workflow instance at start time,
- Stored in a workflow variable by an earlier activity, or
- Saved to an updatable *task* parameter accessible via `APEX_HUMAN_TASK` API.

Copyright © 2026, Oracle and/or its affiliates.

25.1.9 Getting Approval or User Input in a Workflow

Wait for user approval or input by creating a task or sending a notification.

Your workflow runs the steps in a business process. Sometimes, it needs input from a user. They might need to confirm a manual task, approve or reject something, or enter data so the workflow can proceed. The user needs a page where they can see what's needed and enter their response. But workflows run in the background where there's no "current user" to show a page to. Instead, use one of these familiar patterns:

- Create a work item in a to-do list the user checks periodically.
- Send the user a push notification or email, and wait for them to act on it.

Either way, the user clicks a link, opens a page, and provides their input. The workflow waits in the background. It moves on only after the user responds, whether that's in a few minutes or a few days.

- Assigning Work to a User's Task Inbox
  Assign work to users so APEX manages the task lifecycle and continues the workflow when completed.
- Waiting for a Signal to Continue
  Send users a page link, wait for their response, and continue the workflow from application logic.

25.1.9.1 Assigning Work to a User's Task Inbox

Assign work to users so APEX manages the task lifecycle and continues the workflow when completed.

Use a *Task Definition* shared component to track a work item to assign to one or more people. The task *Subject* line you configure appears in the task list "inbox" of all assignees. They are all *potential owners* of the task. The first available user *claims* the task and becomes its *actual owner*. Once claimed, it no longer appears in *other* potential owners' task lists.

By configuring a *Task Details Page*, users can click a task's subject in this "inbox" to review its details. You can generate it in the task definition editor and adjust it in Page Designer as needed. Your enhancements to the page can include saving additional data to updatable task parameters or to application tables.

Some tasks require only that the owner confirm they've completed the action it represents. Others seek an *outcome* indicating the owner's approval or rejection. A task can be open-ended, or made time-sensitive by configuring a due date and expiration behavior. Your workflow uses *Human Task - Create* activities to coordinate which work items need to be done in what order. The APEX engine automatically waits for a potential owner to complete a task before proceeding to the next activity in the workflow. The figure illustrates the workflow as a postal employee delivering tasks to three different users task inboxes. One reviews tasks from a smartphone. Another sees tasks on a tablet. A third person works on tasks from a laptop.

Copyright © 2026, Oracle and/or its affiliates.

Figure 25-22 Workflow Delivers Tasks to Potential Owners' Task List "Inbox"



Using this approach, you get many built-in features that enterprise applications often require. APEX manages the "lifecycle" of a task from creation to completion. Task definitions support more than just a priority and deadlines. Your dynamic task assignment can account for end-user vacations. Users can request and receive more info about a task, delegate it to a colleague, or participate in a threaded discussion about it with teammates when collaboration is required. You can relate each task *instance* to a particular row in a database table, and your task definition can *react* to changes in the state by notifying users or executing custom business logic. In fact, by defining an action on the task *Create* event, you can notify a user a new work item has been added to their to-do list, encouraging them to check their task "inbox".

- Deciding if Task Needs an Outcome
  Choose an action task to track completion or an approval task to capture an outcome.
- Associating a Table with a Task
  Associate a task with a table or query so task logic can reference related row data.
- Accepting Input Values in Parameters
  Use task parameters for values the task cannot get from its Actions Source row.
- Determining Task Participants
  Assign potential owners and business administrators using case-sensitive usernames.
- Understanding Task States
  Understand task states and the actions that move a task through its lifecycle.
- Creating Task "Inbox" Page for Users
  Create a Unified Task List page that shows users the tasks they need to act on.
- Collecting Data Using a Human Task
  Create a human task from a workflow to collect user input before the flow continues.

Copyright © 2026, Oracle and/or its affiliates.

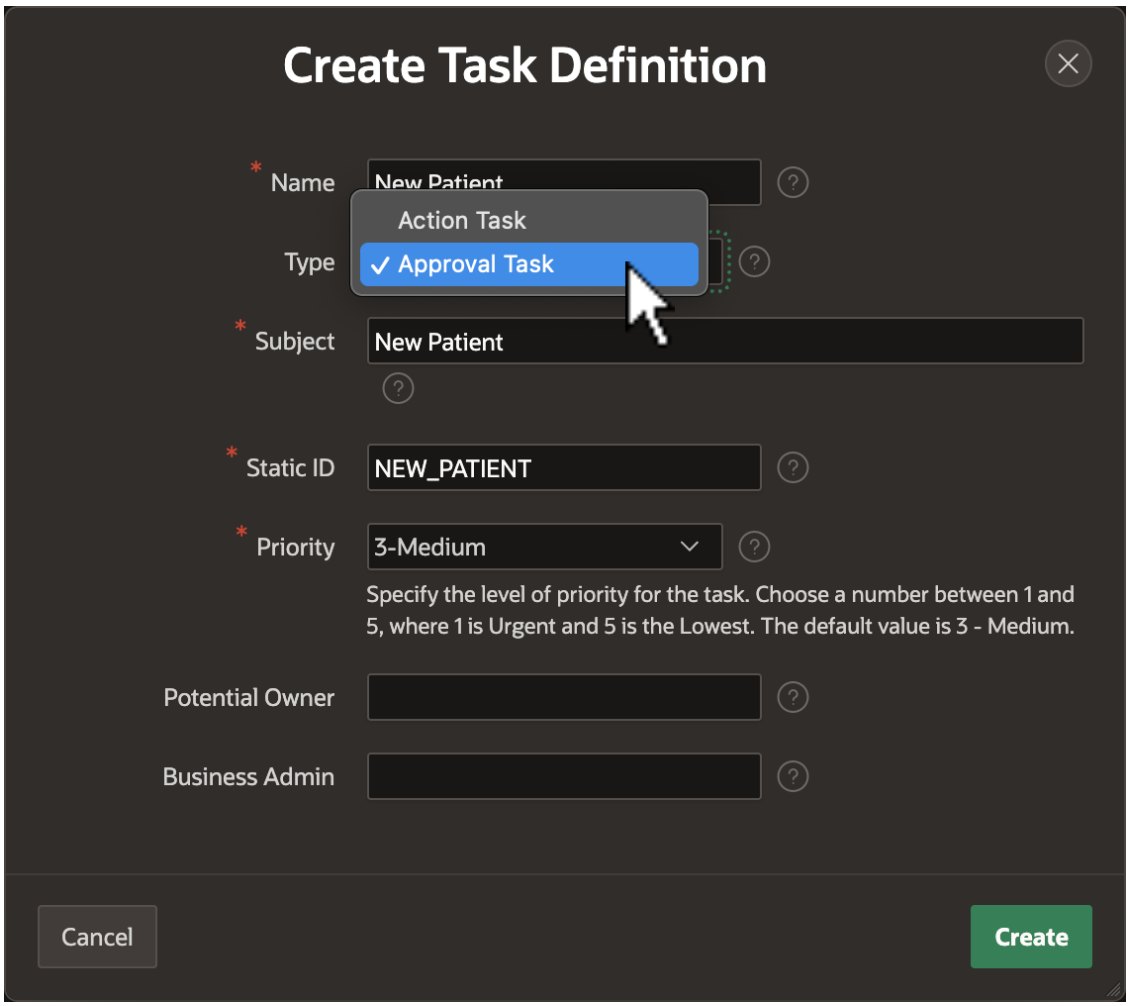
- Using and Tuning the Task Details Page
  Explore how users work with a Task Details page and how developers can tune it.
- Letting an Initiator Approve a Task
  Enable initiators to approve their own tasks when the business process requires it.
- Handling Time-Sensitive Tasks
  Configure due dates and expiration behavior for time-sensitive tasks.
- Reacting to Task Change Events
  Use task actions to react to lifecycle events while a workflow waits.

25.1.9.1.1 Deciding if Task Needs an Outcome

Choose an action task to track completion or an approval task to capture an outcome.

When you create a *Task Definition* shared component, as shown below, set the *Type* to *Action Task* to track *only* that an assignee has completed the task. Choose *Approval Task* instead to capture the actual owner's `APPROVED` or `REJECTED` outcome. If you realize later that you picked the wrong task type, then you'll need to try again by creating a new task definition. You can only set this property at create time.

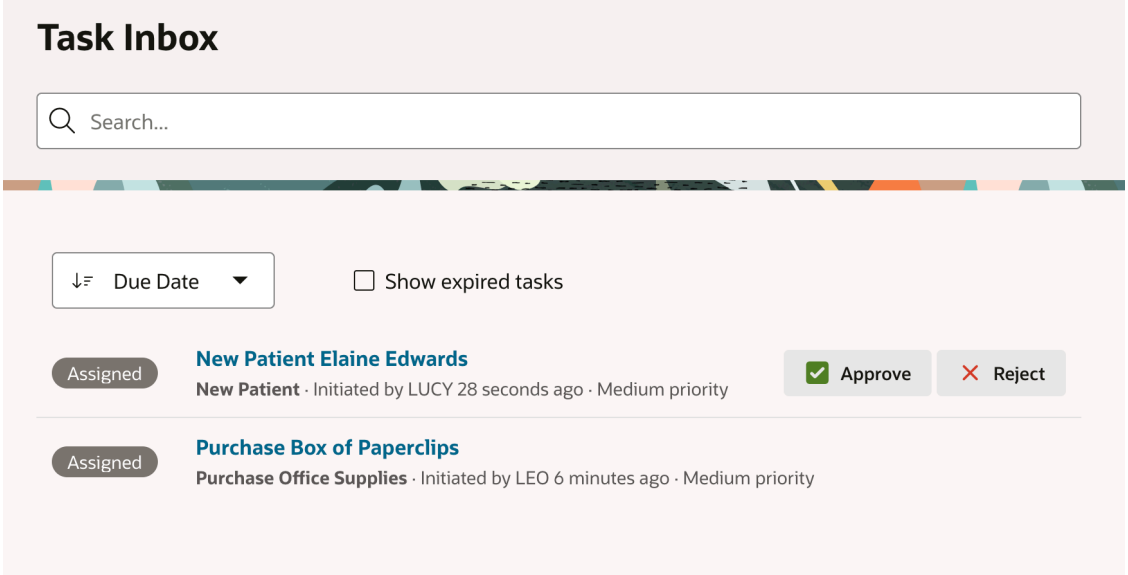
Figure 25-23 Choosing an Action Task or Approval Task at Create Time



Copyright © 2026, Oracle and/or its affiliates.

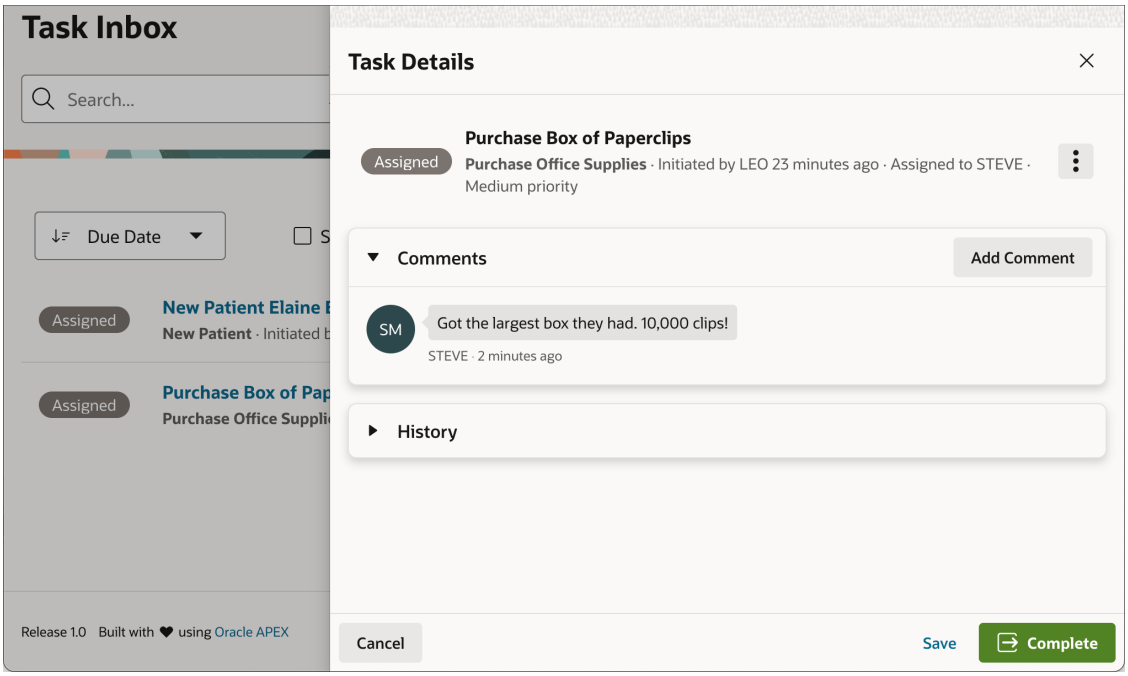
The task type determines the buttons end users will see in the Task List and Task Details pages when your app assigns them tasks to work on. As shown below, by default approval tasks display quick action buttons to approve or reject the task, while action tasks do not.

Figure 25-24 Approval Tasks Show Quick Action Buttons in the Task List



Similarly, while the default task detail page for an approval task shows *(Approve)* and *(Reject)* buttons, as shown below, the generated page for an *action task* has just a *(Complete)* button instead.

Figure 25-25 Action Tasks Have a (Complete) Button in Task Details Page



Copyright © 2026, Oracle and/or its affiliates.

25.1.9.1.2 Associating a Table with a Task

Associate a task with a table or query so task logic can reference related row data.

An end-user task often relates to a business entity in your application data model. A *task definition* models that work item, and a *task instance* represents one example of it for a particular row in that business entity's table. Task assignment and your actions that react to task status changes tend to reference or modify data in this related business entity row.

APEX simplifies this common usage pattern by letting you:

- configure the *Actions Source* table and primary key column in a task definition, and
- supply a *Details Primary Key* value of a corresponding row when creating a task instance.

Then, wherever your app *creates* a new task instance, remember to supply the primary key value of the business entity row to which the new task relates.

This brief investment of time delivers a strong productivity payoff. Without writing any code or queries, you can reference the business entity row's details *anywhere* in the task definition using *any* column name. Just use this syntax:

- `&NAME.` in text substitutions, and
- `:NAME` in code and queries.

For example if your *Actions Source* table's primary key column is `ID`, then you can reference `:ID` in action PL/SQL code or participant assignment expressions or queries. In email notifications use the substitution `&ID.` instead.

> **Tip**
>
> For even more powerful automatic data access in your task definition, change the *Actions Source* type from *Table* to *SQL Query*. Your query typically includes a primary table and its primary key column, and can optionally join in any other related data. It ***must*** return exactly one row. If your task accepts a *Details Primary Key*, your `WHERE` clause can reference its value using the `:APEX$TASK_PK` bind variable. Then throughout the workflow you can use `&NAME.` or `:NAME` notation to reference the value of any column in your *Actions Source* query's `SELECT` list.

25.1.9.1.3 Accepting Input Values in Parameters

Use task parameters for values the task cannot get from its Actions Source row.

Use the pre-defined `APEX$TASK_PK` parameter to reference the value of the *Details Primary Key* passed in when a task instance is created. For many use cases, it is the only parameter value you need. Assuming you have *also* configured the task definition's *Actions Source* table or query, this combination can often provide all necessary context information. However, if there are *additional* values unique to the pending work item that are *not (yet) available* as Actions Source column values, you can define explicit task parameters to accept them when the task is created.

Copyright © 2026, Oracle and/or its affiliates.

> **Tip**
>
> You can optionally make task parameters *updatable*. These are useful to model *additional* data that the task assignee must provide as part of reviewing or approving the task.

When defining a parameter, you provide a user-friendly label that can be in mixed case and include spaces. You also provide an alphanumeric static ID. You can reference any parameter's value throughout the task definition using the `&STATIC_ID.` or `:STATIC_ID` notation depending on the context. The APEX engine manages task parameter storage and makes them queryable in a generic way through the `APEX_TASK_PARAMETERS` dictionary view.

A simple example of a task parameter for a *Salary Change* approval task is the *New Proposed Salary*, with a Static ID of `P_PROPOSED_SALARY`. Until the change of salary is approved, this value only exists as part of the task instance for approvers to consider. If one of them approves the task, then a task *action* you define on the *Completed* event with *Approved* outcome can update the `SAL` column in the `EMP` table for the employee to the new, approved value. Assuming you configured `EMP` as the *Actions Source* table for this task, and passed in the employee's `EMPNO` value as the *Details Primary Key* when creating the task, your action code can perform this update using a simple statement like:

```
update emp
   set sal = :P_PROPOSED_SAL
 where empno = :EMPNO;
```

> **Tip**
>
> Imagine you find your task definition is not logically related to *any* particular table in your application data model, and that you are compelled to create *many* parameters to capture input data for the task. This can be a sign your application data model needs a new table to track the task in question. Consider defining a new table to represent the particular request, with columns for all the data you found yourself defining task parameters for. Should that make sense for your use case, then associate this new table with your task definition as its *Actions Source*. After doing that, the built-in `APEX$TASK_PK` parameter may be the only parameter you need. Your page or workflow inserts a row into this new table, then passes its primary key to the task flow as its *Details Primary Key* value. A follow-on benefit is that it is simpler to query all of the data related to the task from other parts of your application.

25.1.9.1.4 Determining Task Participants

Assign potential owners and business administrators using case-sensitive usernames.

A task participant is either a potential owner or business administrator. A *potential owner* is a user you expect to perform the work the task represents, while a *business administrator* can help address a problem relating to the task.

For example, if user `DAVID` is the potential owner of the *New Patient* approval task, he sees it in his *My Tasks* list. He's also the one who can click *(Approve)* or *(Reject)*. If user `CARLA` is the task's business administrator, she sees it in her *Admin Tasks* list. She can reassign the task to another colleague if a response is due today but `DAVID` is unexpectedly absent. `CARLA` the business administrator can also adjust a task's due date.

Copyright © 2026, Oracle and/or its affiliates.

> **⚠ Caution**
>
> Participant usernames are case-sensitive. User `DAVID` can login with the username `david`. However, if you assign user `david` to a task, David does not see it in his task list because `david` does not match the `DAVID` value the APEX engine returns for the currently logged-in username (`APP_USER`). If `:APP_USER` returns `DAVID`, then use `DAVID` as the participant username.

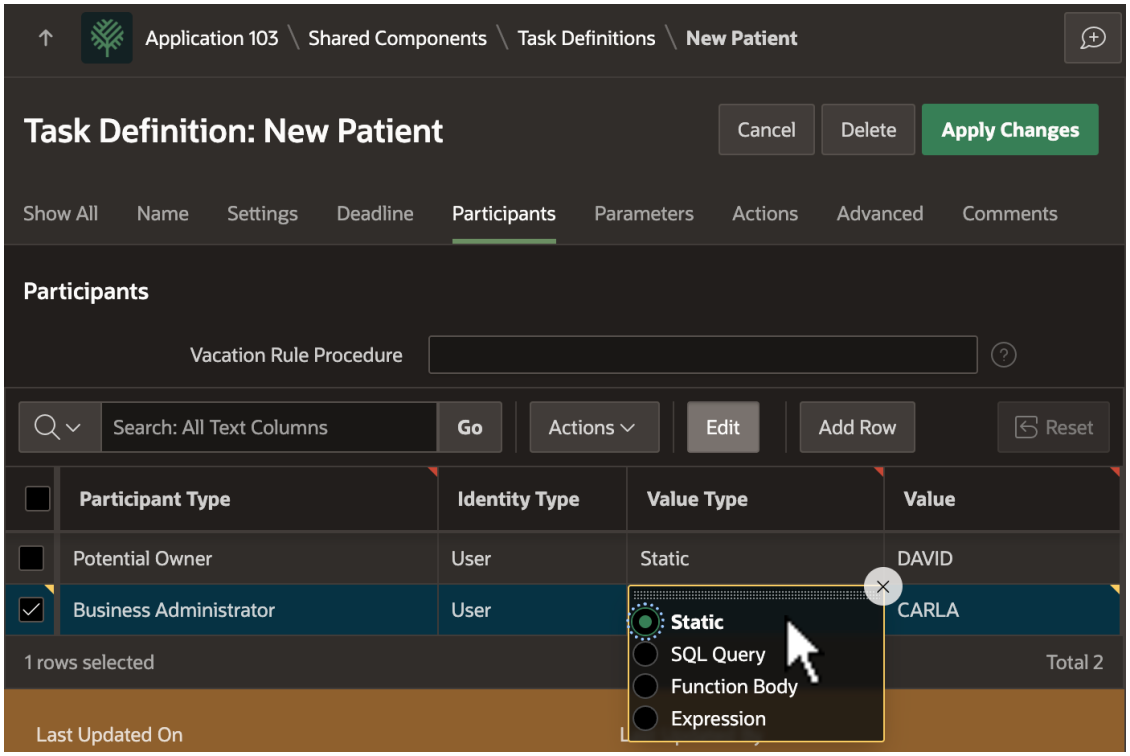
- Setting Static Task Participants
  Set fixed task participants when a simple process always assigns work to the same users.
- Assigning Participants Dynamically
  Assign task participants dynamically with queries, expressions, or functions that return case-sensitive usernames.
- Passing Participants in a Parameter
  Pass task assignees in a parameter and use it to assign potential owners.
- Accounting for Vacations or Absences
  Use a vacation rule procedure to add alternate task participants for absent users.

25.1.9.1.4.1 Setting Static Task Participants

Set fixed task participants when a simple process always assigns work to the same users.

As shown below, in a simple application, either or both of the *Potential Owner* or *Business Administrator* participants can be a static value like `DAVID` or `CARLA`. But normally you use one of the *Value Type* settings like *SQL Query*, *Function Body*, or *Expression* to dynamically determine one or both of these participant lists.

Figure 25-26 Configuring Task Participants



Copyright © 2026, Oracle and/or its affiliates.

> **Caution**
>
> Participant usernames are case-sensitive. If `:APP_USER` returns `DAVID`, then use `DAVID`.

25.1.9.1.4.2 Assigning Participants Dynamically

Assign task participants dynamically with queries, expressions, or functions that return case-sensitive usernames.

To assign task participants dynamically, use a participant *Value Type* of *SQL Query*, *Function Body*, or *Expression*. When using the *SQL Query* option, your single column `SELECT` list returns one case-sensitive username per row. For the *Function Body* or *Expression* options, return a comma-separated list of case-sensitive usernames.

Your query, function, or expression can use bind variable syntax to reference task parameter values, action source column names, application items, or other APEX substitution strings.

> **Tip**
>
> Workflows execute in the background where there is no "logged-in username". So the `APP_USER` substitution returns `nobody` in a workflow context. Use `APEX$WORKFLOW_INITIATOR` to reference the username that initiated the workflow, or `APEX$TASK_INITIATOR` for the username that initiated the task. If neither of these is the user you need, define a workflow parameter or task parameter to accept a username for other purposes.

For easiest maintainability, your expression can invoke a PL/SQL package function as well. The function can be specific to the task at hand or be generic in nature and accept values like the task ID and task details primary key as PL/SQL `IN` parameters:

```
your_app_pkg.owners_for_task(:APEX$TASK_ID, :APEX$TASK_PK)
```

By passing the value of pre-defined bind variables like `APEX$TASK_ID` and `APEX$TASK_PK`, your generic function can lookup related information from APEX dictionary views like `APEX_TASKS` and from your application table using the primary key value. If needed, you can join in task *definition* information from the `APEX_APPL_TASKDEFS` view, using the value of the `TASK_DEF_STATIC_ID` column in `APEX_TASKS`.

> **Caution**
>
> Participant usernames are case-sensitive. If `:APP_USER` returns `DAVID`, then use `DAVID`.

25.1.9.1.4.3 Passing Participants in a Parameter

Pass task assignees in a parameter and use it to assign potential owners.

A common requirement is to let the page or workflow that creates the new task instance pass in the list of task assignees. This is easy to accomplish by defining a parameter – for example

Copyright © 2026, Oracle and/or its affiliates.

named `P_ASSIGNEE_LIST` – and referencing it in an expression consisting of just the bind variable:

```
:P_ASSIGNEE_LIST
```

You could also use it in a *SQL Query*, but in that case you need to return one row per case-sensitive username supplied. For example, if the input value is a colon-separated list of usernames you could use the *SQL Query* option with the query:

```
select column_value
from apex_string.split(:P_ASSIGNEE_LIST,':')
```

> ⚠ **Caution**
>
> Participant usernames are case-sensitive. If `:APP_USER` returns `DAVID`, then use `DAVID`.

25.1.9.1.4.4 Accounting for Vacations or Absences

Use a vacation rule procedure to add alternate task participants for absent users.

Sometimes, stakeholders involved in task assignments are out of the office. Your task definition can use a vacation rule procedure to run custom business logic after the APEX engine determines participants using your original *Participants* configuration. The rule affects both initial task participant assignment, as well as task delegation.

You can also specify a vacation rule procedure at application level. If you do that, a task uses it unless it specifies a more specific rule of its own. You can optionally create the procedure in a PL/SQL package. It needs the following signature using *exactly* the indicated parameter names, modes, and types, in the given order:

```
procedure your_vacation_rule (
  p_param    in apex_human_task.t_vacation_rule_input,
  p_result  out apex_human_task.t_vacation_rule_result )
```

To process the list of existing participants, access the `original_participants` list from the `p_param` input record. If any alternative assignees are needed, populate the `participant_changes` list in the output record. For each participant change you populate the original username, the username who replaces them, and a reason explaining why that is recorded in the task history. The participant changes you return automatically become additional participants on the task.

> ✔ **Tip**
>
> The participant changed reason is logged in the `PARTICIPANT_CHANGED_REASON` column of the `APEX_TASK_HISTORY` view and is available among the result columns returned by the `GET_TASK_HISTORY` function in the `APEX_HUMAN_TASK` package. This information is not included by default in a generated Task Details page, but is straightforward to add when required.

When configuring an out-of-office assignment handler, specify either:

Copyright © 2026, Oracle and/or its affiliates.

- *your_vacation_proc*, or
- *yourpackage.your_vacation_proc*

as the value of the *Vacation Procedure* property. The following example vacation rule procedure adds SUSAN as an additional approver for DAVID due to some personal time off. In your application, you can read absence assignments from a table, but this code illustrates the fundamentals.

```
procedure handle_absences(
    p_param   in apex_human_task.t_vacation_rule_input,
    p_result out apex_human_task.t_vacation_rule_result )
is
    l_participant apex_human_task.t_task_participant;
    l_change      apex_human_task.t_task_participant_change;
    ------------------------------------------------------
    function potential_owner(
        p_username in varchar2)
        return apex_human_task.t_task_participant
    is
    begin
        return apex_human_task.t_task_participant(
                apex_human_task.c_task_potential_owner,
                apex_human_task.c_task_identity_type_user,
                p_username);
    end potential_owner;
    ------------------------------------------------------
    function absent_user_cover(
        p_original_username in varchar2,
        p_covering_username in varchar2,
        p_reason            in varchar2)
        return apex_human_task.t_task_participant_change
    is
    begin
        return apex_human_task.t_task_participant_change(
            potential_owner(p_original_username),
            potential_owner(p_covering_username),
            p_reason);
    end absent_user_cover;
begin
    -- Loop over original participants
    for j in 1..p_param.original_participants.count loop
        l_participant := p_param.original_participants(j);
        -- If the participant is DAVID as a potential owner
        if     l_participant.type  = apex_human_task.c_task_potential_owner
           and l_participant.value = 'DAVID'
        then
            -- Add participant change: SUSAN covers for DAVID's personal time
off
            p_result.participant_changes(p_result.participant_changes.count +
1) :=
                absent_user_cover('DAVID','SUSAN','Personal Time Off');
            p_result.has_participant_changes := true;
        end if;
    end loop;
end handle_absences;
```


Copyright © 2026, Oracle and/or its affiliates.

> ⚠ **Caution**
>
> Participant usernames are case-sensitive. If `:APP_USER` returns `DAVID`, then use `DAVID`.

25.1.9.1.5 Understanding Task States

Understand task states and the actions that move a task through its lifecycle.

Every task instance has a state. It reflects the status of the task as it progresses through its lifecycle from creation to closure. Each state has a corresponding *state code*. The state is a mixed-case string like *Information Requested*, while the state code is uppercase and contains no spaces like `INFO_REQUESTED`. With the exception of this example, all other state codes are the uppercase version of the corresponding state name.

When you create a task, as shown in the diagram below, it starts in the *Unassigned* state when it has multiple potential owners, or the *Assigned* state when there is just one. The arrows in the diagram indicate the action that changes the state of a task instance. Once a task is *Assigned*, if it's an action task, the actual owner can complete it. For an approval task, they can approve or reject it.

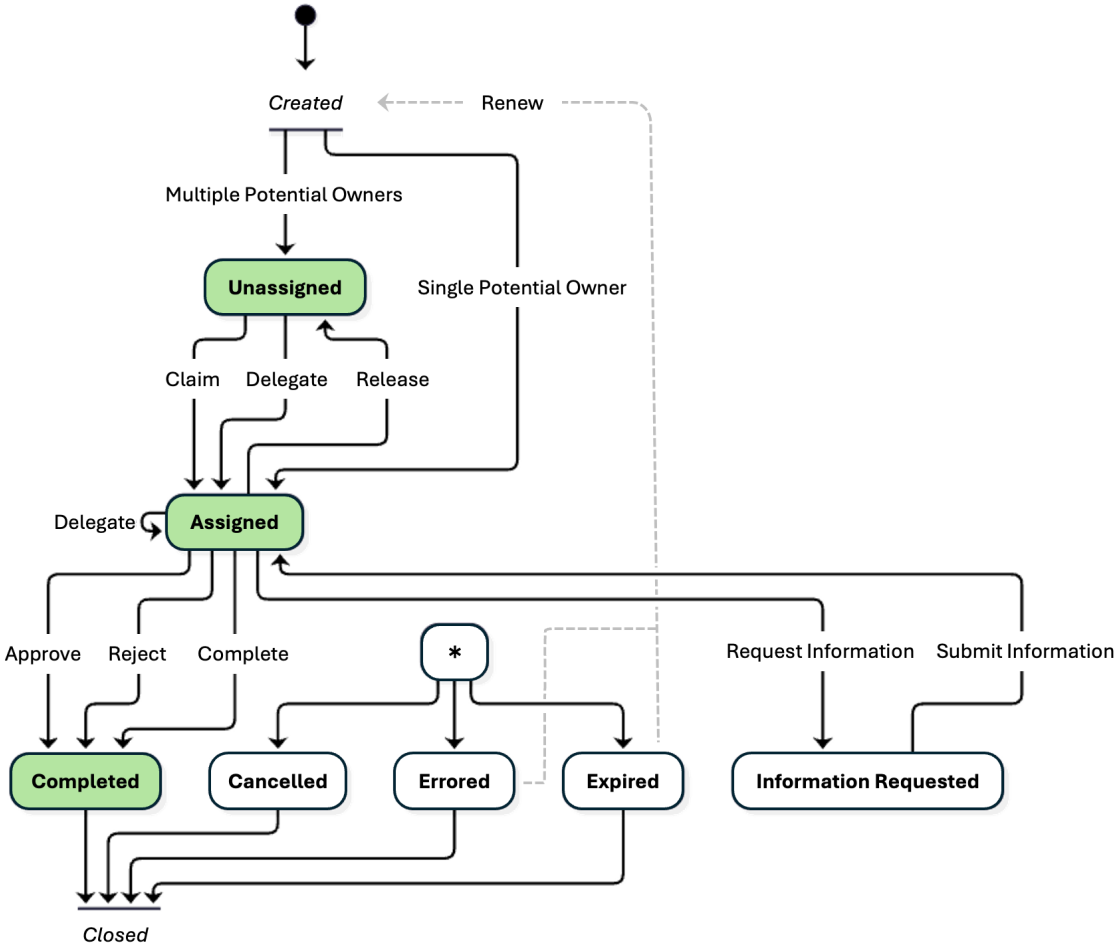
Before they take action on the task, the actual owner can request additional information from the initiator of the task, which makes it appear in the initiator's task list until they supply the requested information. Users can comment on a task at any time, which doesn't change its state. That's why the *Comment* action is not reflected in the diagram.

A task's business administrator can cancel it at any time, add a new participant, and delegate it to a different actual owner. They can also renew a task that has expired or encountered an error in order to retry the lifecycle.

The flow diagram depicts the states of the task lifecycle and the actions that change state. It starts with **Unassigned**, progresses to **Assigned**, and continues to **Completed**. Optionally, the task can be **Cancelled**, **Errored**, **Expired**, or have additional **Information Requested**.

Copyright © 2026, Oracle and/or its affiliates.

Figure 25-27 Task Instance Lifecycle: States and Actions that Change State



25.1.9.1.6 Creating Task "Inbox" Page for Users

Create a Unified Task List page that shows users the tasks they need to act on.

Choose *Unified Task List* in the *Create Page Wizard* to add a task "inbox" page for end users.

- Tailoring Task List to Purpose
  Choose a report context to control which task instances a Unified Task List page shows.
- Showing Only Current Application Tasks
  Limit a Unified Task List page to tasks from the current application.
- Including Extra Columns in Task List
  Expose additional `GET_TASKS` columns so the task list can show more task details.
- Limiting Which Types of Tasks Appear
  Add a WHERE clause to show only selected task definition types.
- Joining Extra Tables into Task List
  Join application tables into the task list to display related business data.

25.1.9.1.6.1 Tailoring Task List to Purpose

Choose a report context to control which task instances a Unified Task List page shows.

Copyright © 2026, Oracle and/or its affiliates.

When creating a *Unified Task List* page, the *Report Context* you choose determines the filtered list of task instances the page displays at runtime.

Table 25-1 Report Context Determines Which Task Instances Display

Report Context	Instances Where Current User Is	API Constant
My Tasks	• Potential Owner (when task is ***not*** assigned) • Actual Owner (when task is assigned)	`MY_TASKS`
Admin Tasks	Business Administrator	`ADMIN_TASKS`
Initiated by Me	Initiator	`INITIATED_BY_ME`

The wizard creates a new *Unified Task List* page that contains:

- A *Content Row* region to display the task list,
- A *Smart Filters* region to narrow the task list in various useful ways, and
- An *Order By Item* to let users sort the list by creation date or due date.

You can use the page as it comes, or customize it in Page Designer as needed. Create as many task inbox pages as your app requires, choosing an appropriate *Report Context* in each one as needed.

25.1.9.1.6.2 Showing Only Current Application Tasks

Limit a Unified Task List page to tasks from the current application.

By default, the task list displays tasks from all applications in the workspace. If instead you want it to only display tasks related to the *current* application, one small change is required. As shown below, the *Content Row* region's query retrieves the task list from the `GET_TASKS` function in the `APEX_HUMAN_TASK` package. This function returns a table of task information records. By using the `table()` operator, this result works just like a regular table in a `SELECT` statement.

Notice that the `p_application_id` parameter is commented out using two consecutive dashes (`--`), the PL/SQL line comment.

```
select task_id,
       task_type,
         ▒
       badge_state
  from table ( apex_human_task.get_tasks (
                  p_context            => 'MY_TASKS',
                  p_show_expired_tasks => :P33_SHOW_EXPIRED
                  --, p_application_id => :APP_ID
                  ) )
```

To only display tasks related to the *current* application, as shown below, simply remove the two dashes to *uncomment* the `p_application_id` parameter line. This passes the current application id as the value of this parameter, and limits tasks returned to those from the current application only.

```
select task_id,
       task_type,
         ▒
       badge_state
```


Copyright © 2026, Oracle and/or its affiliates.

```
  from table ( apex_human_task.get_tasks (
                  p_context            => 'MY_TASKS',
                  p_show_expired_tasks => :P33_SHOW_EXPIRED
                  , p_application_id => :APP_ID
                  ) )
```

25.1.9.1.6.3 Including Extra Columns in Task List

Expose additional `GET_TASKS` columns so the task list can show more task details.

In a new *Unified Task List* page, the *Content Row* region's `SELECT` list includes only *some* of available task information columns the `GET_TASKS` function returns. To see *all* of the available columns, change the query to use * for the column list as shown below:

```
select *
  from table ( apex_human_task.get_tasks (
                  p_context            => 'MY_TASKS',
                  p_show_expired_tasks => :P33_SHOW_EXPIRED,
                  p_application_id     => :APP_ID
                  ) )
```

This causes Page Designer to show all available column names under the region's *Columns* heading in the *Rendering* tab. The following table highlights a few important `GET_TASKS` result columns.

Table 25-2 Important GET_TASKS Result Columns

Column	Description
`TASK_ID`	Unique ID of this task instance
`SUBJECT`	Subject of this task instance
`STATE_CODE`	Current state of this task instance (e.g. `INFO_REQUESTED`)
`STATE`	Current state of this task instance (e.g. *Information Requested*)
`INITIATOR`	Username who created the task instance
`ACTUAL_OWNER`	Username of assigned potential owner
`DETAIL_PK`	*Details Primary Key* for this instance's related business entity row
`TASK_DEF_STATIC_ID`	Unique task definition identifier for this task instance
`DETAILS_LINK_TARGET`	URL of the task details page for this instance

25.1.9.1.6.4 Limiting Which Types of Tasks Appear

Add a WHERE clause to show only selected task definition types.

In a new *Unified Task List* page, the *Content Row* region's query contains no `WHERE` clause. The user can use the *Smart Filters* region at runtime to narrow the list, but sometimes you may want to *limit* which tasks show up in the list at all. The `table()` operator wrapping the `GET_TASKS` function call lets its results participate in the query like a regular table. So, just add a `WHERE` clause to the query to filter the list. For example, as shown below, to show only tasks based on two specific task definition types you can add a `WHERE` clause that filters on the `TASK_DEF_STATIC_ID` column.

```
select task_id,
       task_type,
```


Copyright © 2026, Oracle and/or its affiliates.

```
      ▒
      badge_state
  from table ( apex_human_task.get_tasks (
                  p_context            => 'MY_TASKS',
                  p_show_expired_tasks => :P33_SHOW_EXPIRED,
                  p_application_id => :APP_ID
                  ) )
 where task_def_static_id in ('NEW_PATIENT','PURCHASE_OFFICE_SUPPLIES')
```

25.1.9.1.6.5 Joining Extra Tables into Task List

Join application tables into the task list to display related business data.

In a new *Unified Task List* page, the *Content Row* region's `select` list includes only columns related to the task itself. Sometimes, it's useful to join in data from other application tables using the *Details Primary Key* value in the join clause.

For example, the following query adds a left join to the `FRC_PATIENTS` table based on the value of the `DETAIL_PK` column for task instances of the `NEW_PATIENT` type. This lets the task list page display the related patient name.

Notice the table alias `T` is added for the `table(apex_human_tasks.get_tasks())` row source, and that table alias is added to the columns in the `SELECT` list coming from that source. In addition, the table alias `P` for the `FRC_PATIENTS` table lets the `PATIENT_NAME` column concatenate the first name and last name values from the related patient table.

```
select t.task_id,
       t.task_type,
       ▒
       t.badge_state,
       p.first_name||' '||p.last_name as patient_name
  from table ( apex_human_task.get_tasks (
                  p_context            => 'MY_TASKS',
                  p_show_expired_tasks => :P33_SHOW_EXPIRED,
                  p_application_id     => :APP_ID
                  ) ) t
  left join frc_patients p
    on p.id = case task_def_static_id
                when 'NEW_PATIENT'
                then to_number(t.detail_pk)
              end
 where task_def_static_id in ('NEW_PATIENT','PURCHASE_OFFICE_SUPPLIES')
```

25.1.9.1.7 Collecting Data Using a Human Task

Create a human task from a workflow to collect user input before the flow continues.

Use a *Human Task - Create* activity in a workflow to create an instance of a work item to be completed. A task definition formalizes the work item by configuring potential owners, parameters, and actions that can react to task lifecycle events. The workflow waits until the task completes its lifecycle before continuing, and can return the task id, the username of the actual owner, and the outcome back into workflow variables.

- Defining Updatable Task Parameters
  Define updatable task parameters to collect values from the task owner.

Copyright © 2026, Oracle and/or its affiliates.

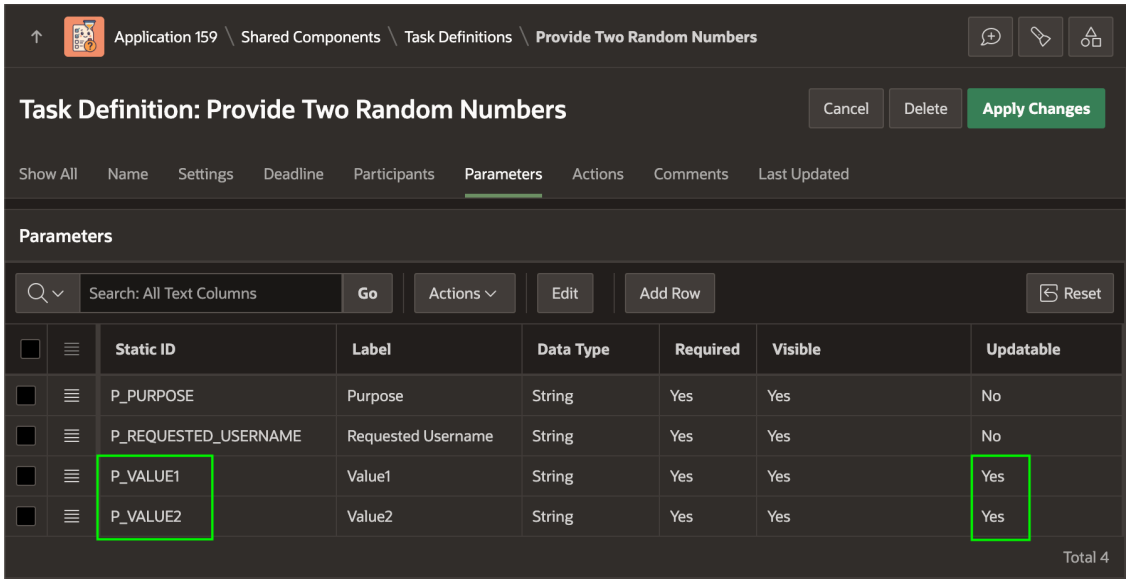
- Controlling Parameter Visibility
  Control which task parameters appear on task details pages.
- Parameterizing the Task Assignee
  Use a task parameter to supply the potential owner at runtime.
- Creating the Human Task Instance
  Create a human task instance and store its task ID and actual owner in workflow variables.
- Passing Workflow Parameter to a Task
  Pass a workflow parameter value into a task definition parameter.
- Assigning Task Parameters to Variables
  Store completed task parameter values in workflow variables for later activities.

25.1.9.1.7.1 Defining Updatable Task Parameters

Define updatable task parameters to collect values from the task owner.

If a task requires the actual owner to supply additional data values, model them as *updatable* task parameters. Notice below that the *Provide Two Random Numbers* task definition accepts two read-only ones to specify the purpose of the request and the requested username. It also defines two *updatable* parameters `P_VALUE1` and `P_VALUE2`. They let the actual owner provide two values while completing the task. The workflow can retrieve these task parameter values using an API in the `APEX_HUMAN_TASK` package.

Figure 25-28 Updatable Action Task Definition Parameters Model User-Supplied Data



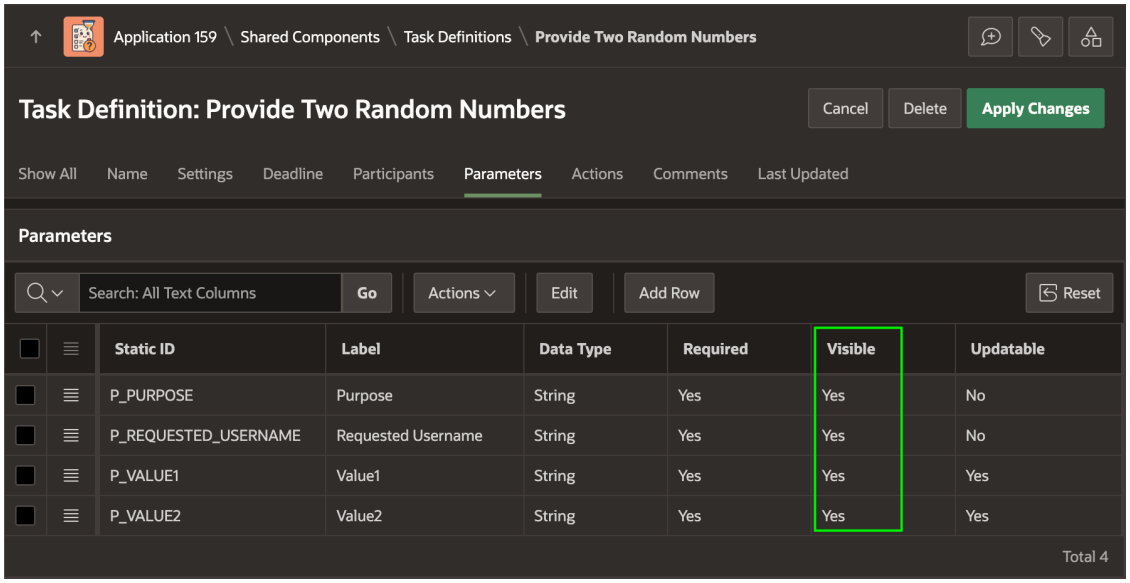
25.1.9.1.7.2 Controlling Parameter Visibility

Control which task parameters appear on task details pages.

Use a parameter's *Visible* setting to control whether it appears by default in the task details page. This visibility hint lets you avoid displaying the parameter even in a *custom* task details page by including `IS_VISIBLE='Y'` filter in your query against the `APEX_TASK_PARAMETERS` view.

Copyright © 2026, Oracle and/or its affiliates.

Figure 25-29 Disable Visible Flag to Hide a Parameter from the Task Details Page

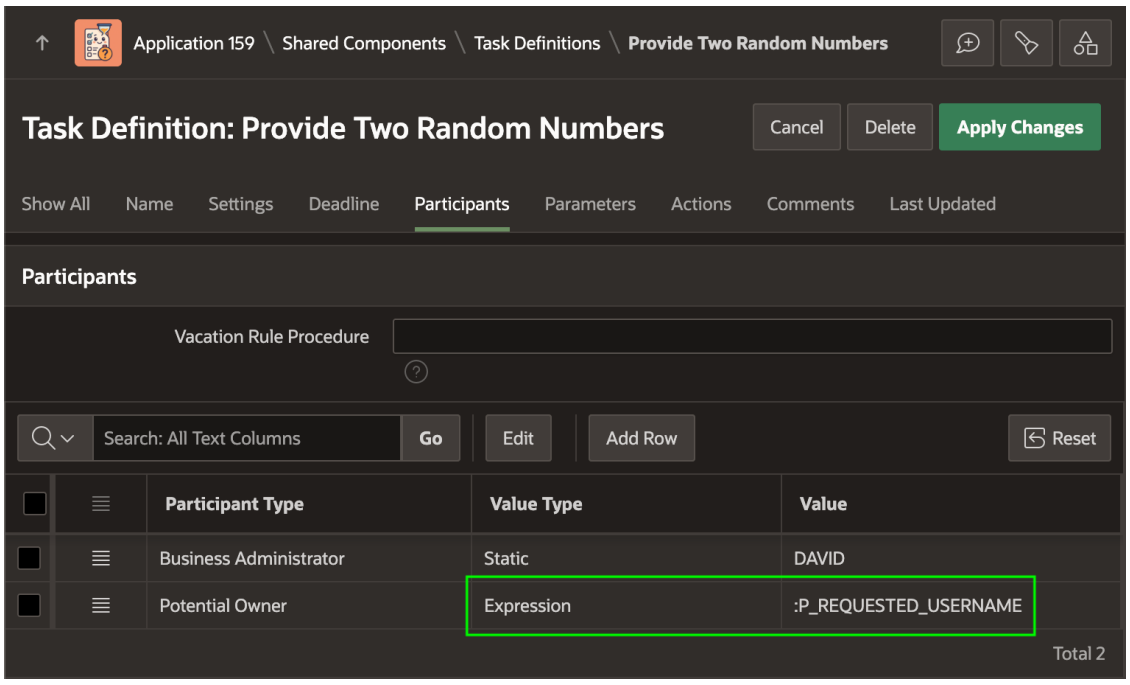


25.1.9.1.7.3 Parameterizing the Task Assignee

Use a task parameter to supply the potential owner at runtime.

The *Provide Two Random Numbers* task definition uses an *Expression* to compute its Potential Owner. As shown below, the simple expression just returns the value of the `P_REQUESTED_USERNAME` parameter by referencing it as a bind variable. This lets the workflow pass in the desired assignee for the task in the *Human Task - Create* activity.

Figure 25-30 Configuring Task Potential Owners Using a Parameter Value



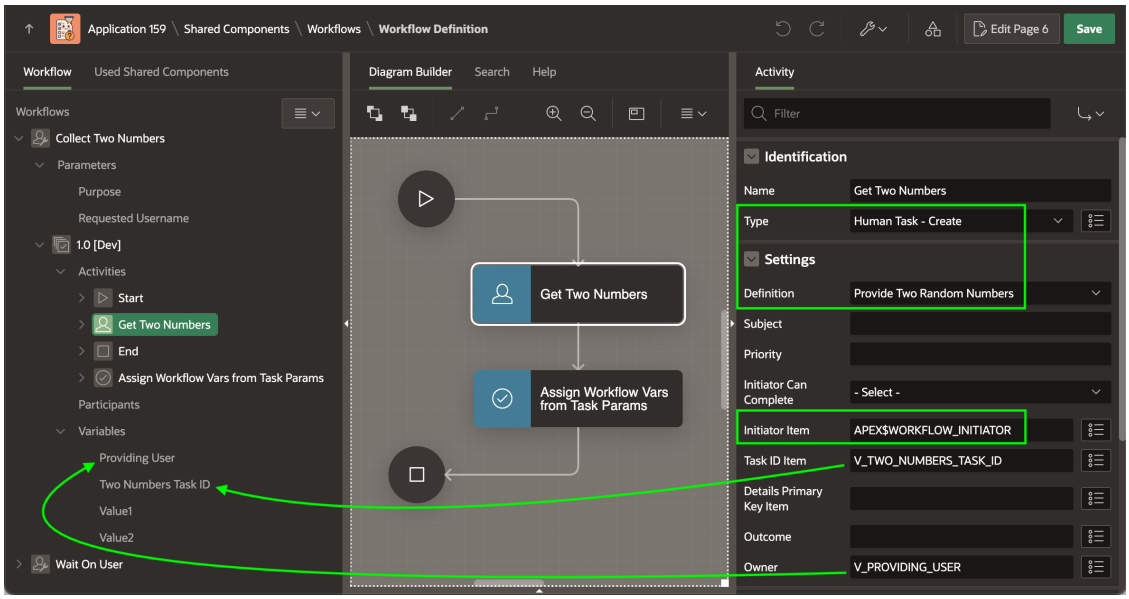
Copyright © 2026, Oracle and/or its affiliates.

25.1.9.1.7.4 Creating the Human Task Instance

Create a human task instance and store its task ID and actual owner in workflow variables.

The *Get Two Numbers* activity of type *Human Task - Create* shown below configures its *Definition* property to use the *Provide Two Random Numbers* task definition. Notice how it uses the special system substitution value `APEX$WORKFLOW_INITIATOR` to pass along the username who started the workflow as the initiator of the task instance it's creating. The *Task ID Item* and *Owner* property values name two workflow variables `V_TWO_NUMBERS_TASK_ID` and `V_PROVIDING_USER` that will store the task id and actual owner of the task instance upon completion.

Figure 25-31 Configuring a Human Task - Create Activity to Use a Task Definition



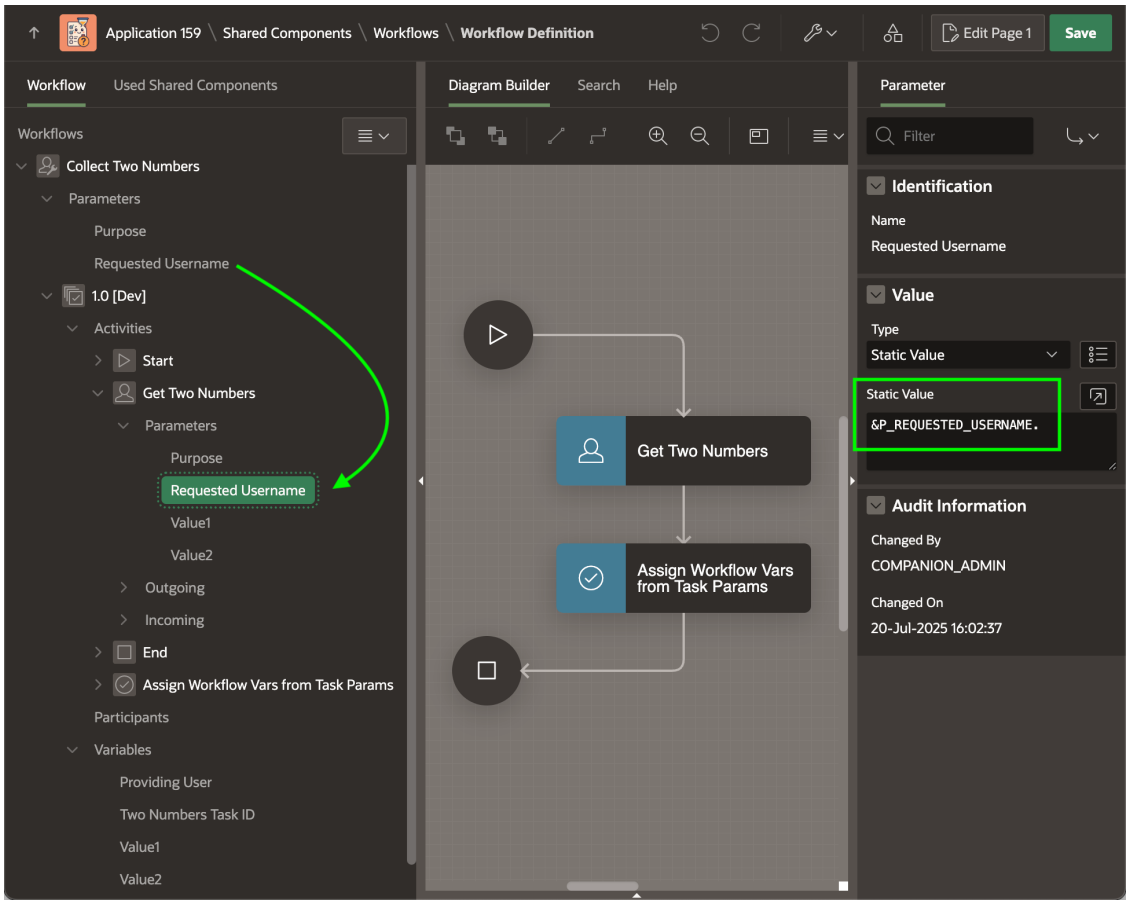
25.1.9.1.7.5 Passing Workflow Parameter to a Task

Pass a workflow parameter value into a task definition parameter.

A workflow configures task definition parameters using corresponding items in the *Workflows* tree indented under the *Human Task - Create* activity's *Parameters* heading. As shown below, the *Get Two Numbers* activity configures the task definition parameter `P_REQUESTED_USERNAME` to the value of the workflow's parameter of the same name using the `&P_REQUESTED_USERNAME.` substitution in the *Static Value* field in the Property Editor.

Copyright © 2026, Oracle and/or its affiliates.

Figure 25-32 Passing the Value of a Workflow Parameter to a Task Definition Parameter



25.1.9.1.7.6 Assigning Task Parameters to Variables

Store completed task parameter values in workflow variables for later activities.

The workflow pauses after creating the human task instance. When the task's lifecycle completes, it proceeds to the next activity. If later workflow activities need to reference the values of the updatable task parameters acquired during the human task lifecycle, they can use the `GET_TASK_PARAMETER_VALUE` function in the `APEX_HUMAN_TASK` package.

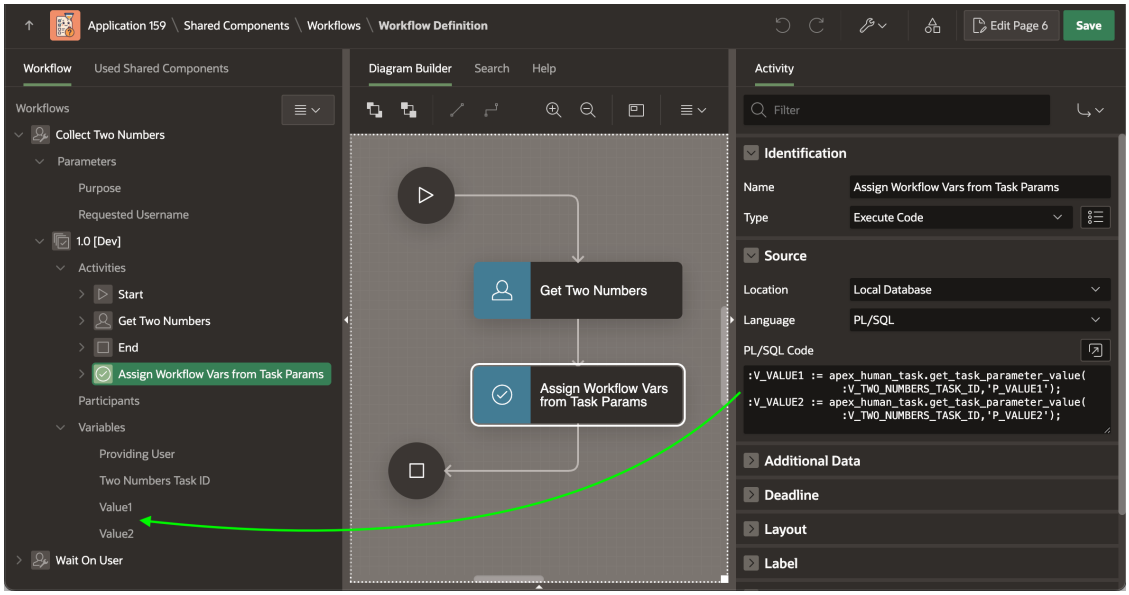
Its first parameter expects the unique id of a task, motivating why it's useful for a *Human Task - Create* activity to return its task ID into a workflow variable. Recall that in this workflow, the return variable for the task ID is `V_TWO_NUMBERS_TASK_ID`.

If you find it more convenient to store a task parameter value in a *workflow* variable, then as shown below, a subsequent *Execute Code* activity can call this same API to assign the updatable task parameter values into workflow variables using code like:

```
:V_VALUE1 := apex_human_task.get_task_parameter_value(
                :V_TWO_NUMBERS_TASK_ID,'P_VALUE1');
:V_VALUE2 := apex_human_task.get_task_parameter_value(
                :V_TWO_NUMBERS_TASK_ID,'P_VALUE2');
```


Copyright © 2026, Oracle and/or its affiliates.

Figure 25-33 Retrieving Task Parameter Values into Workflow Variables



25.1.9.1.8 Using and Tuning the Task Details Page

Explore how users work with a Task Details page and how developers can tune it.

User LUCY starts a *Collect Two Numbers* workflow, passing in a *Purpose* parameter value of "Testing Purposes" and a *Requested User* value of LE0. Both users participate in the task lifecycle using a combination of:

- "Requested Tasks" *Unified Task List* page with Report Context "Initiated by Me"
- "My Tasks" *Unified Task List* page with Report Context "My Tasks"
- "Task Details" page associated with the *Provide Two Random Numbers* task definition.
- "Initiated Workflows" *Workflow Console* page with Report Context "Initiated by Me"
- Viewing Work Items You Requested
  View requested work items with a Unified Task List page filtered to tasks the user initiated.
- Adding a Task Details Page
  Add or assign a Task Details page so users can open a task from its subject link.
- Viewing Tasks Requiring Your Attention
  View assigned work items from a My Tasks page and open a task's details page.
- Requesting More Info for a Task
  Request more information from the task initiator when the assigned work is unclear.
- Responding to Requests for Information
  Respond to a request for information so the task can return to its owner.
- Editing Updatable Parameters
  Edit updatable task parameters from the Task Details page.
- Completing an Action Task
  Complete an action task when its work is done.
- Inspecting Overall Workflow Status
  Inspect workflow details to review status, history, parameters, and variables.

Copyright © 2026, Oracle and/or its affiliates.

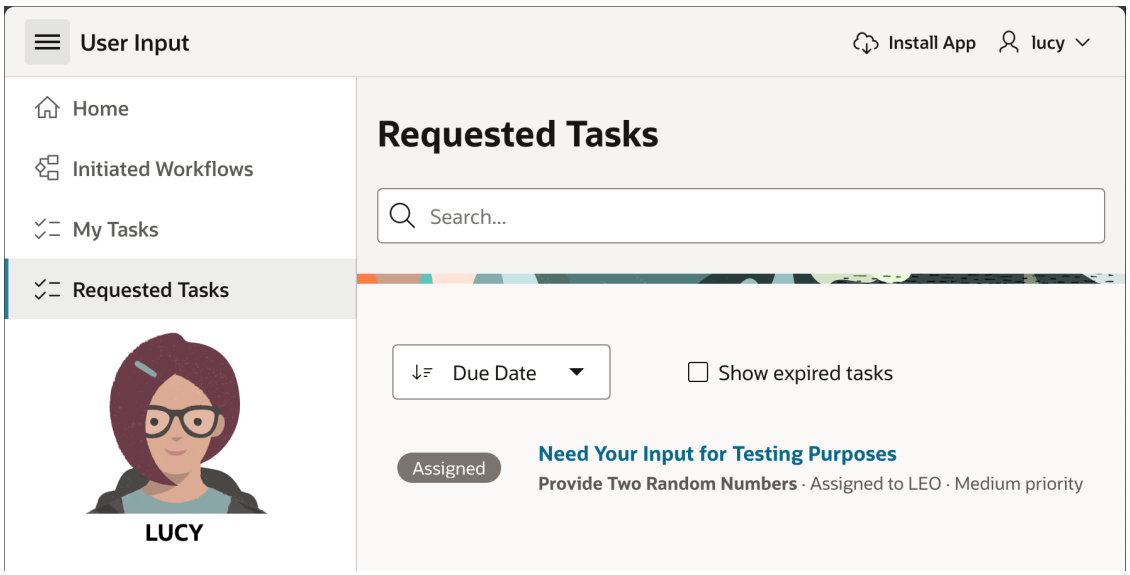
- Fine-Tuning a Task Details Page
  Fine-tune a generated Task Details page for the needs of each task definition.

25.1.9.1.8.1 Viewing Work Items You Requested

View requested work items with a Unified Task List page filtered to tasks the user initiated.

User LUCY visits a *Requested Tasks* page to see a list of requests she's made. In this simple example, there is only the *Provide Two Random Numbers* task definition in the list, but in a real application she would see work items of any type that she initiated. The page shown below is an example of a *Unified Task List* page you can add to an application with the Create Page Wizard. Since its goal is to show users the tasks they initiated, it uses the Report Context of "Initiated by Me". If a task instance has an associated Task Details page, then the task *Subject* line is a link Lucy can click on to review the details of the task.

Figure 25-34 Lucy Initiates a Workflow and Sees It In Her "Requested Tasks" Page



25.1.9.1.8.2 Adding a Task Details Page

Add or assign a Task Details page so users can open a task from its subject link.

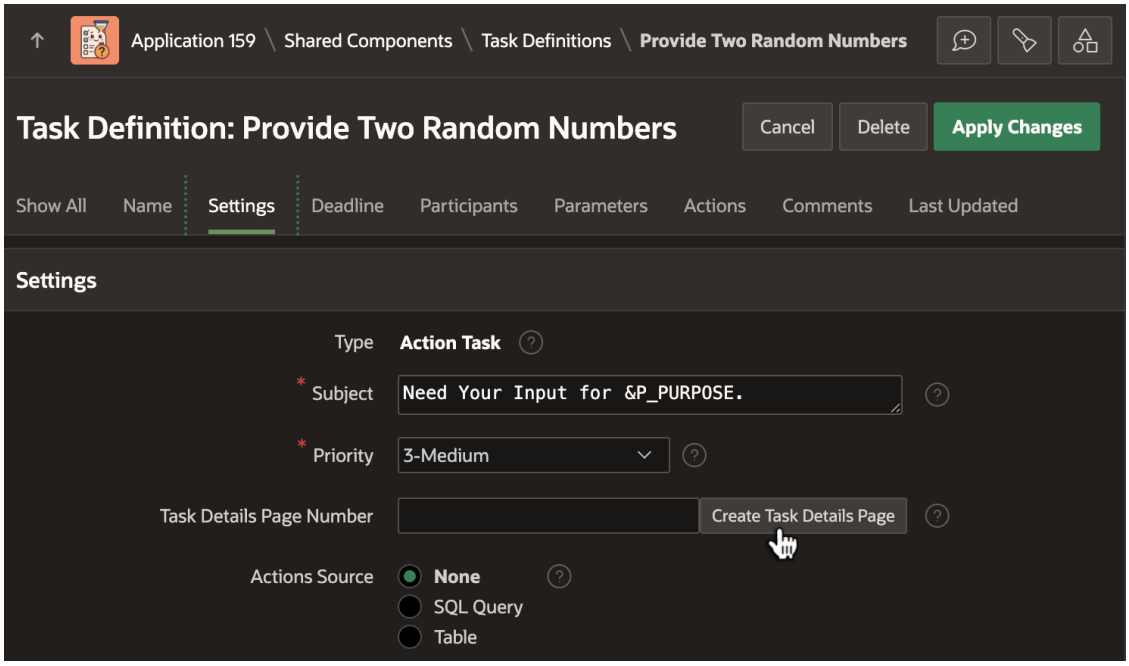
Like all task definitions, the action task *Provide Two Random Numbers* in this example starts off with no Task Details Page. The blank *Task Details Page Number* field below confirms this. Before clicking the *(Create Task Details Page)* button, you can either:

- Leave *Task Details Page Number* blank,
- Enter the number of an existing page, or
- Type the number of your desired new page.

If you leave the field blank, APEX asks your confirmation to proceed using the next available page number.

Copyright © 2026, Oracle and/or its affiliates.

Figure 25-35 Creating a Task Details Page for a Task



In this case, the new Task Details Page gets assigned page number `6`. After you associate a Task Details Page to a task definition, it displays thereafter as a *Task Details Page URL*. It uses the classic format of an APEX page URL:

`f?p=`*`App`*`:`*`Page`*`:`*`Session`*`:`*`Request`*`:`*`Debug`*`:`*`ClearCache`*`:`*`itemNames`*`:`*`itemValues`*

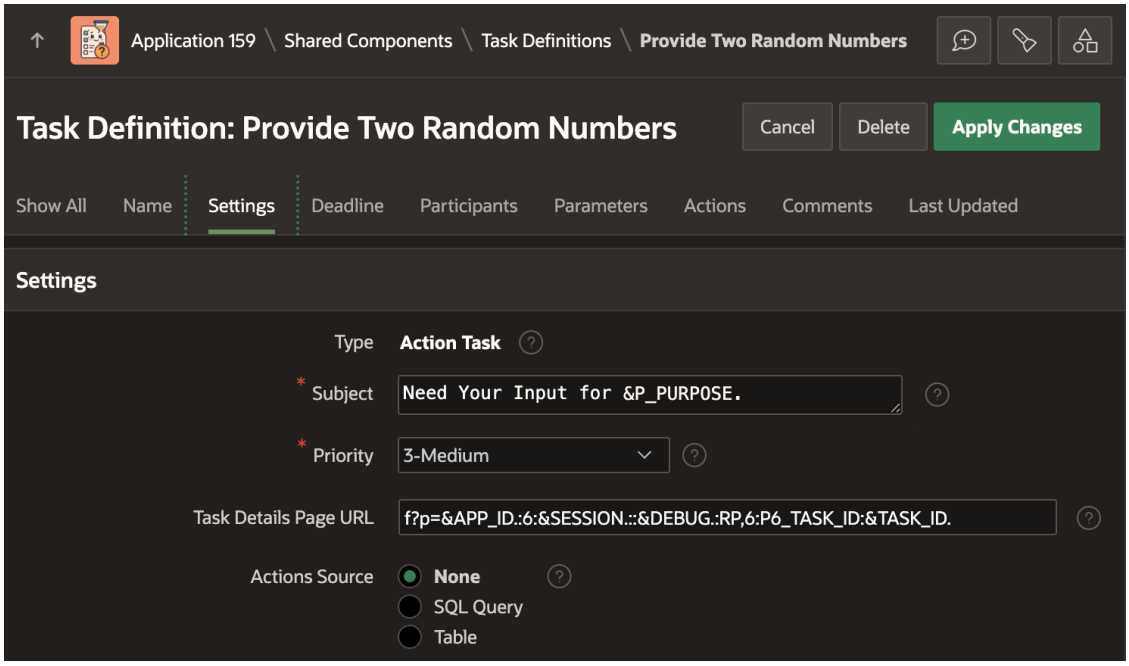
The Task Details Page URL shown below for page 6 looks like this:

```
f?p=&APP_ID.:6:&SESSION.::&DEBUG.:RP,6:P6_TASK_ID:&TASK_ID.
```

It leaves the *`Request`* segment empty and uses substitution values for `&APP_ID.`, `&SESSION.`, `&DEBUG.`, and `&TASK_ID.` in the respective segments of the `p` query string parameter value. In short, the URL navigates to page 6 in the current application, using the current session, and the current debug mode, passing the value of the current task instance's unique ID as the value of the `P6_TASK_ID` parameter in that page.

Copyright © 2026, Oracle and/or its affiliates.

Figure 25-36 Task Details Page URL Appears After You Create or Assign a Page



> **Tip**
>
> In addition to the substitutions mentioned above, your *Task Details Page URL* can also reference the substitution string `&DETAIL_PK.` to pass the value of the *Details Primary Key* of the business entity row associated with the task instance. This can be useful if your task details page needs to display and possibly modify data in that associated business entity row using a *Form* region you add to the generated page as a fine-tuning step.

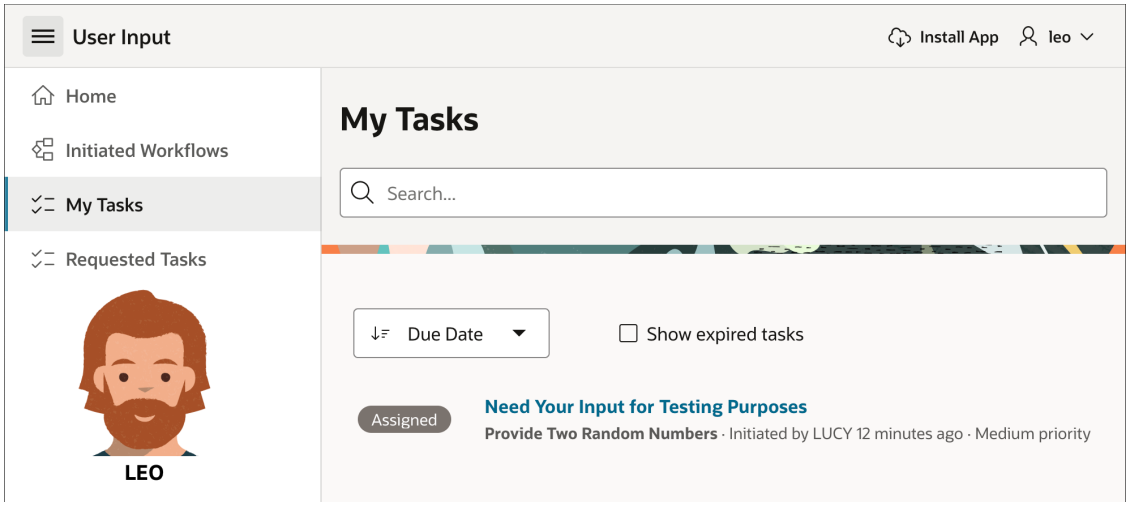
25.1.9.1.8.3 Viewing Tasks Requiring Your Attention

View assigned work items from a My Tasks page and open a task's details page.

User `LEO` visits his *My Tasks* page to view any work items requiring his attention. As shown below, he sees `LUCY` initiated a task that needs his input. To review the details of a task, Leo can click on any task *Subject* link to open its Task Details Page.

Copyright © 2026, Oracle and/or its affiliates.

Figure 25-37 Leo Sees Task Assigned to Him in His "My Tasks" Page



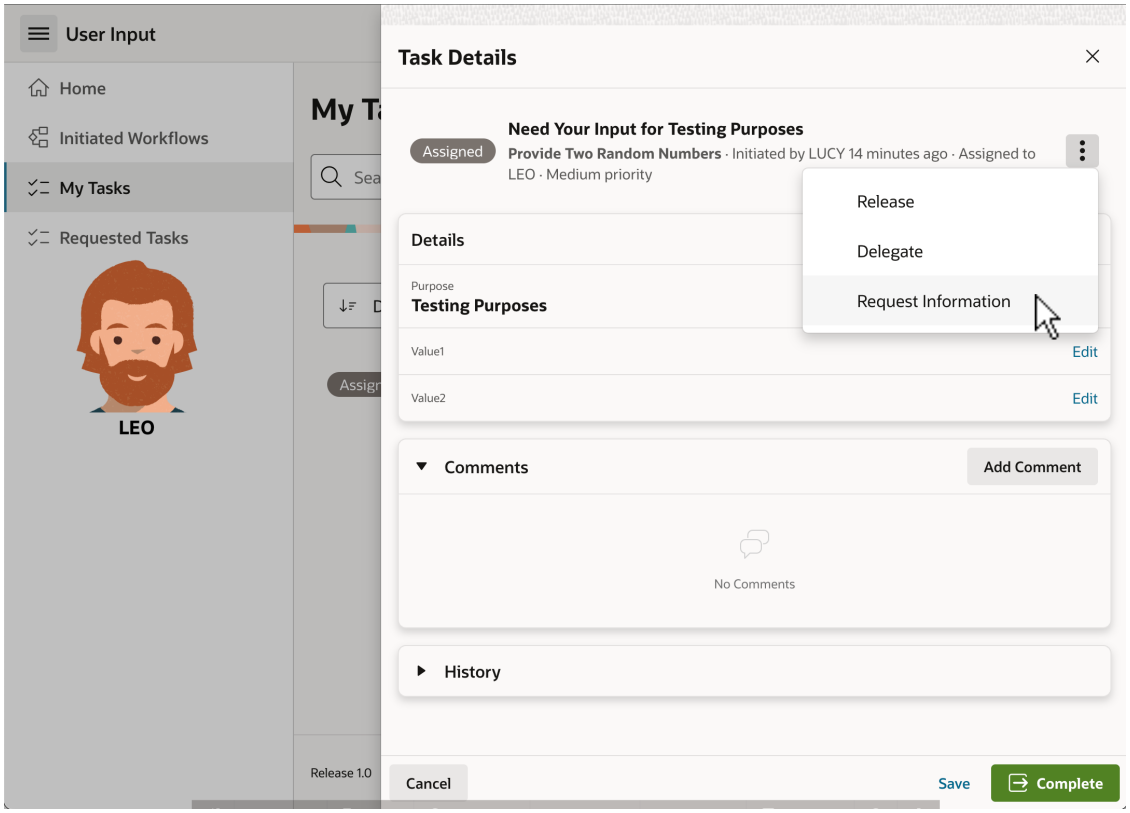
25.1.9.1.8.4 Requesting More Info for a Task

Request more information from the task initiator when the assigned work is unclear.

Clicking on a task *Subject* in the *My Tasks* page opens its Task Details Page. As shown below, by default, it contains sections that display the values of task parameters *Purpose*, *Value1*, and *Value2*, along with a *Comments* and *History* section, too. Leo notices *Edit* links next to the *Value1* and *Value2* parameters, but remains a bit confused. He clicks the three-dot menu and requests more information about the task from Lucy, the requester.

Copyright © 2026, Oracle and/or its affiliates.

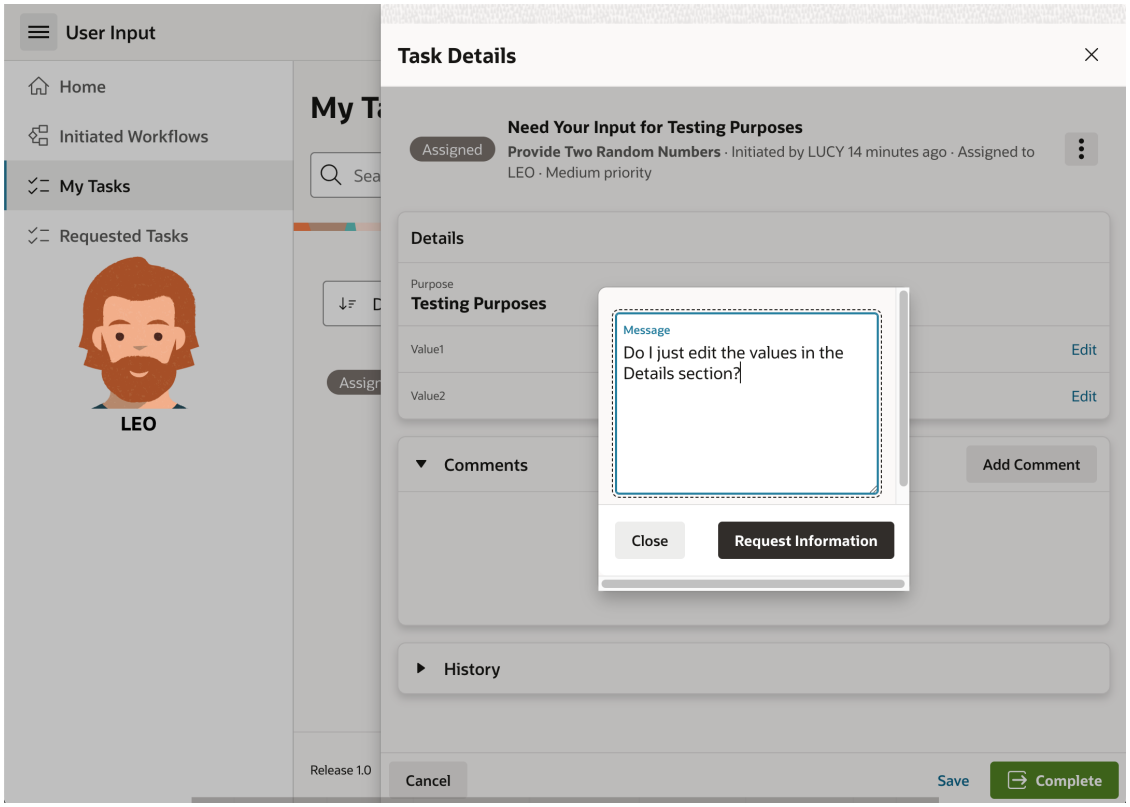
Figure 25-38 Leo Asks for More Information to Clarify the Task



A dialog box appears and, as shown below, Leo types in his question, "Do I just edit the values in the Details section?" and clicks *(Request Information).*

Copyright © 2026, Oracle and/or its affiliates.

Figure 25-39 Leo Enters His More Information Question

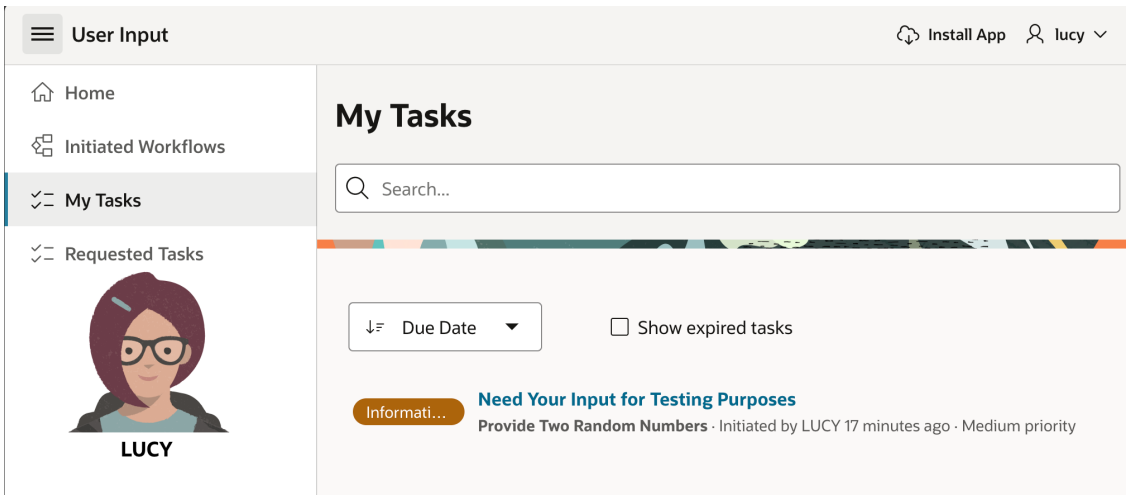


25.1.9.1.8.5 Responding to Requests for Information

Respond to a request for information so the task can return to its owner.

While her *My Tasks* list was previously empty, she now sees the task she requested show up in the list since it requires her attention. The *Information Requested* badge gives her a clue that Leo must have a question.

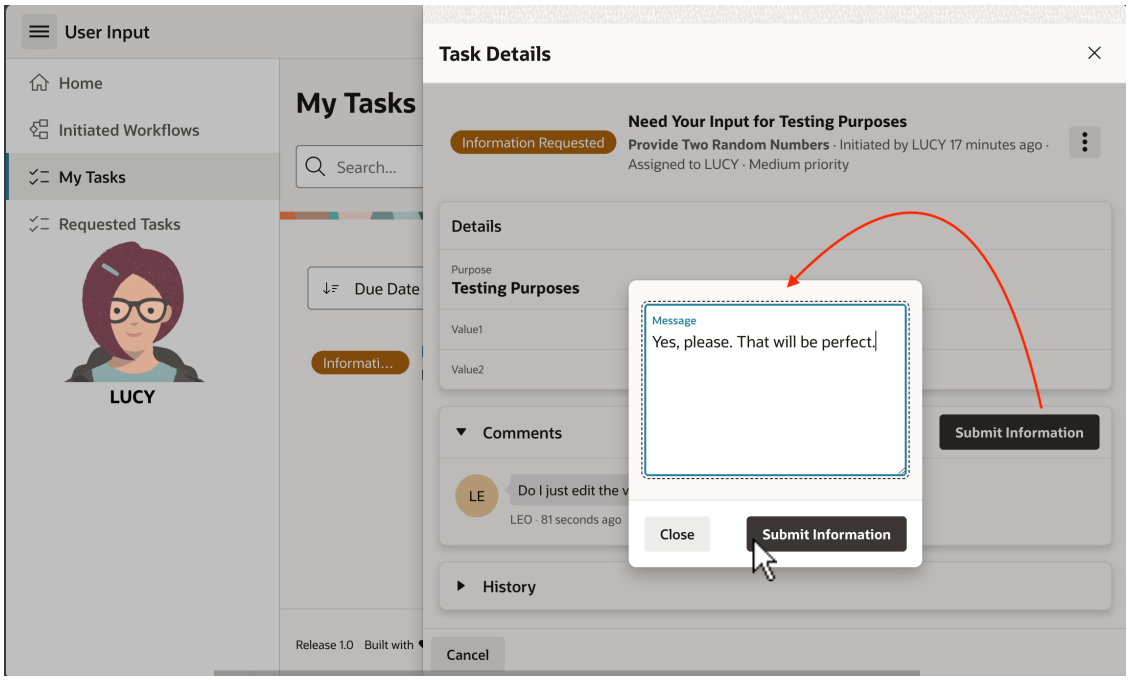
Figure 25-40 Lucy Sees the Task Shows Up in Her "My Tasks" Page



Copyright © 2026, Oracle and/or its affiliates.

Clicking on the *Subject*, the Task Details page opens, and as shown below, Lucy clarifies, "Yes, please. That will be perfect!" Then she clicks *(Submit Information)* to send the task back to Leo along with her clarification.

Figure 25-41 Lucy Clarifies and Submits the Additional Information



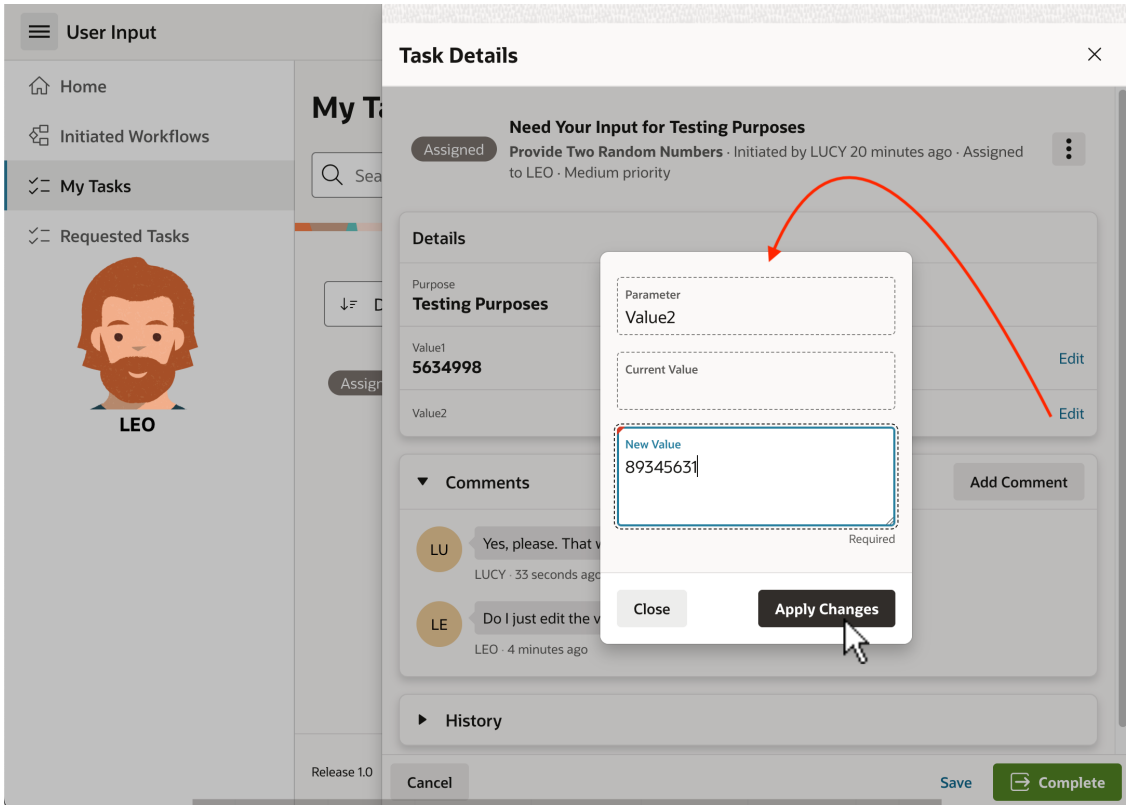
25.1.9.1.8.6 Editing Updatable Parameters

Edit updatable task parameters from the Task Details page.

The work item shows up again in Leo's *My Tasks* page and he clicks to open the Task Details page. He sees the conversation with Lucy in the *Comments* section and proceeds. As shown below, he enters the two values Lucy needs by clicking the *Edit* link next to the *Value1* and *Value2* parameters.

Copyright © 2026, Oracle and/or its affiliates.

Figure 25-42 Leo Enters Parameter Values



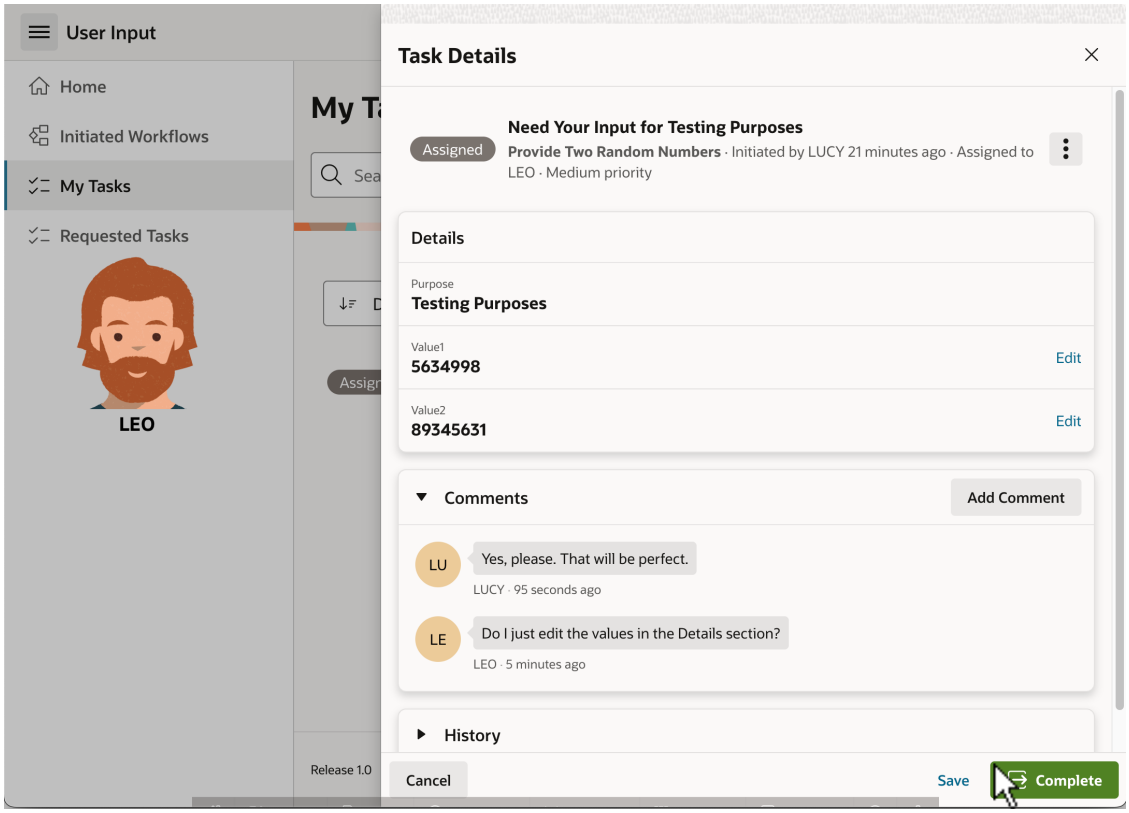
25.1.9.1.8.7 Completing an Action Task

Complete an action task when its work is done.

After entering values for *Value1* and *Value2* in the Task Details page, Leo is done. He clicks the *(Complete)* button to mark the work item as completed.

Copyright © 2026, Oracle and/or its affiliates.

Figure 25-43 Leo Completes the Task



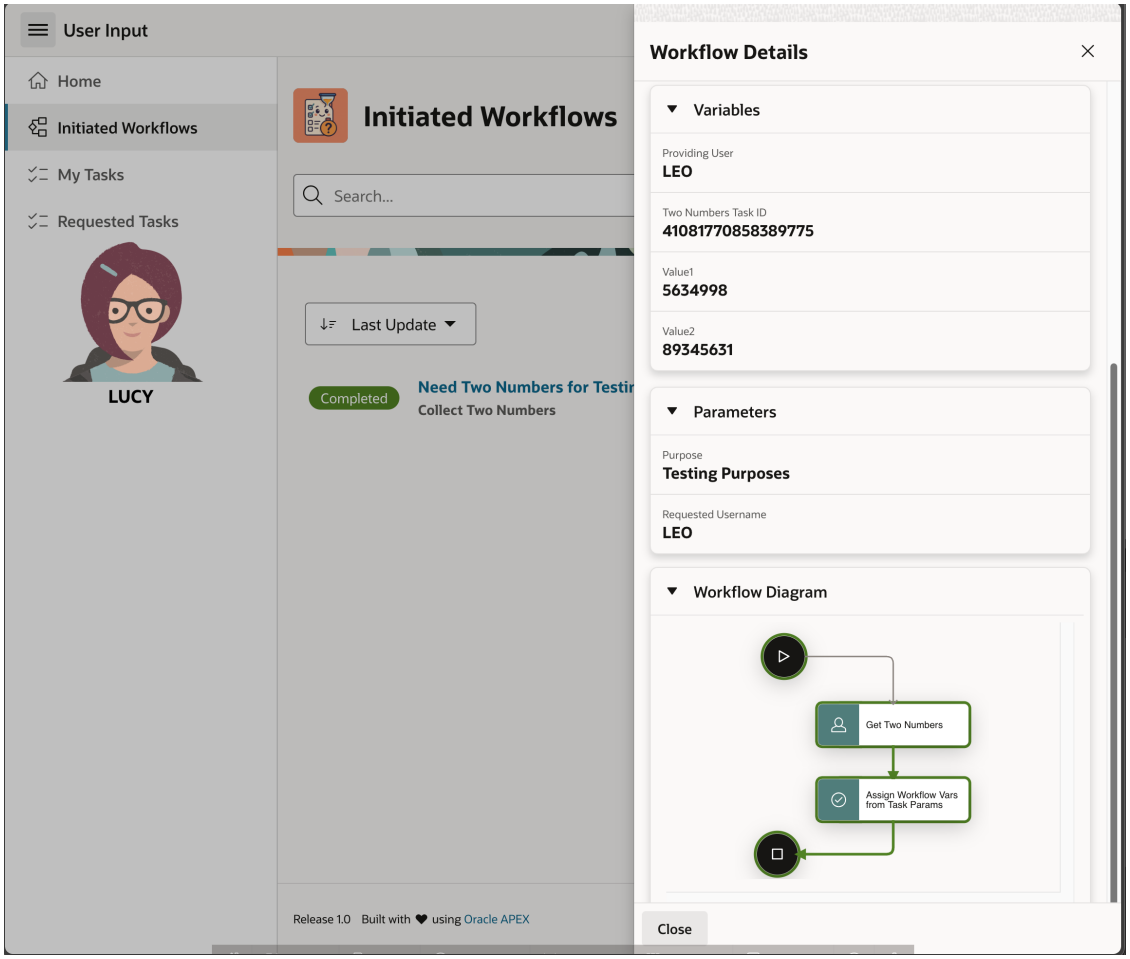
25.1.9.1.8.8 Inspecting Overall Workflow Status

Inspect workflow details to review status, history, parameters, and variables.

From her *Initiated Workflows* page, Lucy can see all the workflows she created. It's a *Workflow Console* page with Report Context of "Initiated by Me". Clicking on the *Title* link, the Workflow Details page opens. As shown below, the workflow details page the wizard generates displays the variable and parameter values as well as visual diagram of the workflow's current status. Scrolling up and down, Lucy can also see the activity steps list and history that are not visible below. She sees the workflow has completed, and notices the *Value1* and *Value2* of the updatable task parameters were assigned to the workflow values of the same names.

Copyright © 2026, Oracle and/or its affiliates.

Figure 25-44 Lucy Reviews Completed Workflow in Her "Initiated Workflows" Page



- Reviewing a Completed Task
  Review completed task details from the list of work items you requested.

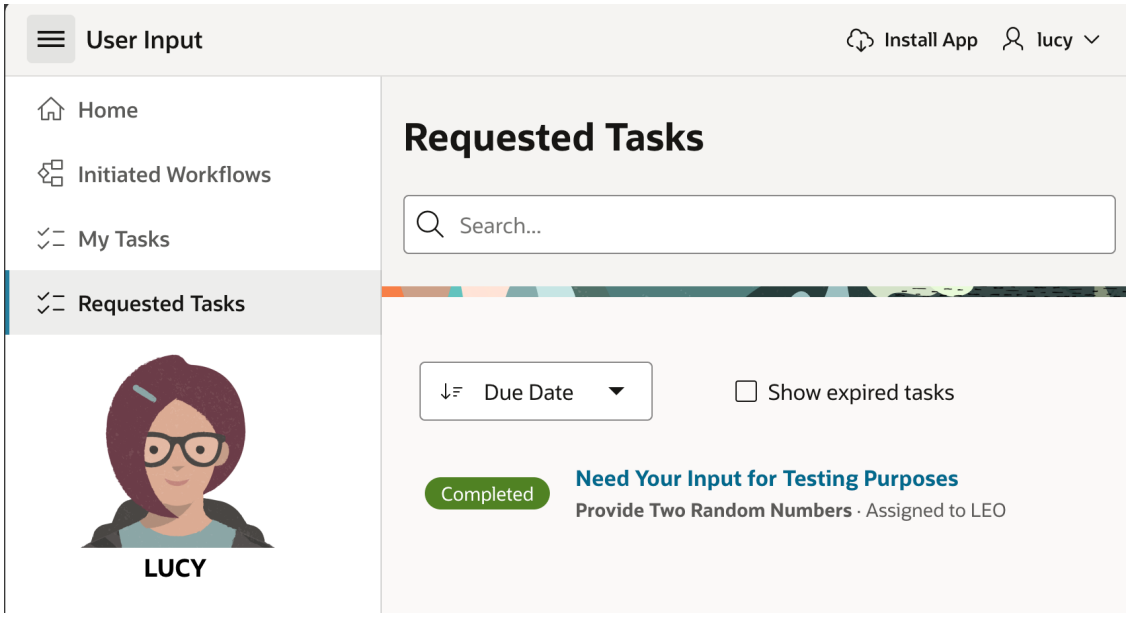
25.1.9.1.8.8.1 Reviewing a Completed Task

Review completed task details from the list of work items you requested.

In her *Requested Tasks* page, Lucy notices the *Completed* badge in the list, and she clicks on the *Subject* link to open the Task Details page to review Leo's work.

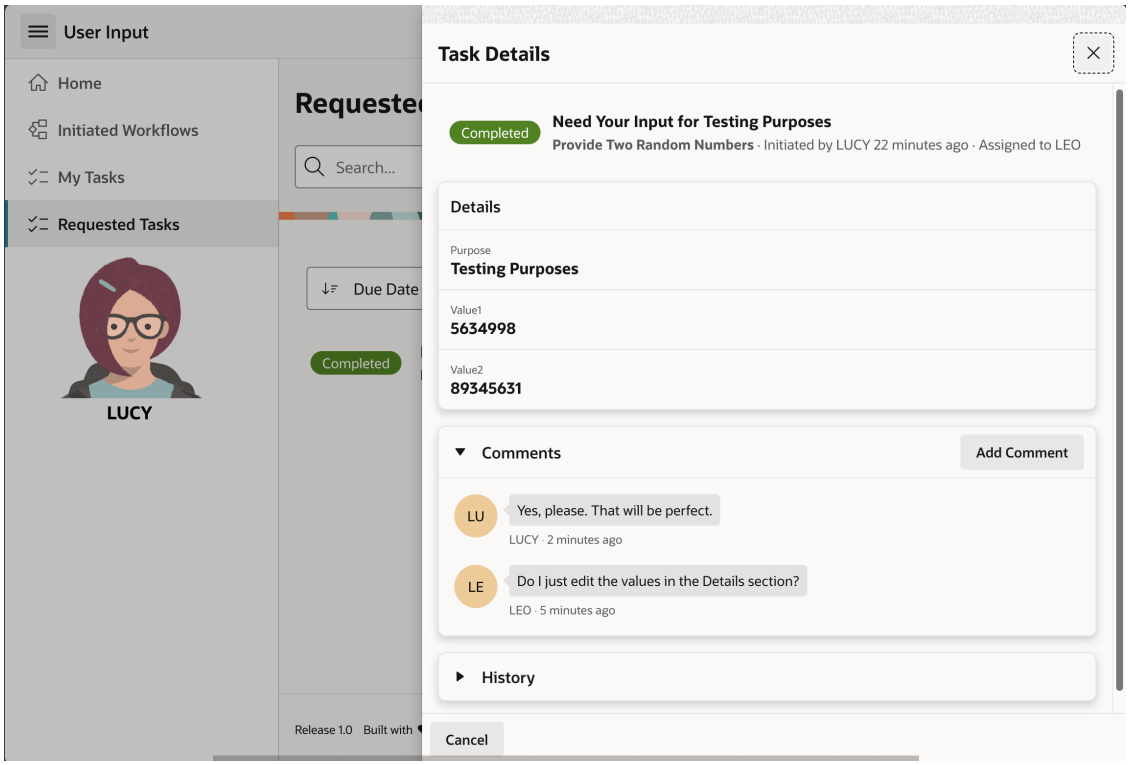
Copyright © 2026, Oracle and/or its affiliates.

Figure 25-45 Lucy Sees Request Completed in Her "Requested Tasks" Page



Lucy reviews the details of the work item and sees the two random numbers Leo entered. Stakeholders can continue adding comments to the task if needed, but otherwise its details are read-only once completed.

Figure 25-46 Lucy Reviews Task Details to See Numbers Leo Supplied



Copyright © 2026, Oracle and/or its affiliates.

25.1.9.1.8.9 Fine-Tuning a Task Details Page

Fine-tune a generated Task Details page for the needs of each task definition.

Your generated Task Details Page is a great starting point. Consider fine-tuning it to meet your particular task definition's needs. It offers useful built-in functionality, but know that it illustrates one *particular* way to use the native *Human Task* page processes and `APEX_HUMAN_TASK` programmatic APIs. Understand what you get by default, then decide on a task by task basis what to remove, add, or change.

While in *theory* you could use a single, generic task definition page with every task definition, the user experience would suffer. In *practice*, if a work item is unique enough to model as a task definition, it benefits from a task-specific Task Details Page. This way, you can show the assignee additional context tailored to that task definition, along with additional fields and functionality as required.

- Adding Page Items to Edit Parameters
  Add page items to give users a better experience when entering task parameter values.
- Making Items Conditionally Read Only
  Make task page items editable only for the assigned owner while the task is active.
- Loading Parameter Values Into Items
  Load updatable task parameter values into page items before rendering.
- Validating Parameter Page Items
  Validate task parameter page items before saving their values back to the task.
- Saving Page Items to Task Parameters
  Save validated page item values back to updatable task parameters.
- Working with a Business Entity Row
  Show or update the related business row from a task-specific details page.

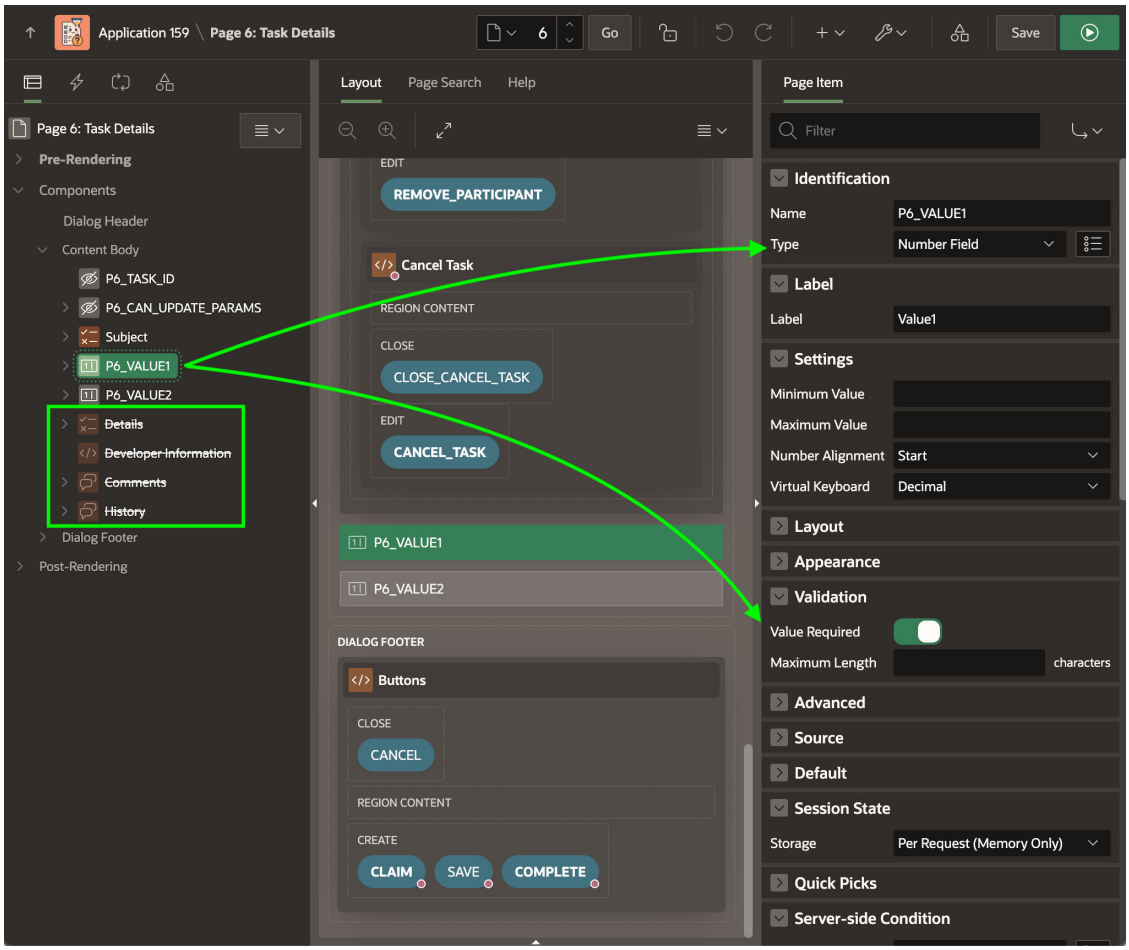
25.1.9.1.8.9.1 Adding Page Items to Edit Parameters

Add page items to give users a better experience when entering task parameter values.

The *Details* region in the generated Task Details page offers a basic parameter editing experience. However, your end users might appreciate entering data using page items. As shown below, you can right-click on a region like *Details*, *Developer Information*, *Comments*, and *History* and choose *Comment Out* to temporarily ignore those regions when the page is run. As shown below, commented out elements display with a line through their label in the tree. After adding page items `P6_VALUE1` and `P6_VALUE2` you can set their *Type* to *Number Field*, enable *Value Required*, and set their *Session State* > *Storage* to *Per Request (Memory Only)*.

Copyright © 2026, Oracle and/or its affiliates.

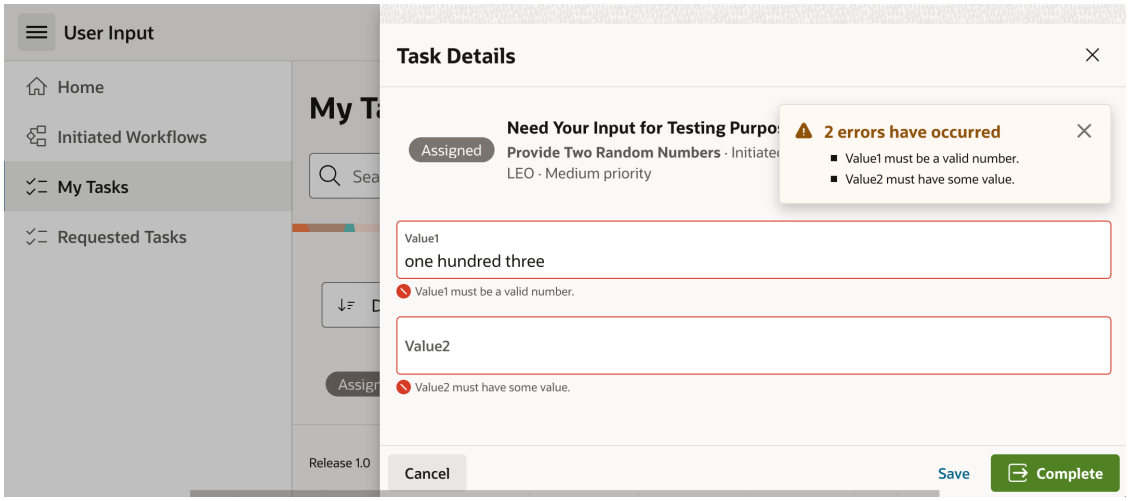
Figure 25-47 Commenting Out Regions and Adding Page Items to Enter Additional Data



Trying out the modified page, you can see that your end user immediately gets a better data entry experience. Any attempts to submit the page with non-numeric values or without filling in a required value gives the normal APEX field validation errors shown below. The figure shows the customized Task Details page for the Provide Two Random Numbers task definition. If Leo submits a *Value1* by typing in words, and leaves *Value 2* blank, he sees the error notification on clicking the (**Complete**) button. It displays *"2 errors have occurred"* and shows the two validation errors *"Value1 must be a valid number"*, and *"Value2 must have some value"*.

Copyright © 2026, Oracle and/or its affiliates.

Figure 25-48 Automatic Field Types and Required Fields Validation



25.1.9.1.8.9.2 Making Items Conditionally Read Only

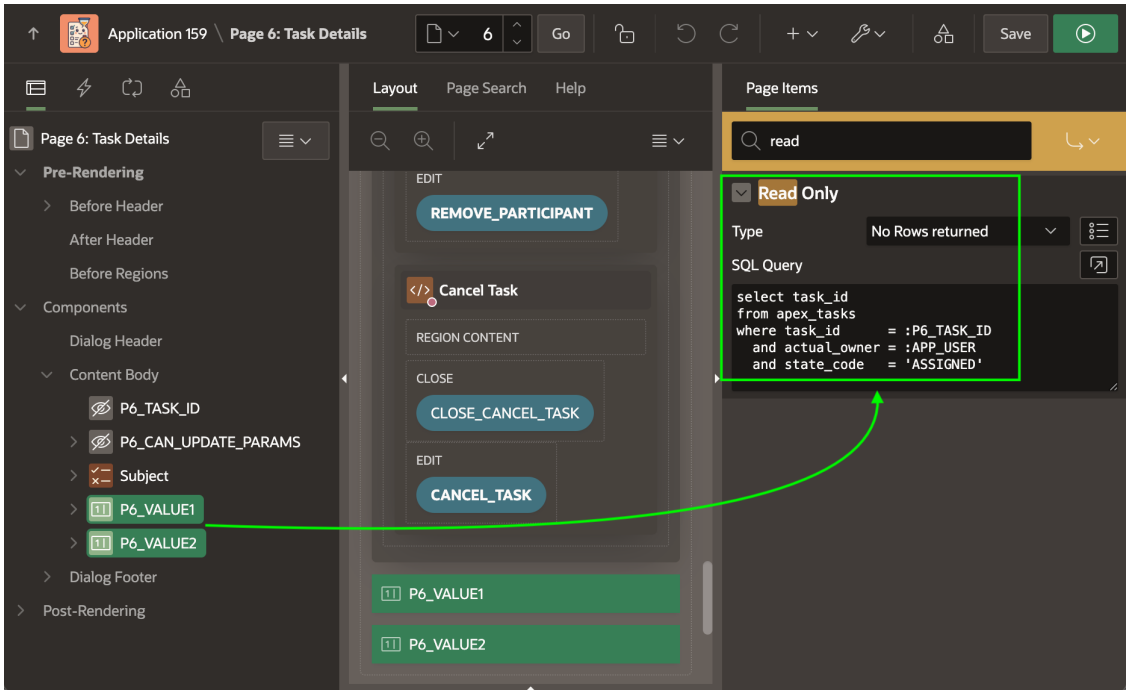
Make task page items editable only for the assigned owner while the task is active.

To make the page items editable only for the assignee, and only while the status of the task is *Assigned*, you can use the *Read Only* settings on `P6_VALUE1` and `P6_VALUE2` as shown below. The fields are be read-only if no row is returned by the query below. It returns a row when the task exists, the current user is the actual owner, and the status is `ASSIGNED`. Otherwise, no row is returned, and the fields render read-only.

```
select task_id
from apex_tasks
where task_id      = :P6_TASK_ID
  and actual_owner = :APP_USER
  and state_code   = 'ASSIGNED'
```


Copyright © 2026, Oracle and/or its affiliates.

Figure 25-49 Making Items Read Only Unless User is Assigned Owner



25.1.9.1.8.9.3 Loading Parameter Values Into Items

Load updatable task parameter values into page items before rendering.

Using an appropriate computation in the *Pre-Rendering* section of the page, as shown below, you can load the value of `P6_VALUE1` from the corresponding updatable task parameter using its *Static ID* `P_VALUE1`.

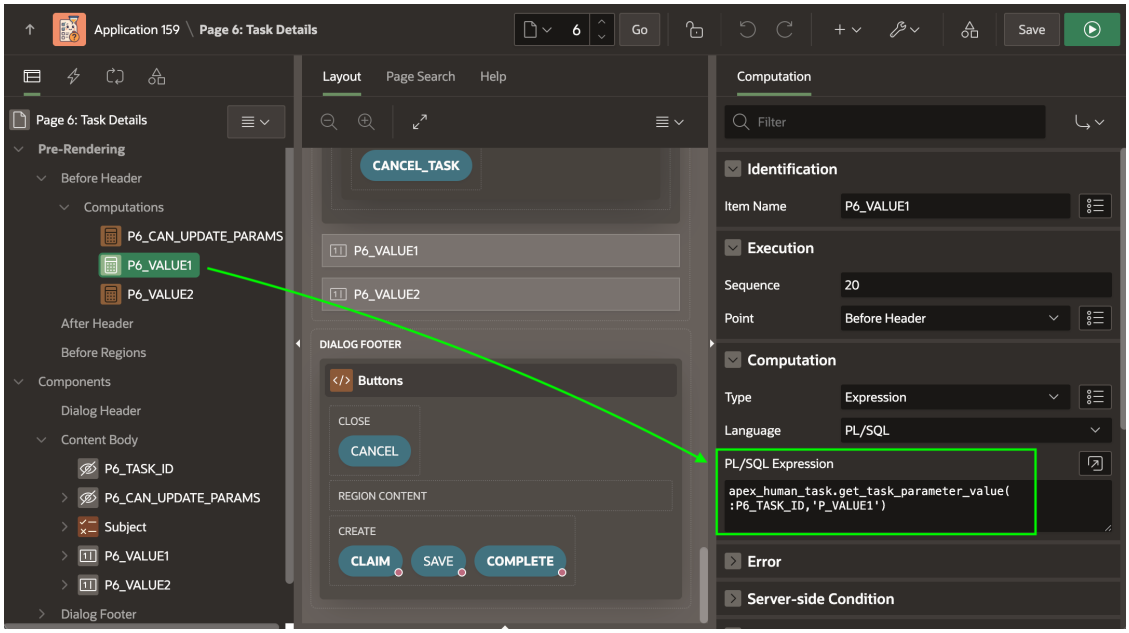
The PL/SQL expression follows, referencing the value of the task id in a hidden page item:

```
apex_human_task.get_task_parameter_value(:P6_TASK_ID,'P_VALUE1')
```

You can add a similar computation to load the value of parameter `P_VALUE2` into `P6_VALUE2`. The figure shows the `P6_VALUE1` computation selected in the component tree and highlights its PL/SQL expression described above that retrieves the value of an updateable task parameter from the current task instance.

Copyright © 2026, Oracle and/or its affiliates.

Figure 25-50 Loading Task Parameter Values Into Page Items Before Page Render



25.1.9.1.8.9.4 Validating Parameter Page Items

Validate task parameter page items before saving their values back to the task.

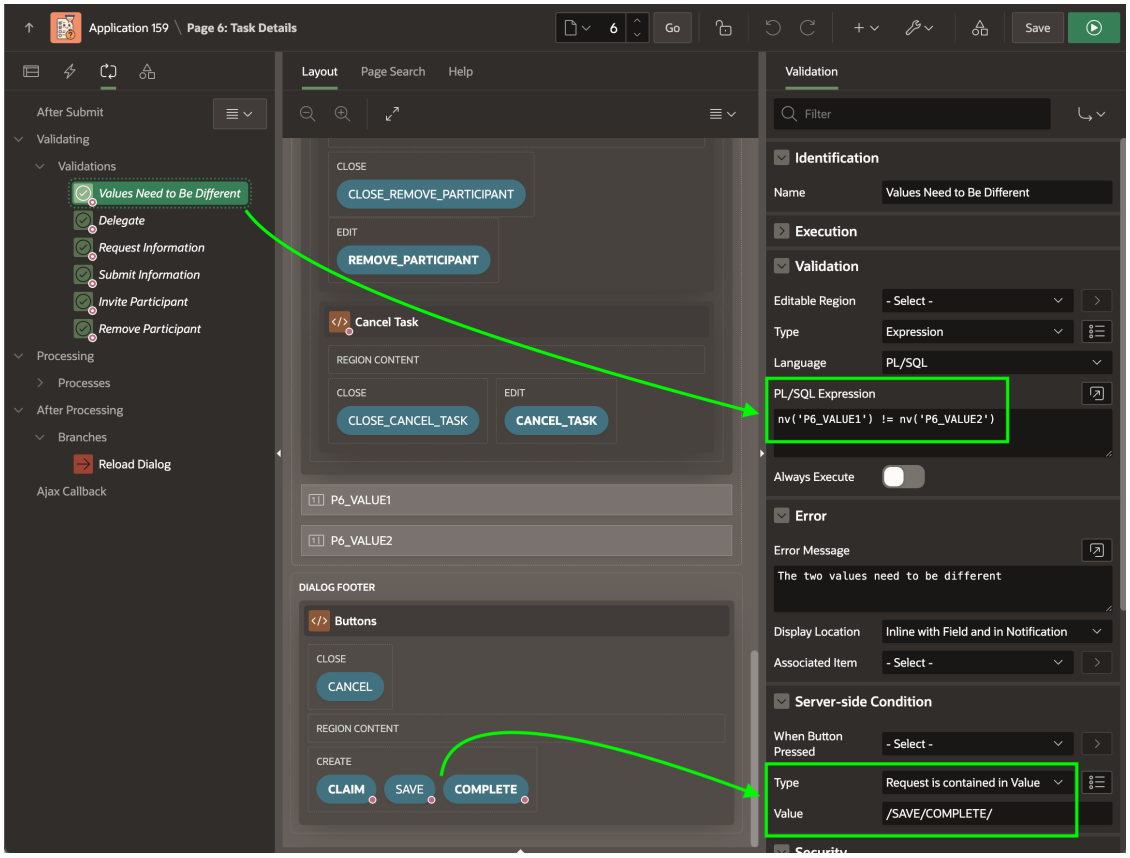
You can use other page features like validations to perform further data checks before letting the parameter values be saved back to the task. For example, imagine a rule that the two numbers provided must be distinct. You can enforce this by adding a *Values Need to Be Different* validation as shown below. It configures an appropriate error message and uses the following PL/SQL expression using the `NV` function to reference the value of a page item as a number.

```
nv('P6_VALUE1') != nv('P6_VALUE2')
```

To ensure the validation is performed only when the user clicks the `SAVE` or `COMPLETE` buttons to submit the page, you can add a *Server-side Condition* that the request is contained in the `/SAVE/COMPLETE/` value. You can use any value separator other than slash as well.

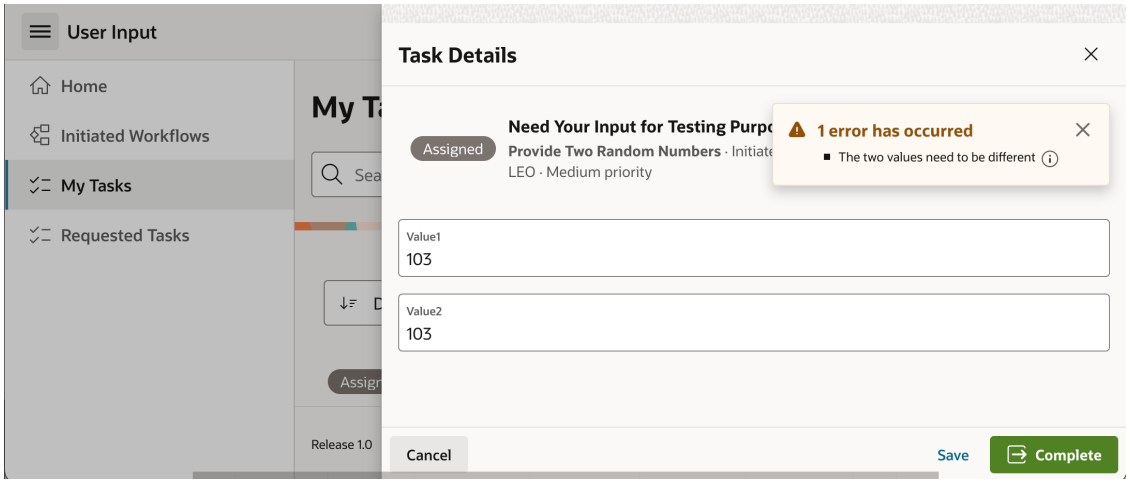
Copyright © 2026, Oracle and/or its affiliates.

Figure 25-51 Configuring a Custom Validator for Page Items in Task Details Page



As shown below, now any user who tries to submit the same value for both page items sees an error *"The two values need to be different"* and must try again.

Figure 25-52 Validation Error if User Enters Same Number Twice



25.1.9.1.8.9.5 Saving Page Items to Task Parameters

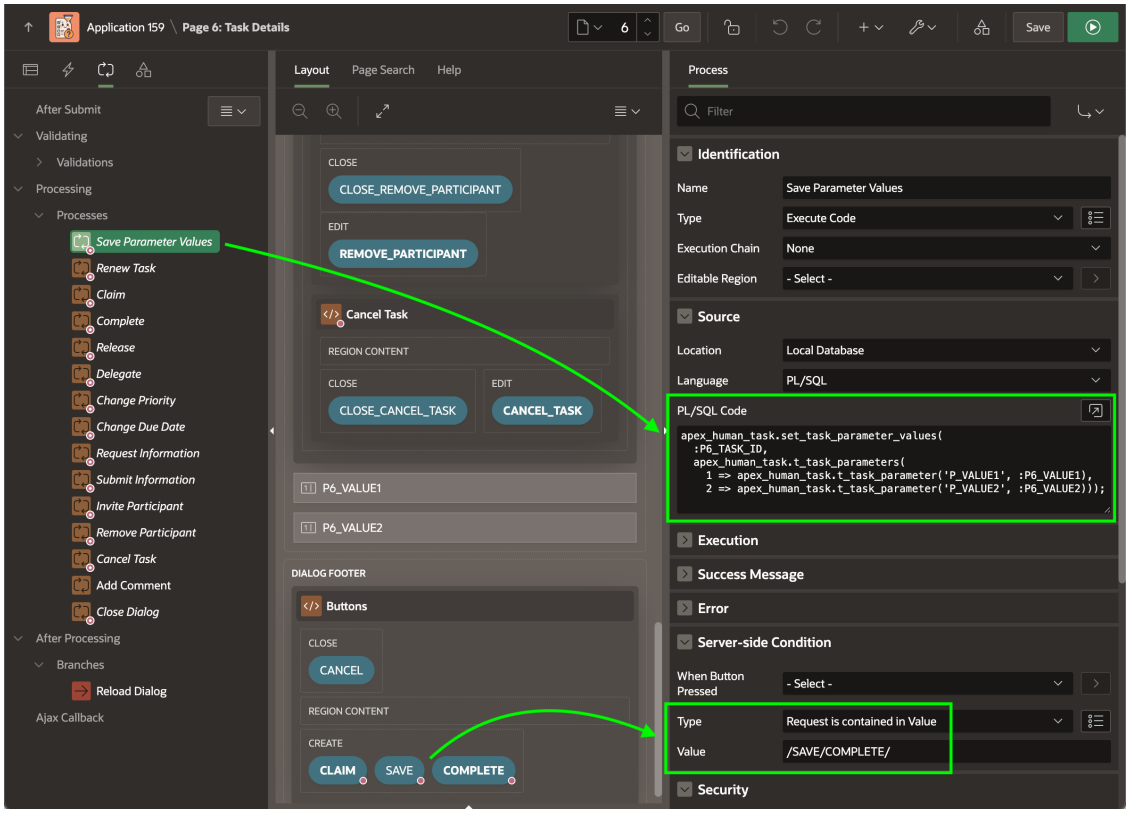
Save validated page item values back to updatable task parameters.

Copyright © 2026, Oracle and/or its affiliates.

If the user's page item values pass validation, you need to save them back to the updatable task parameters. As shown below, you can do this with an *Execute Code* page process configured to fire only when the user clicks the `SAVE` or `COMPLETE` buttons to submit the page. The code to save the parameters looks like this:

```
apex_human_task.set_task_parameter_values(
  :P6_TASK_ID,
  apex_human_task.t_task_parameters(
    1 => apex_human_task.t_task_parameter('P_VALUE1', :P6_VALUE1),
    2 => apex_human_task.t_task_parameter('P_VALUE2', :P6_VALUE2)));
```

Figure 25-53 Saving Task Parameter Values



25.1.9.1.8.9.6 Working with a Business Entity Row

Show or update the related business row from a task-specific details page.

If your task is tied to a business entity row and you use a *Details Primary Key*, you may need to view data from that row in your Task Details page to provide additional context. You also might want the actual owner to store additional data right away into columns in that row. Add a *Form* region to your Task Details page to display and edit the appropriate columns from this related data row. If you do, adopt one of the following approaches.

Suppose that your form region's primary key page item is named `P6_ID`. For the form to load its existing column data, populate the value of `P6_ID` with the *Details Primary Key* in the *Pre-Rendering* section of your page ***before*** the *Form - Initialization* page process executes. Do that in one of two ways:

- Use the hidden `P6_TASK_ID` to query the `DETAIL_PK` from the `APEX_TASKS` view, or

Copyright © 2026, Oracle and/or its affiliates.

- Edit the task definition's *Task Details Page URL* to pass `P6_ID` as `&DETAIL_PK.`

To adopt the first approach, add a *Pre-Rendering* computation for `P6_ID` with the query:

```
select detail_pk
  from apex_tasks
 where task_id = :P6_TASK_ID
```

With the second approach, the `P6_TASK_ID` value is already set by the Task Details Page URL.

In either case, the *Form - Initialization* page process in the *Pre-Rendering* section of your page loads all of the *Form* region's columns using that primary key value. Enable the *Query Only* switch on any columns that should be read-only in the page. If you let the assignee edit any of the columns to acquire additional data, then ensure you have a *Form - Automatic DML* page process related to the form region in your page's *Processing* tab. It should have a server-side condition to trigger on for `SAVE` and `COMPLETE` buttons. It automatically saves the changes back to the business entity row when the user submits the page. For an *approval* task, the buttons are named `APPROVE` and `REJECT` instead of `COMPLETE`.

> **Tip**
>
> If updating the business entity row *immediately* is not appropriate, then use updatable task parameters instead to capture the pending additional data. Then define a task action on the *Complete* event to update business entity row columns with task parameter values at completion time.

25.1.9.1.9 Letting an Initiator Approve a Task

Enable initiators to approve their own tasks when the business process requires it.

By default, a user who initiates an approval task isn't allowed to approve their own request. Even if their username is among the participants as a potential owner, the APEX engine filters out the initiator's username to enforce this common business practice. However, if you need to allow a requester to approve their own task to meet a business requirement, then enable the approval task definition's *Initiator Can Complete* switch to allow it.

25.1.9.1.10 Handling Time-Sensitive Tasks

Configure due dates and expiration behavior for time-sensitive tasks.

By default, a task is open-ended. Potential owners get to it when they can. When a task is time-sensitive, you an assign it a due date. By choosing an appropriate *Due On Type*, you set the due date using:

- A time interval relative to the creation date – e.g. 2 days from now
- An exact date and time computed using an expression, function body, or SQL Query, or
- The first occurrence of a `DBMS_SCHEDULER` expression – e.g. "weekly on Tuesdays at 9am"

When tasks have a due date, they display additional information in the *Unified Task List* "inbox" page and users can sort by the due date to work on the most time-critical tasks first. At any time, a task's business administrator can change the due date if necessary.

Once a task goes beyond its due date, end users can see that it's overdue in their task list, but by default nothing else automatic happens. To *automatically* handle an overdue task, configure the *Expiration Behavior* to one of the following optional strategies:

Copyright © 2026, Oracle and/or its affiliates.

- *Expire* – Close the task with the status of *Expired*, or
- *Renew* – Close the task as *Expired*, then create a new "renewal" task if it has not yet exceeded your maximum renewal count.

When using one of these expiration behaviors, you can notify assignees about the impending deadline using a task action on the *Before Expire* event. You can declaratively send a reminder email or push notification, or execute custom code. When it finally expires, you can react to that with an action on the *Expired* event.

If you opt for the creation of a *renewal task*, know that it is a *new* task instance. It is *linked* to the previous now-expired task. When viewing a renewal task in a task details page, the end-user can optionally view the history of the complete "task renewal chain" if needed. Your dynamic participant assignment logic can reference the value of `:APEX$TASK_RENEWAL_COUNT` and `:APEX$MAX_RENEWAL_COUNT` to decide whether to assign the renewal task in a different way.

- Letting an Initiator Approve a Task
  Enable initiators to approve their own tasks when the business process requires it.

25.1.9.1.10.1 Letting an Initiator Approve a Task

Enable initiators to approve their own tasks when the business process requires it.

By default, a user who initiates an approval task isn't allowed to approve their own request. Even if their username is among the participants as a potential owner, the APEX engine filters out the initiator's username to enforce this common business practice. However, if you need to allow a requester to approve their own task to meet a business requirement, then enable the approval task definition's *Initiator Can Complete* switch to allow it.

25.1.9.1.11 Reacting to Task Change Events

Use task actions to react to lifecycle events while a workflow waits.

Each *Human Task - Create* activity in a workflow creates a task instance, then *waits* for that task instance to complete its lifecycle before continuing to the next activity in the flow. Define task *actions* to react to changes in the task instance's state *during* a task's lifecycle.

- Choosing Event for an Action
  Choose the task event that should trigger an email, push notification, or code action.
- Accessing Task Information
  Reference task source columns and built-in substitutions from task actions.
- Notifying Assignees on Create
  Email potential owners when a new task requires their attention.
- Updating Columns on Completion
  A common requirement is to react to the outcome of an approval task by updating one or more columns in associated business entity row. You do this with an action on the *Complete* event.

25.1.9.1.11.1 Choosing Event for an Action

Choose the task event that should trigger an email, push notification, or code action.

When creating an action, choose an event from the table below that determines when your action triggers. Then, decide whether you want to send an email, send a push notification, or execute code. If the action should trigger only in certain conditions, configure an appropriate server-side condition.

Copyright © 2026, Oracle and/or its affiliates.

When using the *Execute Code* option, you *can* write in-line PL/SQL as part of the action definition. However, for easiest maintenance it's best to organize your code into a PL/SQL package procedure or function, then invoke that packaged program unit from the action's PL/SQL block.

> **Tip**
>
> Call `ADD_TO_HISTORY` in the `APEX_HUMAN_TASK` package from your *Execute Code* action to add a custom message to the task history.

Table 25-3 Pre-Defined Task Events

Event	Happens When
Create	Initiator creates task
Claim	Potential owner claims task
Delegate	Business administrator delegates task
Update Comment	A user adds a new comment to task
Update Priority	Business administrator changes task priority
Update Parameter	Actual owner sets the value of updatable task parameter
Release	Actual owner releases task so another potential owner can claim it
Cancel	Business administrator cancels task
Request Information	Actual owner request more information, typically from task initiator
Submit Information	User submits requested additional information
Before Expire	Less than *Before Expire Interval* left before task expires
Expire	Task expired
Complete	Actual owner completes task

25.1.9.1.11.2 Accessing Task Information

Reference task source columns and built-in substitutions from task actions.

When configuring an action, you can reference any of the *Actions Source* columns by name using either `:NAME` in PL/SQL code or `&NAME.` in email or push notification substitutions. You can also reference any of the built-in substitution values below:

Table 25-4 Pre-Defined Task Substitutions

Name	Purpose
`APEX$TASK_ID`	Unique ID of task instance
`APEX$TASK_PK`	*Details Primary Key* value of related business entity row
`APEX$TASK_STATE`	Mixed-case
`APEX$TASK_OUTCOME`	In *Complete* action, outcome of `APPROVED`, `REJECTED`
`APEX$TASK_CREATED_ON`	Date task instance was created
`APEX$TASK_SUBJECT`	Task subject
`APEX$TASK_DUE_ON`	Date task is due (or null if no due date)
`APEX$TASK_INITIATOR`	Case-sensitive username of task initiator
`APEX$TASK_OWNER`	Case-sensitive username of task's actual owner
`APEX$TASK_RENEWAL_COUNT`	Number of times task has been renewed

Copyright © 2026, Oracle and/or its affiliates.

Table 25-4 (Cont.) Pre-Defined Task Substitutions

Name	Purpose
APEX$TASK_MAX_RENEWAL_COUNT	Maximum task renewal count
APEX$TASK_PREVIOUS_ID	Unique ID of previous task in renewal "chain"
APEX$TASK_TEXT	In *Update Comment* action, text of comment added.

In addition, an *Update Parameter* action can use the HAS_TASK_PARAM_CHANGED, GET_TASK_PARAMETER_VALUE, and GET_TASK_PARAMETER_OLD_VALUE functions in the APEX_HUMAN_TASK package to get more information on which parameters have changed, what any parameter's current value is, and what a parameter's previous value was.

25.1.9.1.11.3 Notifying Assignees on Create

Email potential owners when a new task requires their attention.

Some users check their task "inbox" frequently, while others may appreciate getting notified about a new task that requires their attention. You can configure a task action to email the potential owners of a newly created task.

- Determining Assignees Generically
  Use a package function to determine potential owners for a task at runtime.
- Computing Subject and Action Values
  Compute task subject text and action values in the Actions Source query.
- Formatting Email with Templates
  Format task notification emails with templates and placeholder values.
- Sending Email on Task Creation
  Send an email on task creation using Actions Source values and template placeholders.
- Studying Generic Assignee Code
  Study package functions that return task owners and their email addresses.

25.1.9.1.11.3.1 Determining Assignees Generically

Use a package function to determine potential owners for a task at runtime.

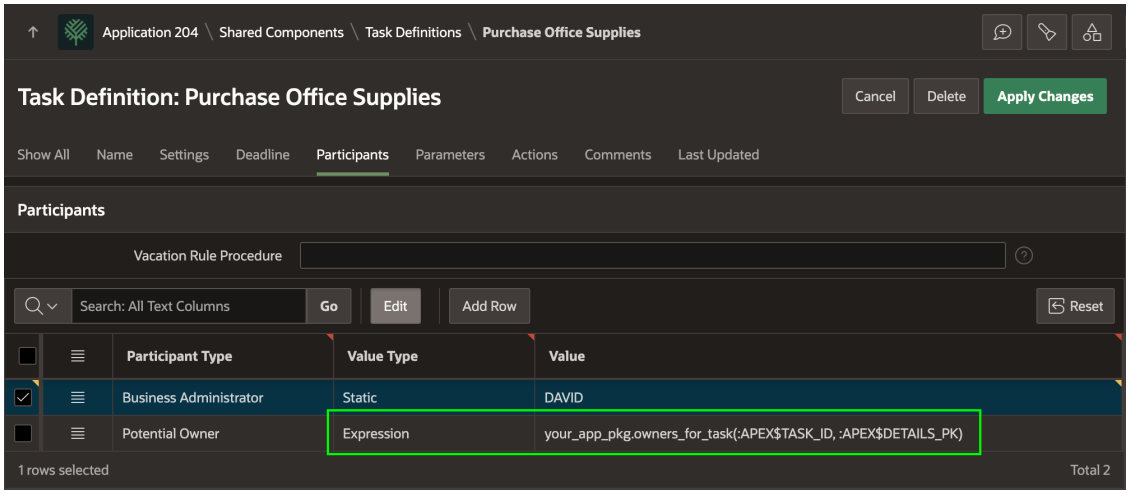
The *Purchase Office Supplies* task definition in this example configures a static username DAVID as the business administrator, and uses an *Expression* to call an OWNERS_FOR_TASK function in the package YOUR_APP_PKG. This dynamically determines the list of potential owners. It returns a comma-separated list of case-sensitive usernames. The function accepts the task ID and primary key of the business entity related to the task instance as parameters. The *Purchase Office Supplies* task definition does *not* use an associated business entity row, nor does it accept a *Details Primary Key* at creation time. However, this function signature illustrates a best practice for providing key information that generic task owner assignment logic typically needs.

The figure shows the Purchase Office Supplies task definition in the Task Definition edit page in **App Builder** and highlights the PL/SQL expression in use for its *Potential Owner* participants:

```
your_app_pkg.owners_for_task(:APEX$TASK_ID,:APEX$DETAILS_PK)
```


Copyright © 2026, Oracle and/or its affiliates.

Figure 25-54 Participants for Purchase Office Supplies Task Definition



The `your_app_pkg` specification looks like this:

```
package your_app_pkg as
    -------------------------------------------------
    -- Returns comma-separated list of case-sensitive
    -- usernames who are potential owners for task id
    -- passed in
    -------------------------------------------------
    function owners_for_task(
        p_task_id   in number,
        p_detail_pk in varchar2)
        return         varchar2;
    -------------------------------------------------
    -- Returns comma-separated list of email
    -- addresses of potential owners for task id
    -- passed in
    -------------------------------------------------
    function owner_emails_for_task(
        p_task_id   in number,
        p_detail_pk in varchar2)
        return         varchar2;
end your_app_pkg;
```

25.1.9.1.11.3.2 Computing Subject and Action Values

Compute task subject text and action values in the Actions Source query.

The *Purchase Office Supplies* task definition in this example accepts two parameters. As shown below, one *required* parameter, `P_ITEMS_TO_BUY`, lets the requester list the items to purchase. A second optional parameter, `P_STORE`, lets them indicate a preferred store to purchase the items from. If the initiator provides both items and a preferred store, then the *Subject* should be:

```
Please Buy Coffee and Bagels at Zabar's
```

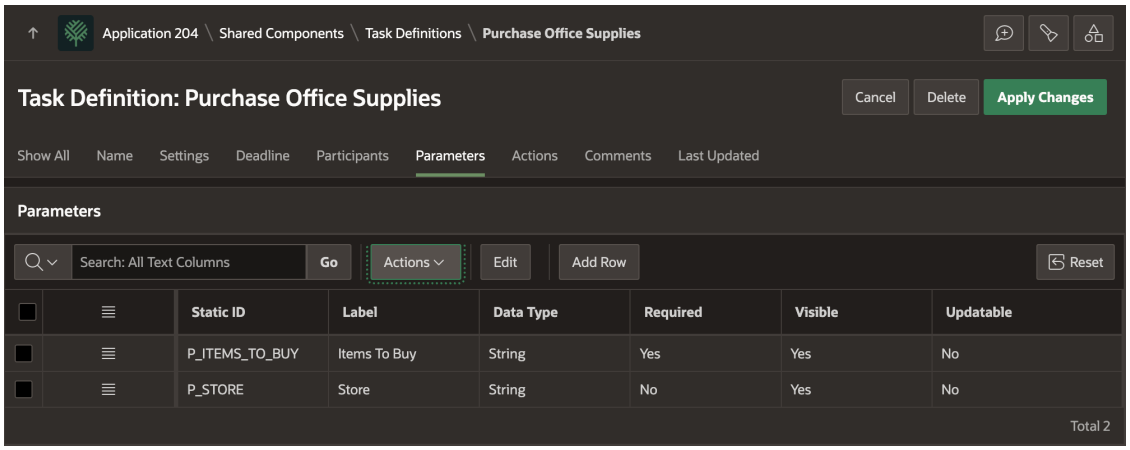

Copyright © 2026, Oracle and/or its affiliates.

In contrast, if the user only supplies a value for the items to buy, the *Subject* should be:

```
Please Buy 2GB Credit Card Sized Hard Drive
```

The task definition *Subject* field does not allow HTML Directives for conditional substitution, but it is still straightforward to get a computed task subject. The figure shows the **Parameters** tab of the Task Definition edit page in **App Builder** showing one required parameter `P_ITEMS_TO_BUY` and one optional parameter `P_STORE`.

Figure 25-55 Parameters for Purchase Office Supplies Task Definition



Your *Actions Source Query* can retrieve one row containing any number of columns to provide information to reference in the task definition's *Subject* or actions. Notice below that the query computes a dynamic subject line using a `CASE` expression to include the optional store information in the title.

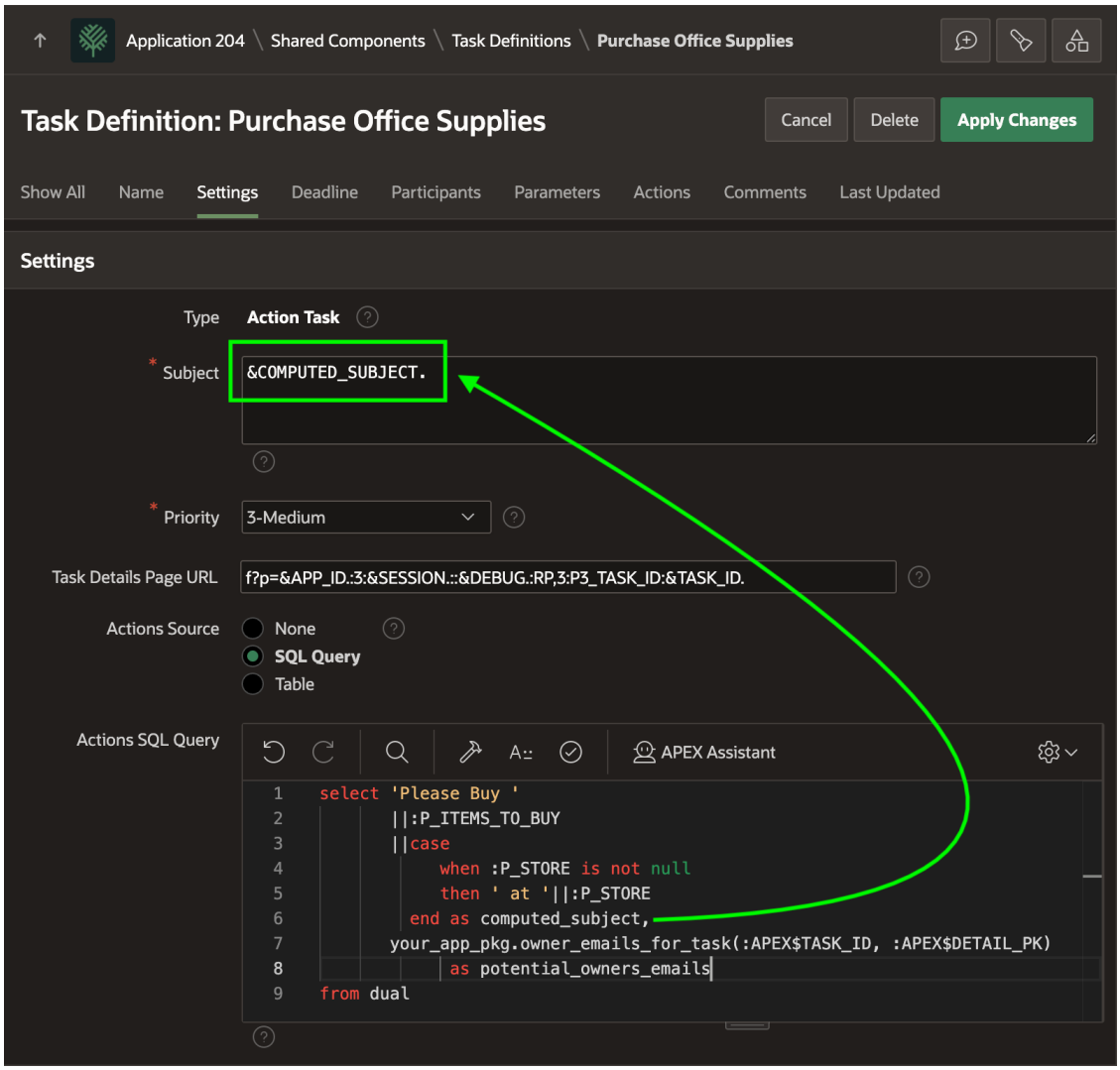
It also invokes the `OWNER_EMAILS_FOR_TASK` function to retrieve a comma-separated list of *emails* of the potential owners. You can then reference the `COMPUTED_SUBJECT` and `POTENTIAL_OWNER_EMAILS` column aliases wherever needed in the task definition. The query looks like this:

```
select 'Please Buy '
       ||:P_ITEMS_TO_BUY
       ||case
            when :P_STORE is not null
            then ' at '||:P_STORE
         end as computed_subject,
       your_app_pkg.owner_emails_for_task(:APEX$TASK_ID, :APEX$DETAIL_PK)
            as potential_owners_emails
from dual
```

As shown below, the task definition's *Subject* references the dynamically computed subject using the `&COMPUTED_SUBJECT.` notation. The figure shows the **Settings** tab of the Task Definition edit page in **App Builder** and highlights how the **Subject** can use `&COMPUTED_SUBJECT.` to reference a column from the **Actions SQL Query**.

Copyright © 2026, Oracle and/or its affiliates.

Figure 25-56 Actions Source Query Retrieves Computed Title and Potential Owner Emails



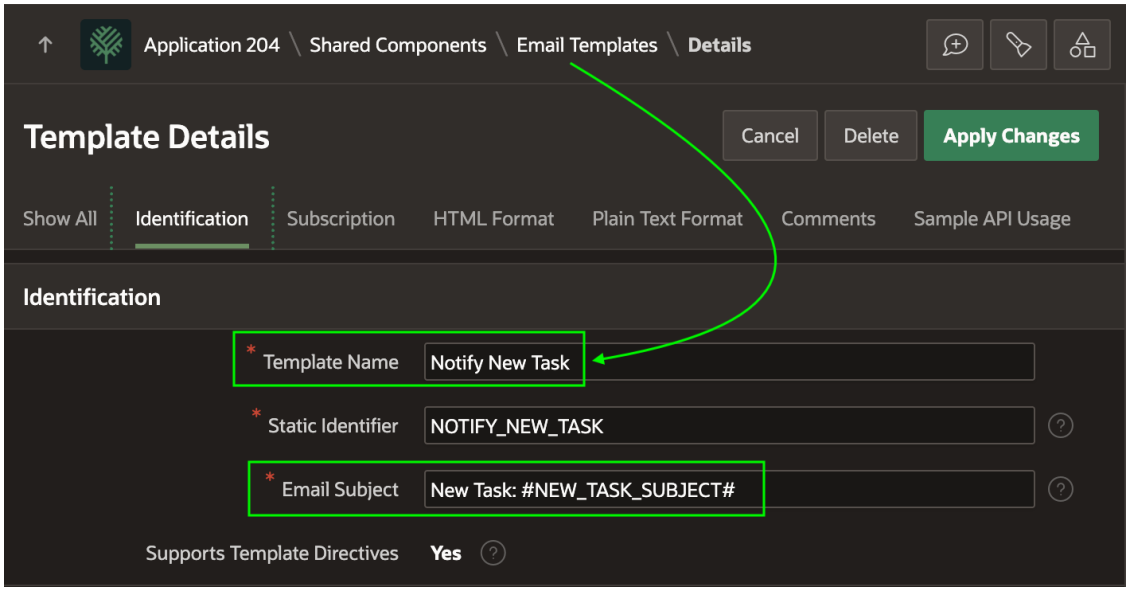
25.1.9.1.11.3.3 Formatting Email with Templates

Format task notification emails with templates and placeholder values.

When sending email, you can reference an *Email Template*. This shared component defines the email subject line along with a message header, body, and footer. These sections can all use `#NAME#` substitutions as placeholders for data you pass in anywhere you use the template to send a mail. Notice how the *Notify New Task* email template below references a placeholder name `#NEW_TASK_SUBJECT#` in the *Email Subject* field.

Copyright © 2026, Oracle and/or its affiliates.

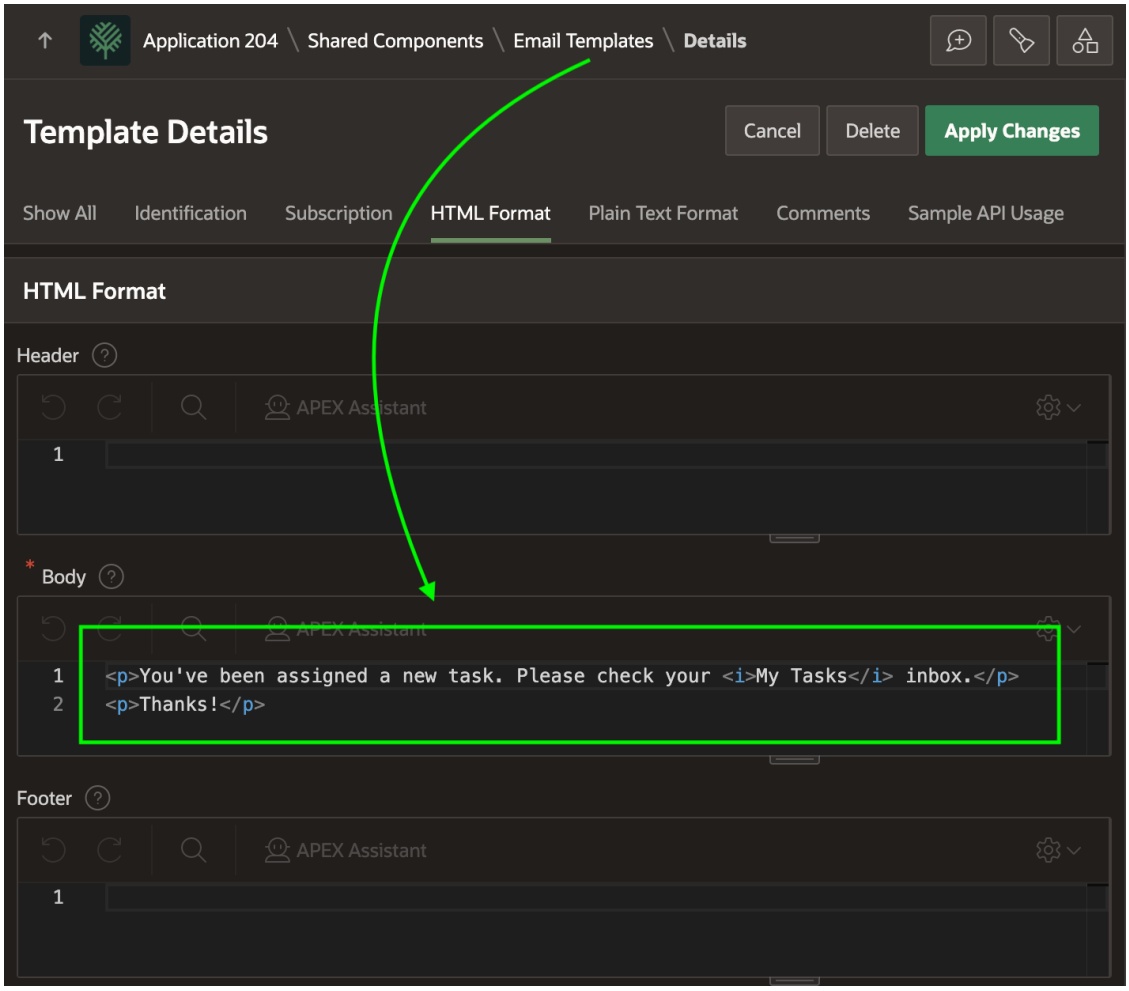
Figure 25-57 Email Template Used to Send New Task Notification



In the *HTML Format* section of the email template edit page, you can see this simple example only defines a static HTML body, but this section as well as *Header* and *Footer* can also mention additional placeholders if needed.

Copyright © 2026, Oracle and/or its affiliates.

Figure 25-58 Email Body HTML in the Template for New Task Notification



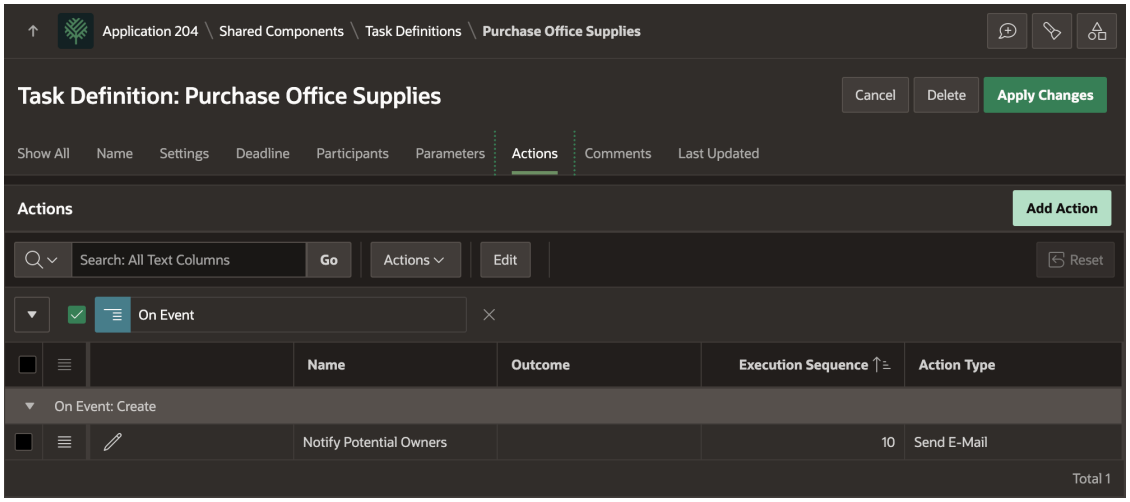
25.1.9.1.11.3.4 Sending Email on Task Creation

Send an email on task creation using Actions Source values and template placeholders.

In the *Actions* section, the *Purchase Office Supplies* action task definition configures a single action on the *Create* task lifecycle event. The action type is *Send E-Mail*.

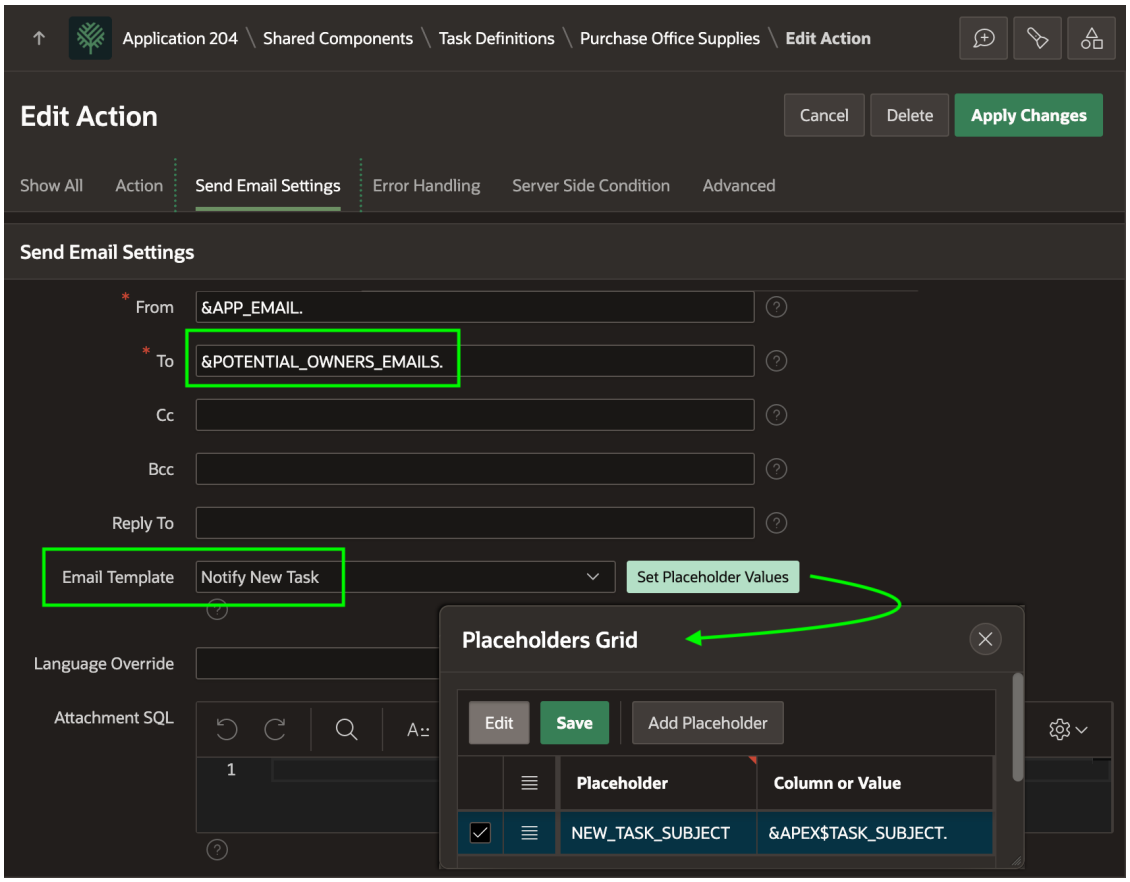
Copyright © 2026, Oracle and/or its affiliates.

Figure 25-59 Create Action on Purchase Office Supplies Task Definition



Notice below how the *Create* action's *To* field references the *Actions Source Query* column `POTENTIAL_OWNERS_EMAILS` in the *Send Email Settings* section. It also uses the *Notify New Task* email template. Clicking on the *(Set Placeholder Values)* button opens the *Placeholders Grid* dialog shown below. It lets you configure values for all placeholders in the email template. Here, the `NEW_TASK_SUBJECT` placeholder uses the task's subject with the `&APEX$TASK_SUBJECT.` syntax.

Figure 25-60 Send E-Mail Action on Task Create Event



Copyright © 2026, Oracle and/or its affiliates.

25.1.9.1.11.3.5 Studying Generic Assignee Code

Study package functions that return task owners and their email addresses.

The `your_app_pkg` contains two generic functions for returning the list of potential owners for a task, as well as a list of their emails. The package body code looks like the following. Notice the `OWNER_EMAILS_FOR_TASK` queries the `APEX_TASK_PARTICIPANTS` view using the supplied task id to find the potential owner participants, and joins that view with the `APEX_WORKSPACE_APEX_USERS` view to get the email addresses of the potential owners. It uses the `LISTAGG` function with a comma as the delimiter to return a single string of distinct email addresses.

```
package body your_app_pkg as
    ------------------------------------------------
    function owners_for_task(
        p_task_id   in number,
        p_detail_pk in varchar2)
        return         varchar2
    is
       l_ret varchar2(4000);
    begin
        -- Your custom logic here to return comma-separated
        -- list of case-sensitive user names.
        return l_ret;
    end owners_for_task;
    ------------------------------------------------
    function owner_emails_for_task(
        p_task_id               in number,
        p_detail_pk             in varchar2)
        return         varchar2
    is
        l_ret varchar2(4000);
    begin
         select listagg(distinct u.email,',')
           into l_ret
           from apex_task_participants p
           join apex_workspace_apex_users u
             on u.user_name = p.participant
          where p.task_id = p_task_id
            and p.participant_type = 'POTENTIAL_OWNER';
         return l_ret;
    end owner_emails_for_task;
end your_app_pkg;
```

25.1.9.1.11.4 Updating Columns on Completion

A common requirement is to react to the outcome of an approval task by updating one or more columns in associated business entity row. You do this with an action on the *Complete* event.

The columns you update depend on whether the business entity related to the task definition represents:

- A primary business entity, like an employee, or
- A request for approval, related data, and an approval status.
- Updating Entity Columns on Complete
  Update the related business entity row when an approval task completes.

Copyright © 2026, Oracle and/or its affiliates.

- Updating Request Columns on Complete
  Update request and business entity rows when an approval task completes.

25.1.9.1.11.4.1 Updating Entity Columns on Complete

Update the related business entity row when an approval task completes.

Assume a *Salary Change* task definition uses the primary business entity table `EMP` as its *Actions Source* table. The task definition defines parameters like:

- `P_PROPOSED_SALARY` – to store the suggested new employee monthly pay, and
- `P_MOTIVATION` – to let the requester summarize the reason for the change.

The task definition defines a subject like the following, referencing the value of the Action Source table `ENAME` column and the parameter name.

```
Review &ENAME. Salary Change to &P_PROPOSED_SALARY.
```

When a *Salary Change* task completes successfully, it has an outcome of either `APPROVED` or `REJECTED`. Your *Execute Code* action on the *Complete* event needs to update the `SAL` column to the `P_PROPOSED_SALARY` value for `EMP` table row related to the task instance. Your code looks like:

```
-- If approved, update sal using Action Table EMPNO column value
if :APEX$TASK_OUTCOME = 'APPROVED' then
   update emp
      set   sal = :P_PROPOSED_SALARY
    where empno = :EMPNO;
end if;
```

25.1.9.1.11.4.2 Updating Request Columns on Complete

Update request and business entity rows when an approval task completes.

Consider a *Salary Change* task definition that uses the following `EMP_SAL_CHANGE_REQUEST` table as its Actions Source table. A row in this table represents the *request* to change a particular employee's salary.

```
create table emp_sal_change_request (
    id            number
                  generated by default on null as identity
                  constraint emp_sal_change_request_id_pk primary key,
    empno         number
                  constraint emp_sal_change_request_empno_fk
                  references emp,
    new_salary    number,
    motivation    varchar2(255 char),
    status        varchar2(8 char)
);
```

For this task definition, the built-in *Details Primary Key* parameter suffices, since you can already reference all the columns in the `EMP_SAL_CHANGE_REQUEST` row it identifies. There is no need to define additional parameters.

Notice the `EMP_SAL_CHANGE_REQUEST` table includes an `EMPNO` foreign key column. So, to show the `ENAME` value in the task subject, you need to change from using the Actions Source *table* to

Copyright © 2026, Oracle and/or its affiliates.

using a *query* instead. This lets you join in the EMP table to include the related ENAME and existing SAL value using a query like this:

```
select r.id as request_id, r.empno, r.new_salary, e.ename, e.sal
  from emp_sal_change_request r
  join emp e
    on e.empno = r.empno
 where r.id = :APEX$TASK_PK
```

Notice above that if your task definition subject or actions don't *need* to reference the current value of particular columns like MOTIVATION and STATUS you can leave those out of the SELECT list.

Your task definition can now use a subject like the following, referencing the value of the Action Source query's ENAME, SAL, and NEW_SALARY columns.

```
Review &ENAME. Salary Change from &SAL. to &NEW_SALARY.
```

As before, when a *Salary Change* task completes successfully, it has an outcome of either APPROVED or REJECTED. Your action on the *Complete* event of type *Execute Code* needs to:

- Record the outcome of the salary change request in the STATUS column, then
- Update the SAL column to the NEW_SALARY value for EMP table row identified by the EMPNO value in the employee salary change request row related to the task instance.

Your code looks like:

```
-- Record the outcome in the salary change request row
update emp_sal_change_request
   set status = :APEX$TASK_OUTCOME
 where id = :REQUEST_ID;
-- If approved, update sal using Action Table query EMPNO column value
if :APEX$TASK_OUTCOME = 'APPROVED' then
   update emp
      set   sal = :NEW_SALARY
    where empno = :EMPNO;
end if;
```

25.1.9.2 Waiting for a Signal to Continue

Send users a page link, wait for their response, and continue the workflow from application logic.

Your workflow can notify a user with a *Send E-Mail* or *Send Push Notification* activity. Either can include a link in the message. By clicking or tapping the link, the recipient navigates to the page to provide their input. After notifying the user, the next step in the flow can be a *Wait* activity with no *Timeout Type* set. This causes the workflow instance to wait *indefinitely* until some code calls CONTINUE_ACTIVITY in the APEX_WORKFLOW package to tell the workflow to move on. When invoking this method to continue a workflow activity that's currently waiting, you need to pass in:

- The unique ID of the workflow instance
- The unique *Static ID* of the activity to continue (typically a *Wait* activity)

Copyright © 2026, Oracle and/or its affiliates.

When continuing a waiting activity, you can also optionally pass a list of name/value pairs. If any of the names in this list match the names of workflow variables, their values get assigned automatically to the corresponding value you return.

- Enabling Deep Linking to a Specific Page
  Enable Deep Linking so notification links can open a specific application page.
- Generating Page URL for Notification
  Generate an absolute page URL for email or push notification links.
- Emailing User a Page Link
  Email a user an absolute page link generated from workflow activity data.
- Delivering Push Notification with Page Link
  Send a push notification with an absolute page link generated from workflow activity data.
- Waiting Until Application Logic Continues
  Pause a workflow with a Wait activity until application logic continues it.
- Navigating to Page Using Email Link
  Open a notification link to enter requested data on a deep-linked page.
- Retrieving Title and Activity Labels
  Retrieve workflow and activity labels to show context on the data entry page.
- Signaling to Continue and Returning Data
  Continue a waiting workflow activity and return submitted values to workflow variables.
- Inspecting Resulting Workflow Instance
  Inspect workflow variables and diagram progress after a waiting activity continues.
- Evaluating Benefits of Using Tasks Instead
  Consider using tasks when user input needs tracking, reminders, and a task inbox.

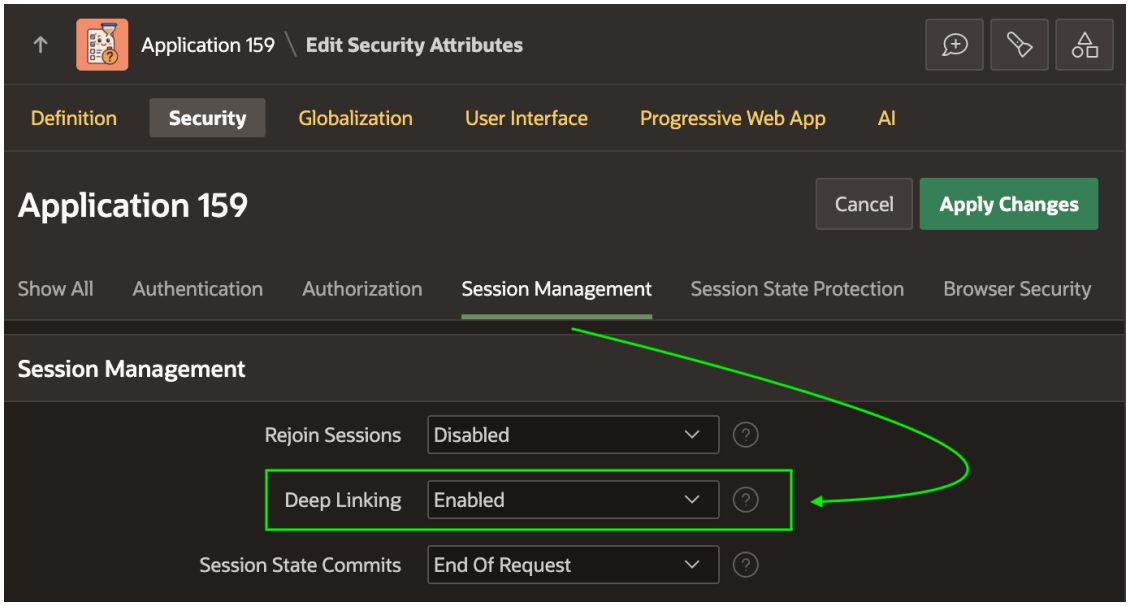
25.1.9.2.1 Enabling Deep Linking to a Specific Page

Enable Deep Linking so notification links can open a specific application page.

Normally, a link to your application navigates the user to the application home page after login. If you want the user to navigate to a *specific page* by clicking on a link, as shown below, you need to enable *Deep Linking* under *Shared Components* > *Security Attributes* for your application.

Copyright © 2026, Oracle and/or its affiliates.

Figure 25-61 Enabling Deep Linking so User's Click Can Directly Link to Page



25.1.9.2.2 Generating Page URL for Notification

Generate an absolute page URL for email or push notification links.

After enabling deep linking, use a function like the one below to generate a link to the page for the user to provide additional details. The `GET_URL` function in the `APEX_PAGE` package returns a *relative* URL by default. This means the *`https://example.com`* part of the URL is omitted. The browser interprets such a link relative to the page in which you embed it. Relative URLs are fine in an application page, but when generating a page link for an email or push notification, use an *absolute* URL instead. The user can click on this complete URL from any context.

Notice the `ABSOLUTE_PAGE_URL` function uses `v('APEX$WORKFLOW_ID')`. In a workflow activity context, this variable returns the workflow instance ID. The function assumes the target page number *n* has two hidden page items `Pn_WORKFLOW_ID` and `Pn_ACTIVITY_STATIC_ID`. Finally, note that it passes `true` to `p_absolute_url` and `0` to `p_session`.

```
function absolute_page_url(
    p_page                          in number,
    p_activity_static_id            in varchar2)
    return                             varchar2
is
    c_workflow_page_item constant varchar2(255)
        := 'P'||p_page||'_WORKFLOW_ID';
    c_activity_static_id_item constant varchar2(255)
        := 'P'||p_page||'_ACTIVITY_STATIC_ID';
begin
    return apex_page.get_url(
            p_page         => p_page,
            p_items        => c_workflow_page_item
                              ||','
                              ||c_activity_static_id_item,
            p_values       => v('APEX$WORKFLOW_ID')
                              ||','
```


Copyright © 2026, Oracle and/or its affiliates.

```
                                ||p_activity_static_id,
            p_absolute_url => true,
            p_session      => 0,
            p_debug        => 'NO');
end absolute_page_url;
```

25.1.9.2.3 Emailing User a Page Link

Email a user an absolute page link generated from workflow activity data.

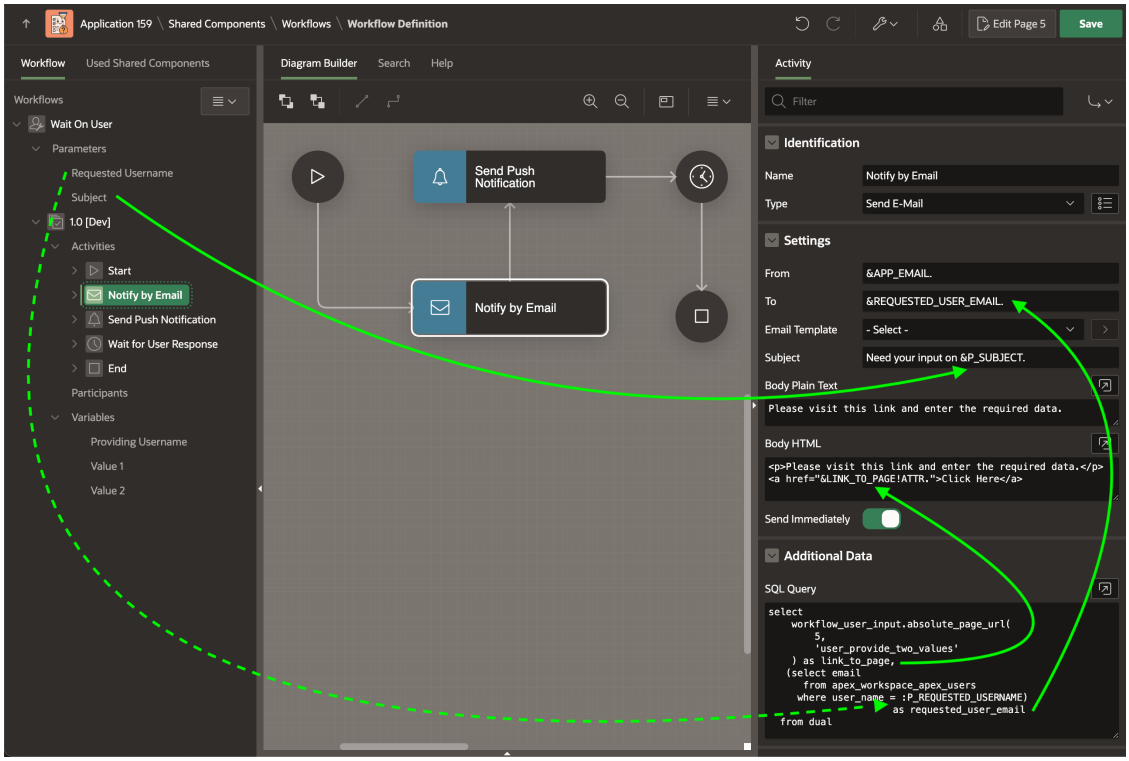
To notify a user via email with a link to a data entry page, use a *Send E-Mail* activity. Since a workflow runs asynchronously in the background, the `APP_USER` system variable returns `nobody`. Therefore, this example workflow accepts a `P_REQUESTED_USERNAME` parameter, along with a `P_SUBJECT` parameter. The *Additional Data* query below on the *Send E-Mail* activity calls an `ABSOLUTE_PAGE_URL` function to generate the link to page 5, also passing the unique static ID of the wait activity `user_provide_two_values`. The query aliases the result to the `LINK_TO_PAGE` column name. A second `SELECT` list expression uses a subquery to lookup the email address of the requested username from the `APEX_WORKSPACE_APEX_USERS` view, using the value of the `P_REQUESTED_USERNAME` parameter as a bind variable. It aliases that result to `REQUESTED_USER_EMAIL`.

```
select
    workflow_user_input.absolute_page_url(
        5,
        'user_provide_two_values'
    ) as link_to_page,
   (select email
      from apex_workspace_apex_users
     where user_name = :P_REQUESTED_USERNAME) as requested_user_email
  from dual
```

As shown below, the *Send E-Mail* activity's *Subject* uses the `&P_SUBJECT.` substitution, the *To* field references the `&REQUESTED_USER_EMAIL.` from the *Additional Data* query result, and the *Body HTML* references the `&LINK_TO_PAGE!ATTR.` in an `href` attribute of an HTML anchor (`<a>`) element. The `!ATTR` modifier ensures the URL value is correctly formatted for inclusion in an HTML attribute.

Copyright © 2026, Oracle and/or its affiliates.

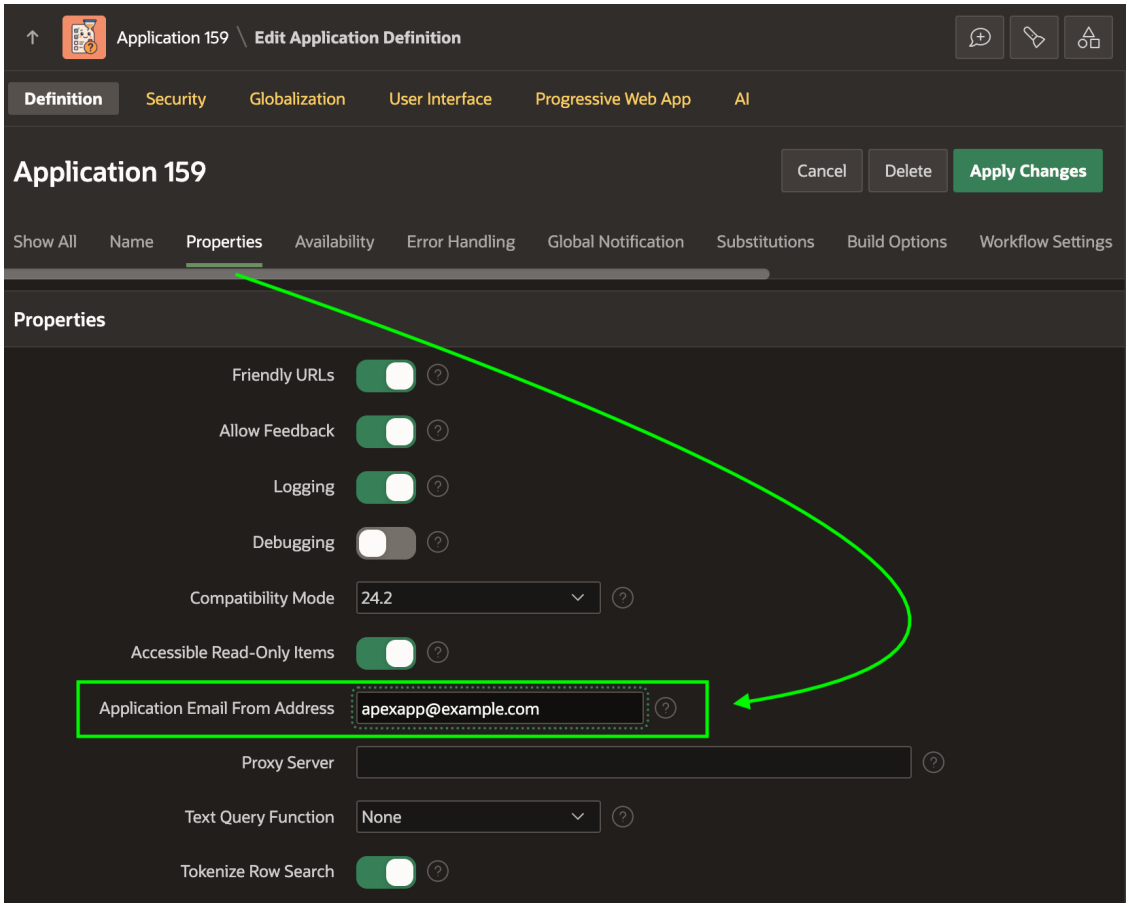
Figure 25-62 Send E-Mail Activity Includes Page Link from Additional Data Query



Also notice the *From* field defaults to the `&APP_EMAIL.` substitution. This is a system variable whose value you configure in the *Shared Components* > *Application Definition* page in the *Properties* section as shown below.

Copyright © 2026, Oracle and/or its affiliates.

Figure 25-63 Configuring the Value of APP_EMAIL Substitution



25.1.9.2.4 Delivering Push Notification with Page Link

Send a push notification with an absolute page link generated from workflow activity data.

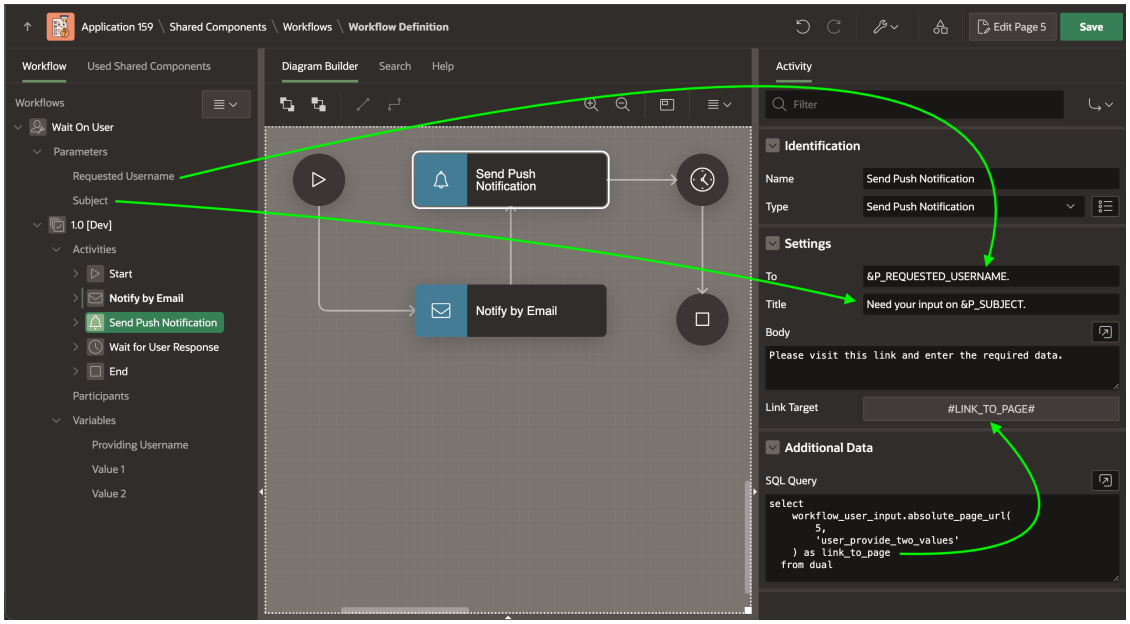
To prompt a user via push with a link to a data entry page, use a *Send Push Notification* activity. Since a workflow runs asynchronously in the background, the `APP_USER` system variable returns `nobody`. Therefore, this example workflow accepts a `P_REQUESTED_USERNAME` parameter, along with a `P_SUBJECT` parameter. The *Additional Data* query below on the *Send Push Notification* activity calls an `ABSOLUTE_PAGE_URL` function to generate the link to page 5, also passing the unique static ID of the wait activity `user_provide_two_values`. The query aliases the result to the `LINK_TO_PAGE` column name.

```
select
    workflow_user_input.absolute_page_url(
        5,
        'user_provide_two_values'
    ) as link_to_page
  from dual
```

As shown below, the *Send Push Notification* activity's *Title* uses the `&P_SUBJECT.` substitution, the *To* field references the `&P_REQUESTED_USERNAME.` parameter, and the *Link Target* references `#LINK_TO_PAGE#` from the *Additional Data* query result.

Copyright © 2026, Oracle and/or its affiliates.

Figure 25-64 Send Push Notification Uses Link from Additional Data Query

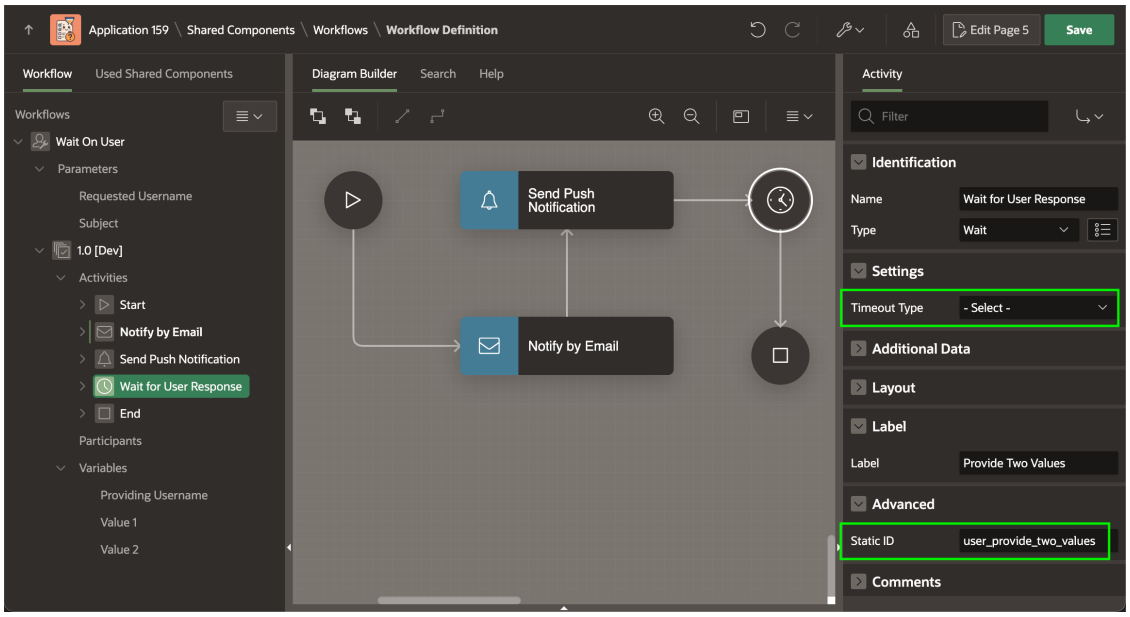


25.1.9.2.5 Waiting Until Application Logic Continues

Pause a workflow with a Wait activity until application logic continues it.

To pause a workflow until your app tells it to continue, use a *Wait* activity with no *Timeout Type* set. The *Static ID* you set on the activity is the value you pass to `CONTINUE_ACTIVITY` in the `APEX_WORKFLOW` package. Notice below that the *Wait* activity's *Static ID* is `user_provide_two_values` and that it leaves *Timeout Type* with no selection.

Figure 25-65 Wait Activity `user_provide_two_values` Has No Timeout Type



Copyright © 2026, Oracle and/or its affiliates.

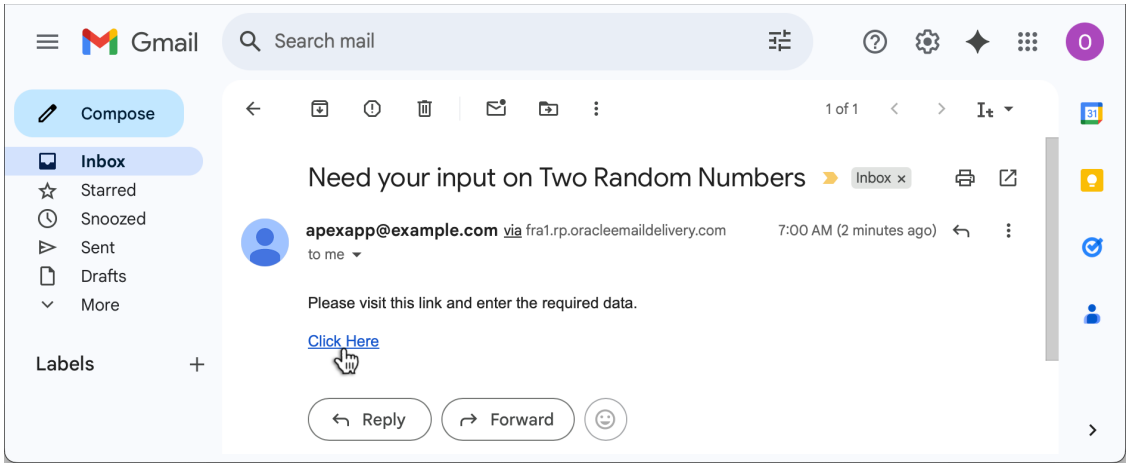
> **Tip**
>
> A native *Wait* activity with no *Timeout Type* is the most common subject of a `CONTINUE_ACTIVITY` call. However, a *custom* workflow activity you create can also wait for completion in the same way. In your process type plug-in with its *Workflow Activity* checkbox checked in the *Supported For* section, check the *Wait for Completion* checkbox in the *Standard Attributes* section. It can also provide a *Completion Function* the APEX engine executes when the custom activity continues.

25.1.9.2.6 Navigating to Page Using Email Link

Open a notification link to enter requested data on a deep-linked page.

When a user receives a notification containing an absolute URL page link, as shown below, they can click the link to navigate to the page.

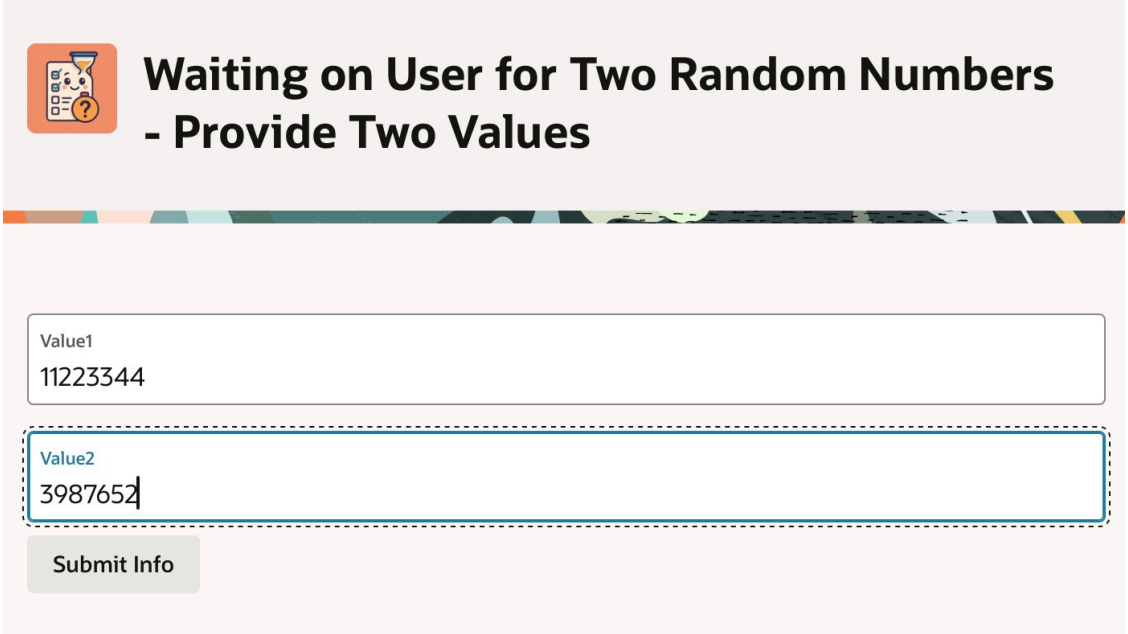
Figure 25-66 Receiving the Notification Email and Clicking the Link



After logging in, since you enabled *Deep Linking*, they navigate to the page where they can provide the additional data and click *(Submit Info)* button to confirm their input. Notice the page uses a *Static Content* region with the *Hero* template to show an application icon and title. The title includes the workflow instance title and the label of the *Wait* workflow activity.

Copyright © 2026, Oracle and/or its affiliates.

Figure 25-67 Page Collecting Two Values from the Notified User



25.1.9.2.7 Retrieving Title and Activity Labels

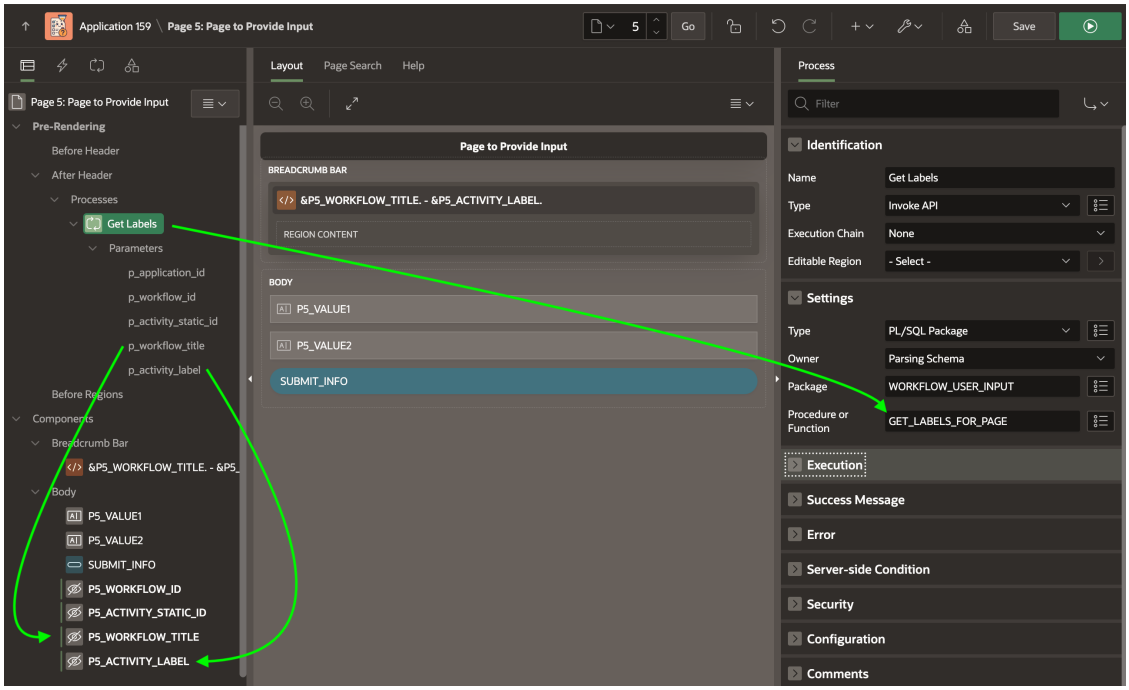
Retrieve workflow and activity labels to show context on the data entry page.

As shown below, the page to collect user input retrieves workflow title and activity label dynamically. It uses an *Invoke API* page process in the *Pre-Rendering* section to call a `GET_LABELS_FOR_PAGE` procedure. This function accepts the workflow instance ID, the activity's static ID from hidden page items, and returns the workflow title and activity label into two *other* hidden page items. Then the *Static Content* region's title references the values of these hidden page items using:

```
&P5_WORKFLOW_TITLE. - &P5_ACTIVITY_LABEL.
```


Copyright © 2026, Oracle and/or its affiliates.

Figure 25-68 Page Collecting Values from the User



The `GET_LABELS_FOR_PAGE` procedure looks like this. It retrieves the title and activity label by joining the `APEX_WORKFLOWS` and `APEX_WORKFLOW_ACTIVITIES` views.

```
procedure get_labels_for_page(
    p_application_id     in  number default null,
    p_workflow_id        in  number,
    p_activity_static_id in  varchar2,
    p_workflow_title     out varchar2,
    p_activity_label     out varchar2)
is
    l_application_id apex_applications.application_id%type;
begin
    l_application_id := coalesce(p_application_id,v('APP_ID'));
    select a.label as activity_label,
           w.title as workflow_title
     into p_activity_label,
          p_workflow_title
     from apex_workflow_activities a
     join apex_workflows w
       on w.workflow_id = a.workflow_id
    where a.application_id = l_application_id
      and w.workflow_id    =  p_workflow_id
      and a.type_code      = 'NATIVE_WORKFLOW_WAIT'
      and a.state          = 'WAITING'
      and a.static_id      = p_activity_static_id;
exception
    when no_data_found then
        null;
end get_labels_for_page;
```


Copyright © 2026, Oracle and/or its affiliates.

25.1.9.2.8 Signaling to Continue and Returning Data

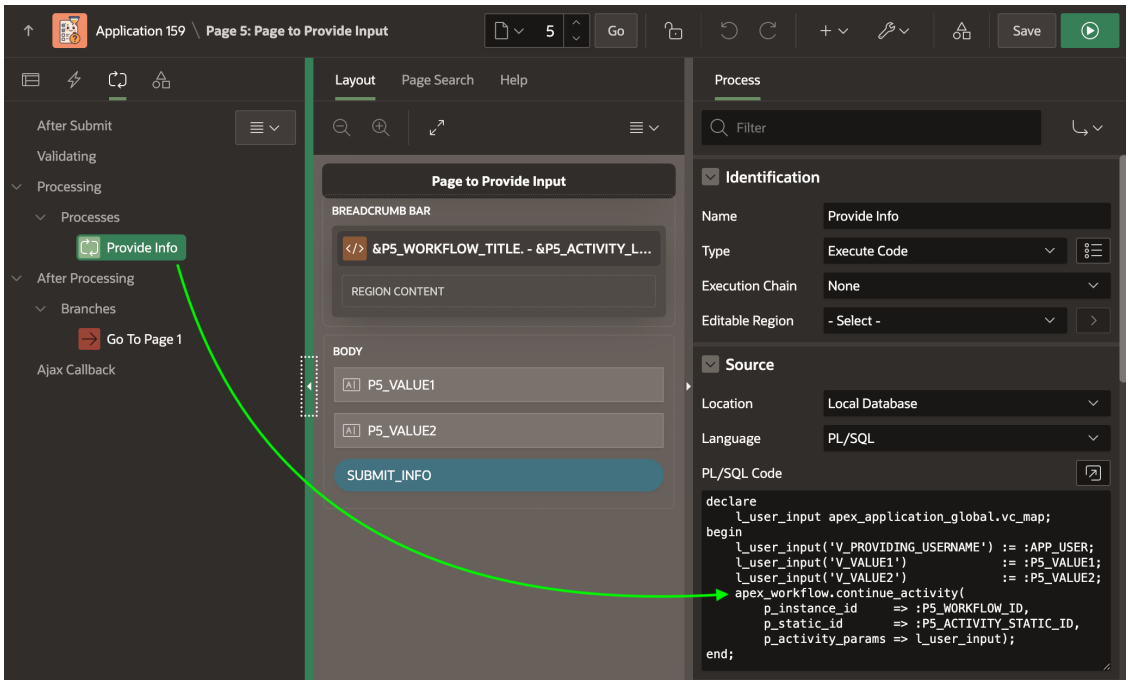
Continue a waiting workflow activity and return submitted values to workflow variables.

The page below contains hidden page items `P5_WORKFLOW_ID` and `P5_ACTIVITY_STATIC_ID`. When the user clicks the *(Submit Information)* button, a *Provide Info* page process in the *Processing* section executes a small block of PL/SQL code.

It first populates a parameter map containing the user's submitted values. Then it calls `CONTINUE_ACTIVITY`, passing the workflow ID, activity static ID, and the parameter map. The map is a set of key/value pairs. Notice that the keys are the *Static ID* values of three workflow variables. Any workflow variable whose *Static ID* matches a key in the parameter map is assigned its corresponding value when the activity continues.

```
declare
    l_user_input apex_application_global.vc_map;
begin
    l_user_input('V_PROVIDING_USERNAME') := :APP_USER;
    l_user_input('V_VALUE1')             := :P5_VALUE1;
    l_user_input('V_VALUE2')             := :P5_VALUE2;
    apex_workflow.continue_activity(
        p_instance_id     => :P5_WORKFLOW_ID,
        p_static_id       => :P5_ACTIVITY_STATIC_ID,
        p_activity_params => l_user_input);
end;
```

Figure 25-69 Page Process Calls CONTINUE_ACTIVITY, Passing Data in Parameter Map



Copyright © 2026, Oracle and/or its affiliates.

> **Tip**
>
> Passing workflow variable values back through the `CONTINUE_ACTIVITY` procedure is only one way to capture end user input in a workflow. Depending on your use case, your page could instead populate the data into columns of an appropriate application table row directly. Using the workflow ID, you can lookup the primary key value of the associated business entity row by retrieving the `DETAIL_PK` value from the `APEX_WORKFLOWS` table where the `WORKFLOW_ID` equals the value you know.

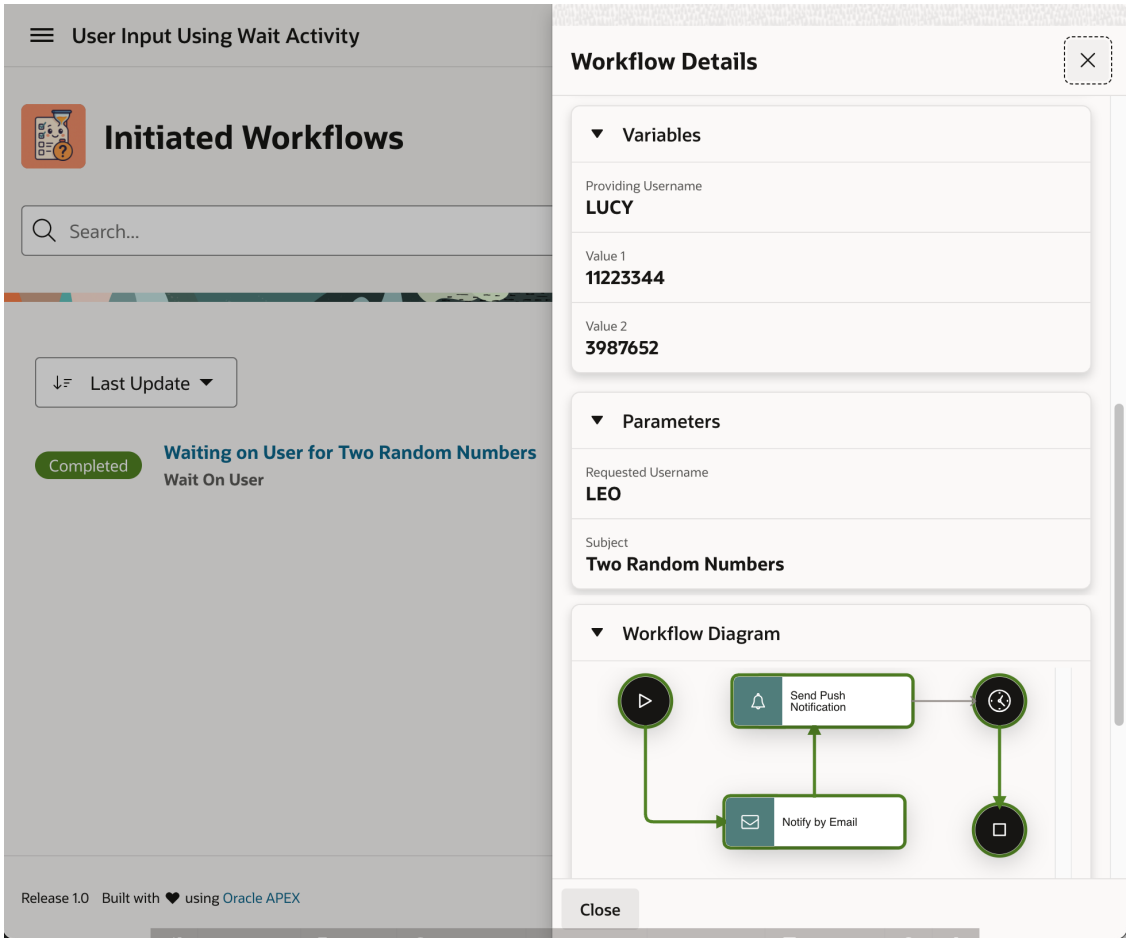
25.1.9.2.9 Inspecting Resulting Workflow Instance

Inspect workflow variables and diagram progress after a waiting activity continues.

If a page calls `CONTINUE_ACTIVITY` in the `APEX_WORKFLOW` package to signal a *Wait* activity to proceed, any workflow variable whose *Static ID* matches a key in the parameter map is assigned its corresponding value. The default workflow detail page you generate when creating a *Workflow Console* in the Create Page Wizard displays the workflow variables and parameter values. It also displays a *Workflow Diagram* showing the workflow path completed so far. As shown below, while the requested username parameter was `LEO`, he must have forwarded the email to his colleague `LUCY` to click on the link and provide the data. Notice in the *Variables* region, the values she provided are reflected, along with her username. The page displays the user-friendly labels and values of the `V_PROVIDING_USERNAME`, `V_VALUE1`, and `V_VALUE2` variables. The *Completed* badge in the *Initiated Workflows* Content Row region confirms this workflow instance is done executing. The visual diagram's highlighted activities also confirm this.

Copyright © 2026, Oracle and/or its affiliates.

Figure 25-70 Inspecting the Details of the Completed Workflow



25.1.9.2.10 Evaluating Benefits of Using Tasks Instead

Consider using tasks when user input needs tracking, reminders, and a task inbox.

Prompting an end user with a link to a data entry page using email or push notification is straightforward. However, if the user dismisses the notification or the message gets lost among other emails, your business process may stall. A workflow could remain stuck waiting for input that never arrives.

To avoid the pitfalls of this simple approach, a more complete solution would:

- Track which data entry requests the application sent,
- Provide users with a list of outstanding requests requiring their input, and
- Remind them if they haven't responded after a reasonable amount of time.

A workflow using a *Task Definition* to formally track a user work item gets these features natively:

- Updatable task parameters can model user input required,
- Task instances track pending work items
- Task actions can notify users when they've been assigned something new
- A *Unified Task List* page is a task "inbox" where users see and complete assigned tasks

Copyright © 2026, Oracle and/or its affiliates.

- Due dates and *Before Expire* actions can remind users about time-sensitive items.

Before building a custom solution to add these features, consider whether using a *Human Task - Create* activity combined with a well-designed task definition can save you significant time and effort.

25.1.10 Identifying Workflow Participants

Assign workflow owners and administrators using case-sensitive usernames.

A workflow participant is either a workflow owner or workflow administrator. A *workflow owner* is a user with overall responsibility for the business process, while a *workflow administrator* helps address a problem relating to the flow. For example, if user `DAVID` is the owner of the *Patient Onboarding* workflow, he's the one who can see it in his workflow console when the context is set to *My Workflows*. He can also retry or terminate the flow. If user `CARLA` is the workflow's business administrator, she'll see it in her *Admin Workflows* console list. She can suspend it, change its variable values if needed to fix a problem, and resume it again.

> ⚠ **Caution**
>
> Participant usernames are case-sensitive. User `DAVID` can login with the username `david`. However, if you assign user `david` to a workflow, David does not see it in his *My Workflows* console list because `david` does not match what the APEX engine returns for the currently logged-in username (`APP_USER`). If `:APP_USER` returns `DAVID`, then use `DAVID` as the participant username.

- Configuring Static Workflow Participants
  To create a new participant, choose *Create Participant* from the context menu of the *Participants* node in the *Workflows* tree.
- Determining Participants Dynamically
  To determine workflow participants dynamically, use a participant *Value* > *Type* of *SQL Query*, *Function Body*, or *Expression*.
- Passing in Workflow Participants
  A common requirement is to let the page or workflow activity that creates a workflow pass in the list of task assignees as an input parameter.

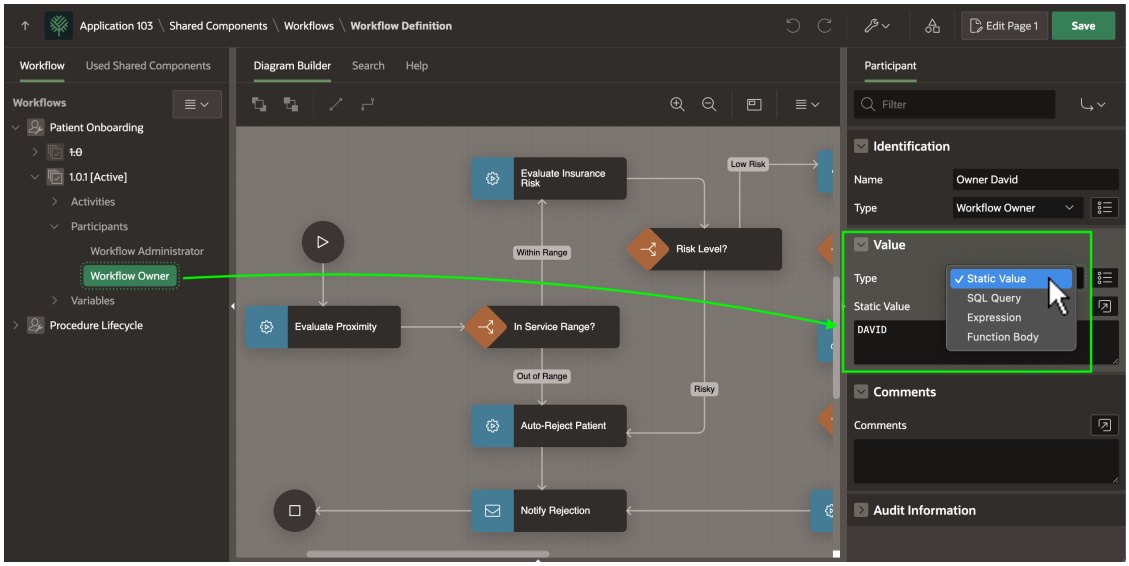
25.1.10.1 Configuring Static Workflow Participants

To create a new participant, choose *Create Participant* from the context menu of the *Participants* node in the *Workflows* tree.

Then set the participant *Type* in the Property Editor. As shown below, in a simple application, either or both of the *Workflow Owner* or *Workflow Administrator* participants can be a static value like `DAVID` or `CARLA`. But normally you use one of the *Value* > *Type* settings like *SQL Query*, *Function Body*, or *Expression* to dynamically determine one or both of these participant lists.

Copyright © 2026, Oracle and/or its affiliates.

Figure 25-71 Configuring Workflow Participants



> **Caution**
>
> Participant usernames are case-sensitive. If `:APP_USER` returns `DAVID`, then use `DAVID`.

25.1.10.2 Determining Participants Dynamically

To determine workflow participants dynamically, use a participant *Value* > *Type* of *SQL Query*, *Function Body*, or *Expression*.

When using the *SQL Query* option, your single column `SELECT` list returns one case-sensitive username per row. For the *Function Body* or *Expression* options, return a comma-separated list of case-sensitive usernames.

Your query, function, or expression can use bind variable syntax to reference workflow parameter values, workflow version *Additional Data* column names, application items, or other APEX substitution strings.

> **Tip**
>
> Workflows execute in the background where there is no "logged-in user". So the `APP_USER` substitution returns `nobody` in a workflow context. Use `APEX$WORKFLOW_INITIATOR` to reference the username that initiated the workflow. If it is not the user you need, define a workflow parameter to accept a username for other purposes.

For easiest maintainability, your expression can invoke a PL/SQL package function as well. The function can be specific to the workflow at hand, or by passing the value of the pre-defined

Copyright © 2026, Oracle and/or its affiliates.

bind variable `APEX$WORKFLOW_ID` your generic function can lookup the related workflow definition using the `APEX_WORKFLOWS` dictionary view's `WORKFLOW_DEF_STATIC_ID` column.

```
your_app_pkg.owners_for_workflow(:APEX$WORKFLOW_ID)
```

> **⚠ Caution**
>
> Participant usernames are case-sensitive. If `:APP_USER` returns `DAVID`, then use `DAVID`.

25.1.10.3 Passing in Workflow Participants

A common requirement is to let the page or workflow activity that creates a workflow pass in the list of task assignees as an input parameter.

This is easy to accomplish by defining a parameter – for example named `P_ASSIGNEE_LIST` – and referencing it in an expression consisting of just the bind variable:

```
:P_ASSIGNEE_LIST
```

Of course, you could also use it in a *SQL Query*, but in that case you need to return one row per case-sensitive username supplied. For example, if the input value is a colon-separated list of usernames, you could use the *SQL Query* option with the query:

```
select column_value
from apex_string.split(:P_ASSIGNEE_LIST,':')
```

> **⚠ Caution**
>
> Participant usernames are case-sensitive. If `:APP_USER` returns `DAVID`, then use `DAVID`.

25.1.11 Listing Workflows and Viewing Details

Create Workflow Console pages to list the workflows a user owns, administers, or initiated.

In the *Create Page Wizard*, create a *Workflow Console* page to list workflow instances. You provide the name for the page that lists workflow instances, and the workflow details page. The *Report Context* you choose determines the filtered list of workflow instances the page displays at runtime.

Table 25-5 Report Context Determines Which Workflow Instances Display

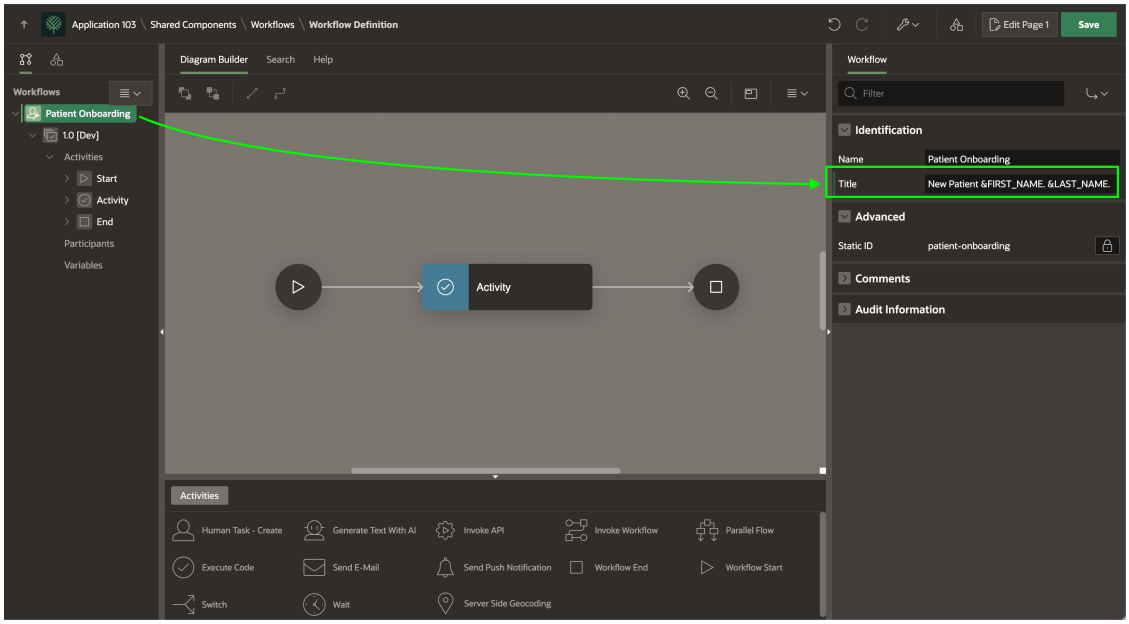
Report Context	Instances Where Current User Is	API Constant
My Workflows	Workflow Owner	`MY_WORKFLOWS`
Admin Workflows	Workflow Administrator	`ADMIN_WORKFLOWS`
Initiated by Me	Initiator	`INITIATED_BY_ME`

You can use the pages as they come, or customize them in Page Designer as needed. The page below is a slightly modified version of an *Admin Workflows* console page. Notice the

Copyright © 2026, Oracle and/or its affiliates.

workflow title you configure is used as the *Title* of the *Content Row* region the generated console page contains.

Figure 25-72 Title Appears in Console Page Listing Workflow Instances



Tip

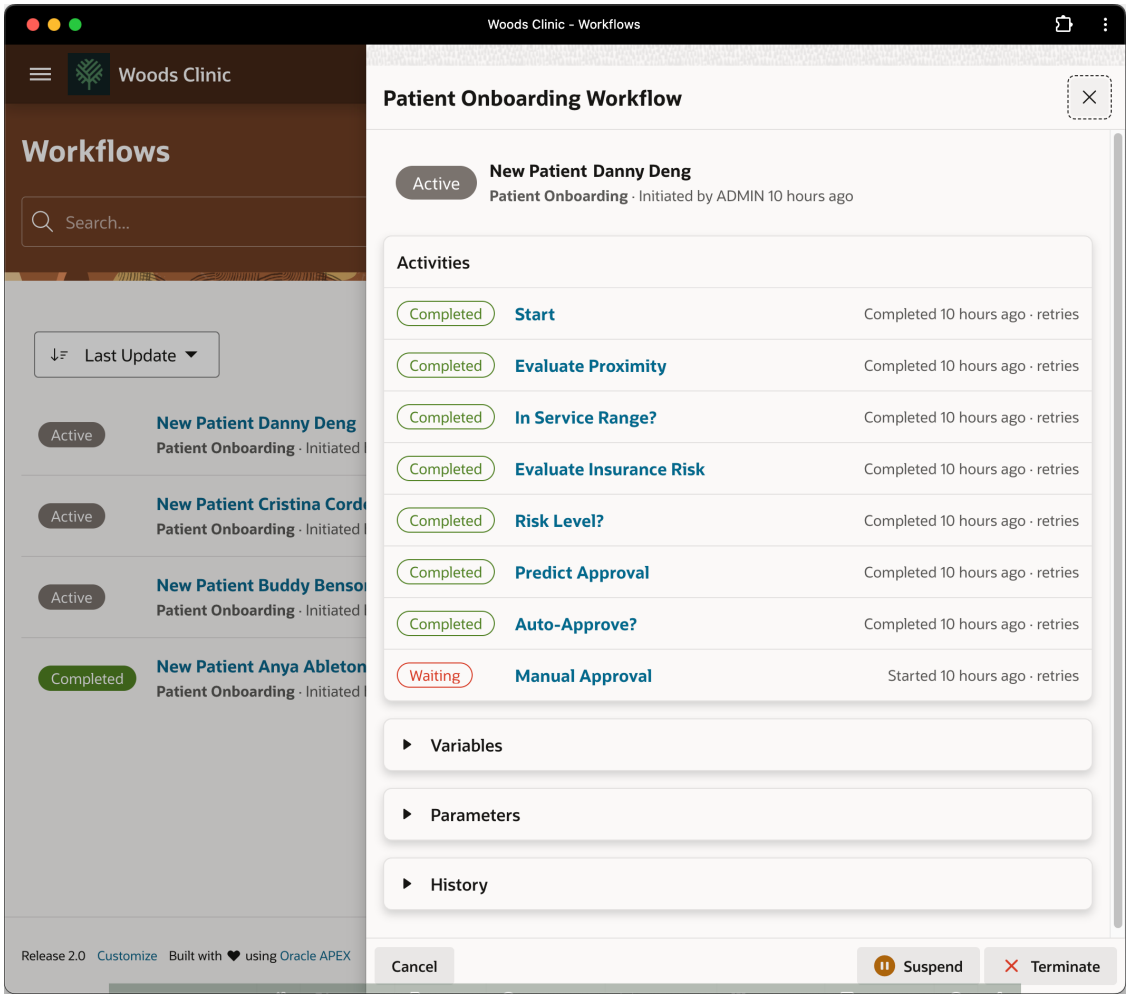
By default, the console's list page shows workflow instances for all applications in the workspace, but you can focus the page to display only workflows for the current application by *uncommenting* the `p_application_id` parameter line in the Content Row region's query shown below. By default, the wizard generates a `SELECT` statement with a *subset* of available columns. Change the query to `select *` to see the complete list of available workflow columns you can reference.

```
select *
  from table ( apex_workflow.get_workflows (
                  p_context => 'ADMIN_WORKFLOWS'
                  --, p_application_id => :APP_ID
                  ) )
```

Clicking on the title of a particular workflow instance like *New Patient Danny Deng*, the generated workflow detail page appears. It shows the list of completed activities for the selected instance. Expand the *Variables*, *Parameters*, and *History* sections to see additional information.

Copyright © 2026, Oracle and/or its affiliates.

Figure 25-73 Workflow Detail Page Displays Status of Selected Instance



25.2 Exploring Patient Onboarding Example

Explore how APEX Workflow automates patient onboarding at the fictitous Woods Clinic.

Consider a concrete use case to explore how an APEX workflow solves real business challenges. Imagine you are the lead developer at *Woods Clinic*, a fictional ambulatory medical clinic. You've been tasked with improving the efficiency of new patient onboarding. You start by understanding the current situation, then use APEX Workflow to deliver business value by automating a number of previously-manual steps.

- Understanding the Current Process
  Understand the manual patient inquiry process before workflow automation.
- Identifying Automation Opportunities
  Identify the patient onboarding steps workflow automation can handle.
- Starting a Patient Onboarding Workflow
  Create a workflow definition and start building its first version in Workflow Designer.
- Associating the Patients Table
  Associate the workflow with the patients table so activities can reference patient row data.

Copyright © 2026, Oracle and/or its affiliates.

- Configuring a Meaningful Title
  Set a meaningful workflow title using values from the related patient row.
- Building Patient Onboarding Visually
  Explore and understand key aspects of building the Patient Onboarding workflow visually.

25.2.1 Understanding the Current Process

Understand the manual patient inquiry process before workflow automation.

A new patient inquires about a particular medical procedure they need. Then, *Woods Clinic* staff members review the provided information to decide if the clinic can accommodate the patient's request.

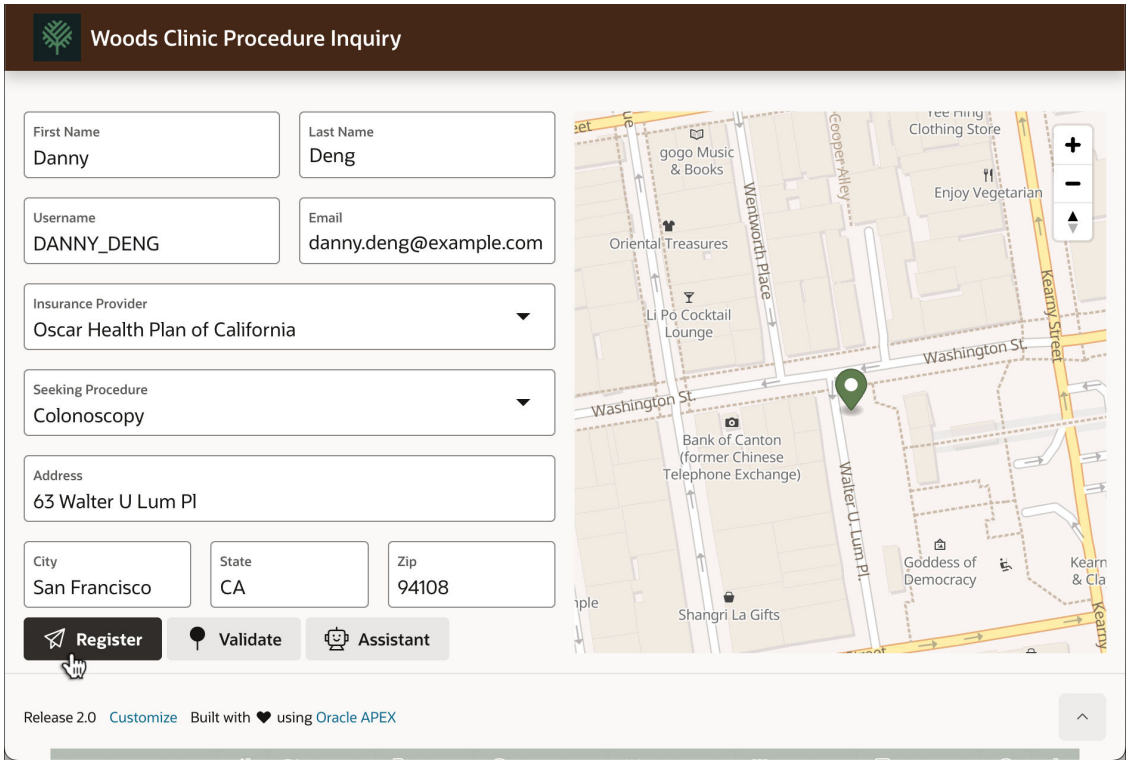
- Inquiring About a Procedure
  Submit a procedure inquiry so the clinic can decide whether it can help.
- Onboarding the Patient Manually
  Review the manual onboarding steps staff follow before approving or rejecting a patient.

25.2.1.1 Inquiring About a Procedure

Submit a procedure inquiry so the clinic can decide whether it can help.

A new patient completes the *Procedure Inquiry* form shown below on the clinic's website to check if the clinic can accommodate a particular medical procedure. They provide contact information, home address, and insurance provider. They also indicate the procedure they seek. In this example, Danny Deng needs a Colonoscopy and hopes Woods Clinic can help.

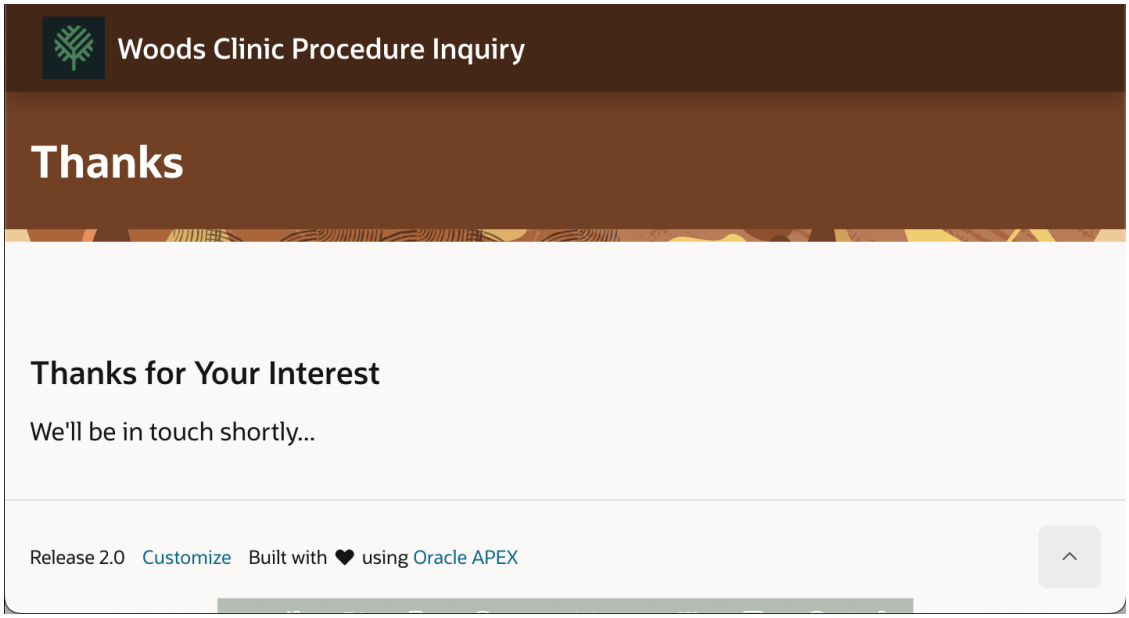
Figure 25-74 Woods Clinic New Patient Procedure Inquiry Form



Copyright © 2026, Oracle and/or its affiliates.

As shown below, after the patient completes this initial registration, they await the clinic's response.

Figure 25-75 After Registering a Procedure Inquiry, a New Patient Awaits a Response



25.2.1.2 Onboarding the Patient Manually

Review the manual onboarding steps staff follow before approving or rejecting a patient.

The clinic's back-office staff currently follows a manual business process to onboard a new potential patient. The steps involve:

- Verifying the patient lives within 400km of the clinic
- Evaluating the insurance reimbursement risk for the procedure they seek by researching past cases
- Deciding whether to approve the new patient's request
- If approved, then provisioning an account for them to access the clinic's portal application
- Periodically checking to see if newly-approved patients logged in to complete their registration

If any of the steps results in a decision to reject the patient's request, a staff member writes an email to the patient apologizing and explaining they need to seek their care at another facility.

25.2.2 Identifying Automation Opportunities

Identify the patient onboarding steps workflow automation can handle.

You identify several opportunities to automate the current manual patient onboarding process. You aim to:

- Automatically reject patients who live outside the clinic's service range
- Proactively reject historically risky combinations of insurance provider and medical procedure

Copyright © 2026, Oracle and/or its affiliates.

- Skip human approval step when the insurer has historically reimbursed the procedure
- Notify rejected patients by email without staff involvement
- Provision approved patient's portal access account and generate a temporary password
- Mark process complete after patient provides their birth date and proof of insurance

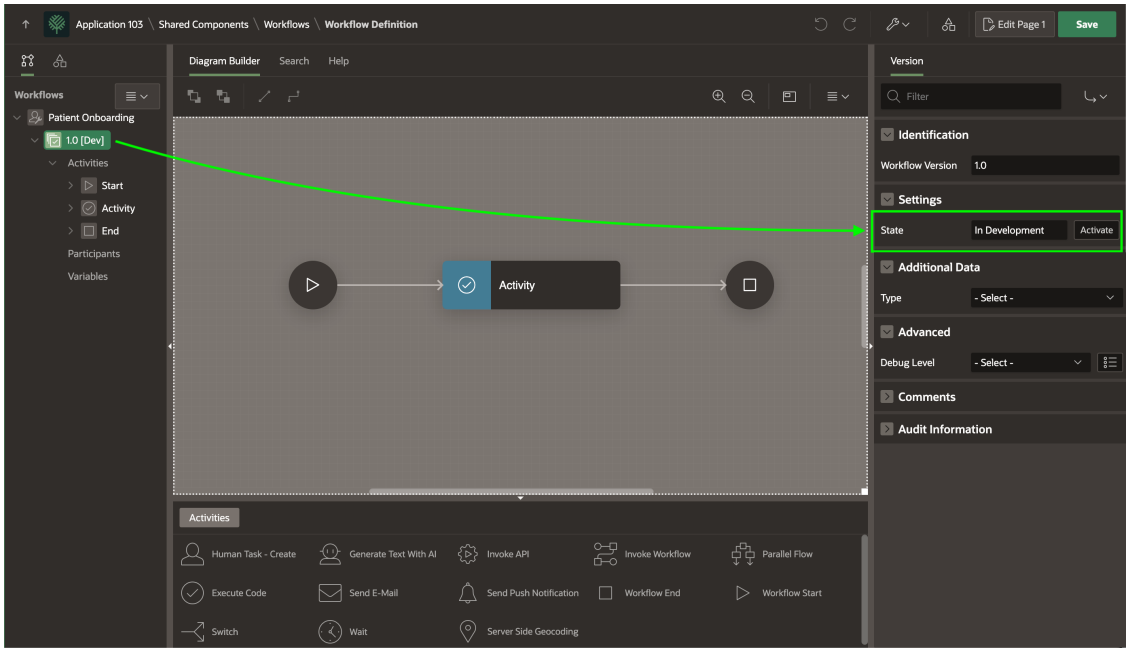
25.2.3 Starting a Patient Onboarding Workflow

Create a workflow definition and start building its first version in Workflow Designer.

In the existing application, you create a new *Workflow* shared component named *Patient Onboarding*. The workflow Diagram Builder feels familiar, since it works like Page Designer. As shown below, your workflow appears in the tree in the *Workflows* tab. Since a business process can change over time, a workflow definition can have multiple versions. Out of habit, you name the first version `1.0`. A version starts in the *In Development* state until you activate it later for use in production. Notice the *1.0 [Dev]* label in the tree below confirms this state. The Property Editor also reflects it.

To get you started, App Builder creates a single *Execute Code* activity sequenced between a *Start* and an *End*. You can change its type and properties as needed, and add other activities and connections to the diagram to build up your flow. As flows get more involved or you identify a reusable "subflow", one workflow can invoke another like a subroutine, passing parameters in and out as required.

Figure 25-76 Workflow Version Starts in Development Mode Until You Activate It



25.2.4 Associating the Patients Table

Associate the workflow with the patients table so activities can reference patient row data.

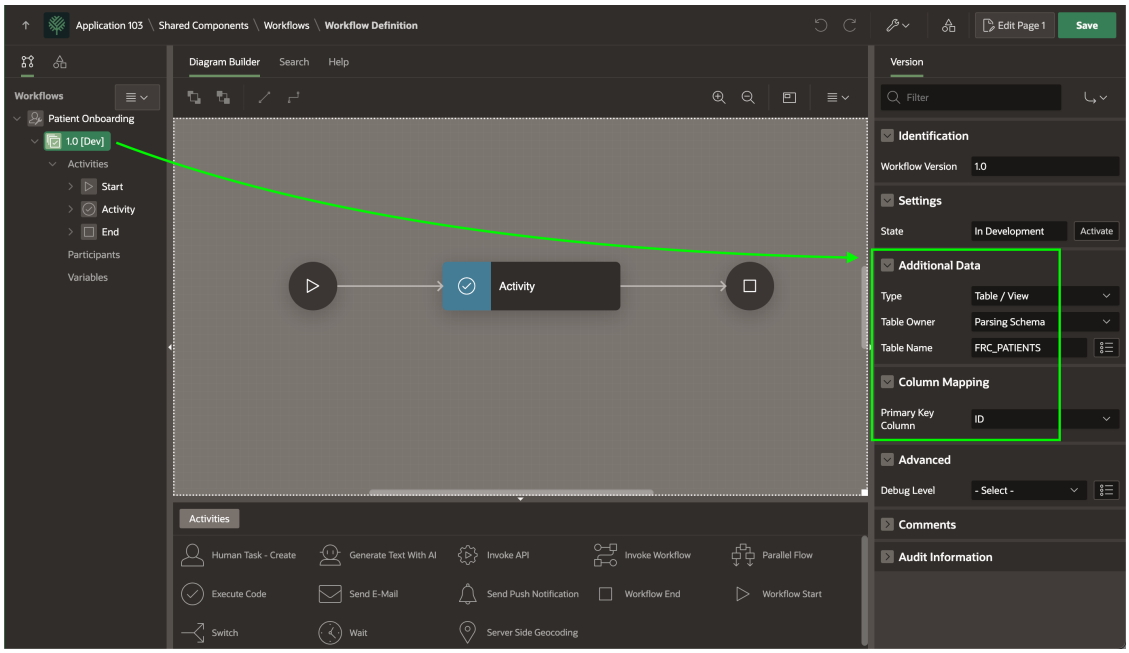
The patient onboarding process relates to patients in the `FRC_PATIENTS` table. Each patient's onboarding journey is a *concrete example* of the process "blueprint" for one *particular* patient. Since the *Patient Onboarding* workflow represents this business process, most of its steps

Copyright © 2026, Oracle and/or its affiliates.

reference or modify the FRC_PATIENTS row whose onboarding progress each *instance* represents.

As shown below, *Additional Data* is a workflow *version* property. Just set the *Patient Onboarding* workflow's table to FRC_PATIENTS and identify ID as its *Primary Key Column*. Then, wherever your app *starts* a new *Patient Onboarding* workflow instance, remember to supply the ID value of the patient to whom the new flow relates. This lets you reference patient details as *&NAME.* or *:NAME anywhere* in the flow using *any* FRC_PATIENTS column name. For example, you can reference the patient's FIRST_NAME and LAST_NAME anywhere you need them in the flow.

Figure 25-77 Getting Started on a Patient Onboarding Workflow



25.2.5 Configuring a Meaningful Title

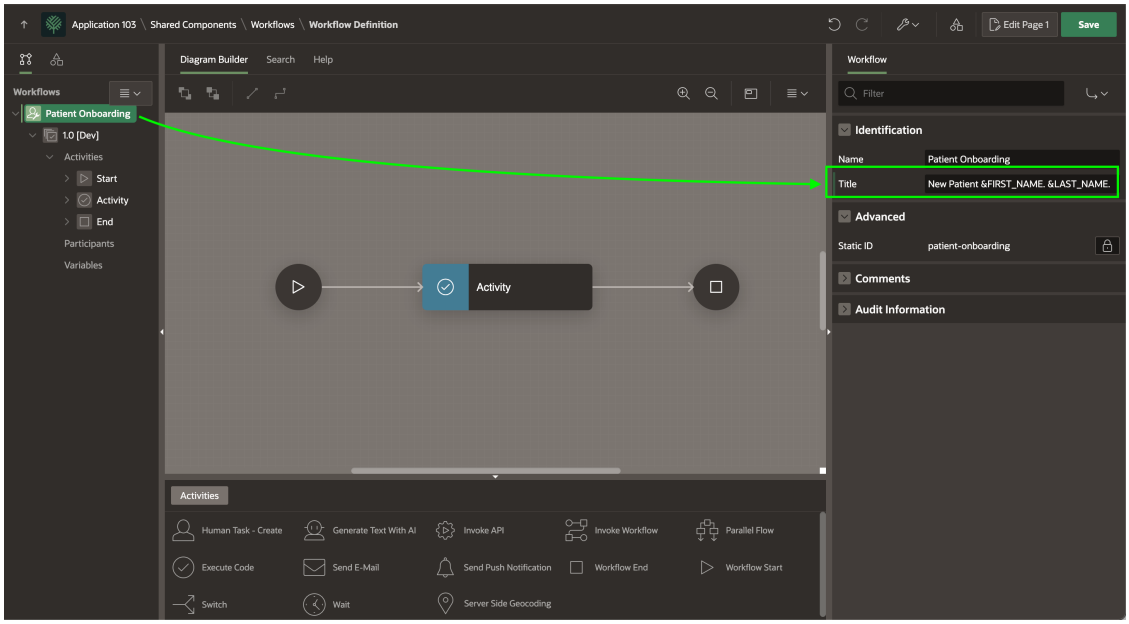
Set a meaningful workflow title using values from the related patient row.

One convenient place to reference patient information is in the workflow's *title*. It appears anywhere users can list "in-flight" workflow instances, so it's important to make it meaningful. As shown below, after configuring your workflow's *Additional Data* table (or query) you can use *&NAME.* notation to include the patient's first name and last name using a *Title* of:

```
New Patient &FIRST_NAME. &LAST_NAME.
```


Copyright © 2026, Oracle and/or its affiliates.

Figure 25-78 Referencing Patient Columns by Name in the Workflow Title



25.2.6 Building Patient Onboarding Visually

Explore and understand key aspects of building the Patient Onboarding workflow visually.

You build business processes visually by adding activities to your workflow diagram, sequencing them with connection arrows, and using the Property Editor to configure their settings. The Woods Clinic *Patient Onboarding* flow uses of many APEX workflow's native features.

- Applying Activities to Patient Onboarding
  Review the native workflow activities used to automate patient onboarding.
- Evaluating Spatial Distance with an API
  Use built-in geospatial functions to check whether a patient lives within service range.
- Switching Path Using a Condition
  Use a Switch activity to decide which path the workflow takes.
- Predicting Results with Machine Learning
  Use historical data to predict likely outcomes and automate confident decisions.
- Visualizing Workflow Progress
  Show a workflow instance's progress in a Workflow Diagram region.
- Awaiting Completion of a Human Task
  Create an approval task and wait for its outcome before the workflow continues.
- Emailing Users a Rejection
  Notify rejected patients with an email template populated from workflow data.
- Provisioning a New User Account
  Provision a portal user account when the workflow approves a patient request.
- Waiting Until Patient Completes Profile
  Wait for the approved patient to complete portal registration before the workflow continues.

Copyright © 2026, Oracle and/or its affiliates.

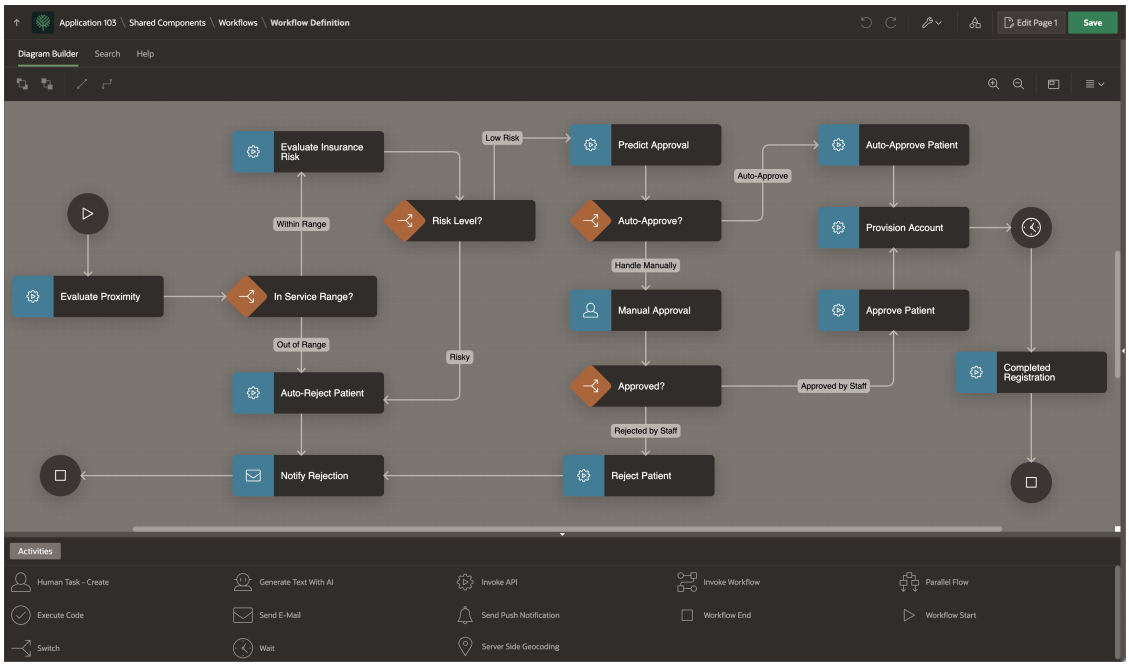
25.2.6.1 Applying Activities to Patient Onboarding

Review the native workflow activities used to automate patient onboarding.

You can create the *Patient Onboarding* workflow shown below to automate the Woods Clinic's manual business process. The diagram uses the following native activities:

- 9 *Invoke API*s run procedures to evaluate proximity and insurance risk, predict approval, and more
- 4 *Switch*s route a patient outside service range, with risky insurance, auto-approved, or rejected
- 1 *Human Task - Create* assigns the patient for manual approval if not auto-approved
- 1 *Send E-Mail* notifies the patient of a rejection, and
- 1 *Wait* pauses until approved patient uses portal to supply proof of insurance and birth date.
- 2 *Workflow End*s terminate the rejected and approved paths separately.

Figure 25-79 Woods Clinic Patient Onboarding Workflow in App Builder



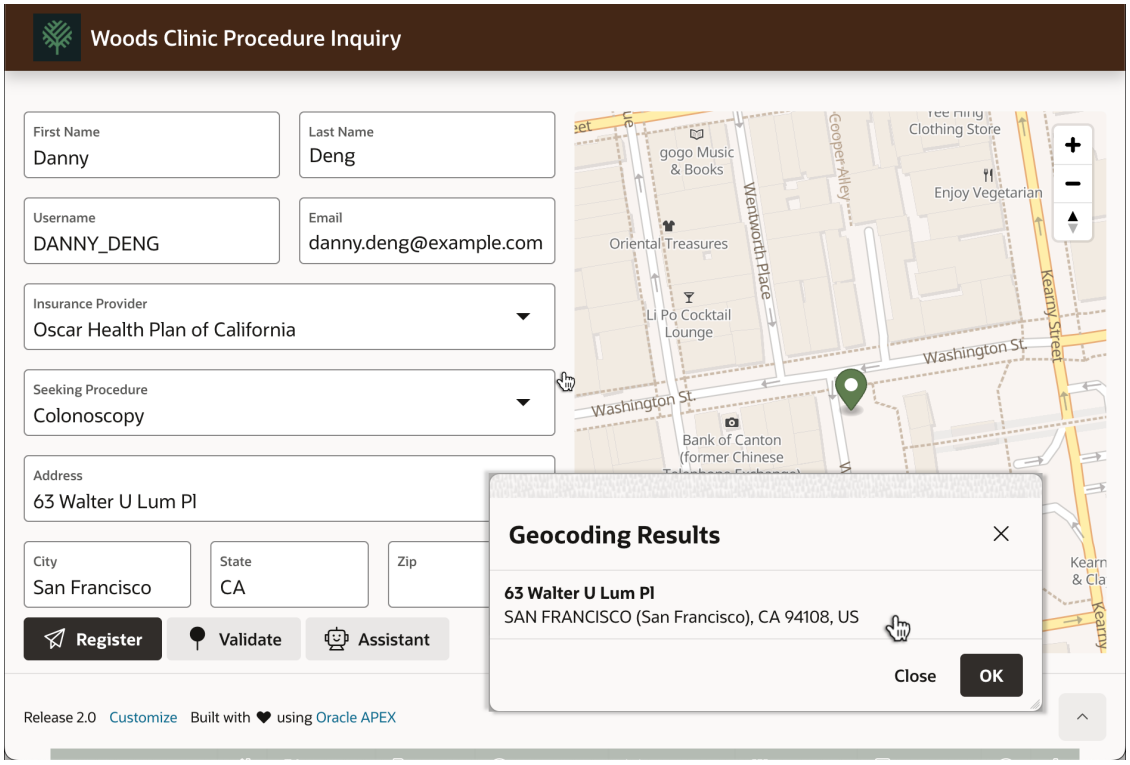
25.2.6.2 Evaluating Spatial Distance with an API

Use built-in geospatial functions to check whether a patient lives within service range.

As shown below, a *Geocoded Address* page item in the registration page identifies the longitude and latitude of a new patient's home address. The page saves these coordinates along with the patient's other data in the `FRC_PATIENTS` table.

Copyright © 2026, Oracle and/or its affiliates.

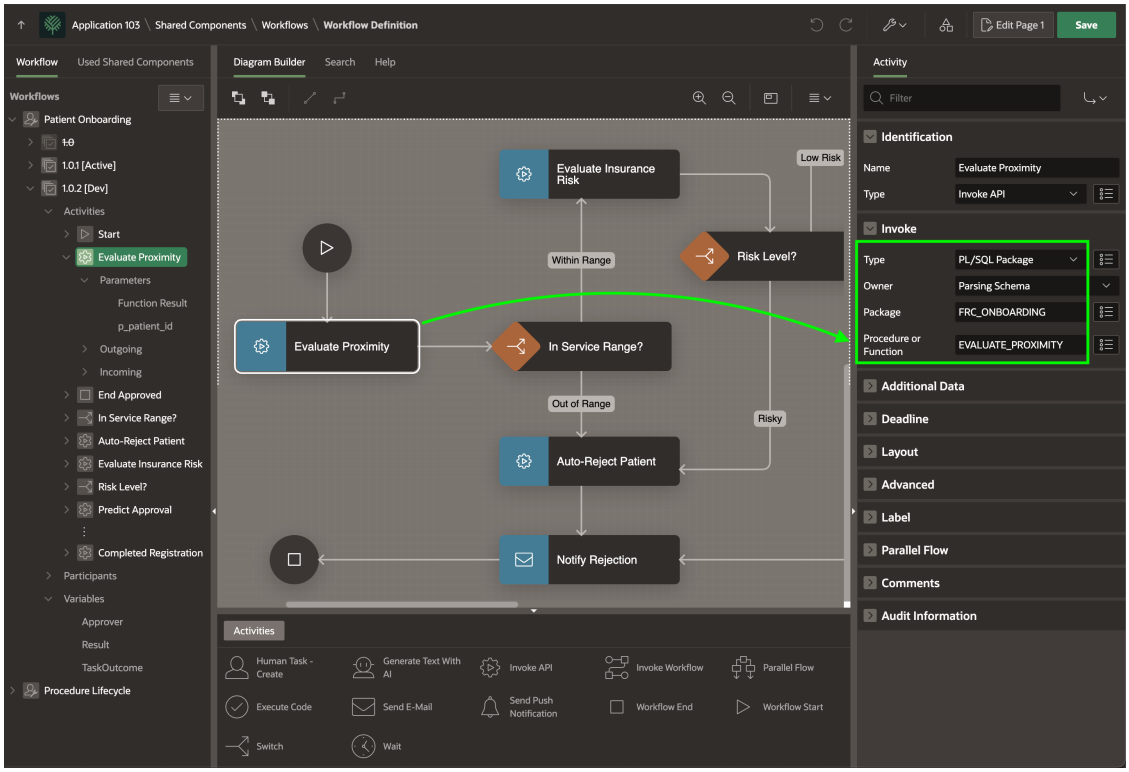
Figure 25-80 Geocoding New Patient Address in Registration Page



The *Evaluate Proximity* activity shown below is an *Invoke API* that calls the `EVALUATE_PROXIMITY` PL/SQL package API to determine if the patient's home address is within the clinic's 400km service radius.

Copyright © 2026, Oracle and/or its affiliates.

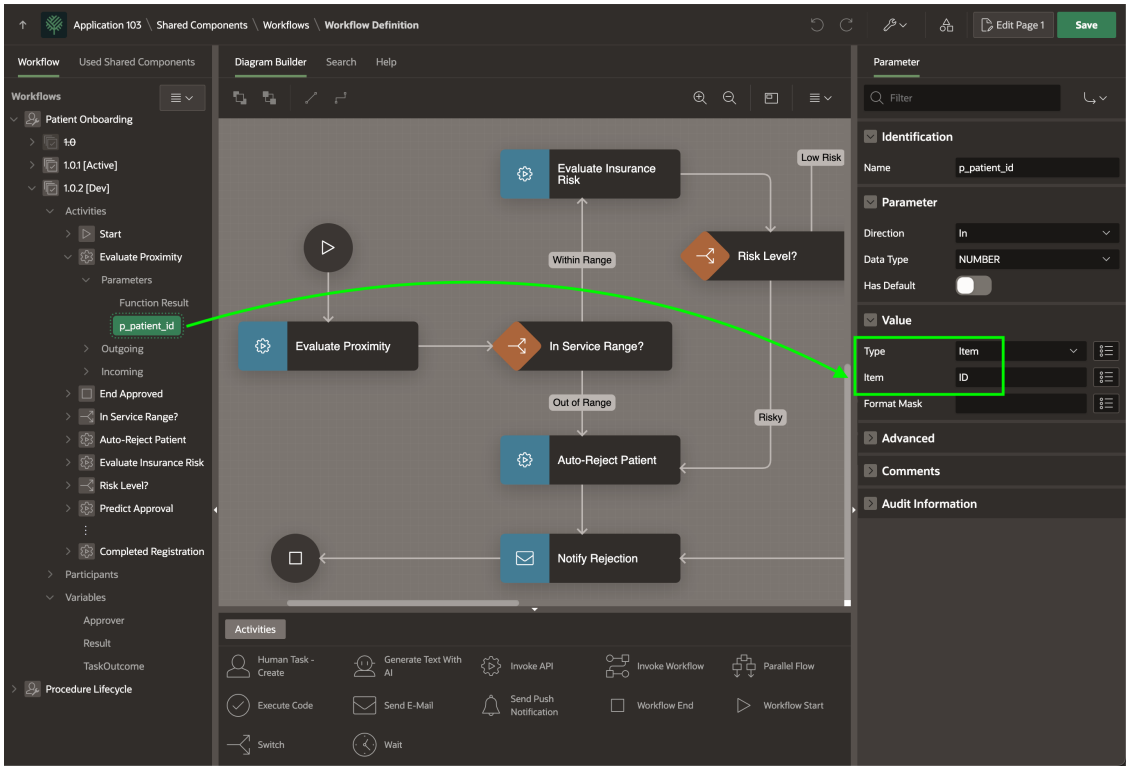
Figure 25-81 Invoking a PL/SQL Package API to Evaluate Service Range



The function accepts a patient ID and returns an outcome string of "`Within Range`" or "`Out of Range`" that the following *In Service Range?* switch activity uses to decide which way to flow. As shown below, you configure the invoked function's parameters using the Property Editor after selecting each parameter name under the activity's *Parameters* heading in the *Workflows* tree. Here, the `P_PATIENT_ID` parameter's value comes from the workflow *Additional Data* column `ID` containing the patient identifier associated to the current workflow instance. The *Function Result* is mapped to the `V_RESULT` workflow variable whose *Result* label shows under the *Variables* heading in the tree.

Copyright © 2026, Oracle and/or its affiliates.

Figure 25-82 Configuring Patient ID as Value of P_PATIENT_ID PL/SQL Parameter



The code for the invoked function appears below. It returns a `VARCHAR2` result of "`Within Range`" if the distance from the clinic to the patient's home address is less than 400km. Otherwise, it returns "`Out of Range`". The function is essentially a single `SELECT…INTO` statement using the `SDO_DISTANCE` function in the `SDO_GEOM` package to compute the distance in kilometers between the clinic and the patient's home address.

```
-- Result = 'Within Range' or 'Out of Range'
function evaluate_proximity(p_patient_id number)
return varchar2
is
    l_result_val varchar2(30);
    /*
     | Assume the clinic is located at:
     | 1244 Cotter Way, Hayward, CA 94541
     */
    c_clinic_lat  constant number := 37.68255;
    c_clinic_long constant number := -122.09026;
begin
    begin
        select case
                when sdo_geom.sdo_distance(
                        sdo_geometry(2001,4326,
                            sdo_point_type(pat.address_longitude,
                                           pat.address_latitude,
                                           null),null,null),
                        sdo_geometry(2001,4326,
                            sdo_point_type(c_clinic_long,
                                           c_clinic_lat,
```


Copyright © 2026, Oracle and/or its affiliates.

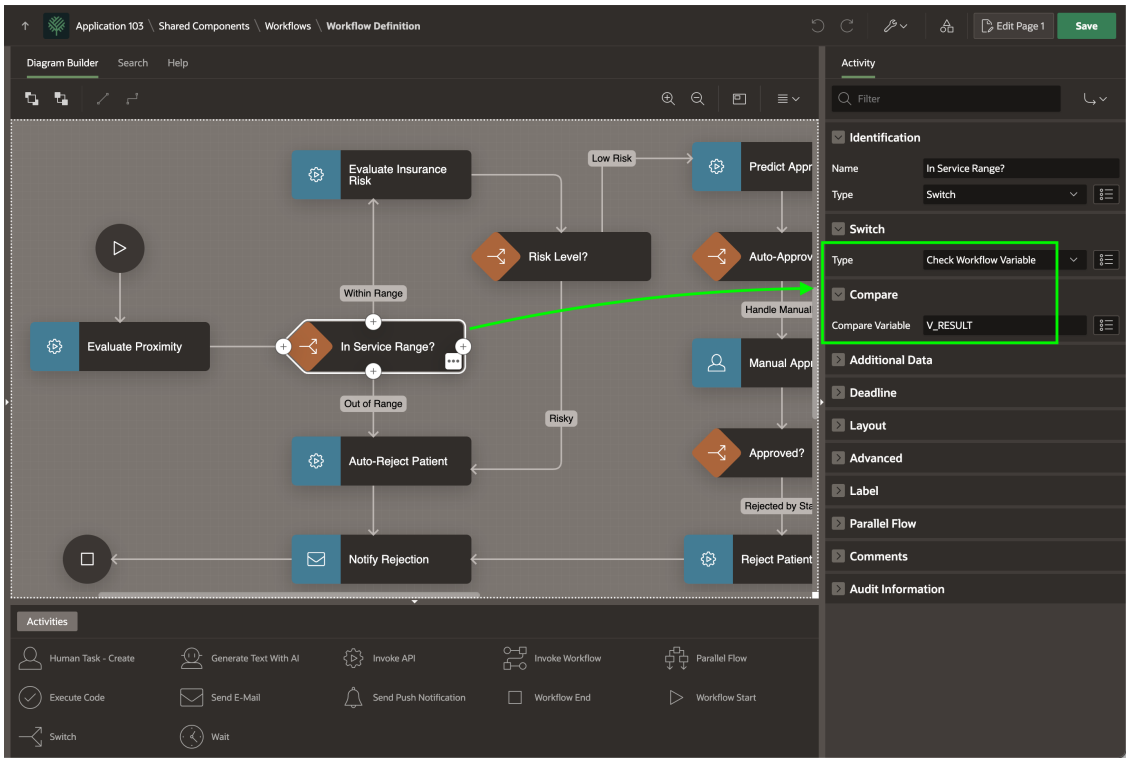
```
                                                null),null,null),
                        tol=>1,
                        unit=>'unit=KM'
                    ) > 400 /* Kilometers */
                  then 'Out of Range'
                  else 'Within Range'
                end
        into l_result_val
        from frc_patients pat
        where pat.id = p_patient_id
          and address_latitude  is not null
          and address_longitude is not null;
    exception
        when no_data_found then
            l_result_val := 'Within Range';
    end;
    return l_result_val;
end;
```

25.2.6.3 Switching Path Using a Condition

Use a Switch activity to decide which path the workflow takes.

The *Evaluate Proximity* activity configures the `V_RESULT` workflow variable to store its *Function Return* value. The subsequent *In Service Range?* is a *Switch* activity that as shown below is set to check this `V_RESULT` variable to decide which path to follow. The *Switch* activity identifies the *source* of the conditional decision. Then, each connection *originating* from that *Switch* activity configures a condition based on this source. When evaluating the next activity, the APEX engine follows the connection path whose condition evaluates to true.

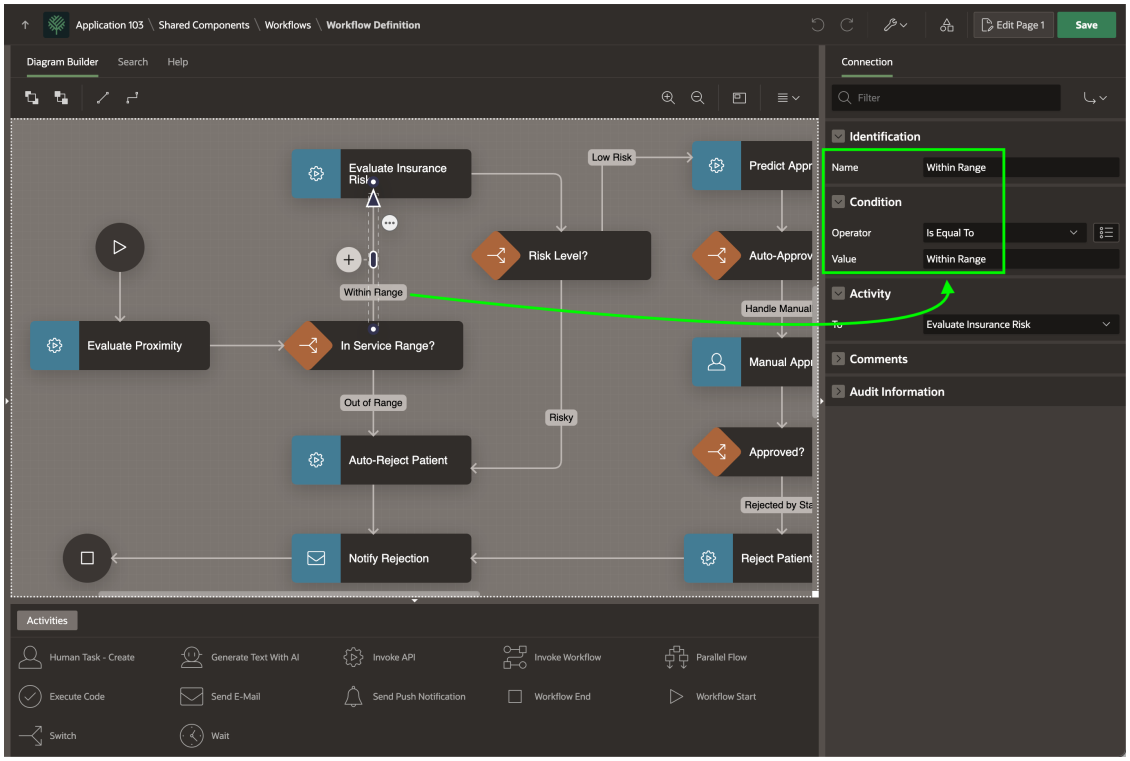
Figure 25-83 Configuring a Switch Activity to Decide Based on a Workflow Variable



Copyright © 2026, Oracle and/or its affiliates.

As shown below, the connection named *Within Range* represents the condition that the *Switch* source `V_RESULT` equals the literal value "`Within Range`". The name of the connection need not coincide with the condition value, but sometimes that makes the diagram the most easy to understand. For example, you could change the connection name to *Inside Service Area*, as long as the condition remains the same, without affecting the workflow's flow.

Figure 25-84 Assigning Outcome Condition to Connection Originating from a Switch



The following table lists all the available Switch types you can use.

Table 25-6 Switch Decision Types

Type	Routes Based On	Outbound Connections
Check Workflow Variable	Distinct values of a variable	One connection per unique value. *Otherwise* evaluated last.
Case	Distinct values of an expression	Any number of connections, each configured with an Execute Sequence. First in sequence to evaluate to *true* is used. Sequence *Otherwise* last if used.
True False Check	True or false result of boolean expression	One *True*, one *False*
If Elsif Else	Evaluating connection conditions in execution sequence order	Any number of connections, each with a Server-side Condition configured and an Execute Sequence. First in sequence to evaluate to *true* is used.

25.2.6.4 Predicting Results with Machine Learning

Use historical data to predict likely outcomes and automate confident decisions.

Copyright © 2026, Oracle and/or its affiliates.

Historical data can be useful to predict the future. After years of considering patient requests manually for onboarding, *Woods Clinic* `FRC_PATIENTS` table holds the institutional knowledge of what patient requests were approved and which were rejected. It potentially holds clues to further automate the patient onboarding business process. You can train a machine learning model then use it to predict patient approvals. If historically, you approved or rejected a situation with high confidence, you can automatically handle similar new cases, leaving only unsure situations to be approved manually.

- Finding Best Algorithm for Your Data
  Use AutoML to find the machine learning model that best predicts your target outcome.
- Predicting Patient Onboarding Outcome
  Invoke a machine learning model to predict patient onboarding outcomes and route the workflow.

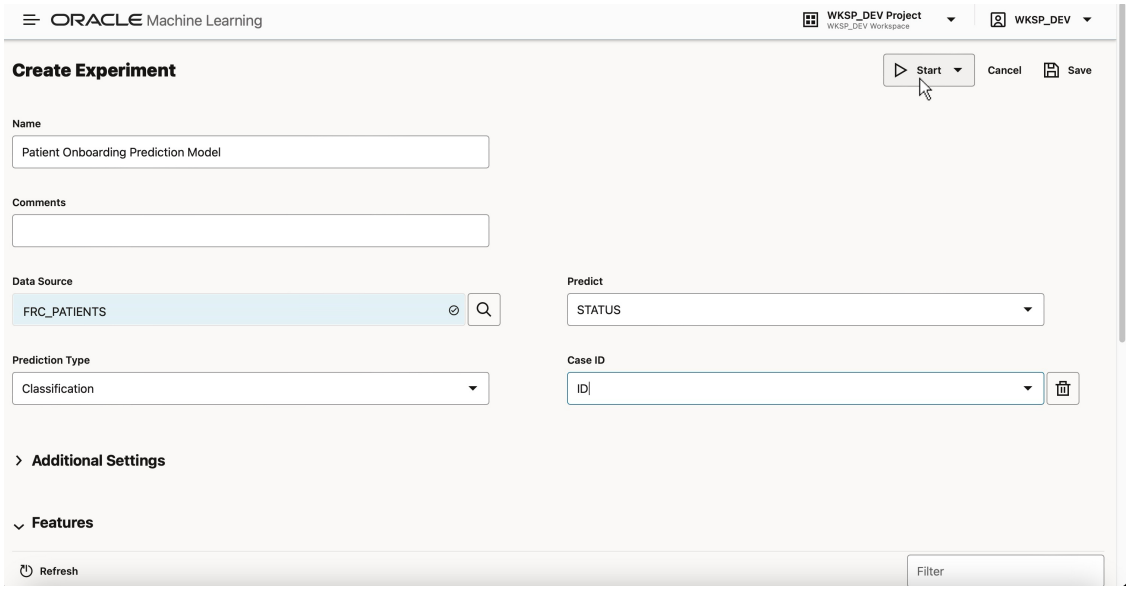
25.2.6.4.1 Finding Best Algorithm for Your Data

Use AutoML to find the machine learning model that best predicts your target outcome.

Your application has access to over 30 different machine learning algorithms. You can use them to "mine" your historical data for interesting correlations. By training a machine learning model using the most appropriate algorithm, your applications can predict future outcomes using that model.

As shown below, Oracle Cloud's *AutoML* tool lets you run an experiment on your historical data to help you determine the machine learning algorithm that best predicts the outcome you seek. Setting up an experiment is simple. You tell the tool which table to analyze, what column represents the outcome you want to predict, and identify the table's primary key column. For the *Woods Clinic* app, you select the `FRC_PATIENTS` table, with its `ID` primary key column, and specify the patient's onboarding `STATUS` as the value you aim to predict.

Figure 25-85 Configuring an Oracle Machine Learning AutoML Experiment

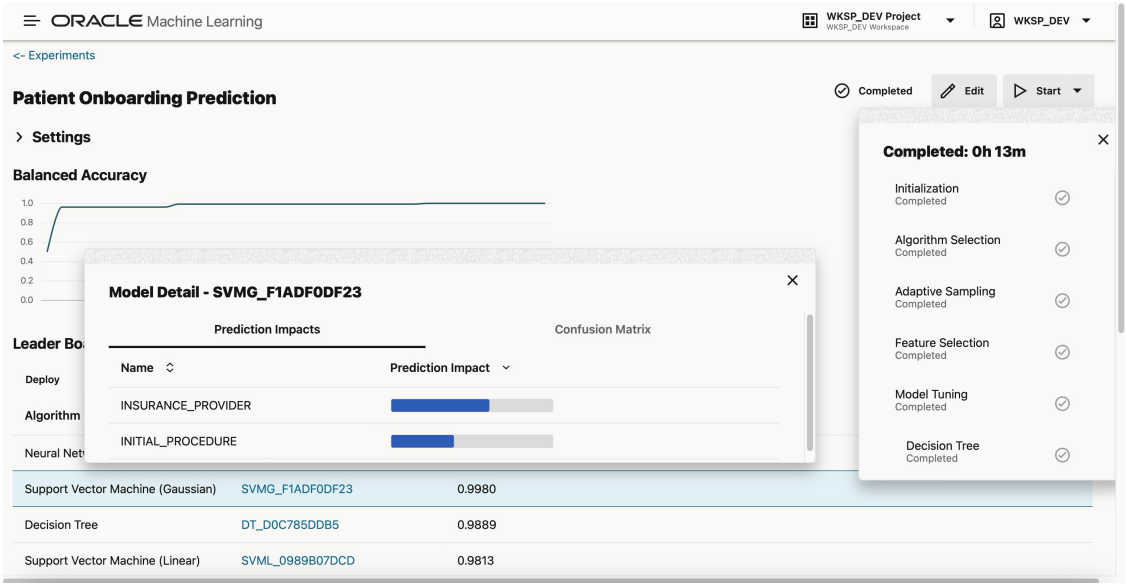


AutoML runs an experiment trying all of its available machine learning algorithms on the *Woods Clinic* patient onboarding data. This experiment takes about 13 minutes to run. As shown below, it keeps a "Leader Board" of the algorithms that deliver the best prediction accuracy. Clicking on the name of any candidate model in the experiment, you can see the

Copyright © 2026, Oracle and/or its affiliates.

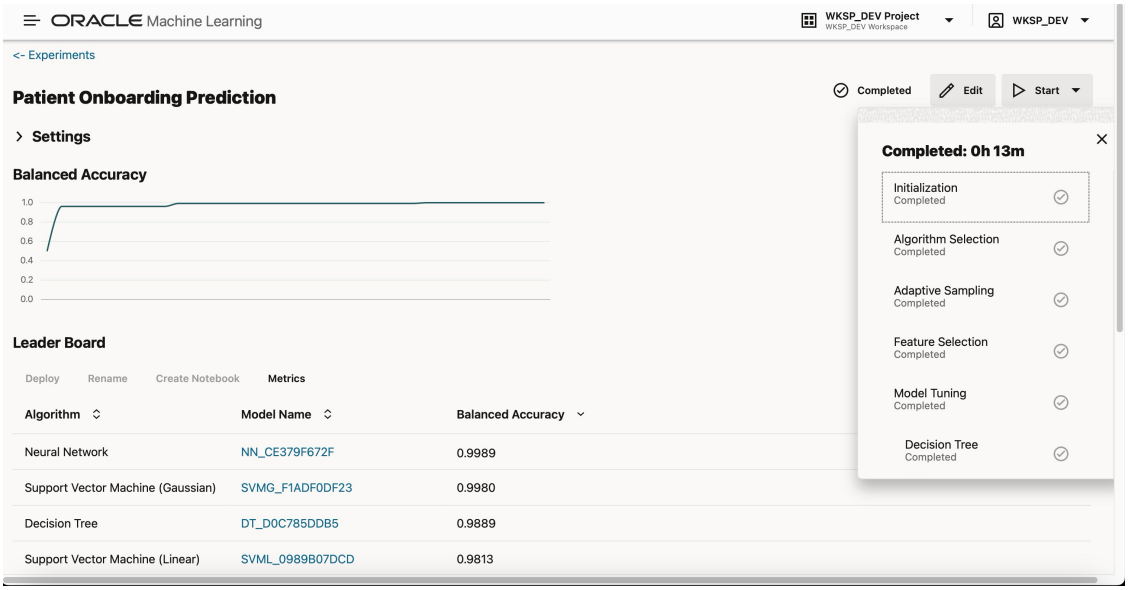
column values that had the strongest impact on predicting the historical outcome. It identifies the INSURANCE_PROVIDER and the INITIAL_PROCEDURE the patient requested during registration as the strongest predictors.

Figure 25-86 Inspecting the Algorithm "Leader Board" Results of the Experiment



The experiment finds the *Neural Network* model with the system-generated name of NN_CE379F672F gives the best results, so you can use that model in your *Patient Onboarding* business process to attempt to automatically predict the patient onboarding approval.

Figure 25-87 Winning Neural Network Model for Woods Clinic Patient Onboarding



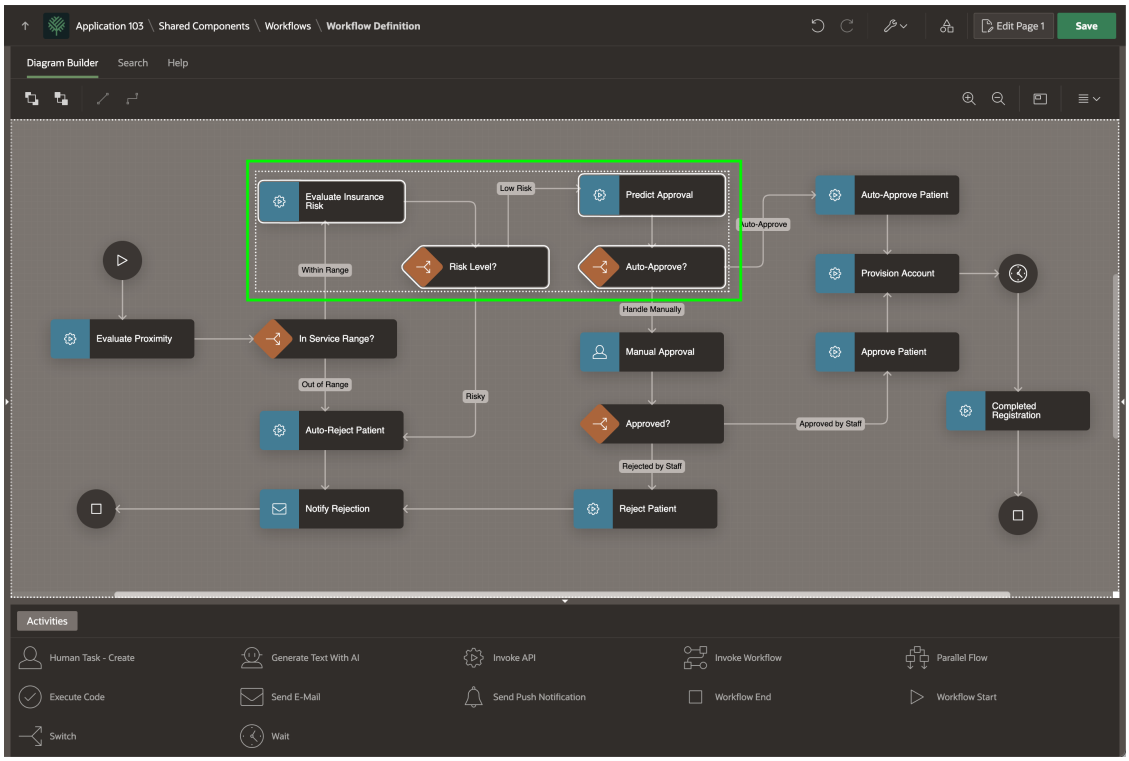
Copyright © 2026, Oracle and/or its affiliates.

25.2.6.4.2 Predicting Patient Onboarding Outcome

Invoke a machine learning model to predict patient onboarding outcomes and route the workflow.

The *Patient Onboarding* workflow uses the machine learning model in two different *Invoke API* activities. First, the *Evaluate Insurance Risk* calls the `EVALUATE_RISK` function to detect if historically a patient with a similar request to the current one was automatically rejected. Office staff tells you that this automatic rejection was due to the patient's insurance not reimbursing the procedure.

Figure 25-88 Two Combinations of Invoke API and Switch Using Machine Learning



The code for `EVALUATE_RISK` appears below. Predicting the outcome of the onboarding patient approval using the Neural Network machine learning model that won the AutoML experiment is a single SQL statement using the `PREDICTION` function. Its `USING` clause feeds the model the values of the current patient's insurance provider and initial procedure. Recall that these were the two columns that most strongly predicted the outcome. If the model predicts the `STATUS` to be "`AutoRejected`", then the function returns the outcome "`Risky`", otherwise it returns "`Low Risk`". The *Evaluate Insurance Risk* activity returns the *Function Result* into the `V_RESULT` workflow variable, and the subsequent *Risk Level?* Switch activity uses that variable containing the outcome value to decide the path to follow.

```
-- Result = 'Low Risk' or 'Risky'
function evaluate_risk(p_patient_id number)
return varchar2 is
    l_result varchar2(30);
begin
    /*
```


Copyright © 2026, Oracle and/or its affiliates.

```
       | Machine Learning Model NN_CE379F672F was the winning model
       | automatically identified by using AutoML.
       */
      select
           prediction (NN_CE379F672F
           using
              pat.insurance_provider  as insurance_provider,
              pat.initial_procedure   as initial_procedure
           )
        into l_result
        from frc_patients pat
       where pat.id = p_patient_id;
      if l_result = 'AutoRejected' then
          return 'Risky';
      else
          return 'Low Risk';
      end if;
end;
```

The *Predict Approval* activity uses *Invoke API* again to call a similar `PREDICT_APPROVAL` function. It accepts the previous prediction as a parameter. If that prediction was "`AutoApproved`" then it returns the outcome string "`Auto-Approved`", otherwise it returns the string "`Manual`. The *Predict Approval* activity returns the *Function Result* into the `V_RESULT` workflow variable, and the subsequent *Auto-Approve?* Switch activity uses that variable containing the outcome value to decide the path to follow.

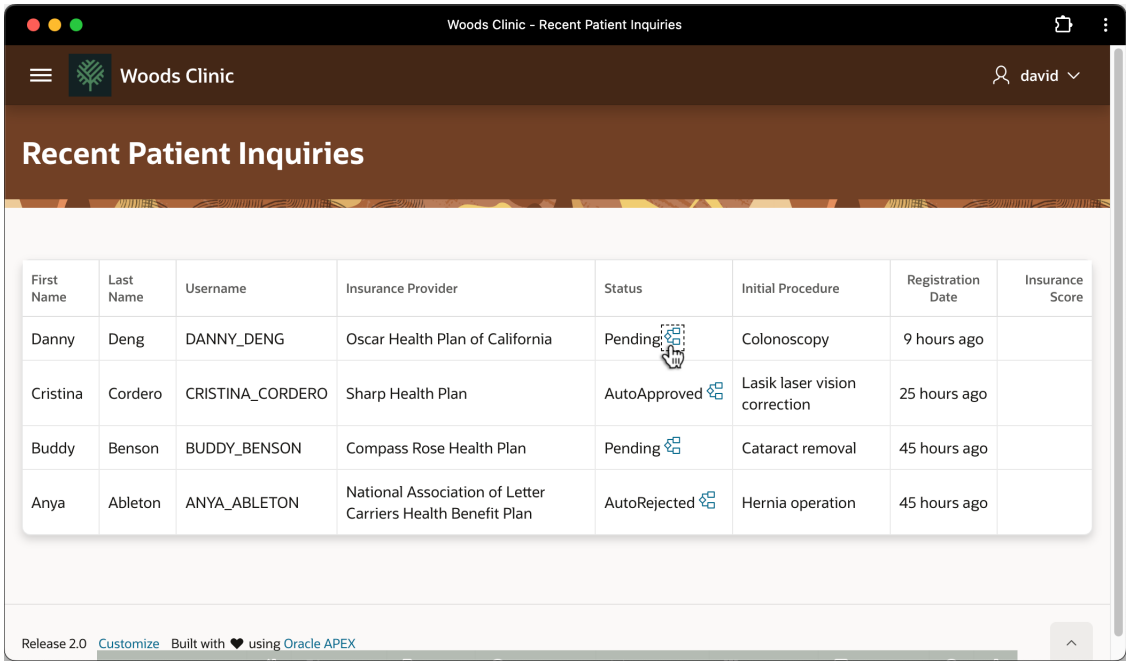
25.2.6.5 Visualizing Workflow Progress

Show a workflow instance's progress in a Workflow Diagram region.

In the *Woods Clinic* app, you can use a *Workflow Diagram* region in a page to visualize the progress of a particular instance. For example, imagine your end users asked to quickly understand where each recent inquiry was in the onboarding process. As shown below, you can add an link icon to your *Recent Patient Inquiries* page to pass the workflow instance ID to a modal page containing a *Workflow Diagram* region.

Copyright © 2026, Oracle and/or its affiliates.

Figure 25-89 Linking to a Workflow Diagram Page to Visualize Progress



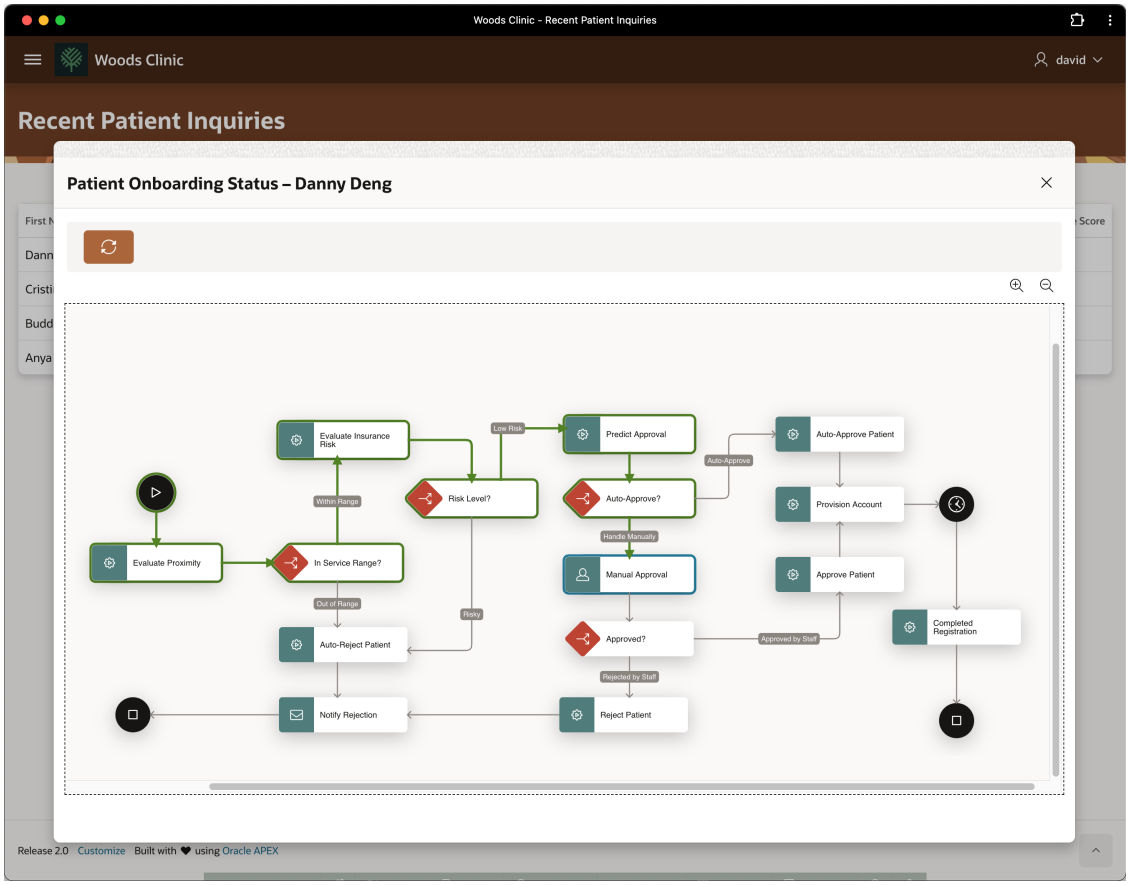
Given the workflow instance ID value, the *Workflow Diagram* region displays its progress as shown below. A colored outline highlights the activities and connections Danny Deng's onboarding journey has taken so far. At a glance, it's easy to see that:

- his home address was within the clinic's service range
- his insurance company had a reasonably low risk of refusing to pay, and
- his procedure request didn't qualify to be auto-approved.

So it's clear that Danny Deng's procedure request currently awaits manual approval by back-office staff

Copyright © 2026, Oracle and/or its affiliates.

Figure 25-90 Visualizing Onboarding Progress of New Patient Danny Deng



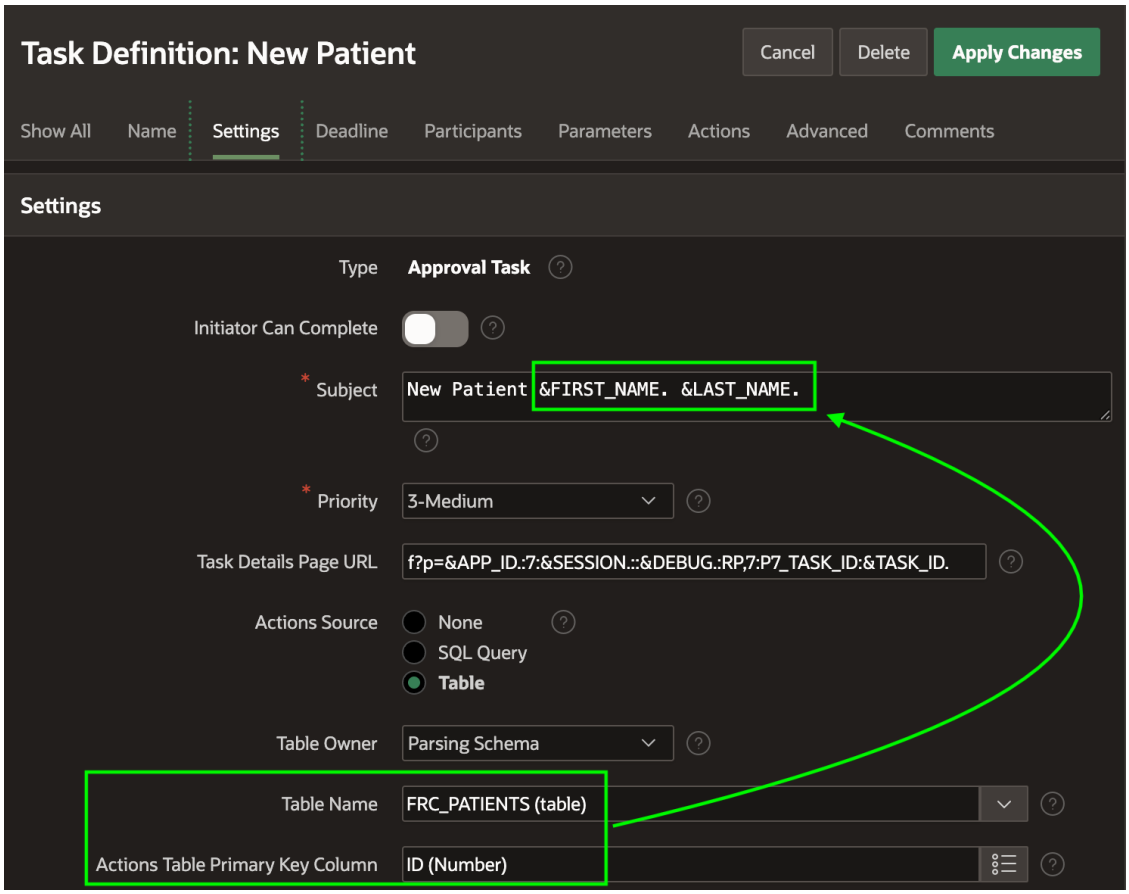
25.2.6.6 Awaiting Completion of a Human Task

Create an approval task and wait for its outcome before the workflow continues.

Your *Woods Clinic* app uses the approval task definition shown below to track the work item for manual approval of a new patient. Its *Actions Source* table is `FRC_PATIENTS` so its *Subject* can reference patient details using substitutions referencing columns like `&FIRST_NAME.` and `&LAST_NAME.` in that table.

Copyright © 2026, Oracle and/or its affiliates.

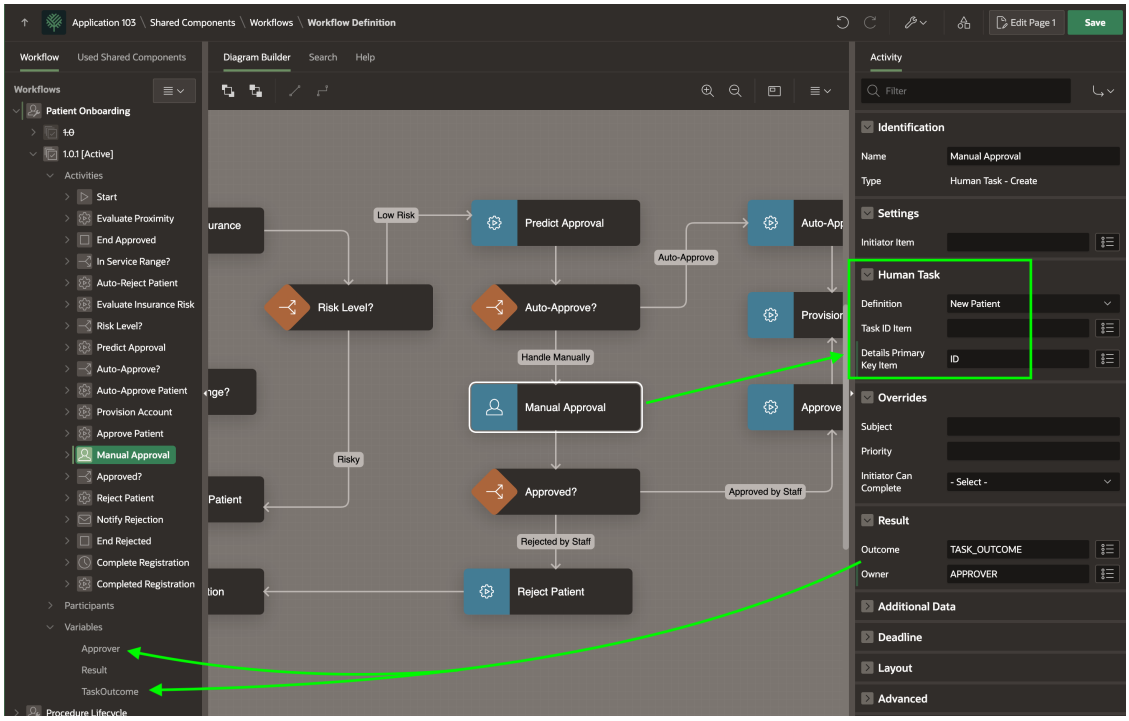
Figure 25-91 Task Definition Subject Uses Column Values from the Actions Source Table



As shown below, the *Patient Onboarding* workflow uses a *Human Task - Create* activity configured to use this task definition. It passes in the patient ID as the *Details Primary Key* value when creating the task using the workflow's *Additional Data* table `ID` column. In its *Result* section, the approval *Outcome* and actual *Owner* are returned into the workflow variables named `TASK_OUTCOME` and `APPROVER`. Since capturing these two values from an approval task is very common, the Workflow Builder creates these two variables automatically when you first add a *Human Task - Create* activity to the diagram. You can rename them, or configure *different* variables to receive these values as well.

Copyright © 2026, Oracle and/or its affiliates.

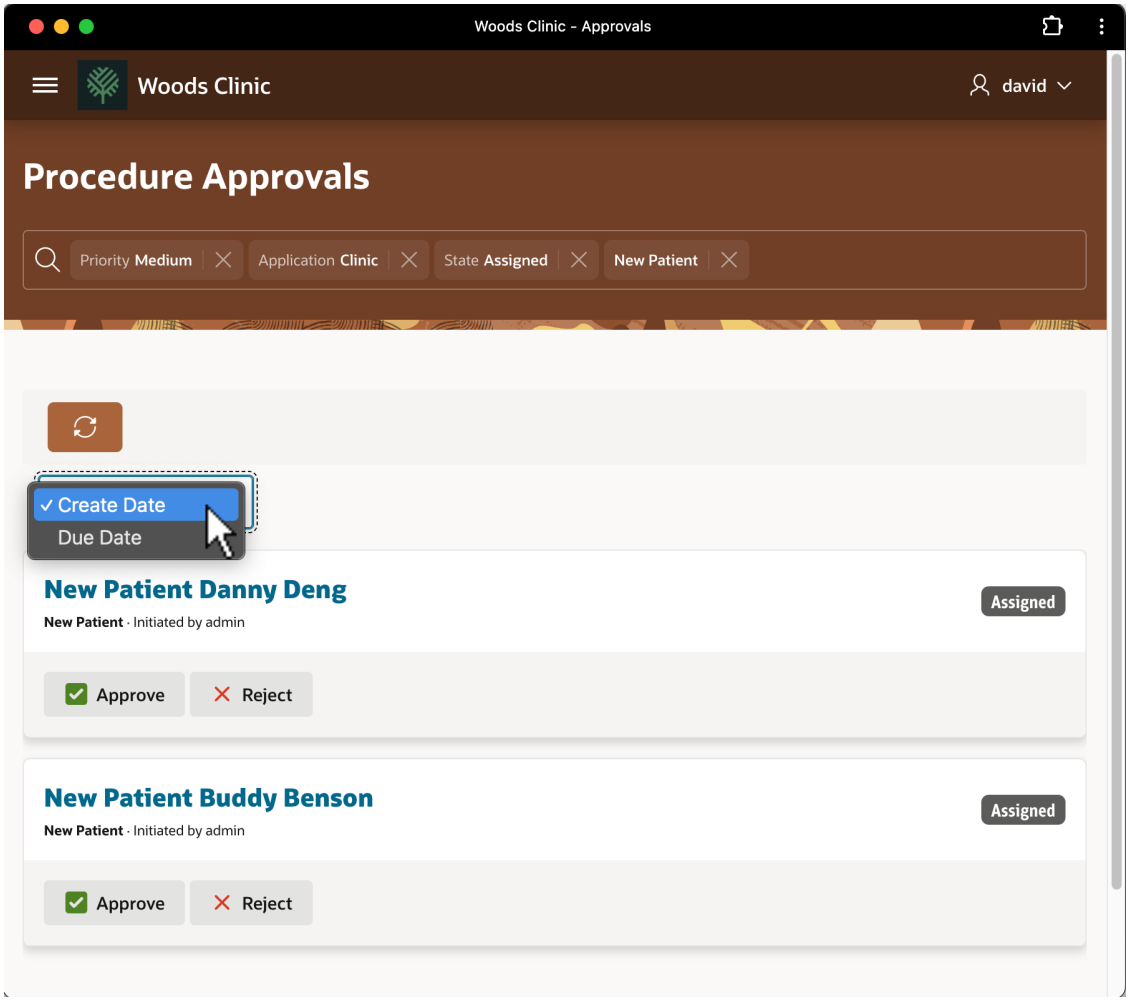
Figure 25-92 Using Human Task Create Activity to Wait for Approval



Your *Patient Onboarding* workflow guides Danny Deng's request to manual approval, so, as shown below, back office user David sees Danny's pending approval in his *Procedure Approvals* task "inbox". You can create this page by picking a *Universal Task List* in the Create Page Wizard, and using the *My Tasks* report context. Notice David has used several Smart Filters to narrow the list by *Priority*, *Application*, and *State*. He also typed in a text search of "New Patient" to further filter the list. He's currently sorting the list by *Create Date*.

Copyright © 2026, Oracle and/or its affiliates.

Figure 25-93 Unified Task List Showing Patient Onboarding Approval Tasks for DAVID



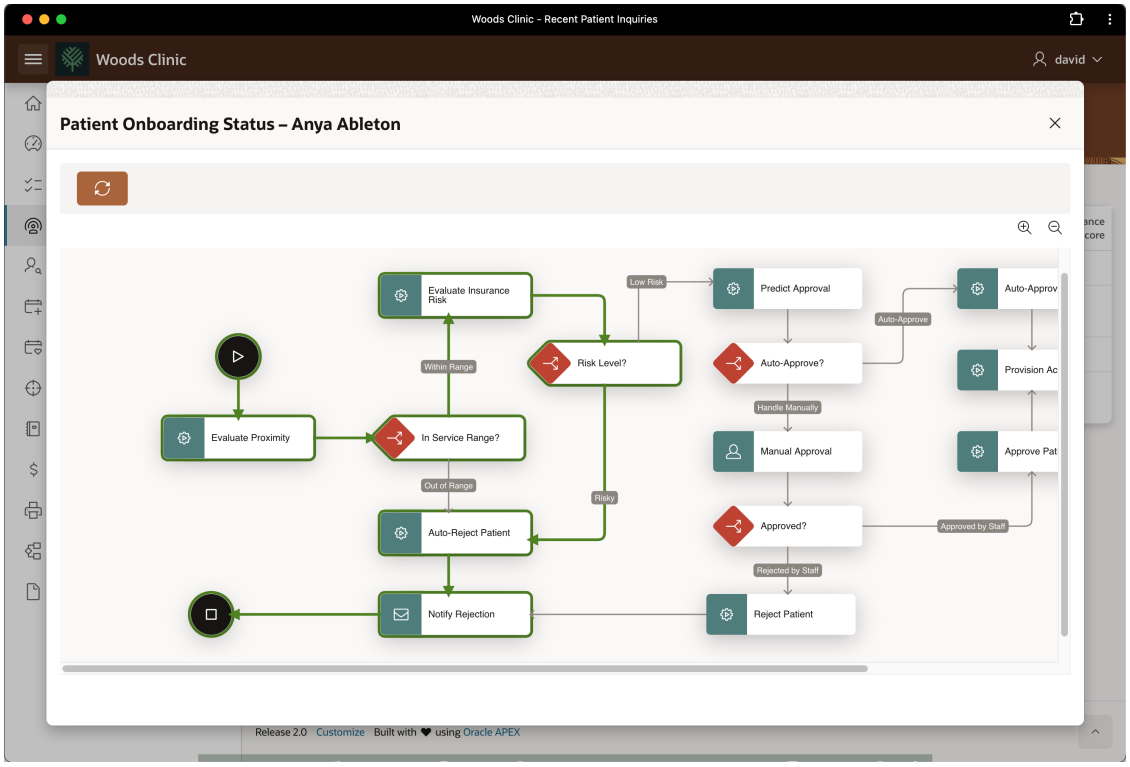
25.2.6.7 Emailing Users a Rejection

Notify rejected patients with an email template populated from workflow data.

From the *Recent Patient Inquiries* page, as shown below, David can see that patient Anya Ableton's request was automatically rejected. The figure shows a Woods Clinic backoffice application modal dialog with a Workflow Diagram region. The dialog title shows *"Patient Onboarding Status - Anya Ableton"*. The workflow diagram is an end-user-friendly view of the same workflow *Patient Onboarding* diagram from **App Builder** Workflow Designer. The color and bolder style of certain activity shape outlines and connection arrows help the user understand the progress. In this example, David sees the *Risk Level?* decision followed the *Risky* path to the *Auto-Reject Patient* activity and on to the *Notify Rejection* activity.

Copyright © 2026, Oracle and/or its affiliates.

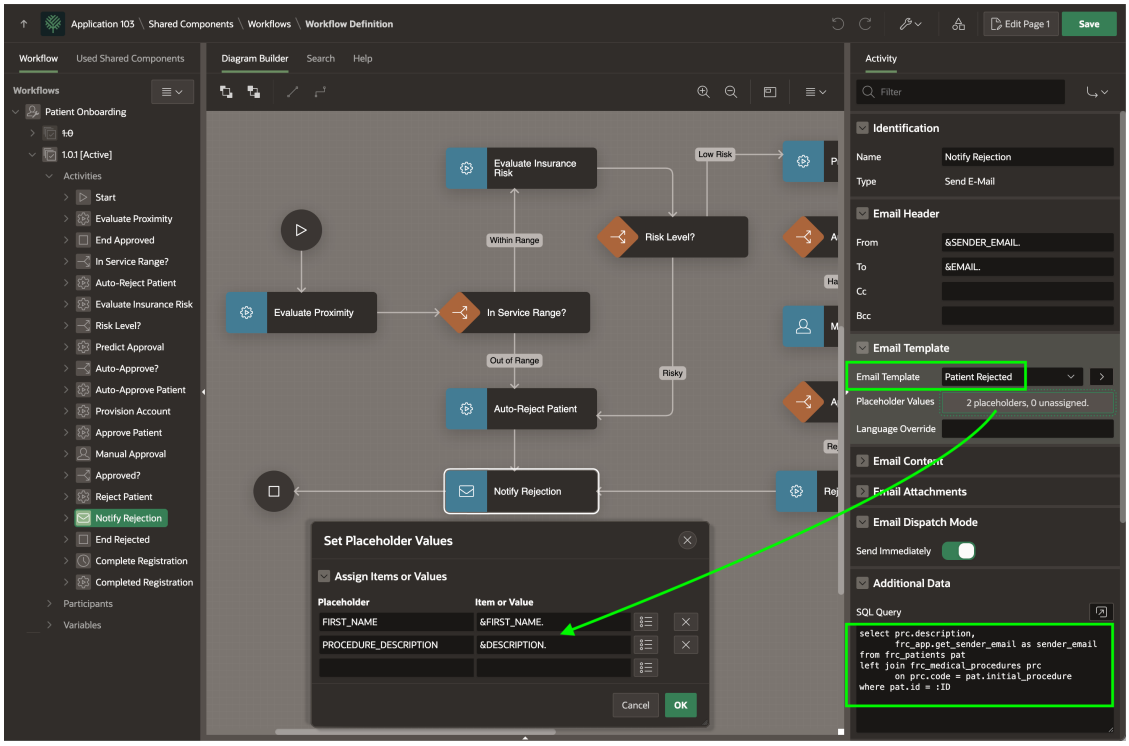
Figure 25-94 Anya Ableton's Procedure Request Was Rejected for Being Too Risky



In the *Patient Onboarding* workflow, your *Notify Rejection* activity of type *Send E-Mail* uses the *Patient Rejected* email template above, and passes `&FIRST_NAME.` and `&DESCRIPTION.` as the values of the template's two placeholders. Notice the *From* field references the `&SENDER_EMAIL.` column from the activity's *Additional Data* query. The value of `&DESCRIPTION.` used for the placeholder also comes from that query.

Copyright © 2026, Oracle and/or its affiliates.

Figure 25-95 Configuring Send Email Activity and Email Template Placeholders



- Emailing or Push Notifying Users
  Notify workflow users with email or push notification activities.
- Using Email Template with Placeholders
  Use an email template with placeholders for workflow notification content.
- Querying Activity Data On Demand
  Query activity-level data to supply email fields and template placeholders.
- Receiving the Rejection Email
  Receive an automatic rejection email when the workflow denies a patient request.

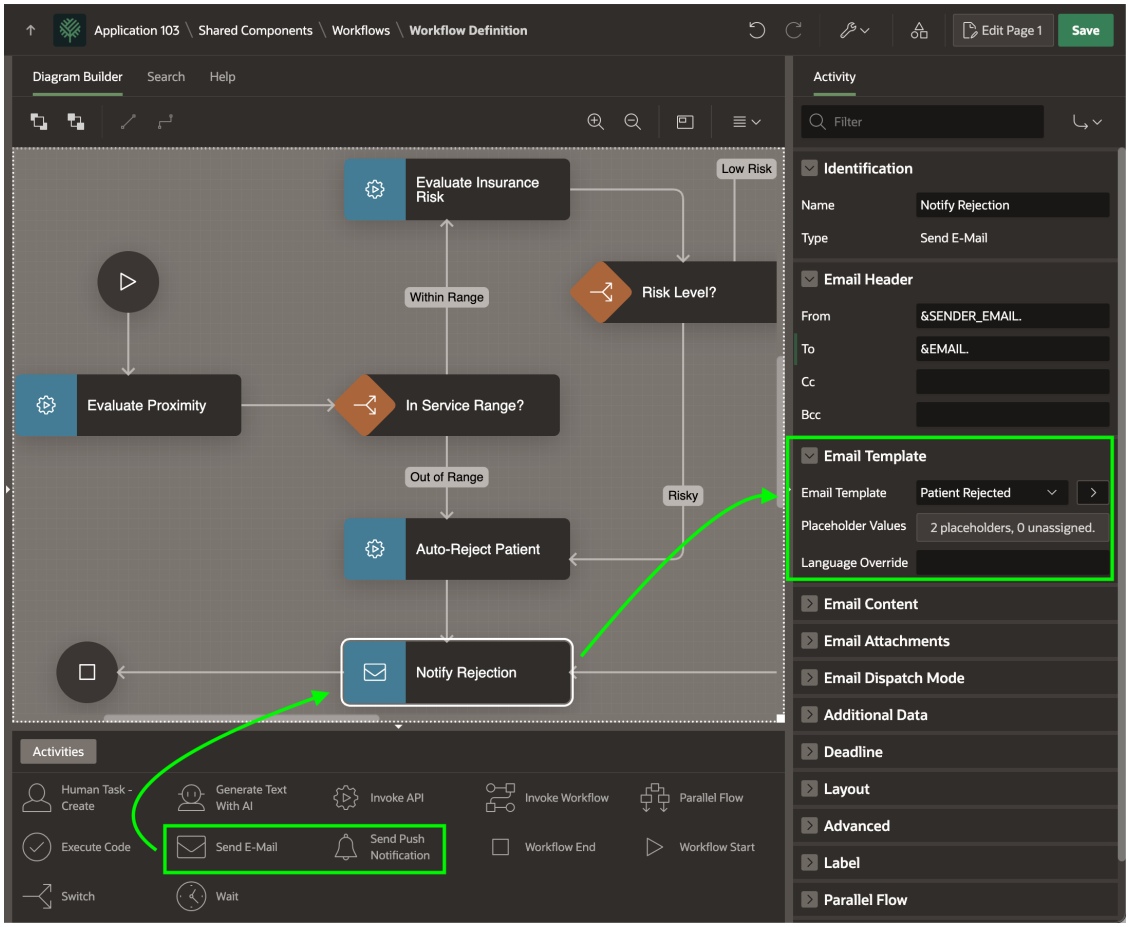
25.2.6.7.1 Emailing or Push Notifying Users

Notify workflow users with email or push notification activities.

It's equally easy to send users a push notification or an email. The figure shows the Workflow Designer palette offers both **Send E-Mail** and **Send Push Notfication** activities. The Woods Clinic app opted for the email option. Notice the *Notify Rejection* Send E-Mail activity is selected in the diagram, and the **Email Template** configured is *Patient Rejected*.

Copyright © 2026, Oracle and/or its affiliates.

Figure 25-96 Native Activities for Alerting Users with Email or Push Notifications



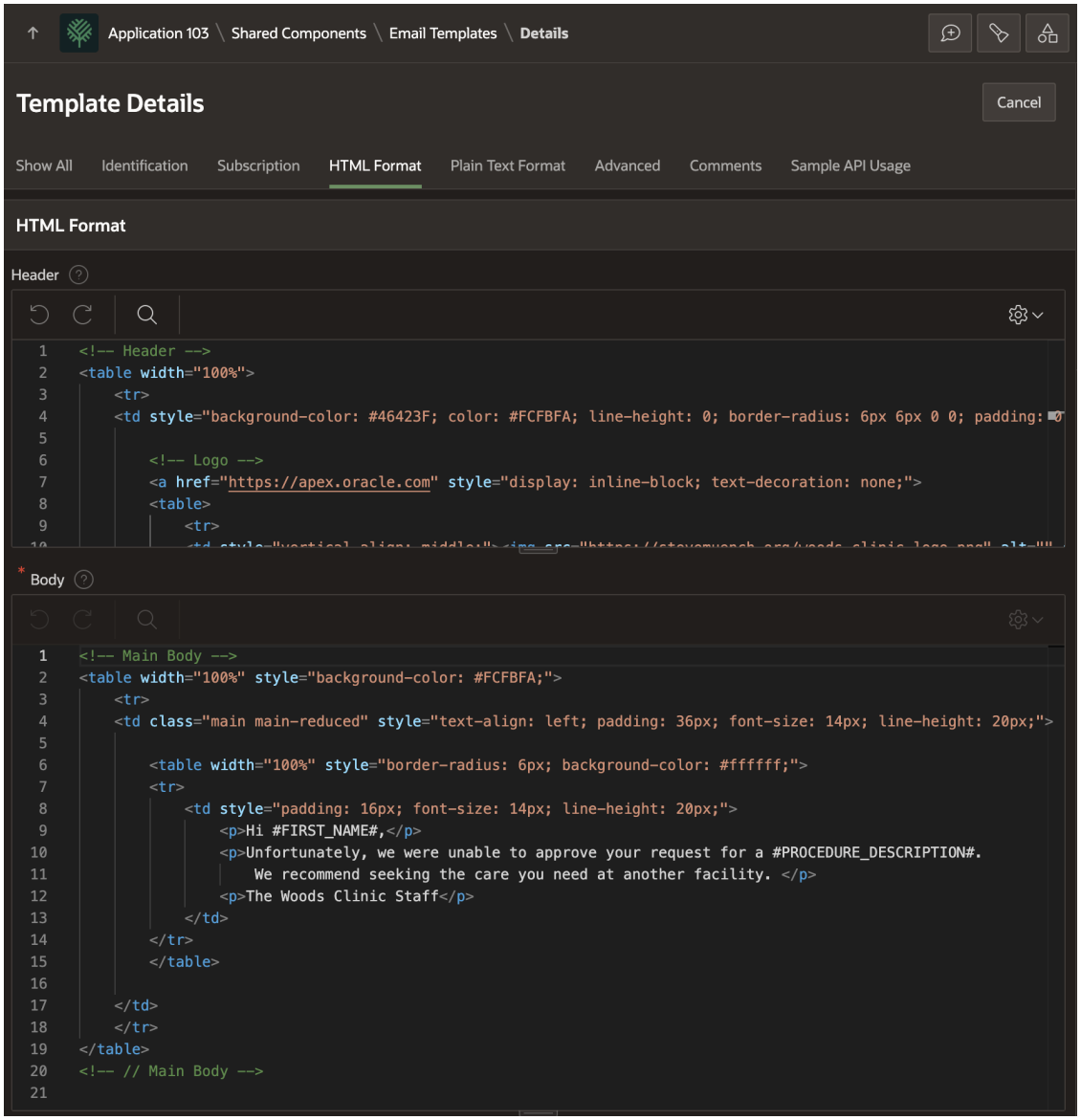
25.2.6.7.2 Using Email Template with Placeholders

Use an email template with placeholders for workflow notification content.

To send the rejection email, you *first* setup an appropriate *Patient Rejected* shared component email template shown below. It includes two placeholders in the email *Body* `#FIRST_NAME#` and `#PROCEDURE_DESCRIPTION#`.

Copyright © 2026, Oracle and/or its affiliates.

Figure 25-97 Rejection Email Template with Two Placeholders



25.2.6.7.3 Querying Activity Data On Demand

Query activity-level data to supply email fields and template placeholders.

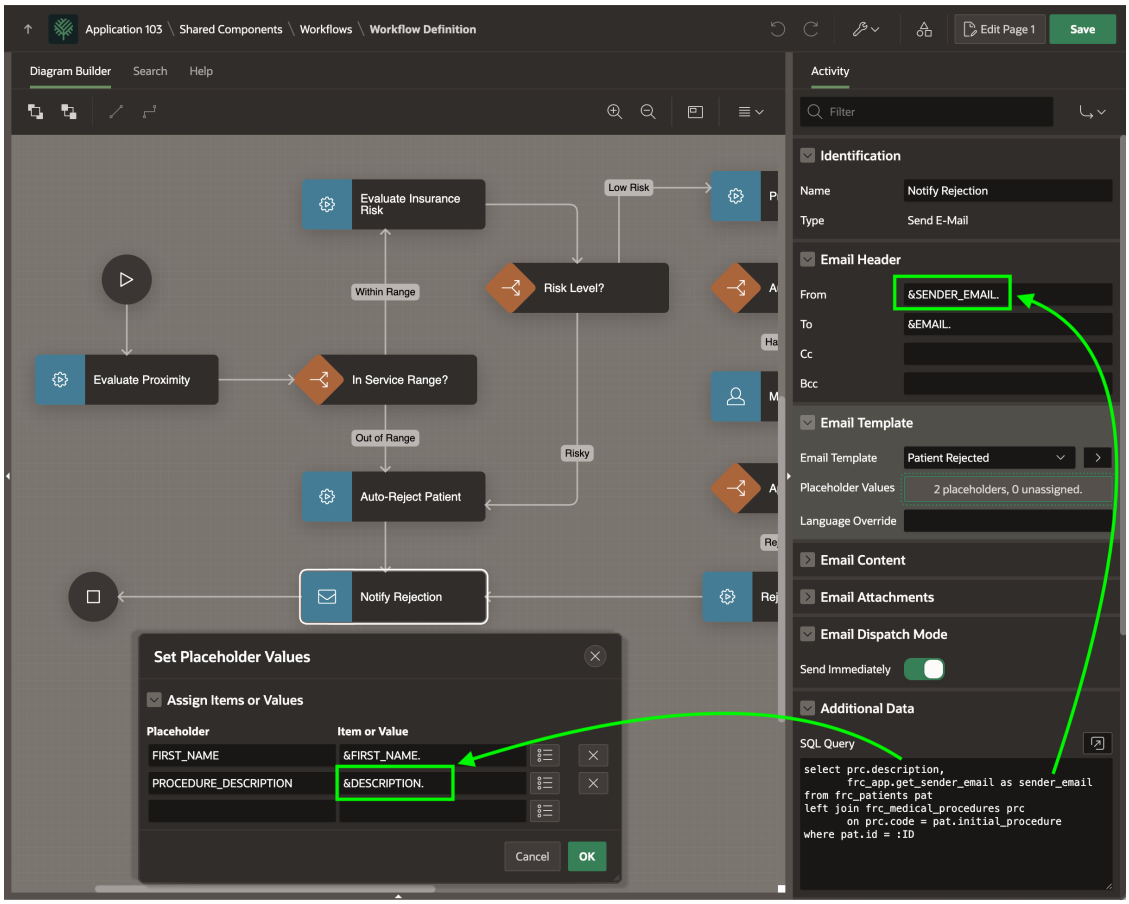
As shown below, the *Notify Rejection* activity uses the following *Additional Data* query. It retrieves the sender's email, and the description of the requested medical procedure by joining in another table. The *From* field references the `SENDER_EMAIL` column value, while a placeholder value uses the `DESCRIPTION` column.

```
select prc.description,
       frc_app.get_sender_email as sender_email
from frc_patients pat
left join frc_medical_procedures prc
       on prc.code = pat.initial_procedure
where pat.id = :ID
```


Copyright © 2026, Oracle and/or its affiliates.

The figure shows the **App Builder** Workflow Designer with the Notify Rejection activity selected in the diagram. The Property Editor shows the **Additional Data** SQL Query above, that includes a `SENDER_EMAIL` column in its select list. The **Email Header** section references its value using `&SENDER_EMAIL.` in the value of its **From** property.

Figure 25-98 Referencing Additional Data Columns in an Activity



25.2.6.7.4 Receiving the Rejection Email

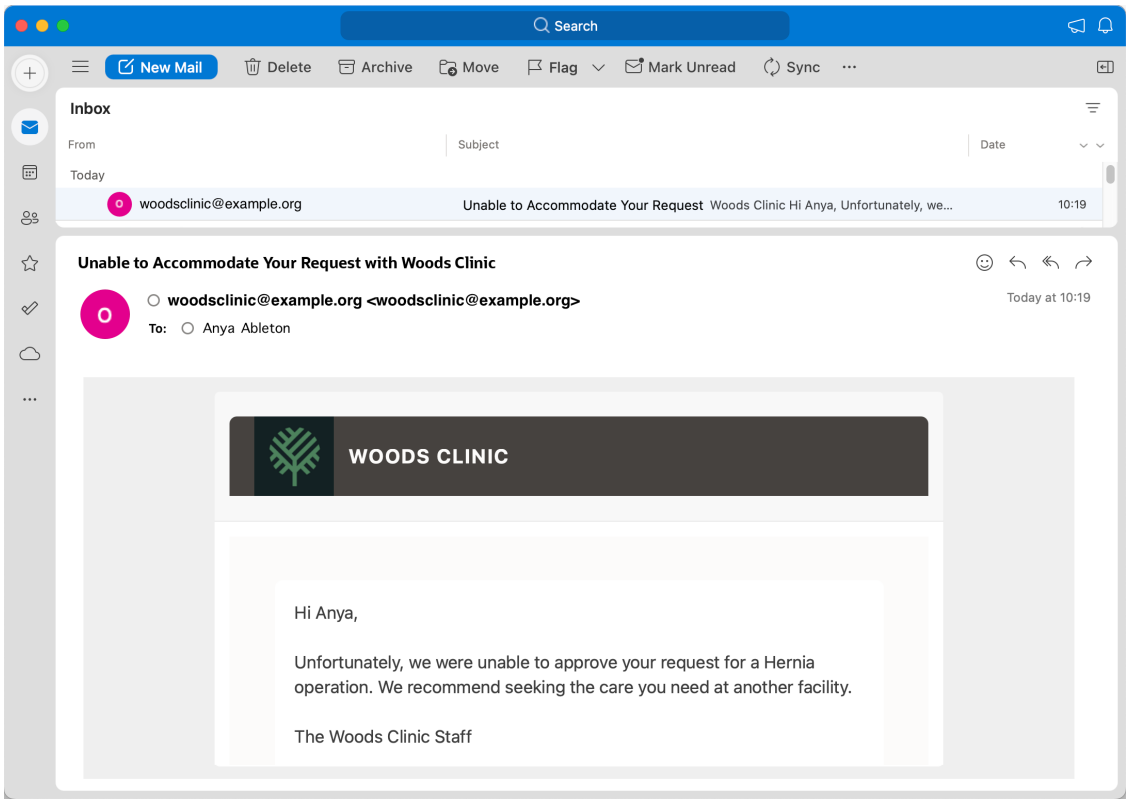
Receive an automatic rejection email when the workflow denies a patient request.

With no back office staff involvement, your *Patient Onboarding* workflow now automatically notifies Anya Ableton that she'll need to seek her procedure at another facility.

When Ms. Ableton's procedure request is rejected, the *Patient Onboarding* workflow automatically notifies her as shown below, with no back office staff involvement. The figure shows Anya's email client with the subject line *"Unable to Accommodate Your Request"* and a message explaining she'll need to seek the care she needs at another facility.

Copyright © 2026, Oracle and/or its affiliates.

Figure 25-99 Anya Ableton Rejection Email from Woods Clinic Patient Onboarding Workflow



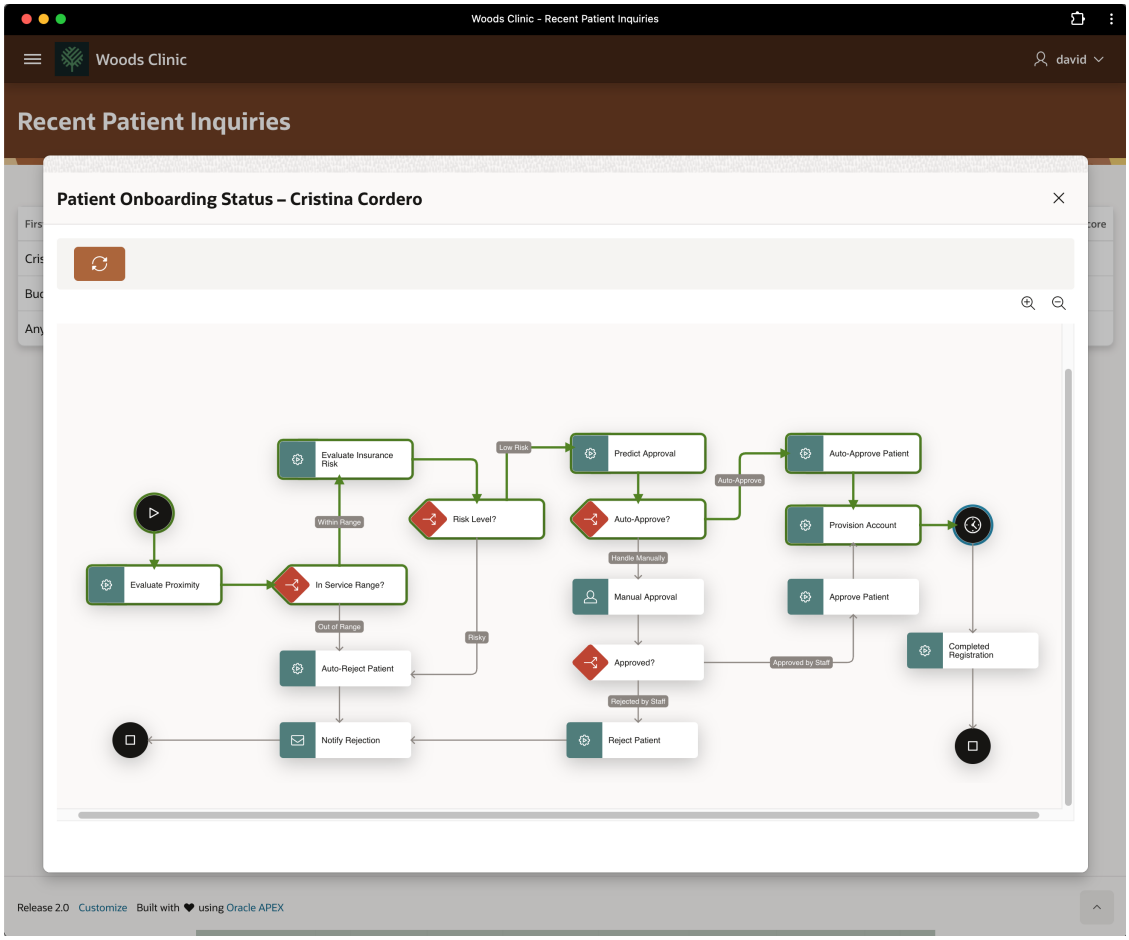
25.2.6.8 Provisioning a New User Account

Provision a portal user account when the workflow approves a patient request.

Consulting the *Recent Patient Inquiries* page again, as shown below, David can see that patient Cristina Cordero's request was automatically approved, her portal account got provisioned, and the workflow is waiting for her to complete her registration.

Copyright © 2026, Oracle and/or its affiliates.

Figure 25-100 Cristina Cordero's Request Was Auto-Approved and Is Now Waiting



- Invoking the Provision Account API
  Invoke a provisioning API to create the approved patient's account and assign its role.
- Inserting Account Creation Request
  Insert an account creation request for a background job to process outside the workflow session.
- Scheduling Create New Account Job
  Schedule the account creation job from SQL Developer Web so it runs as the workspace schema.
- Handling New Account Requests
  Process queued account requests, create users, and email portal credentials from PL/SQL.
- Notifying User of Temporary Password
  Email the approved patient their portal username and temporary password from PL/SQL.
- Receiving New Account Email

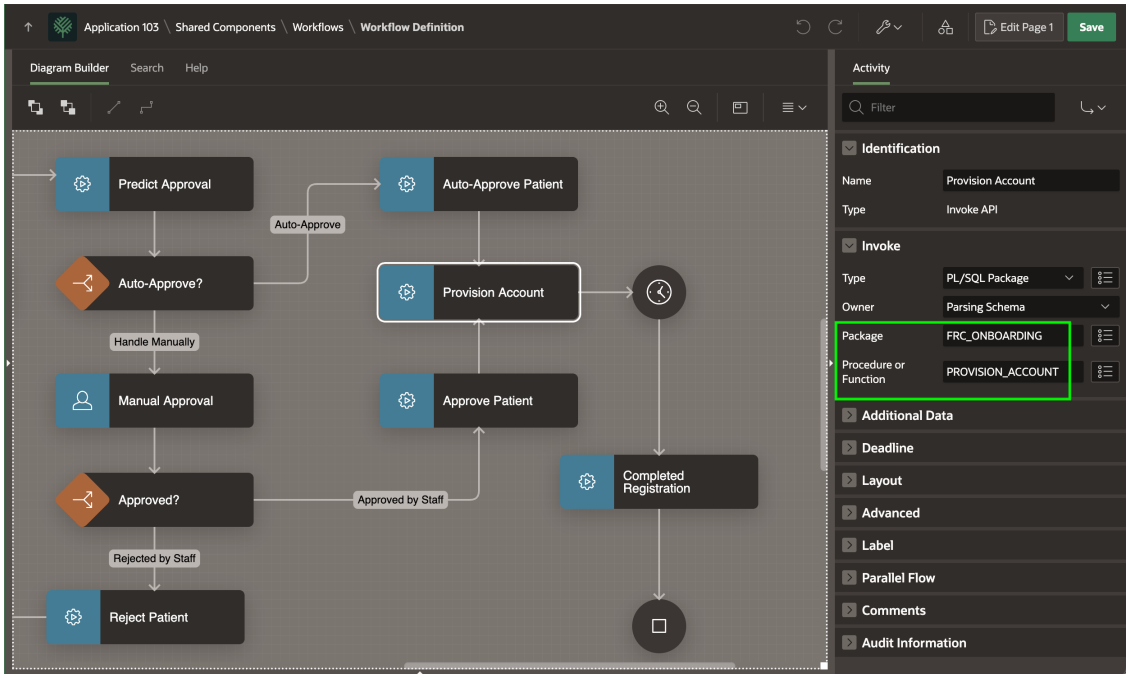
25.2.6.8.1 Invoking the Provision Account API

Invoke a provisioning API to create the approved patient's account and assign its role.

Copyright © 2026, Oracle and/or its affiliates.

As shown below, your *Patient Onboarding* workflow's *Provision Account* activity is an *Invoke API* that calls the `PROVISION_ACCOUNT` procedure.

Figure 25-101 Provision Account Invoke API Activity Creates User Account



The `PROVISION_ACCOUNT` routine is below. It looks up the patient information using the patient ID passed in, then calls a `CREATE_USER` procedure in the `FRC_APEX_USER` package to create the APEX user account. Finally, it calls `ADD_USER_ROLE` in the `APEX_ACL` package to add the new account's username to the Access Control List (ACL) role with *Static ID* `PATIENTS`. The *Woods Clinic* application serves both back office staff and patients. It uses Authorization Schemes based on ACL role membership to automatically show each user appropriate pages and functionality based on their role.

```
procedure provision_account(
    p_patient_id number)
is
    l_patient t_patient    := patient(p_patient_id);
    l_clinic_app_id number := frc_app.clinic_app_id();
begin
    frc_apex_user.create_user(
        p_username   => l_patient.username,
        p_email      => l_patient.email,
        p_patient_id => p_patient_id
    );
    apex_acl.add_user_role(
        p_application_id => l_clinic_app_id,
        p_user_name      => l_patient.username,
        p_role_static_id => 'PATIENTS'
    );
end;
```


Copyright © 2026, Oracle and/or its affiliates.

25.2.6.8.2 Inserting Account Creation Request

Insert an account creation request for a background job to process outside the workflow session.

To create an APEX user, you call the `CREATE_USER` procedure in the `APEX_UTIL` package. However, since this must be run from context *outside* of an APEX application session, the `CREATE_USER` function in the *Woods Clinic* package uses a two-step approach. Here, it simply inserts a row in a table to track the *request* to create a new user account. Then, a `DBMS_SCHEDULER` background job periodically processes the table to actually create the accounts.

```
-- package frc_apex_user
procedure create_user(
    p_username   in varchar2,
    p_email      in varchar2,
    p_patient_id in number)
is
    l_group_id number := apex_util.get_group_id(p_group_name=>'Patients');
begin
    insert into frc_apex_users_queue(
       username,
       group_id,
       email,
       patient_id)
    values (
       upper(p_username),
       l_group_id,
       p_email,
       p_patient_id);
end;
```

25.2.6.8.3 Scheduling Create New Account Job

Schedule the account creation job from SQL Developer Web so it runs as the workspace schema.

Scheduling a background job is simple, but if you run the one-line PL/SQL block below inside in the *SQL Commands* or *SQL Scripts* pages of SQL Workshop, DBMS Scheduler infers a `JOB_CREATOR` of `APEX_PUBLIC_USER`. That generic schema does not have privileges to call `APEX_UTIL.CREATE_USER`.

Instead, use *SQL Workshop* > *SQL Developer Web* if your APEX instance administrator has enabled this integration. This menu option opens *SQL Developer Web* in a new browser tab, automatically logged in as the workspace schema user `WOODS` (the *Woods Clinic* schema). Then, from the *SQL Worksheet* in *SQL Developer Web*, the DBMS Scheduler correctly infers the `WOODS` schema as the Job Creator. Notice that the `JOB_ACTION` parameter passes the PL/SQL block to handle the new users queue, and the `REPEAT_INTERVAL` indicates to run every minute.

```
--
-- NOTA BENE: Run this outside of App Builder so that the
-- ~~~~~~~~~  JOB_CREATOR ends up correctly reflecting WOODS
--            instead of the APEX_PUBLIC_USER proxy user.
--
```


Copyright © 2026, Oracle and/or its affiliates.

```
BEGIN
    DBMS_SCHEDULER.CREATE_JOB (
      job_name            => 'APEX_CREATE_USER_JOB',
      job_type            => 'PLSQL_BLOCK',
      job_action          => 'begin frc_apex_user.handle_new_users_queue;
end;',
      number_of_arguments => 0,
      repeat_interval     => 'FREQ=MINUTELY',
      end_date            => NULL,
      enabled             => TRUE,
      auto_drop           => TRUE,
      comments            => ''
      );
END;
```

25.2.6.8.4 Handling New Account Requests

Process queued account requests, create users, and email portal credentials from PL/SQL.

A `DBMS_SCHEDULER` job calls the procedure below every minute to check the new account creation request table. It loops over any rows in that table, generates a temporary password, creates the new APEX end-user account as part of the `Patients` group that was recorded in the request table. Then it notifies the new user of their patient portal credentials by calling a `SEND_APPROVAL_EMAIL` in the `FRC_ONBOARDING` package. This email *could* have been sent using another *Send E-Mail* workflow activity, but doing it from PL/SQL is an opportunity to see how to send mail programmatically.

> **Tip**
>
> The procedure uses `EXECUTE IMMEDIATE` in this example to avoid a circular dependency between the `FRC_APEX_USER` package containing `HANDLER_NEW_USERS_QUEUE` and the `FRC_ONBOARDING` package containing the rest of the patient onboarding application logic.

```
-- package frc_apex_user
procedure handle_new_users_queue is
    l_temp_password varchar2(255);
begin
    -- set workspace context
    apex_util.set_workspace('WOODS');
    for x in (select * from frc_apex_users_queue)
    loop
        -- Generate temporary password
        l_temp_password := generate_password();
        -- create the apex user in the supplied group if present
        apex_util.create_user
        (
            p_user_name     => x.username,
            p_web_password  => l_temp_password,
            p_email_address => x.email,
            p_group_ids     => x.group_id
        );
        execute immediate q'~
```


Copyright © 2026, Oracle and/or its affiliates.

```
            begin
               frc_onboarding.send_approval_email(
                  :PATIENT_ID,
                  :TEMP_PASSWORD);
            end;
          ~' using x.patient_id,
                   l_temp_password;
          -- remove the user from the queue
          delete from frc_apex_users_queue where id = x.id;
    end loop;
end;
```

25.2.6.8.5 Notifying User of Temporary Password

Email the approved patient their portal username and temporary password from PL/SQL.

The final step of user account provisioning is notifying the user of their new username and temporary password. While you could send the email from the workflow, here you explore the programmatic way to send an email using `SEND_MAIL` in the `APEX_EMAIL` package. It's using an email template for the new account notification that uses a number of placeholder values. Notice that the `P_PLACEHOLDERS` argument expects to receive the placeholder name/value pairs as a JSON document. The local `APPROVAL_PLACEHOLDERS` helper function builds up that JSON document using PL/SQL's native `JSON_OBJECT_T` type and returns it as a CLOB in JSON format.

```
-- in package frc_onboarding
procedure send_approval_email(
    p_patient_id         in number,
    p_temporary_password in varchar2)
is
    l_patient   t_patient := patient(p_patient_id);
    ------------------------------------------------
    function approval_placeholders(
        p_patient            in t_patient,
        p_temporary_password in varchar2)
        return                  clob
    is
        l_json json_object_t := json_object_t;
        l_clinic_app_url varchar2(255);
    begin
        l_clinic_app_url := frc_app.url_for_page(
                                frc_app.clinic_app_id,'home');
        l_json.put('FIRST_NAME',
                   p_patient.first_name);
        l_json.put('PROCEDURE_DESCRIPTION',
                   p_patient.procedure_description);
        l_json.put('PORTAL_URL',
                   l_clinic_app_url);
        l_json.put('USERNAME',
                   lower(p_patient.username));
        l_json.put('TEMPORARY_PASSWORD',
                   p_temporary_password);
        return l_json.stringify();
    end;
begin
    apex_mail.send(
```

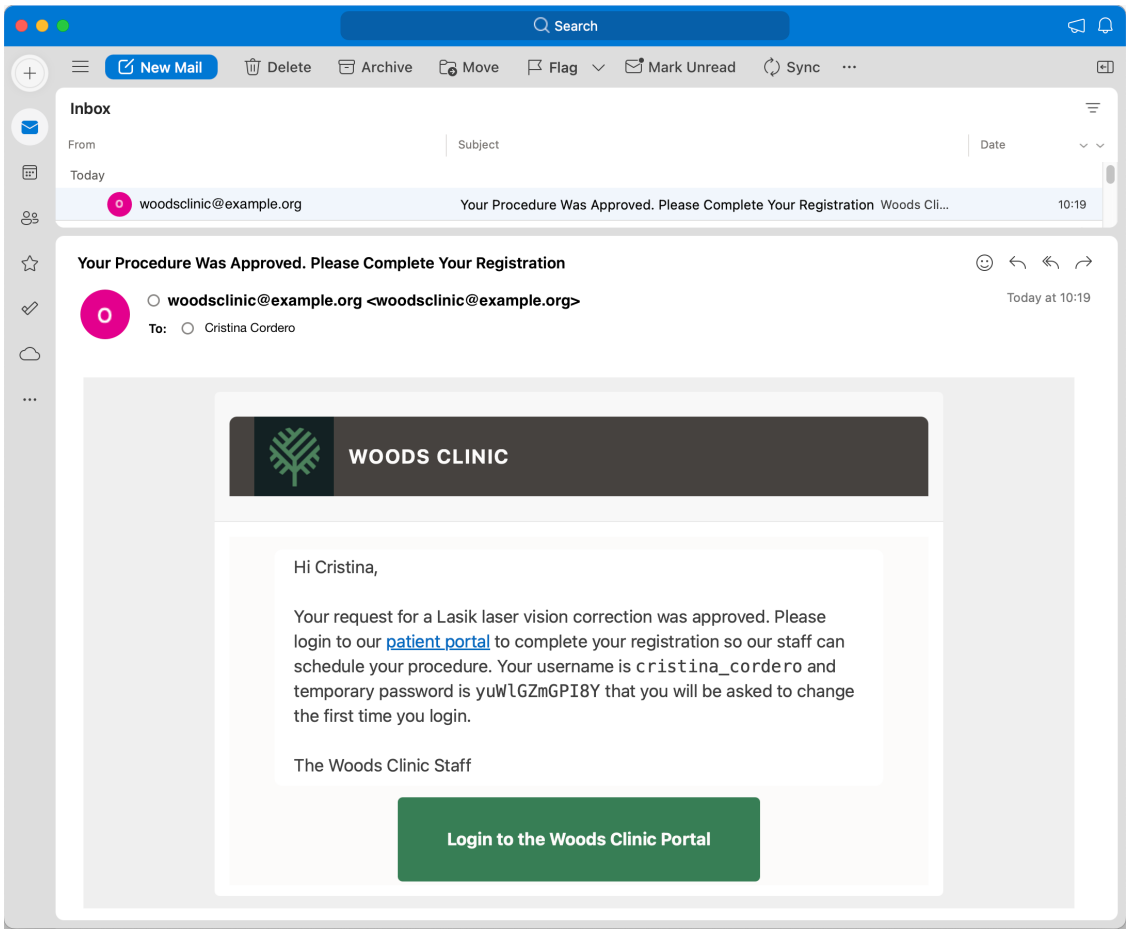

Copyright © 2026, Oracle and/or its affiliates.

```
            p_to                => l_patient.email,
            p_from              => frc_app.get_sender_email,
            p_template_static_id => frc_app.c_approved_email_template,
            p_placeholders      => approval_placeholders(
                                       l_patient,
                                       p_temporary_password),
            p_application_id    => frc_app.onboarding_app_id());
        -- Force mail to be sent immediately
        apex_mail.push_queue;
end;
```

25.2.6.8.6 Receiving New Account Email

As shown below, your *Patient Onboarding* workflow's *Provision Account* activity results in automatically sending Cristina Cordero the credentials for her new account.

Figure 25-102 Cristina Cordero Receives Her New Account Credentials



25.2.6.9 Waiting Until Patient Completes Profile

Wait for the approved patient to complete portal registration before the workflow continues.

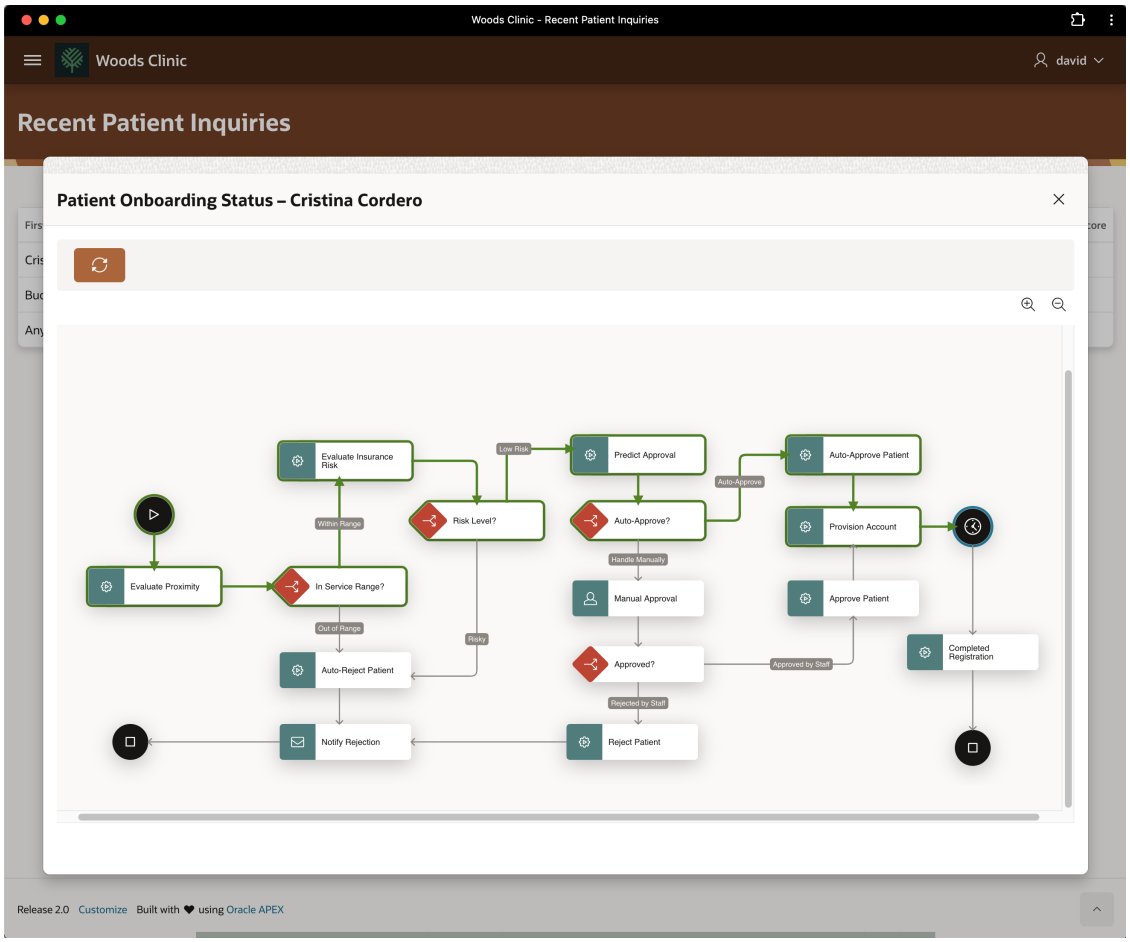
Copyright © 2026, Oracle and/or its affiliates.

As shown below, patient Cristina Cordero's request is waiting for her to provide her date of birth and proof of insurance on the portal. The diagram show the activity path the workflow took to get to this moment:

- *Evaluate Proximity* and *In Service Range?* determined she was within range
- *Evaluate Insurance Risk* and *Risk Level?* determined her insurance was low risk
- *Predict Approval* and *Auto-Approve?* approved her based on historical data
- *Provision Account* set up her account and mailed her initial credentials for registration.

Now it's clear in the diagram that the *Patient Onboarding* workflow for Cristina Cordero is waiting for her to complete her registration, before proceeding on to the last leg of the "journey": marking her registration as completed.

Figure 25-103 Cristina Cordero's Request Was Auto-Approved and Is Now Waiting



- [Waiting Indefinitely for a Custom Signal]
  Use a Wait activity without a timeout to pause until custom application logic continues it.
- [Completing the New Patient Profile]
  Complete portal registration by entering birth date and insurance details.
- [Signaling the Workflow to Continue]
  Save the patient's registration data and signal the waiting workflow to continue.

Copyright © 2026, Oracle and/or its affiliates.

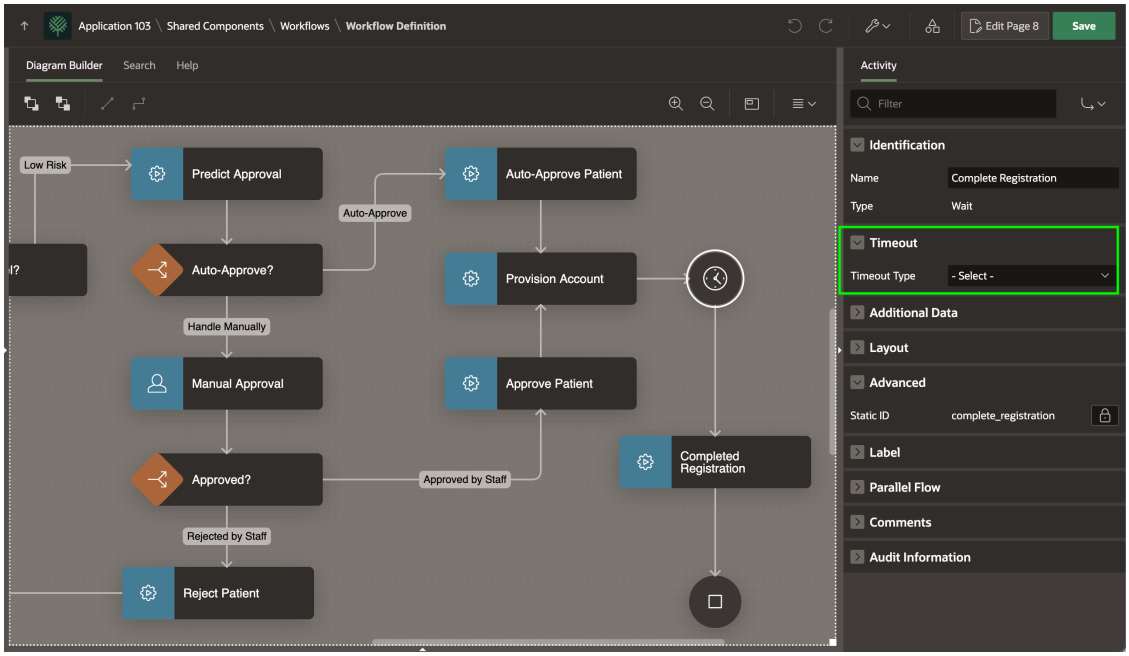
- Understanding Onboarding is Complete
  Confirm that completing registration updates the portal and finishes the onboarding workflow.

25.2.6.9.1 Waiting Indefinitely for a Custom Signal

Use a Wait activity without a timeout to pause until custom application logic continues it.

Your *Patient Onboarding* workflow uses a *Wait* activity to pause until the new patient completes their profile by providing a date of birth and proof of insurance scan. If a *Unified Task List* "inbox" is not the kind of user experience you want for your end users, you can use the combination of a *Wait* activity and a later call to `CONTINUE_ACTIVITY` in the `APEX_WORKFLOW` package to craft a custom user experience. Notice that by ***not*** choosing a *Timeout Type*, you opt to wait indefinitely until your own application logic in another part of the application signals the waiting activity should continue.

Figure 25-104 Configuring an Activity to Wait Indefinitely



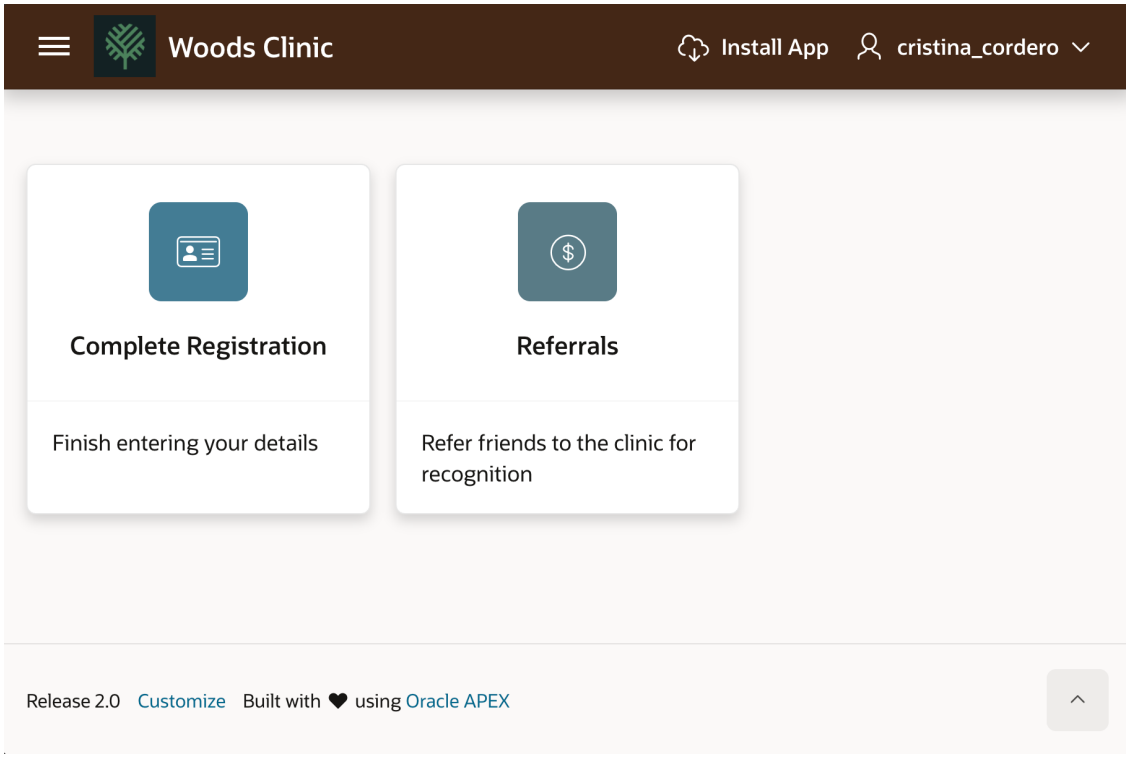
25.2.6.9.2 Completing the New Patient Profile

Complete portal registration by entering birth date and insurance details.

Cristina clicks the link in her notification email and logs into the portal for the first time. After changing her initial password, she sees tiles for functionality available to her. To complete her profile, she clicks on *Complete Registration*.

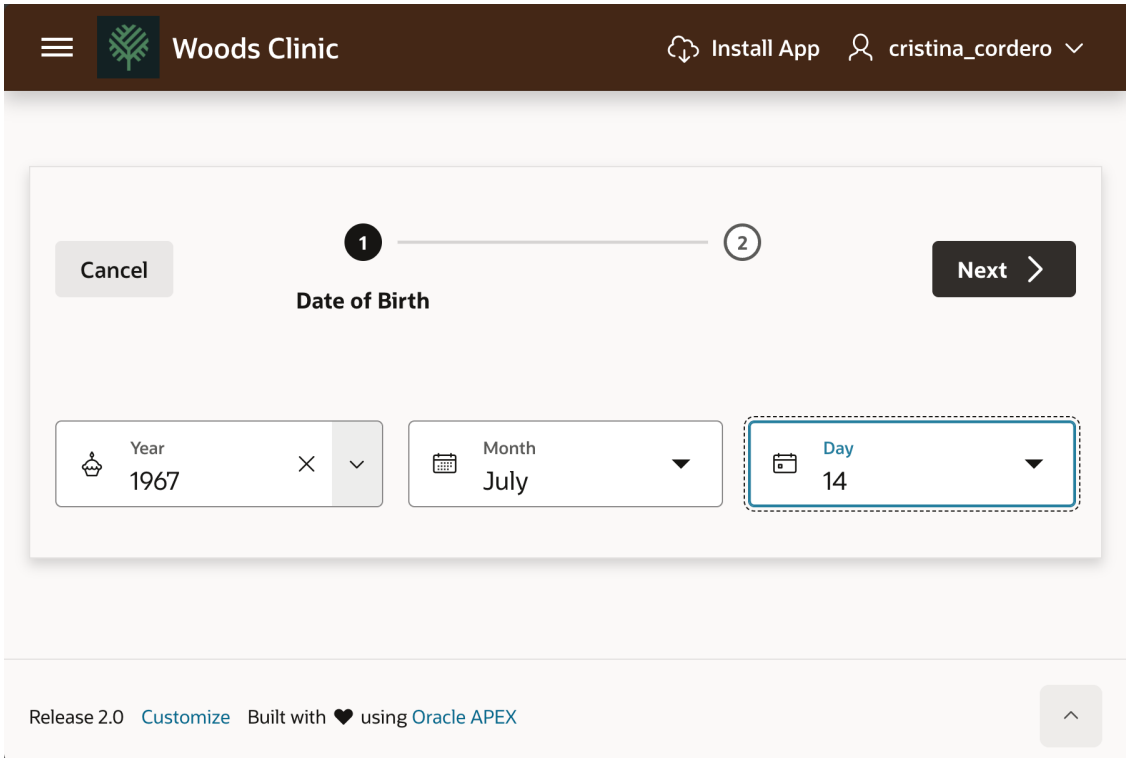
Copyright © 2026, Oracle and/or its affiliates.

Figure 25-105 Patient Portal for Cristina Cordero on First Login



As shown below, Cristina enters her date of birth on the first step of the *Complete Registration* wizard, then clicks *(Next >)*.

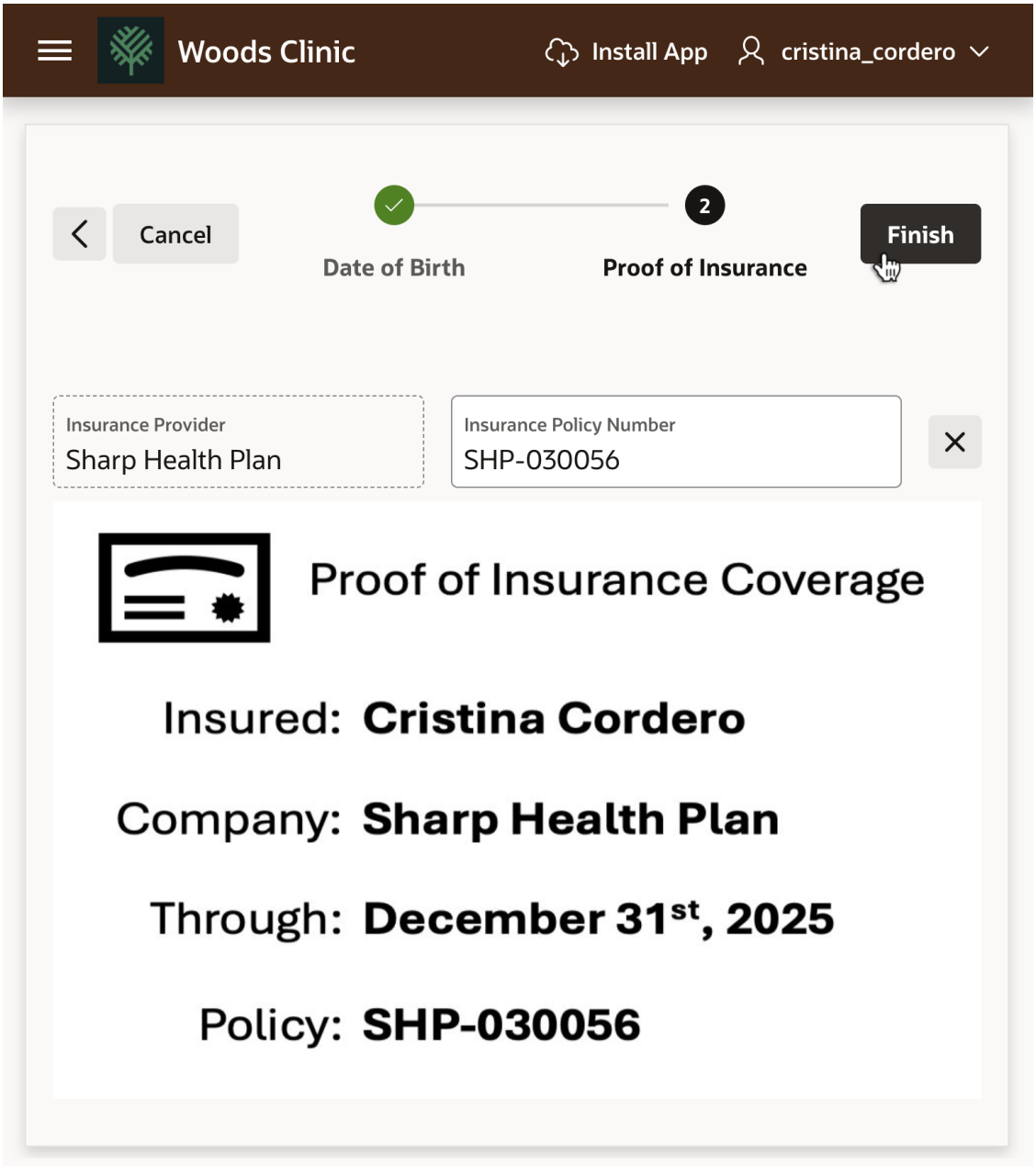
Figure 25-106 Cristina Cordero Completes Registration - Date of Birth



Copyright © 2026, Oracle and/or its affiliates.

On step 2 of the wizard, she uploads her proof of insurance and enters her insurance policy number, then clicks *(Finish)*.

Figure 25-107 Cristina Cordero Completes Registration - Proof of Insurance



25.2.6.9.3 Signaling the Workflow to Continue

Save the patient's registration data and signal the waiting workflow to continue.

The last page of the *Complete Registration* wizard uses an *Invoke API* page process to call the `COMPLETE_REGISTRATION` procedure below. Each data item the wizard collected gets passed in by declaratively configuring the parameter value expressions. The value of an *Image Upload* page item provides the unique file name in the `APEX_APPLICATION_TEMP_FILES` table where the APEX engine stores the uploaded file for application-specific processing.

Copyright © 2026, Oracle and/or its affiliates.

Notice that after retrieving the uploaded proof of insurance scan image blob from the temporary file table, it updates Cristina Cordero's `FRC_PATIENTS` row with date of birth and proof of insurance information. For fraud detection purposes, the wizard also uses a *Get Current Position* dynamic action step in a *Page Load* event handler to record the latitude and longitude of the user's position when uploading the proof of insurance.

After updating the patient row, the procedure calls `CONTINUE_ACTIVITY` in the `APEX_WORKFLOW` package to signal that Cristina Cordero's *Patient Onboarding* workflow should now proceed.

```
procedure complete_registration(
    p_patient_id              in number,
    p_year_of_birth           in number,
    p_month_of_birth          in number,
    p_day_of_birth            in number,
    p_insurance_policy_number in varchar2,
    p_scan_uploadname         in varchar2,
    p_scan_latitude           in number,
    p_scan_longitude          in number)
is
    l_scan_blob   blob;
    l_params apex_application_global.vc_map;
begin
    -- get uploaded insurance proof scan blob from temp storage
    for scan in (select blob_content
                   from apex_application_temp_files
                  where name = p_scan_upload_name)
    loop
        l_scan_blob := scan.blob_content;
    end loop;
    -- update patient to include insurance proof scan blob
    update frc_patients
    set insurance_proof_scan = l_scan_blob,
        date_of_birth = to_date(
                        to_char(p_year_of_birth)||
                        to_char(p_month_of_birth,'00')||
                        to_char(p_day_of_birth,'00'),
                        'YYYYMMDD'),
        insurance_policy_number   = p_insurance_policy_number,
        insurance_proof_latitude  = p_scan_latitude,
        insurance_proof_longitude = p_scan_longitude
    where id = p_patient_id;
    -- Continue onboarding workflow that's waiting for new patient
    -- registration to be complete. Use workflow id from the most
    -- recent onboarding workflow associated with this patient id
    for wf in (select workflow_id
                 from apex_workflows
                where workflow_def_static_id = 'patient_onboarding'
                  and detail_pk = to_char(p_patient_id)
                order by start_time desc
                fetch first row only)
    loop
        -- Continue the complete_registration "Wait" activity
        apex_workflow.continue_activity(
            p_instance_id     => wf.workflow_id,
            p_static_id       => 'complete_registration',
            p_activity_params => l_params);
```


Copyright © 2026, Oracle and/or its affiliates.

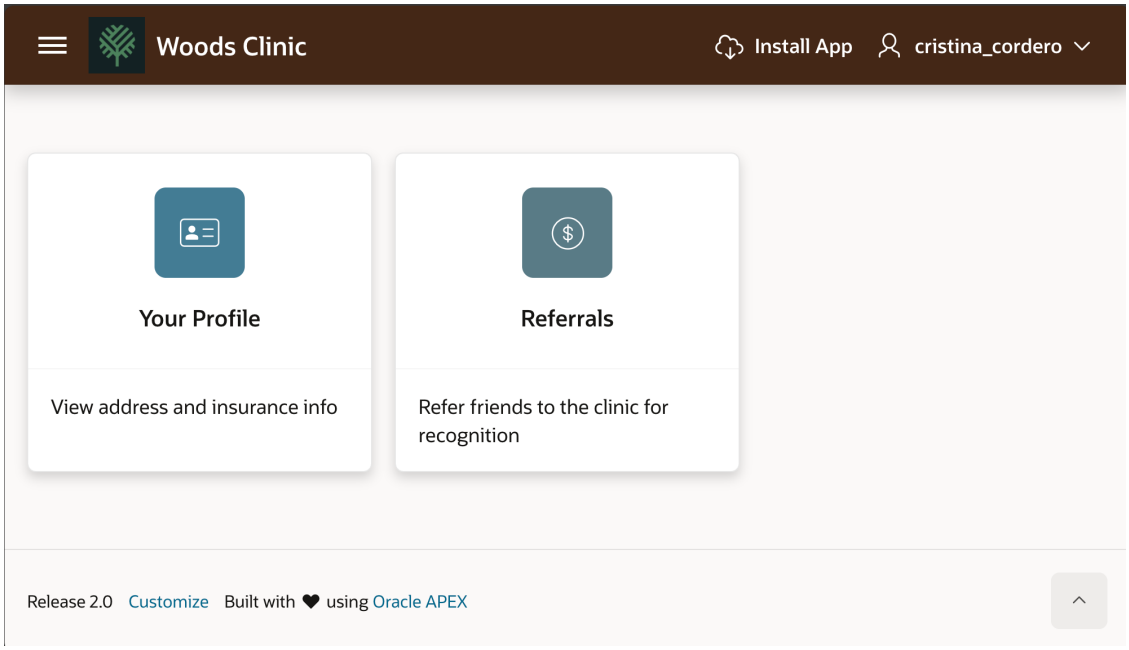
```
    end loop;
end;
```

25.2.6.9.4 Understanding Onboarding is Complete

Confirm that completing registration updates the portal and finishes the onboarding workflow.

After completing her profile, as shown below, Cristina Cordero sees the *Complete Registration* tile is replaced by a *Your Profile* tile. This confirms she provided all necessary information the clinic requires.

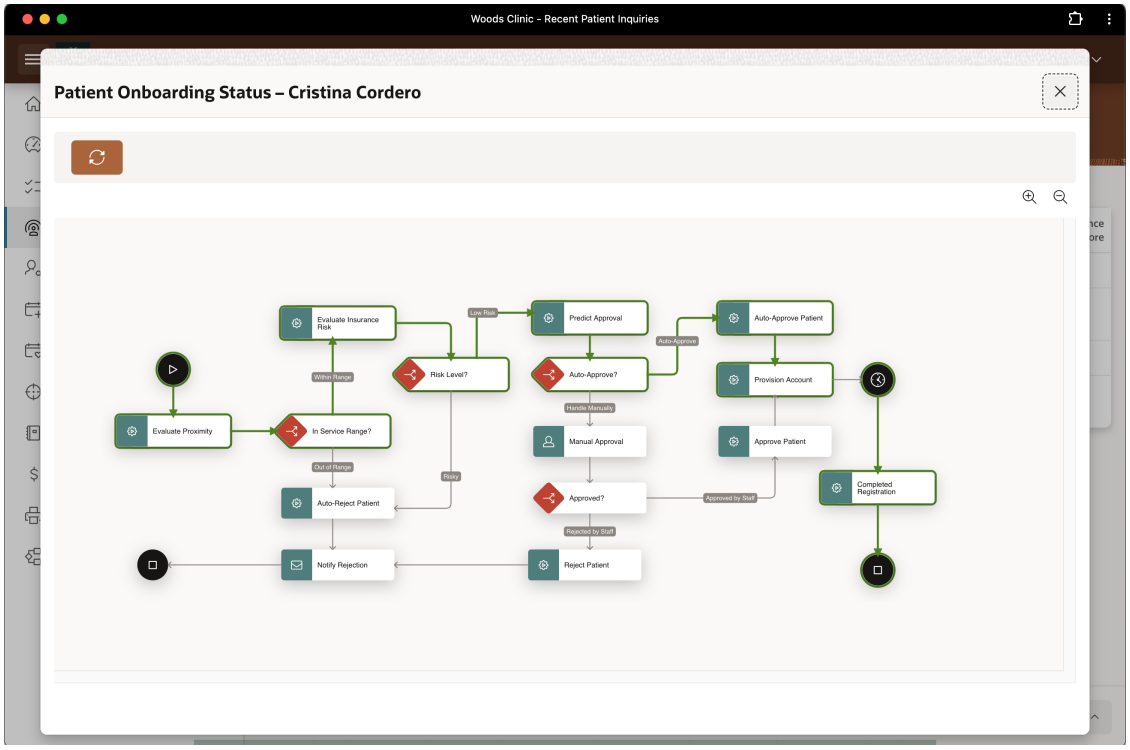
Figure 25-108 After Completing Registration, New Patients Portal Dashboard Options Change



In the back office, David confirms Cristina Cordero completed her *Patient Onboarding* workflow by glancing at the visual workflow diagram. As shown in the figure, now the *End* activity is highlighted on the path that ended by running the *Completed Registration* activity.

Copyright © 2026, Oracle and/or its affiliates.

Figure 25-109 Verifying Cristina Cordero's Patient Onboarding is Completed



25.3 Automating Business Entity Lifecycle Logic

Use a trigger and automation to wake workflow lifecycle logic on business entity status changes.

Your workflows can wait for data change events to implement custom business logic *visually* as a business entity row progresses through its natural "lifecycle".

- Learning the Medical Procedure Lifecycle
  Understand the patient procedure lifecycle tracked by the `STATUS` column.
- Seeing the Medical Procedure Lifecycle
  Visualize patient procedures by status to see how work flows through the clinic.
- Modeling the Medical Procedure Lifecycle
  Model status-driven business logic with a workflow that waits for procedure status or schedule changes.
- Starting the Lifecycle Workflow
  Start a procedure lifecycle workflow when onboarding creates the patient procedure row.
- Following a Medical Procedure Lifecycle
  Follow a patient procedure through its full lifecycle in the Woods Clinic app.
- Notifying Workflow of Data Changes
  Notify a waiting workflow about data changes with events an automation processes periodically.

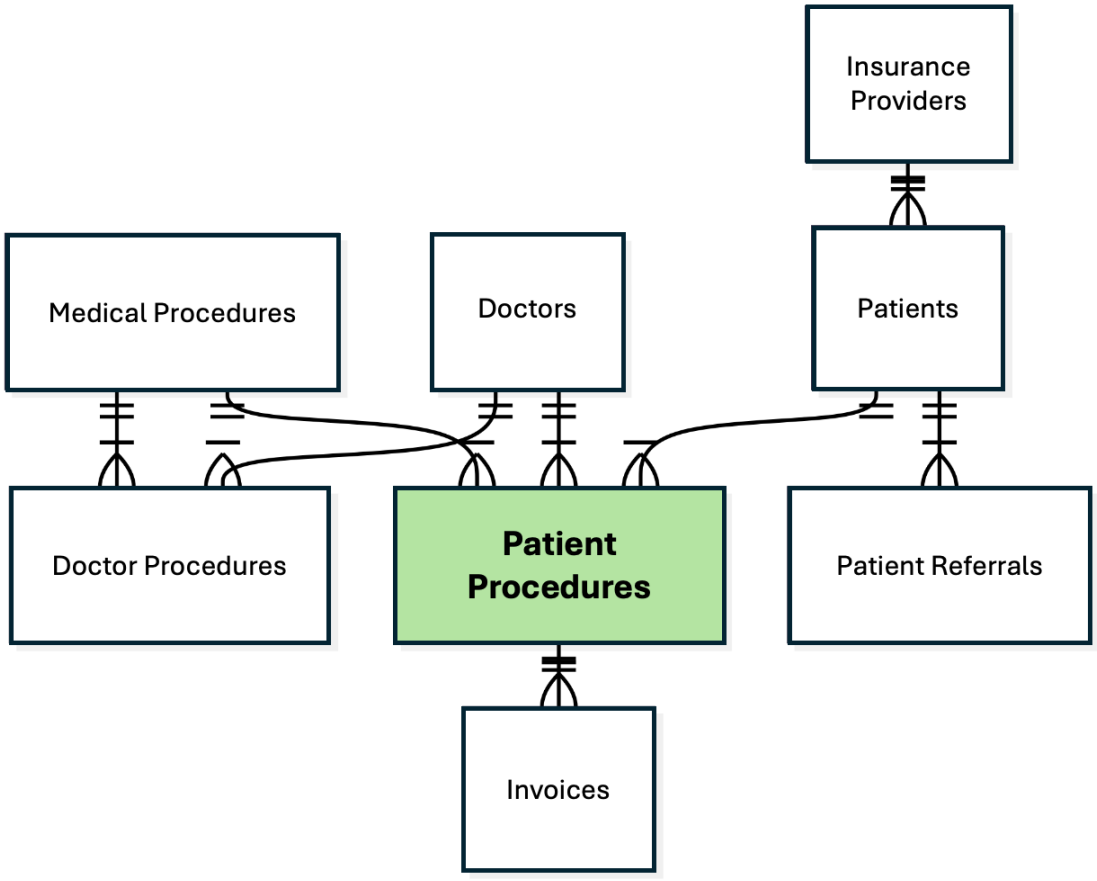
Copyright © 2026, Oracle and/or its affiliates.

25.3.1 Learning the Medical Procedure Lifecycle

Understand the patient procedure lifecycle tracked by the `STATUS` column.

In your *Woods Clinic* application, once a patient's initial procedure request is approved, you track the medical procedure for the patient as a new row in the *Patient Procedures* table. You can see below how it fits into the application data model. The figure shows an entity relationship diagram of the business entities involved in the Woods Clinic application. This scenario focuses on the *Patient Procedure*. It tracks a particular *Patient*'s visit at which a selected clinic *Doctor* performs a certain *Medical Procedure* they need. Each patient has an *Insurance Provider* and can make *Patient Referrals* to help bring in new patients. Multiple doctors might perform a given procedure, so *Doctor Procedures* tracks that skills matrix. Finally, a patient procedure might have one or more *Invoices* to bill the insurance provider and the patient who pays the uncovered amount.

Figure 25-110 Patient Procedures Table in the Woods Clinic Data Model



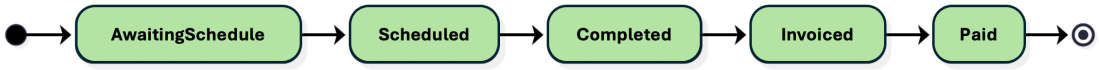
Your `FRC_PATIENT_PROCEDURES` table has a `STATUS` column whose value reflects the current status of the "Patient Procedure" business entity.

After speaking with *Woods Clinic* staff, you draw yourself the diagram below of a medical procedure's natural flow from creation to closure. You think of it as the "lifecycle" of this business entity. You learn the following:

Copyright © 2026, Oracle and/or its affiliates.

- The procedure starts out "*Awaiting Schedule*".
- When staff set a date, time, and doctor for the procedure, it becomes *Scheduled*.
- Once performed, the procedure becomes *Completed*
- After receiving the insurance reimbursement, the balance is *Invoiced* to the patient.
- As soon as the patient pays, it's finally marked *Paid*.

Figure 25-111 Patient's Medical Procedure Statuses Define its Lifecycle



25.3.2 Seeing the Medical Procedure Lifecycle

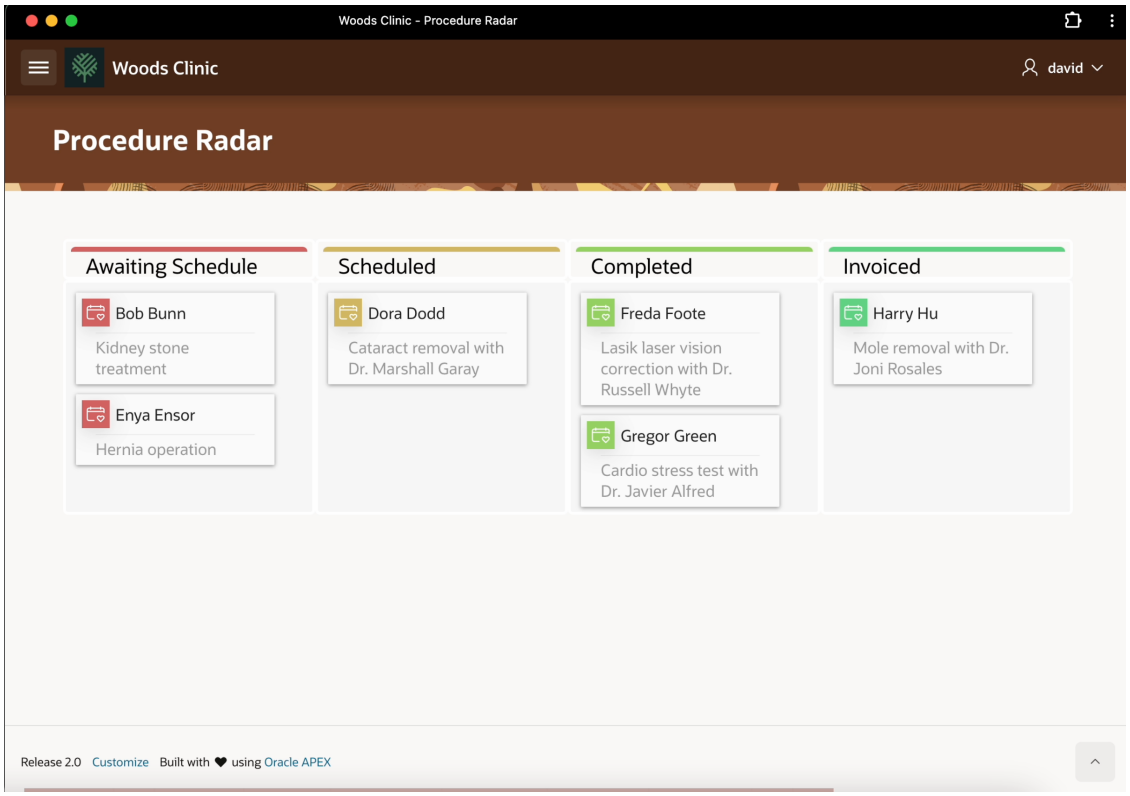
Visualize patient procedures by status to see how work flows through the clinic.

The medical procedure's lifecycle is not only for internal tracking purposes. As shown below, your *Procedure Radar* page uses the status value to help all staff understand how medical procedures are "flowing" through the clinic. The Kanban board shows:

- **Awaiting Schedule**
  - *Bob Bunn (Kidney stone treatment)*
  - *Enya Ensor (Hernia operation)*
- **Scheduled**
  - *Dora Dodd (Cataract removal with Dr. Marshall Garay)*
- **Completed**
  - *Freda Foote (Lasik laser vision correction with Dr. Russel Whyte)*
  - *Gregor Green (Cardio stress test with Dr. Javier Alfred)*
- **Invoiced**
  - Harry Hu (Mole removal with Dr. Joni Rosales)

Copyright © 2026, Oracle and/or its affiliates.

Figure 25-112 Kanban Board Shows Users How Procedures Progress Through Their Lifecycle



25.3.3 Modeling the Medical Procedure Lifecycle

Model status-driven business logic with a workflow that waits for procedure status or schedule changes.

The *Woods Clinic* business process for patient procedures closely aligns with these statuses. More specifically, it involves the steps back office staff take when a procedure *changes* statuses. From talking to *Woods Clinic* staff, you learn when a patient's medical procedure is:

- *Scheduled* – staff send an email to the patient
- *Completed* – staff prepare a draft invoice to review with the insurance provider
- *Invoiced* – staff email the invoice to the client for payment
- *Paid* – staff consider the procedure's lifecycle complete.

You can use a workflow with a *Wait* activity as shown below to automate this medical procedure lifecycle. Notice the *Wait* activity has the *Static ID* unique identifier of `status_or_date_changed` and that it has left the *Timeout Type* unset. This makes the workflow wait indefinitely until your application logic signals it to continue.

At that time, it can populate a workflow variable `V_WHAT_CHANGED` with the value:

- `STATUS` – if the medical procedure status has changed, or
- `SCHEDULE` – if its schedule has changed.

The *Switch* activities can use the value of the `V_WHAT_CHANGED` variable and *Additional Data* columns like `STATUS`, `DOCTOR`, and `PROCEDURE_DATE` of the associated `FRC_MEDICAL_PROCEDURES`

Copyright © 2026, Oracle and/or its affiliates.

row to perform appropriate business logic. Most of the paths in the workflow connect back to the *Wait* activity to await further data changes. However, when `STATUS` reflects the final "`Paid`" status, the workflow ends.

Figure 25-113 Procedure Lifecycle Workflow Coordinates Application Logic



Using this approach, you can automate the previously manual steps in the medical procedure lifecycle so that now, when a medical procedure's status becomes:

- *Scheduled* – your workflow emails the patient
- *Completed* – your workflow prepares the draft invoice as a pixel-perfect PDF document
- *Invoiced* – your workflow emails the invoice as an attachment to the client for payment
- *Paid* – your workflow marks the procedure as complete.

25.3.4 Starting the Lifecycle Workflow

Start a procedure lifecycle workflow when onboarding creates the patient procedure row.

The final step of the `patient_onboarding` workflow uses an *Invoke API* activity to call the `COMPLETED_REGISTRATION` procedure below in the `FRC_ONBOARDING` package. It inserts a row in `FRC_PATIENT_PROCEDURES` to track the patient's medical procedure to perform at the clinic. Just after that, it calls the `START_WORKFLOW_FOR` helper procedure to create the `procedure_lifecycle` workflow instance related to the newly-assigned patient procedure ID.

```
-- in frc_onboarding package
procedure completed_registration(p_patient_id number)
is
    l_patient t_patient := patient(p_patient_id);
    l_new_procedure_id number;
begin
    -- insert initial procedure
    for j in (select l_patient.id as patient_id,
                     code,
                     'AwaitingSchedule' as status,
                     price,
                     duration_hours
```


Copyright © 2026, Oracle and/or its affiliates.

```
                 from frc_medical_procedures
                where code = l_patient.initial_procedure)
    loop
        insert into frc_patient_procedures(
            patient_id,
            medical_procedure,
            status,
            amount_due,
            duration_hours)
        values (
            j.patient_id,
            j.code,
            j.status,
            j.price,
            j.duration_hours)
        return id into l_new_procedure_id;
        start_workflow_for(l_new_procedure_id);
    end loop;
end;
```

The `START_WORKFLOW_FOR` helper method starts a workflow with *Static ID* `procedure_lifecycle`, passing in the medical procedure ID for the *Details Primary Key* value. It calls the `START_WORKFLOW` function in the `APEX_WORKFLOW` package.

```
-- internal in package frc_procedure_lifecycle
-------------------------------------------------------------------
-- Start a procedure_lifecycle workflow instance associated with
-- the supplied patient procedure id
-------------------------------------------------------------------
procedure start_workflow_for(
    p_procedure_id in number)
is
    l_workflow_id number;
begin
   l_workflow_id := apex_workflow.start_workflow(
                        p_application_id => frc_app.clinic_app_id,
                        p_static_id      => 'procedure_lifecycle',
                        p_detail_pk      => p_procedure_id,
                        p_initiator      => 'ADMIN');
end start_workflow_for;
```

Tip

When you don't plan to use a function's result, to avoid a PL/SQL compilation error, assign it to a local variable that you then ignore. Failing to do this in the above example would yield the error:

```
PLS-00221: 'START_WORKFLOW' is not a procedure or is undefined
```


Copyright © 2026, Oracle and/or its affiliates.

25.3.5 Following a Medical Procedure Lifecycle

Follow a patient procedure through its full lifecycle in the Woods Clinic app.

Woods Clinic staff member *David* approved patient Danny Deng's medical procedure. Danny got his confirmation email with login credentials from your `patient_onboarding` workflow. He logged in and completed his registration. This concluded the patient onboarding workflow, whose last step is to create the patient procedure row and its corresponding `procedure_lifecycle` workflow. The steps that follow illustrate how David takes Danny's medical procedure through its lifecycle at the clinic.

- Launching Woods Clinic Application
  Launch the Woods Clinic application like a native app with persistent authentication.
- Scheduling a Medical Procedure
  Schedule a medical procedure and let the workflow notify the patient.
- Receiving Medical Procedure Schedule
  Receive the scheduled procedure details by email.
- Discharging Patient After Procedure
  Complete a patient procedure and generate draft discharge instructions with AI.
- Preparing Pixel-Perfect PDF Invoice
  Generate a pixel-perfect PDF invoice in the background with a report query and layout.
- Reviewing Invoice with Insurance Provider
  Review the prepared invoice and enter the insurance coverage amount.
- Regenerating Invoice in Background
  Regenerate an invoice in the background while preventing duplicate runs for the same procedure.
- Invoicing Patient for the Balance
  Mark the invoice complete so the workflow emails it to the patient and advances the procedure lifecycle.
- Acknowledging Receipt of Payment
  Record the patient's payment so the procedure reaches its final status.
- Confirming the Lifecycle is Complete
  Confirm the paid procedure completed its workflow and full business lifecycle.

25.3.5.1 Launching Woods Clinic Application

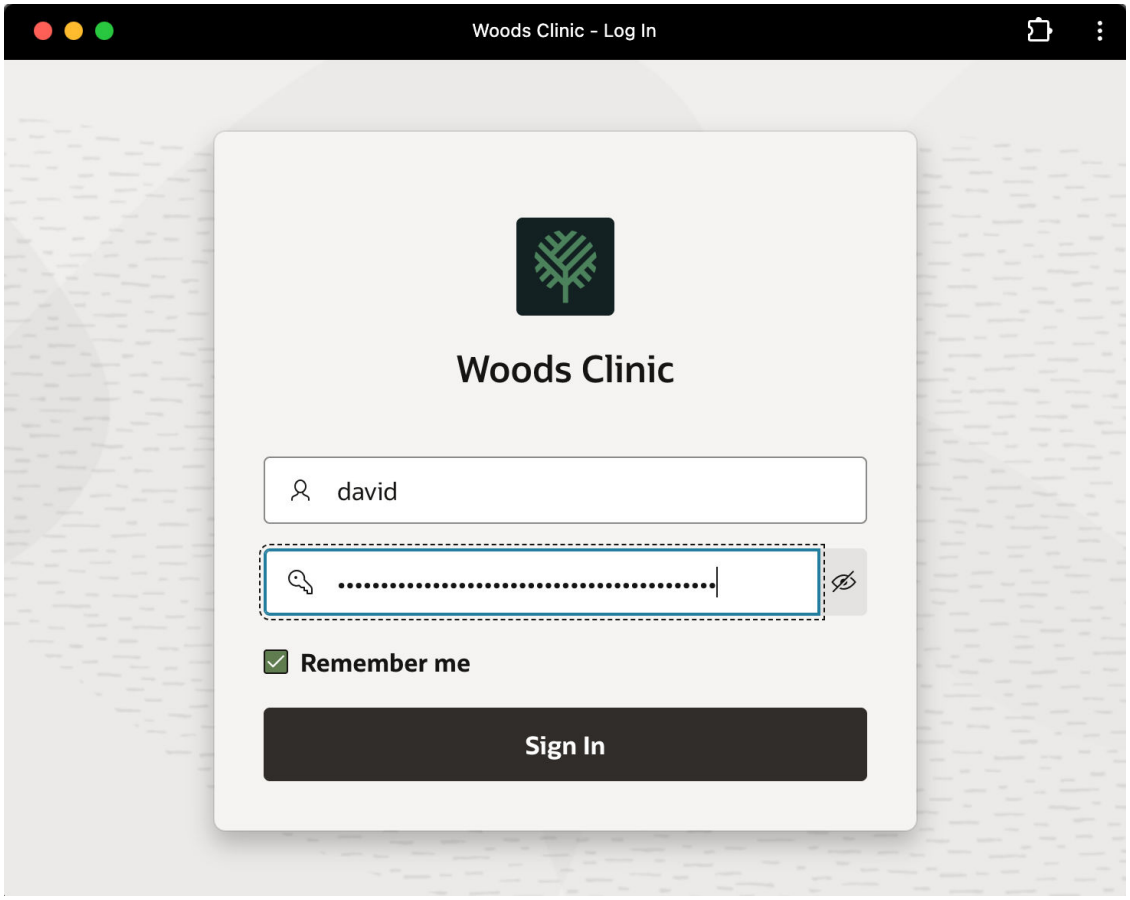
Launch the Woods Clinic application like a native app with persistent authentication.

As shown below, earlier in the week when David logged into the *Woods Clinic* application, he checked the *Remember me* checkbox on the login page. He avoids having to type in his password every time he launches the app for the next 30 days.

> **Tip**
>
> You enable this feature in APEX instance administration in the *Manage Instance* > *Security* settings. Set *Allow Persistent Auth* to *Yes* and adjust the *Persistent Authentication Lifetime Days* setting from the default 30 days if needed.

Copyright © 2026, Oracle and/or its affiliates.

Figure 25-114 On David's Last Login, He Ticked the Remember Me Checkbox



As shown below, today David starts his day by launching the *Woods Clinic* app from the dock on his iMac. You enable the *Progressive Web App* (PWA) installable switch in your application definition, so with one click David added the app to his desktop to launch like other native Mac apps. He can do the same on his iPad and iPhone to launch the app with a tap on his home screen. When combined with the persistent authentication feature, this lets end-users immediately get to work just like they do with other apps they use everyday like Outlook, Slack, and more.

Figure 25-115 David Launches Woods Clinic APEX App from His Dock



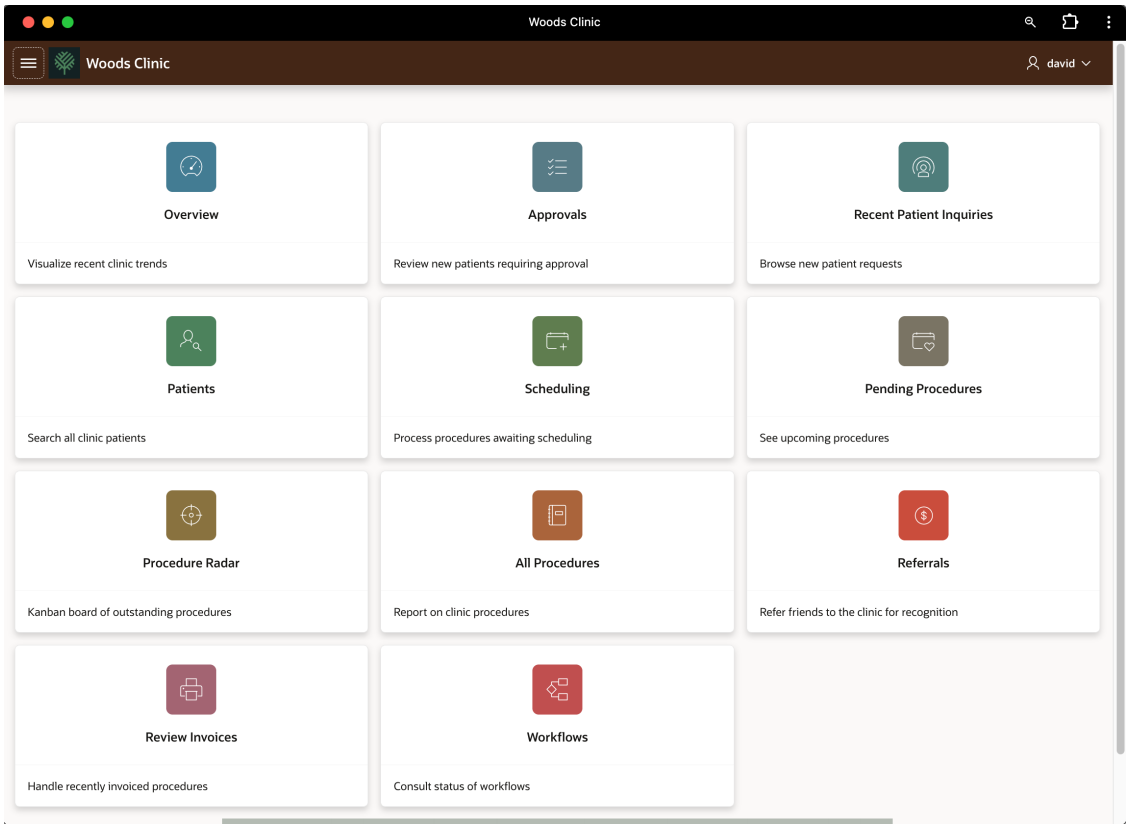
Your *Woods Clinic* app opens in its own window like a native desktop application. As shown below, a PWA application has no browser address bar. Using a *List* region associated with your

Copyright © 2026, Oracle and/or its affiliates.

Navigation Menu shared component, the home page shows a tile for each screen David has access to as clinic staff member.

> **Tip**
>
> Using built-in access control list security enforcement, the same home page shows a totally different set of tiles when a *patient* like Danny Deng logs in.

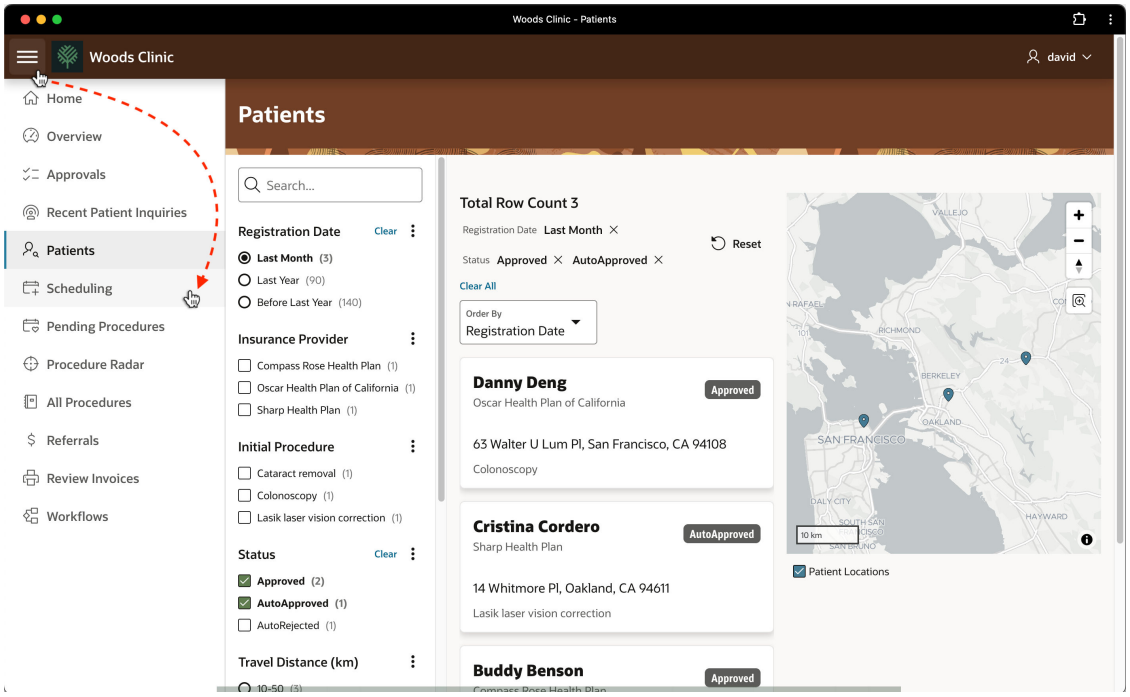
Figure 25-116 Home Page for Woods Clinic App Launched as a Native App



From any screen in the app, David can click the "hamburger" icon with three-horizontal lines in the upper left corner. This pops open the collapsible side menu. As shown below, he uses it to navigate to the *Scheduling* page to see if any medical procedures await him there.

Copyright © 2026, Oracle and/or its affiliates.

Figure 25-117 Navigating to Scheduling Page from a Different Screen



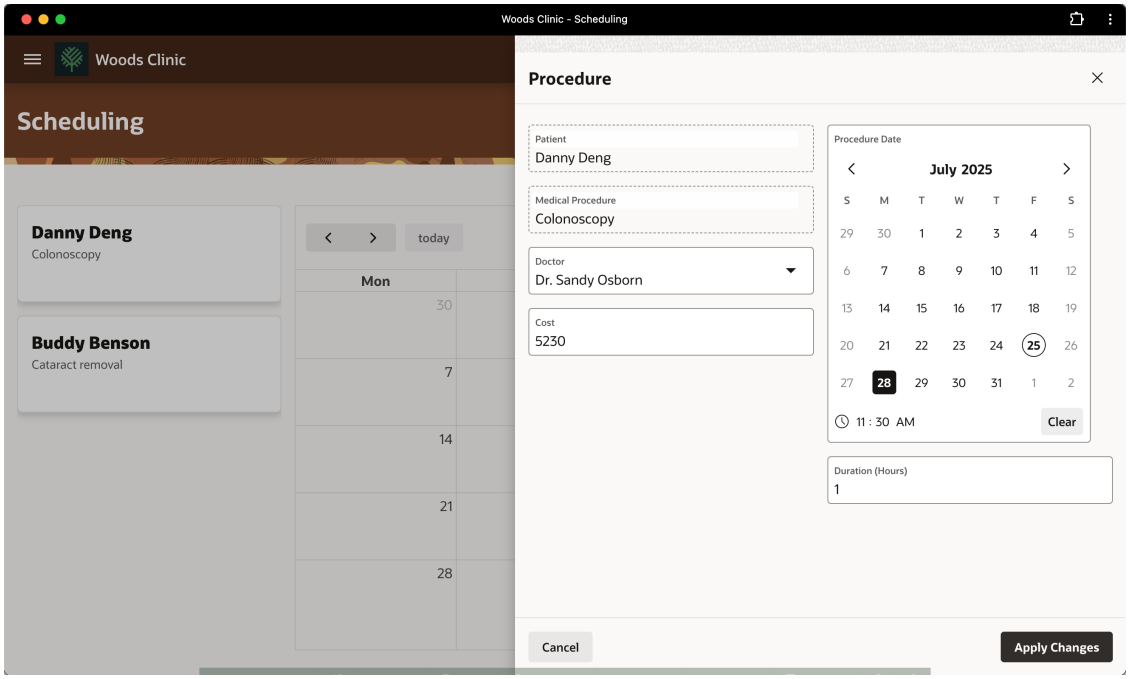
25.3.5.2 Scheduling a Medical Procedure

Schedule a medical procedure and let the workflow notify the patient.

David visits the *Scheduling* screen and sees that Danny Deng's medical procedure needs a date and doctor assigned. As shown below, he clicks on Danny's card to open his procedure details and schedules the exam for July 28th at 11:30am with Dr. Sandy Osborn.

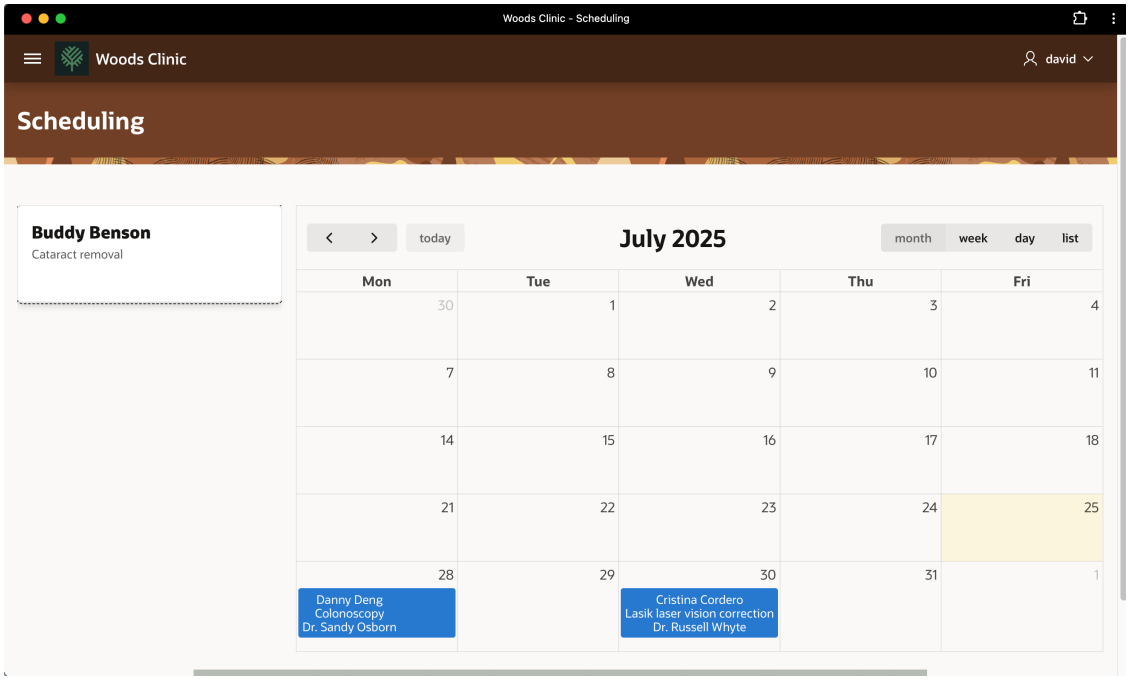
Copyright © 2026, Oracle and/or its affiliates.

Figure 25-118 David Schedules Danny Deng's Procedure



When David confirms the scheduling details, as shown below, Danny's procedure appears on the calendar and no longer shows in the pending list.

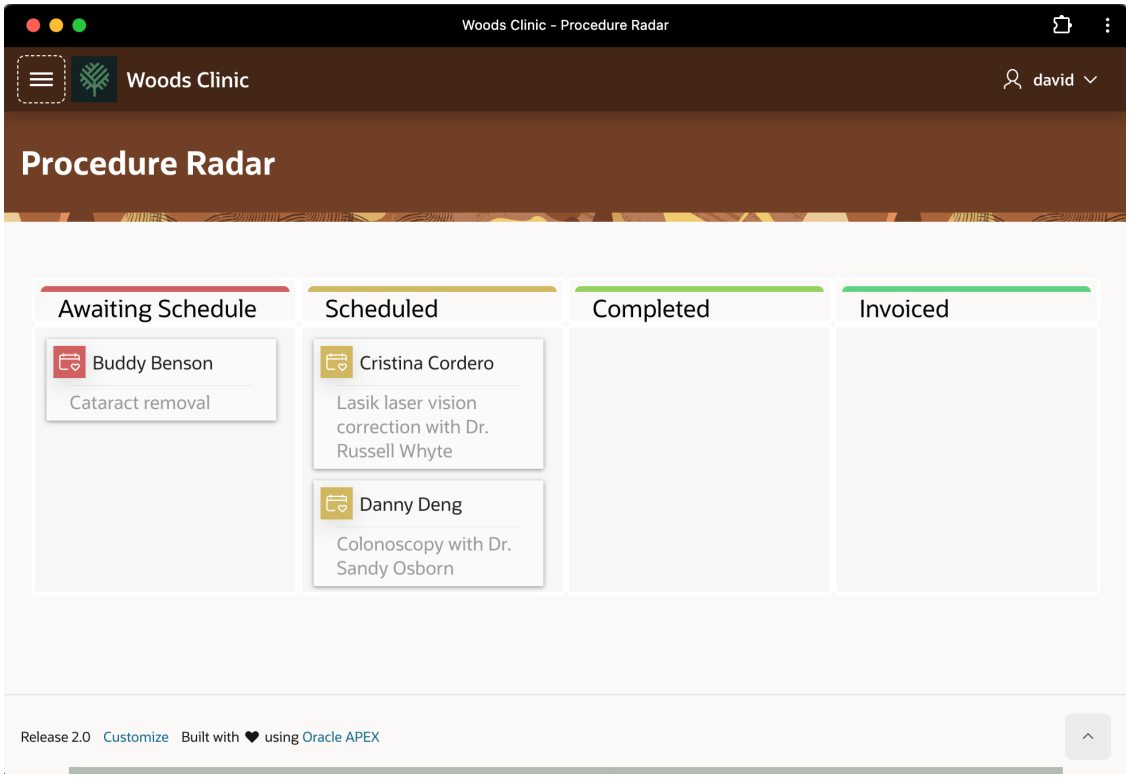
Figure 25-119 Scheduled Medical Procedures Appear on the Calendar



Checking the *Procedure Radar* page, as shown below, David now sees Danny Deng's procedure in the *Scheduled* column on the Kanban board.

Copyright © 2026, Oracle and/or its affiliates.

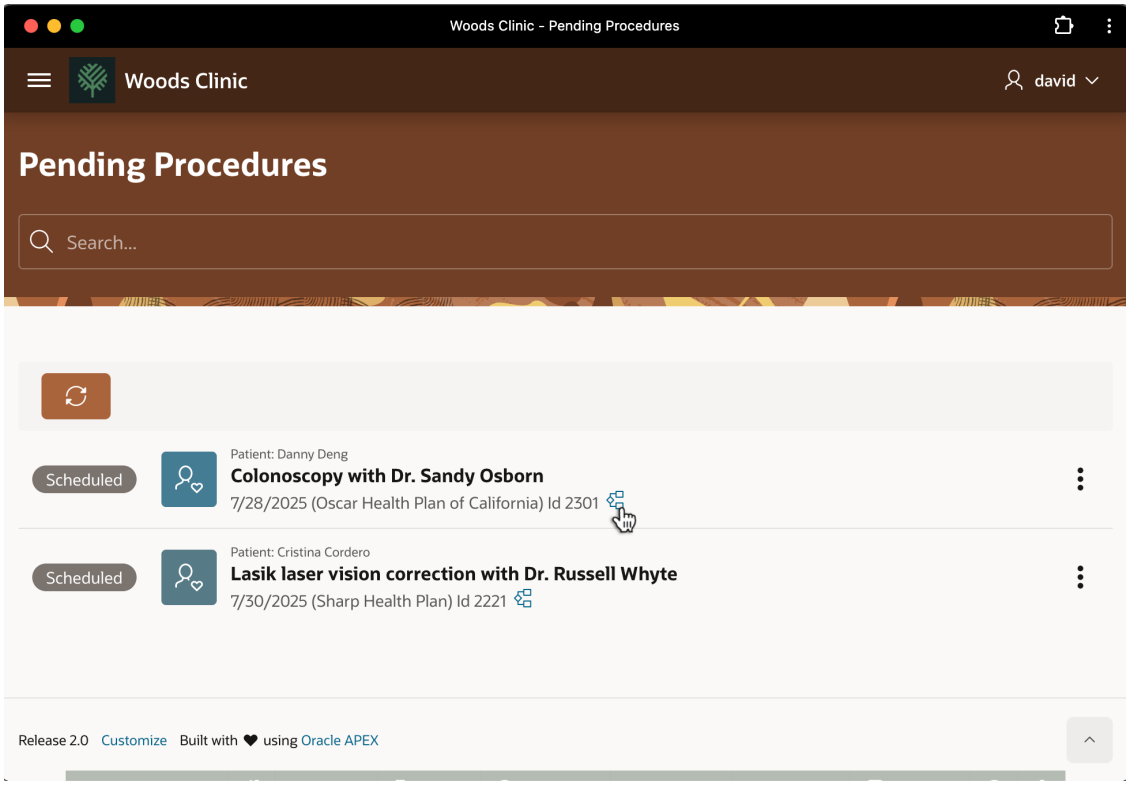
Figure 25-120 Office Staff See Danny Deng's Procedure Scheduled on Kanban Board



In the background, your *Procedure Lifecycle* workflow reacts to the change in status to "`Scheduled`". It automates notifying Danny Deng of the details. Later in the morning, when David visits the *Pending Procedures* page, as shown below, he clicks on the workflow icon next to *Danny's* procedure in the list.

Copyright © 2026, Oracle and/or its affiliates.

Figure 25-121 Checking a Workflow from Pending Procedures List

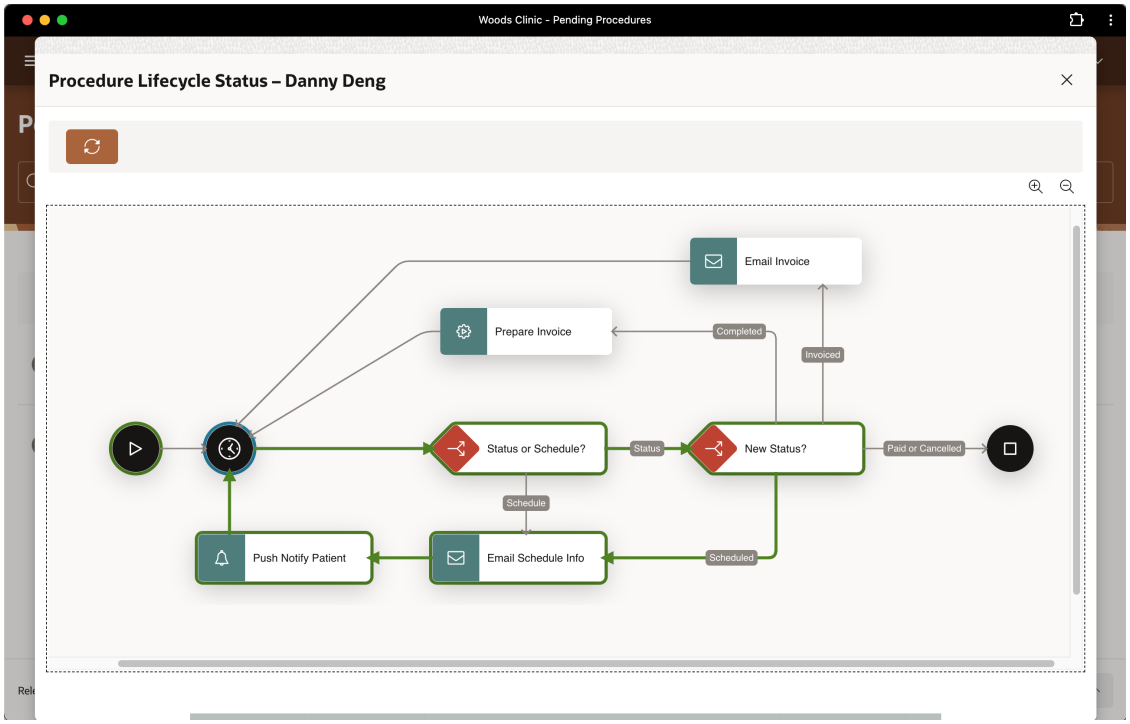


The *Procedure Lifecycle Status* page for Danny Deng appears. It gives David a progress report at a glance. As shown below, he observes colored highlights on the workflow diagram that confirm it sent the patient email and push notifications.

> **Tip**
>
> APEX only sends the push notification if Danny Deng opted-in to receiving these mobile notifications after installing the *Woods Clinic* app on his smartphone, tablet, or desktop computer. If he hasn't explicitly allowed them, APEX silently skips your workflow's attempt to deliver the message.

Copyright © 2026, Oracle and/or its affiliates.

Figure 25-122 Confirming Danny Deng Was Notified of Appointment



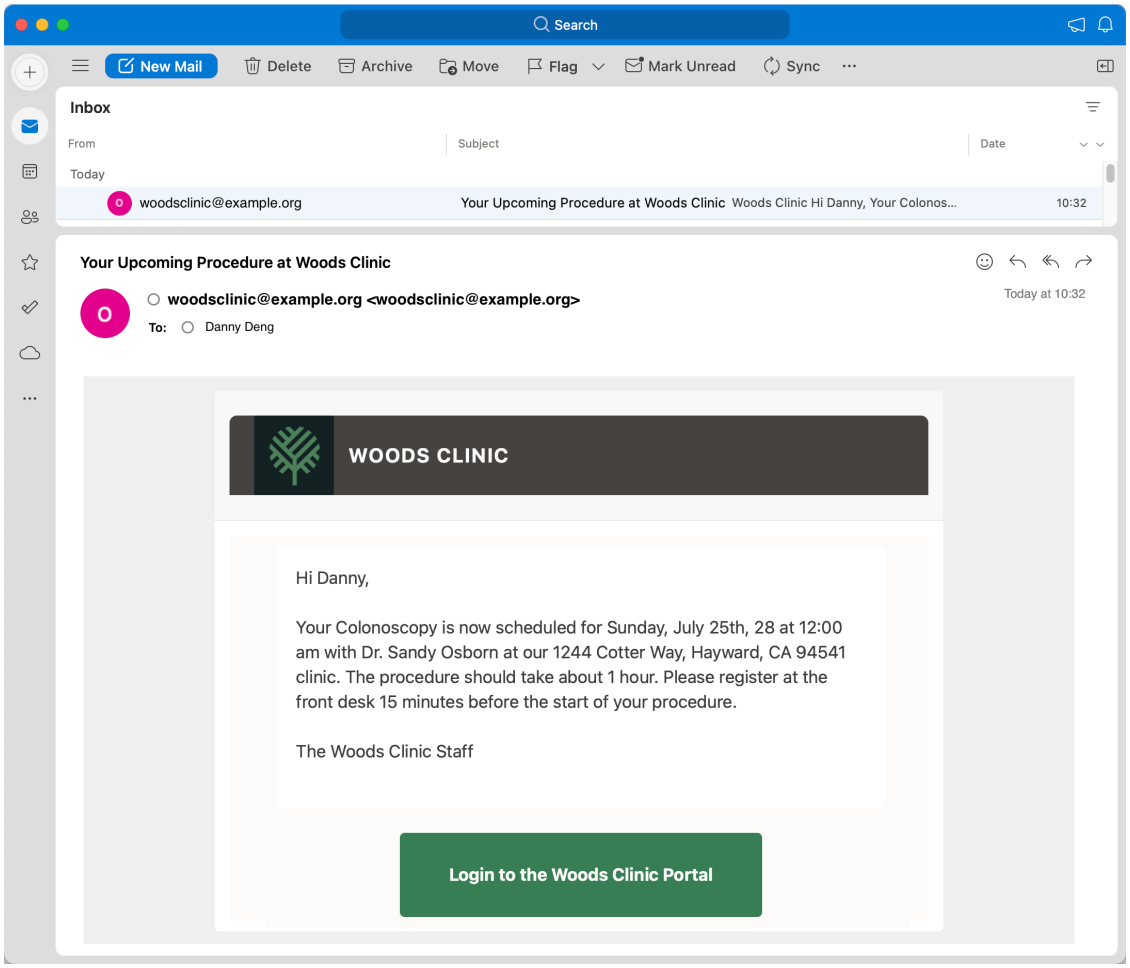
25.3.5.3 Receiving Medical Procedure Schedule

Receive the scheduled procedure details by email.

Danny receives the email shown below and he makes a note of the date and time. As the message advises, he'll plan to show up 15 minutes early to register on the day.

Copyright © 2026, Oracle and/or its affiliates.

Figure 25-123 Danny Deng Gets Email with His Appointment Information



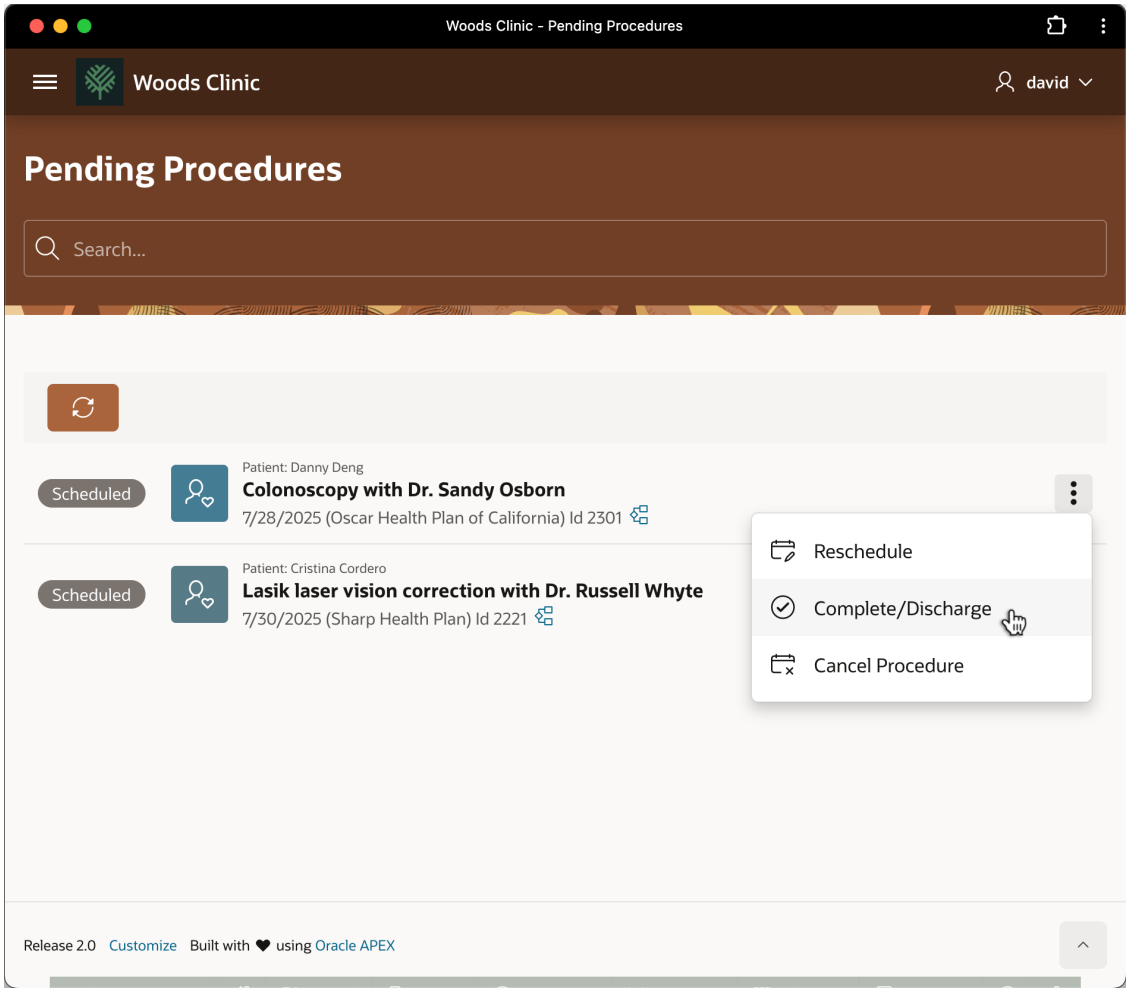
25.3.5.4 Discharging Patient After Procedure

Complete a patient procedure and generate draft discharge instructions with AI.

On the day of the appointment, when the procedure is over, Danny stops at the front desk before leaving. As shown below, David uses the three-vertical-dots dropdown menu on Danny's procedure record to open the *Discharge* page.

Copyright © 2026, Oracle and/or its affiliates.

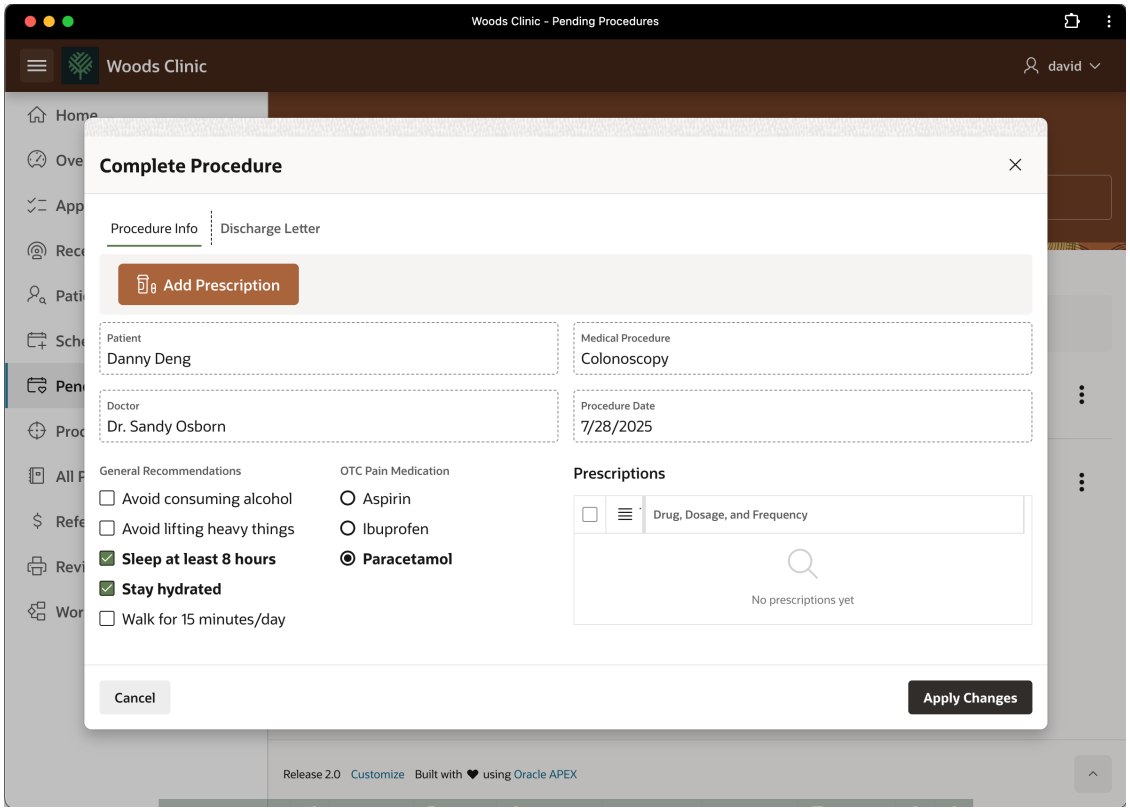
Figure 25-124 Discharging the Patient and Completing the Procedure



To send Danny on his way with some advice, on the *Procedure Info* tab, David ticks some common recommendations like sleeping at least 8 hours and to stay hydrated. He recommends Paracetamol if Danny experiences any discomfort.

Copyright © 2026, Oracle and/or its affiliates.

Figure 25-125 Entering Patient Discharge Instructions

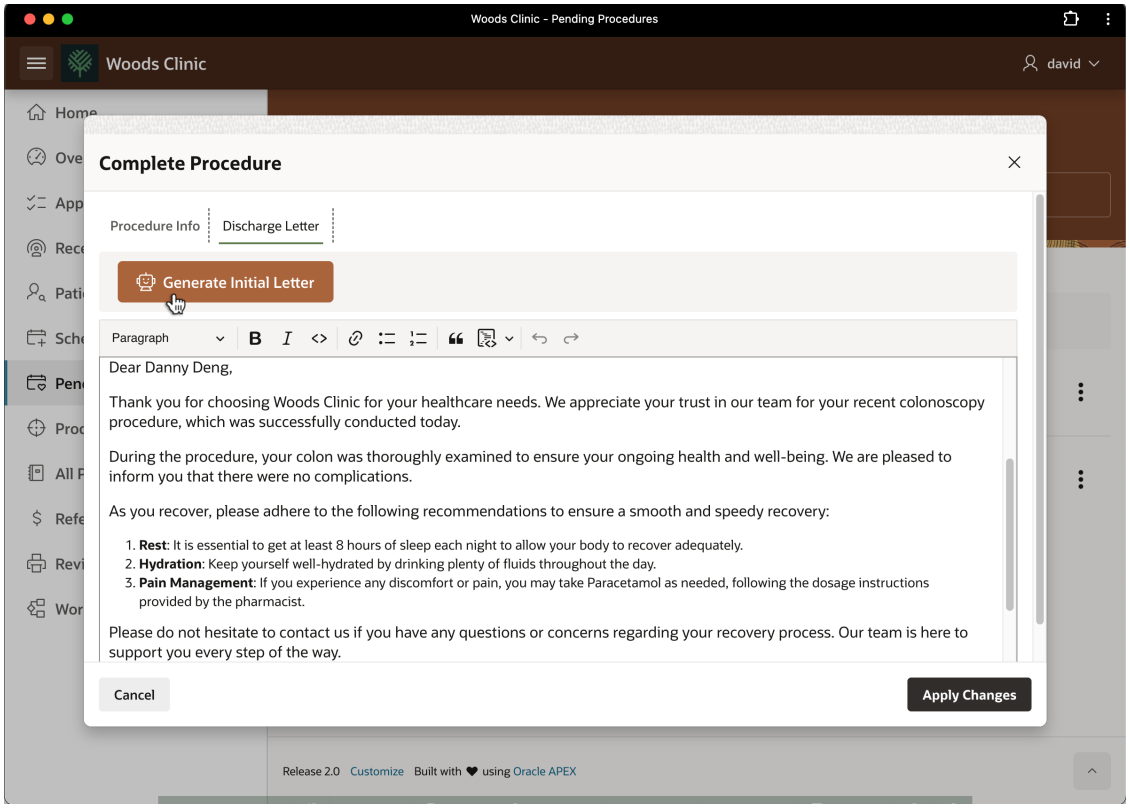


On the *Discharge Letter* tab, David clicks the *(Generate Initial Letter)* button to get a head start on writing Danny's discharge letter. David adjusts the initial letter as needed, prints out a copy for Danny, and clicks *(Apply Changes)* to complete the discharge step.

> **Tip**
>
> Your page's *(Generate Initial Letter)* button uses a dynamic action event handler on the button click to run a native *Generate Text with AI* dynamic action step. Its related AI Configuration contains the system prompt and RAG source to use a Generative AI Service to prepare a draft of the discharge letter with Danny's procedure information and David's discharge recommendations included. This productivity idea has proven to be a big time saver and is a real hit with staff.

Copyright © 2026, Oracle and/or its affiliates.

Figure 25-126 Preparing Discharge Letter Using Generative AI



25.3.5.5 Preparing Pixel-Perfect PDF Invoice

Generate a pixel-perfect PDF invoice in the background with a report query and layout.

By using a report query and report layout, your *Procedure Lifecycle* workflow engages a remote printing service to generate a pixel-perfect invoice document on-demand, in the background.

- Printing Documents from a Workflow
  Print documents from a workflow by combining a Report Query and Report Layout.
- Retrieving Invoice Data
  Define a Report Query that supplies invoice data as JSON for the report layout.
- Formatting the Invoice in a Layout
  Create a Word report layout that includes invoice data using special tags.
- Generating the Data-Driven Document
  Generate a PDF invoice and store it for review, download, and email delivery.

25.3.5.5.1 Printing Documents from a Workflow

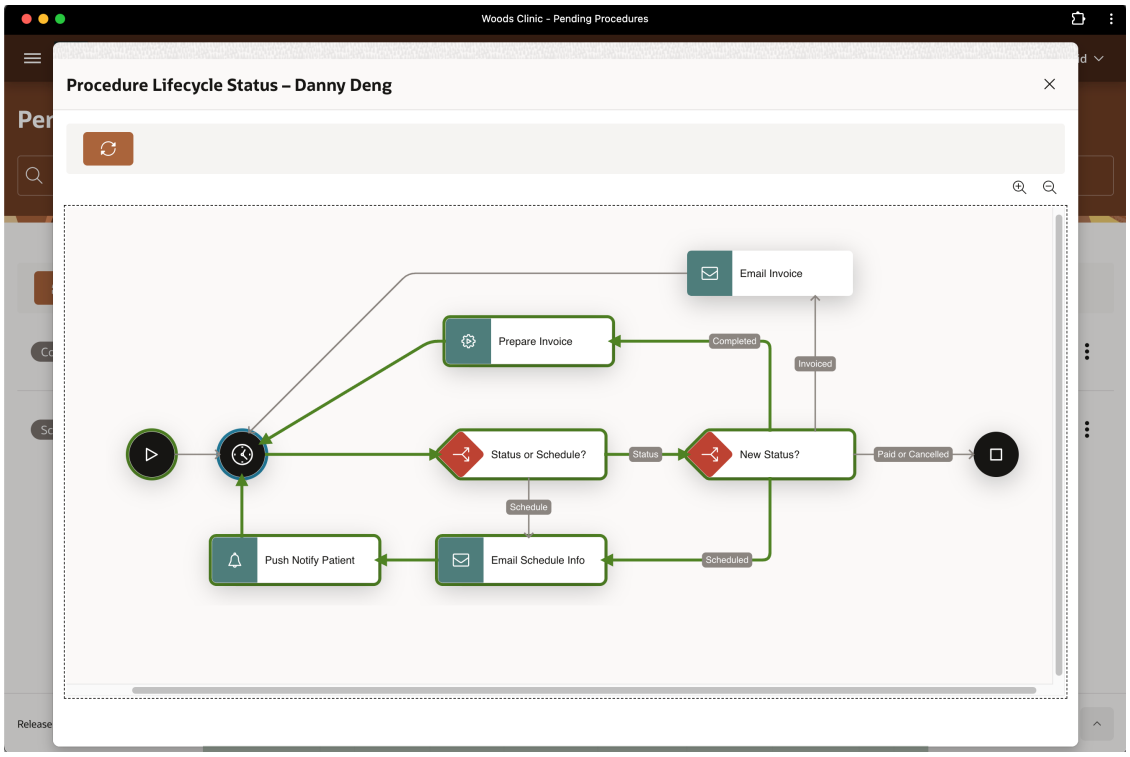
Print documents from a workflow by combining a Report Query and Report Layout.

As David sees from the lifecycle diagram below, in the background, your *Procedure Lifecycle* workflow reacts when Danny Deng's medical procedure status changes to "`Completed`". It prepares a draft invoice for *Woods Clinic* staff to review with Danny's insurance provider.

Copyright © 2026, Oracle and/or its affiliates.

The workflow's *Prepare Invoice* activity invokes the `PREPARE_PROCEDURE_INVOICE` procedure in the `FRC_CLINIC` package. First, it inserts a row in the `FRC_INVOICES` table with a reference to the medical procedure ID. Then, it generates the pixel-perfect invoice by combining two shared components: a Report Query and a Report Layout.

Figure 25-127 Invoice Prepared After Procedure Completed



25.3.5.5.2 Retrieving Invoice Data

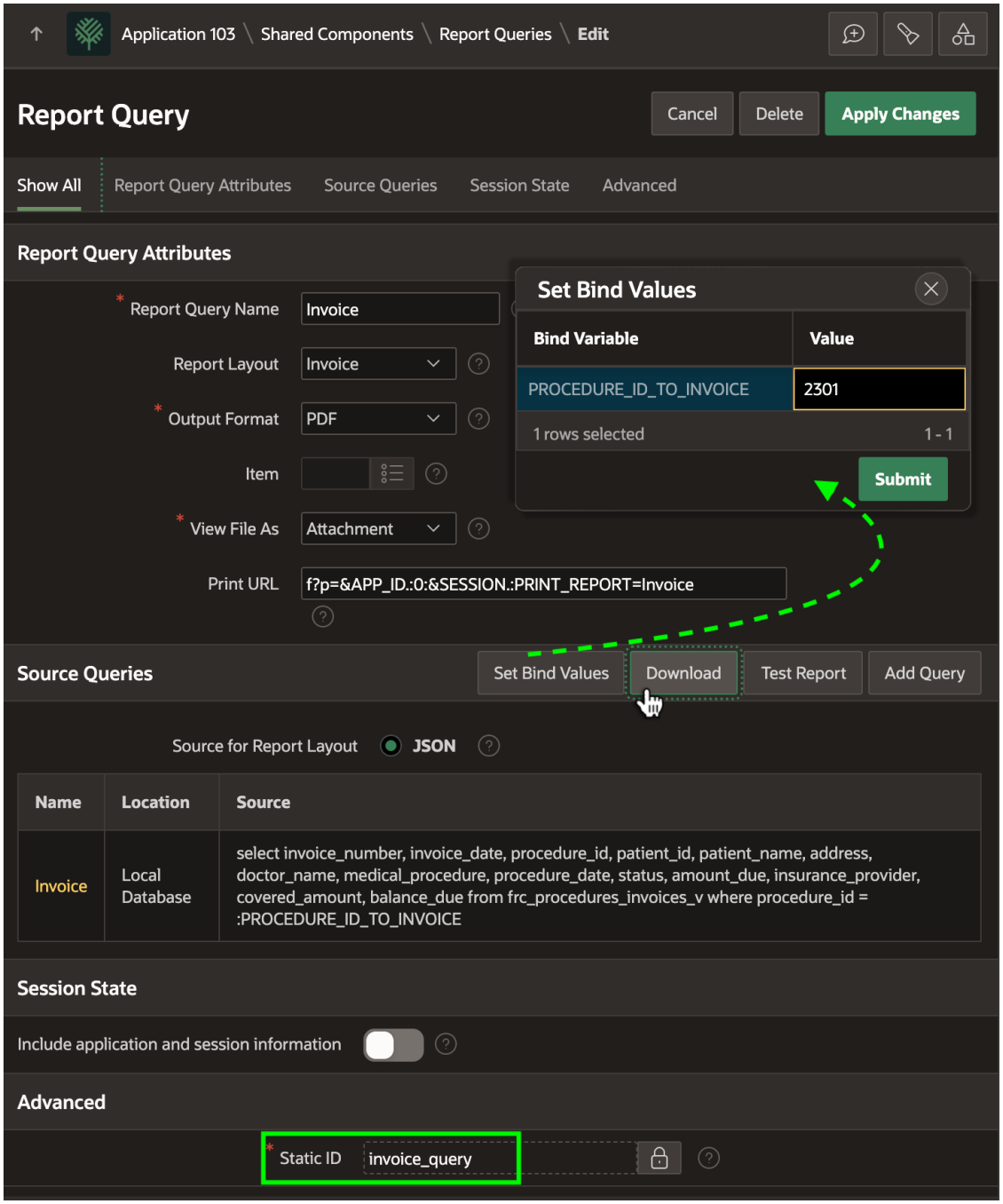
Define a Report Query that supplies invoice data as JSON for the report layout.

Your *Invoice* Report Query shared component shown below defines the data the invoice needs. You provide one or more *Source Queries* whose results the APEX engine includes in a single JSON document sent to the remote printing service. For the invoice, a *single* query suffices.

Using the *Set Bind Values* dialog, you can set example values for any bind variables your source queries reference. Then, clicking the *(Download)* button produces a JSON file containing example data. You use it to learn the names to reference to inject data values in appropriate places in your report layout. You use the *Static ID* `invoice_query` to identify the Report Query definition at document generation time.

Copyright © 2026, Oracle and/or its affiliates.

Figure 25-128 Report Query Provides JSON Data for Pixel-Perfect PDF Report Layout



The *Invoice* source query appears below. It uses a `PROCEDURE_ID_TO_INVOICE` application item as a bind variable for the medical procedure whose ID needs an invoice printed.

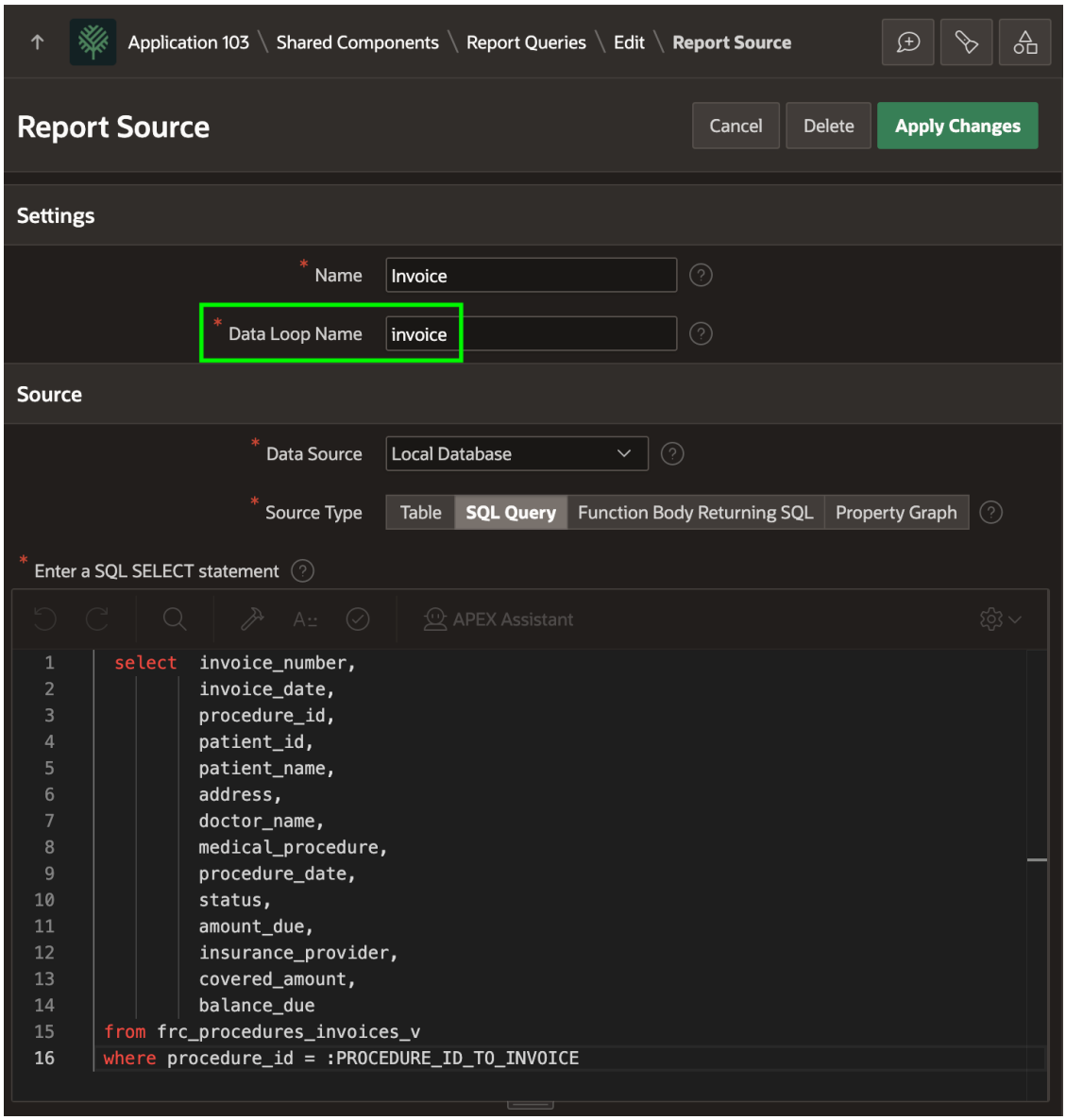
```
select  invoice_number,
        invoice_date,
        procedure_id,
        patient_id,
        patient_name,
        address,
```


Copyright © 2026, Oracle and/or its affiliates.

```
        doctor_name,
        medical_procedure,
        procedure_date,
        status,
        amount_due,
        insurance_provider,
        covered_amount,
        balance_due
from frc_procedures_invoices_v
where procedure_id = :PROCEDURE_ID_TO_INVOICE
```

Notice this query in the *Report Source* edit page below, as well as the *Data Loop Name*. This name identifies the array containing this source query's result rows in the JSON data document.

Figure 25-129 Report Source Configures SQL Query and Data Loop Name



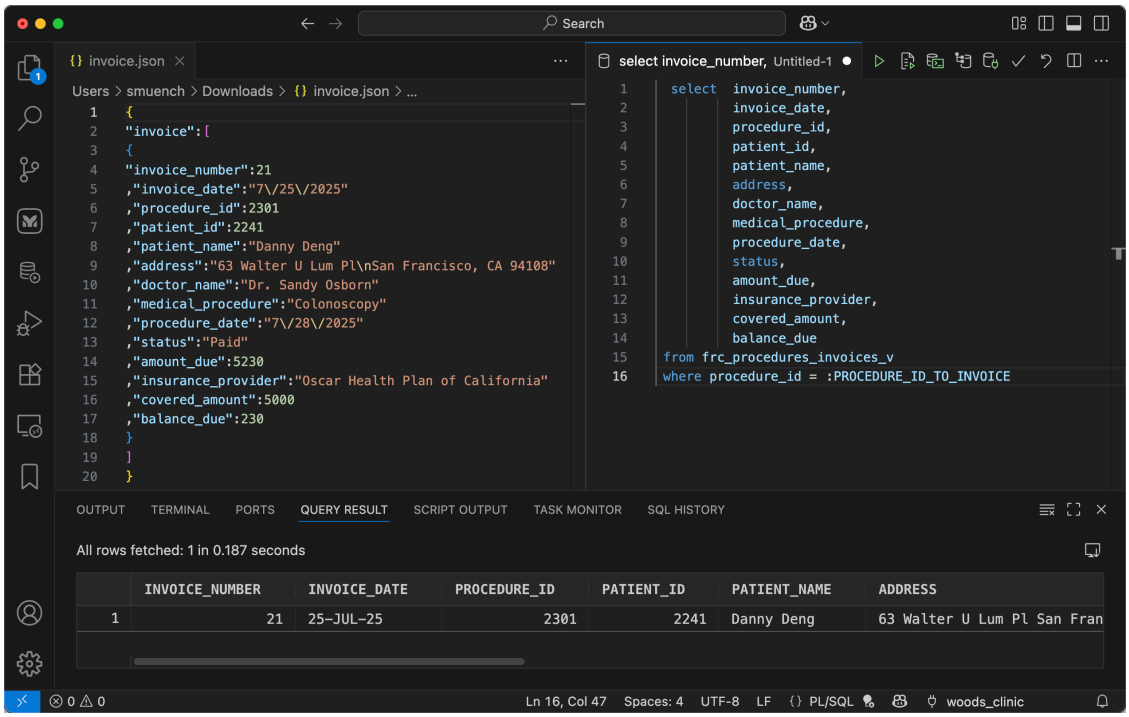
Copyright © 2026, Oracle and/or its affiliates.

After downloading an example JSON data document for medical procedure 2301, you can open it in an editor like Visual Studio Code shown below. Notice the "`invoice`" *Data Loop Name* and its corresponding array of a single row of JSON medical procedure query results. The property names in this document are vital when creating your report layout.

> **Tip**
>
> If you use VS Code, you will find the SQL Developer extension productive. As shown below, it lets you easily experiment with queries from the same code editor environment.

The figure shows Visual Studio Code with the SQL Developer Extension installed. On the left, the downloaded JSON appears containing the data that will drive the report layout. On the left, a SQL Developer worksheet tab where you can run SQL queries and observe the results.

Figure 25-130 Studying Report Query JSON and Testing Query in VS Code



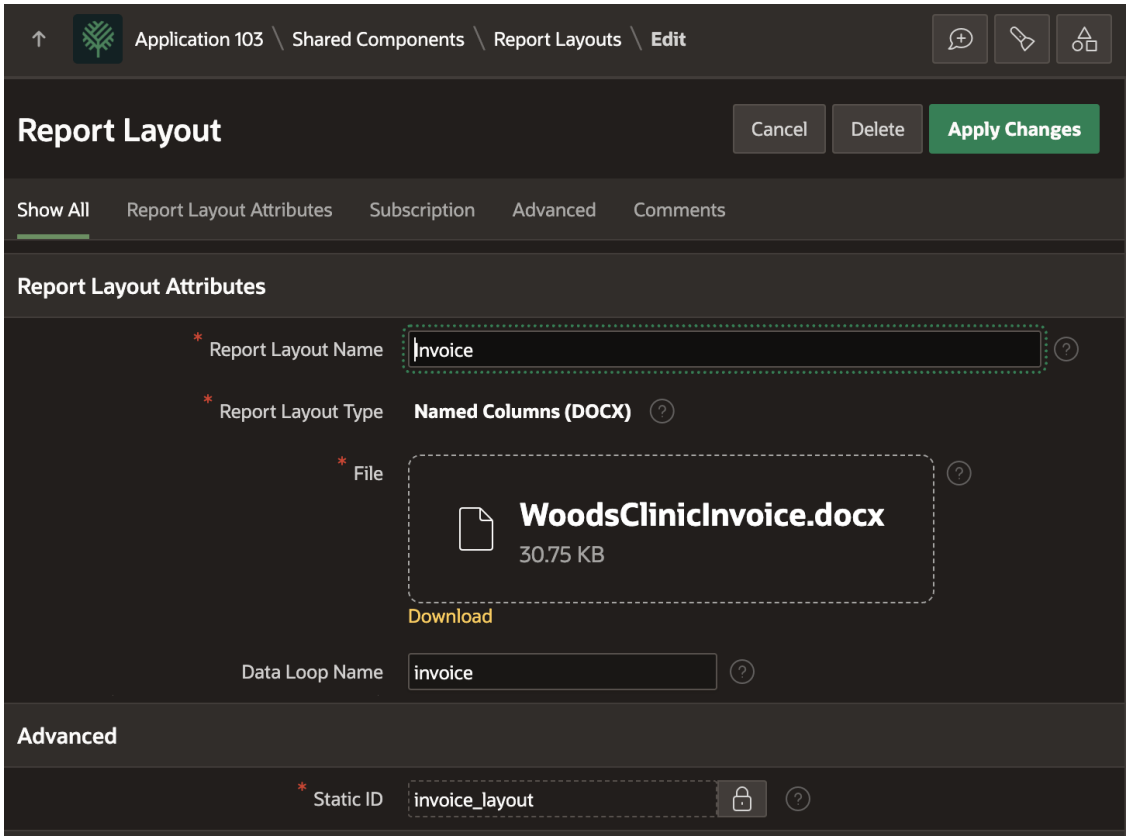
25.3.5.5.3 Formatting the Invoice in a Layout

Create a Word report layout that includes invoice data using special tags.

Your *Invoice* Report Layout shared component shown below defines a *Static ID* you use when generating the invoice. It includes the Microsoft Word document template that details how the invoice looks and where it includes data.

Copyright © 2026, Oracle and/or its affiliates.

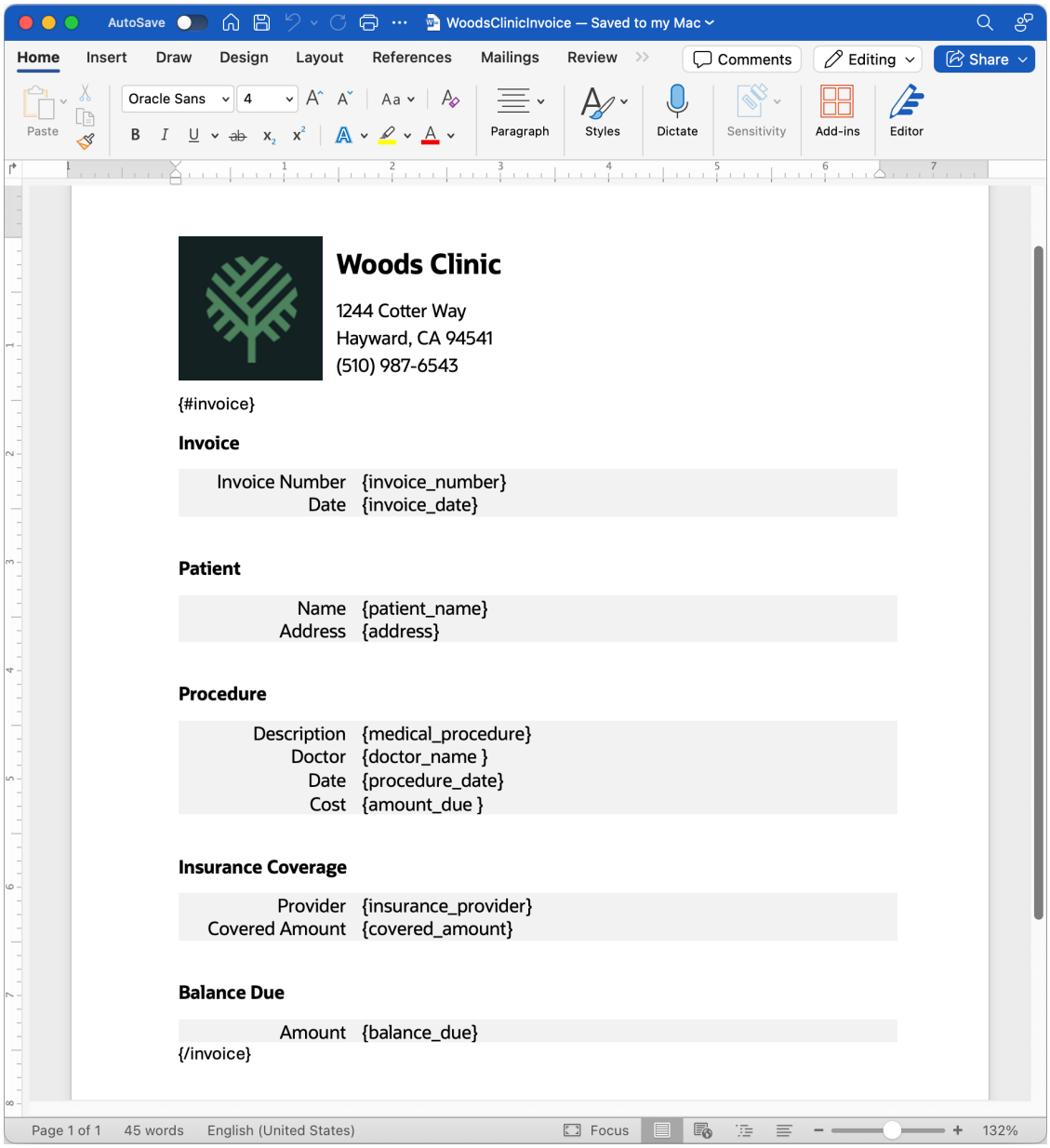
Figure 25-131 Report Layout Provides Word Document Template for Invoice



Your Microsoft Word document for the invoice uses the tag syntax the Oracle Document Generator cloud service defines to loop over JSON data arrays and include values of data object properties. Notice the `{#invoice}` tag opens the data loop and the matching `{/invoice}` closes the loop. The `invoice` name in these tags is the *Data Loop Name* you defined in your Report Query, which used it as the JSON array name for the query result rows. Inside this loop, tags like `{patient_name}` and `{balance_due}` among others reference the relative names of JSON properties in the row of `invoice` data.

Copyright © 2026, Oracle and/or its affiliates.

Figure 25-132 Microsoft Word Document Defines Pixel-Perfect PDF Layout



25.3.5.5.4 Generating the Data-Driven Document

Generate a PDF invoice and store it for review, download, and email delivery.

The workflow's *Prepare Invoice* activity ends up calling the `GENERATE_INVOICE` procedure below in the `FRC_CLINIC` package. This procedure looks up the procedure ID from the invoice ID, sets that value into the application item `PROCEDURE_ID_TO_INVOICE` used as a bind variable in the report query, and calls `GENERATE_DOCUMENT` in the `APEX_PRINT` package to create the pixel-perfect PDF invoice as a `BLOB`. Notice it passes the *Static ID* values of the Report Query and Report Layout, as well as the current application ID using `V('APP_ID')`.

Finally, it updates the `FRC_INVOICES` row to assign the generated PDF to its `DOCUMENT` column. This lets staff review it interactively in the *Woods Clinic* app and download it. It also helps your workflow attach it to the email it eventually sends to the patient.

Copyright © 2026, Oracle and/or its affiliates.

> **Tip**
>
> The multiple calls to the `UPDATE_INVOICE_STATUS` procedure provide feedback that appears in the *Review Invoices* page in the app. The status info helps users see the report generation is in progress in the background.

```
-- in frc_clinic package
procedure generate_invoice(
    p_invoice_id in number)
is
    l_pdf blob;
    l_procedure_id number;
begin
    -- Lookup patient procedure id for invoice for report query bind var
    select patient_procedure_id
      into l_procedure_id
      from frc_invoices
     where id = p_invoice_id;
    -- Set application item used as a bind variable in report query
    apex_session_state.set_value('PROCEDURE_ID_TO_INVOICE',l_procedure_id);
    update_invoice_status(p_invoice_id,'Processing');
    commit;
    -- Generate the pixel-perfect PDF invoice
    l_pdf := apex_print.generate_document(
                p_application_id          => v('APP_ID'),
                p_report_query_static_id  => 'invoice_query',
                p_report_layout_static_id => 'invoice_layout');
    update frc_invoices
    set status = 'Prepared',
        document = l_pdf
    where id = p_invoice_id;
exception
    when others then
        update_invoice_status(p_invoice_id,'Error');
        commit;
        raise;
end;
```

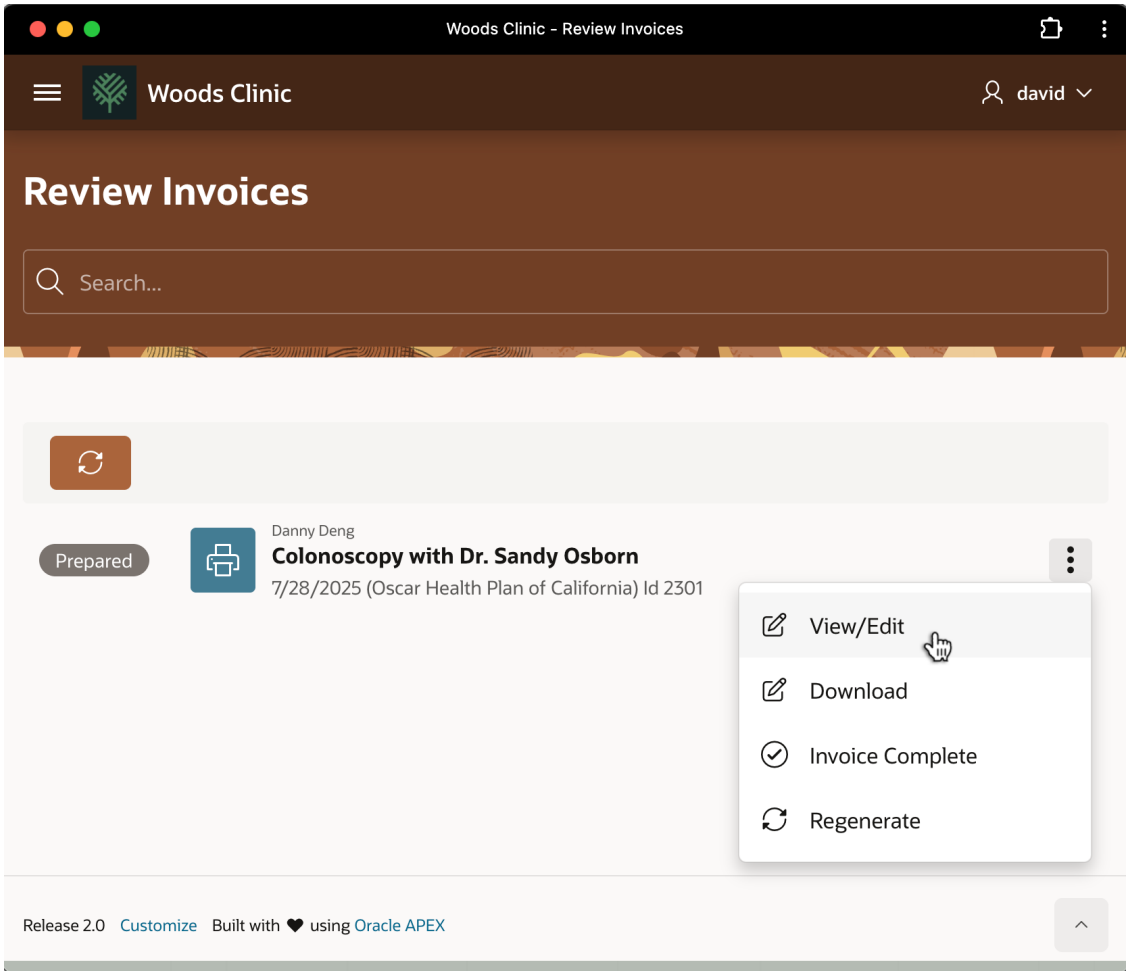
25.3.5.6 Reviewing Invoice with Insurance Provider

Review the prepared invoice and enter the insurance coverage amount.

From the *Review Invoices* page, David sees the "Prepared" status badge on the invoice so he knows it's ready to review. He uses the three-vertical-dots dropdown menu to review the invoice details.

Copyright © 2026, Oracle and/or its affiliates.

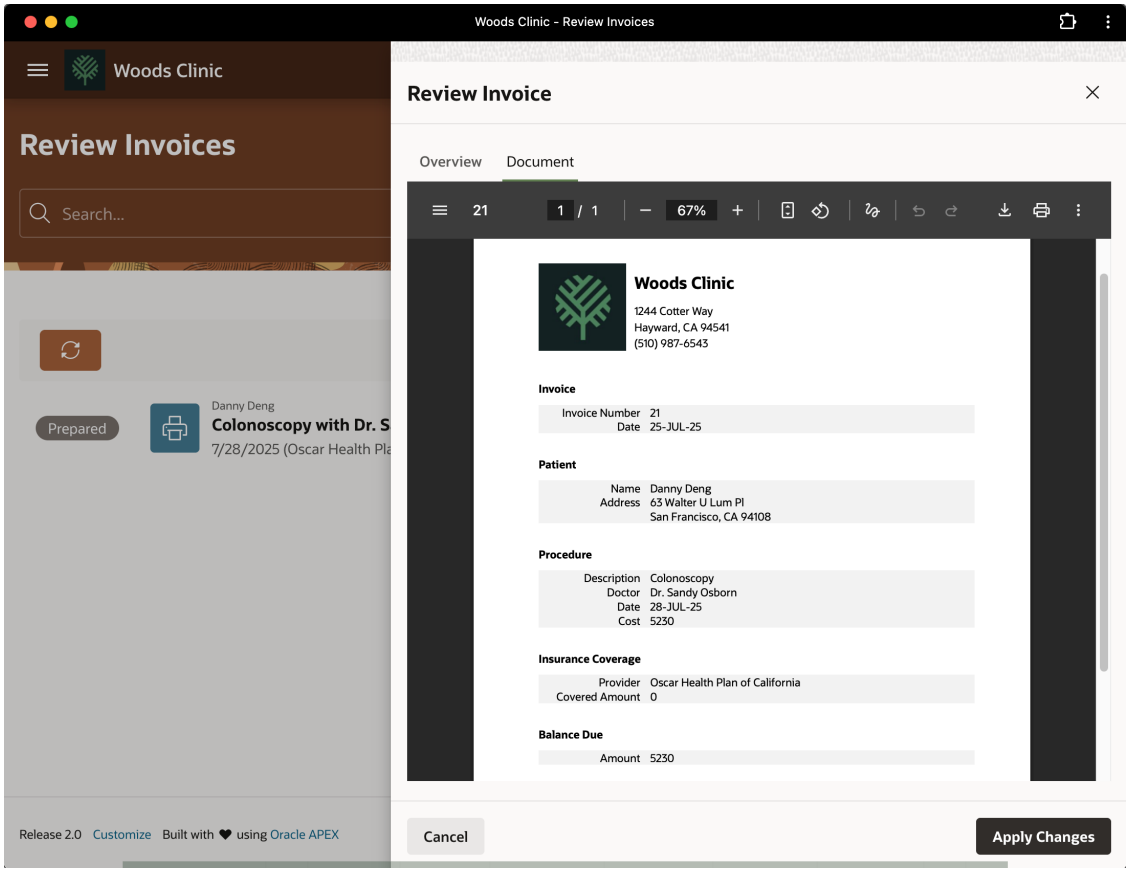
Figure 25-133 Opening the Prepared Invoice Details



On the *Document* tab of the *Review Invoice* dialog that appears, David can inspect the PDF document the patient will eventually get. But for the moment, notice that it starts out with zero (0) dollars in the *Covered Amount* field.

Copyright © 2026, Oracle and/or its affiliates.

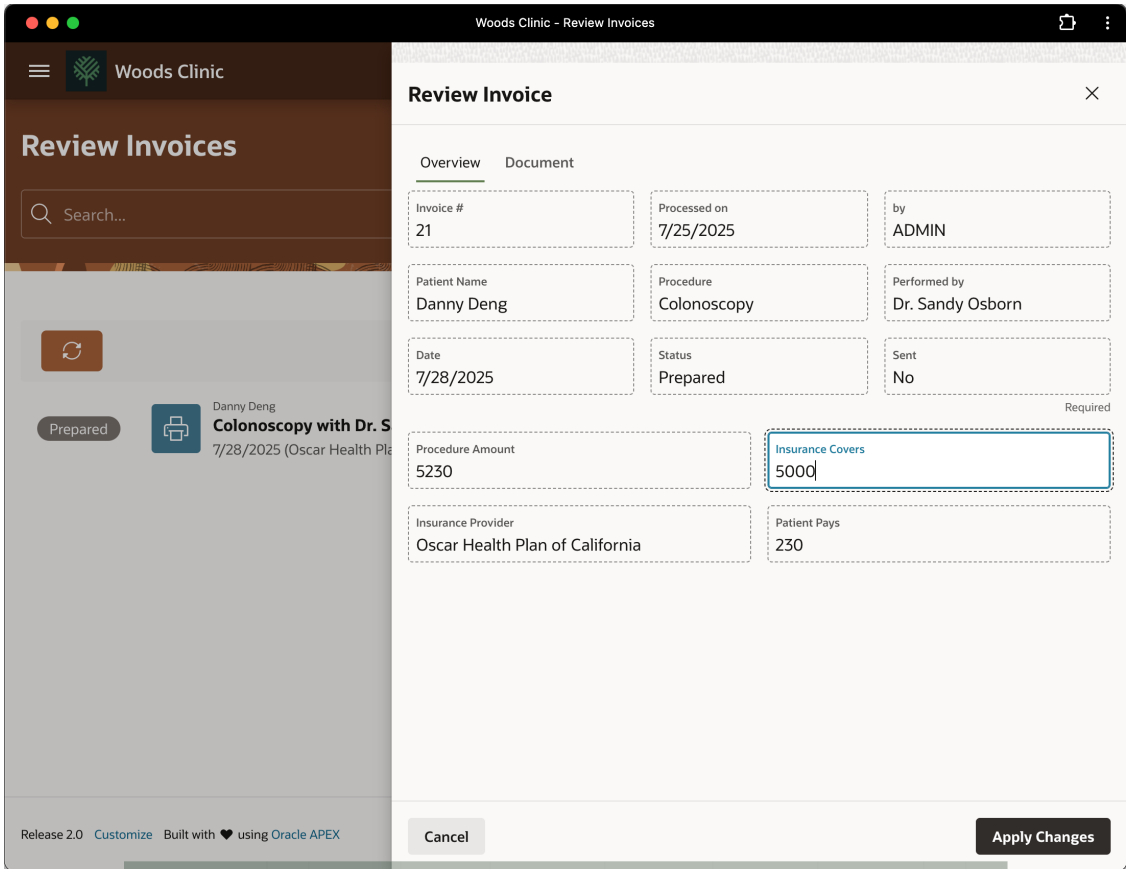
Figure 25-134 Reviewing Pixel-Perfect PDF Invoice



David calls Danny Deng's insurance provider and learns that they will reimburse 5000 dollars of the medical procedure, so on the *Overview* tab David enters that amount in the *Insurance Covers* field. Your page uses a dynamic action event handler on the value change of this field to recalculate the *Patient Pays* field as shown below.

Copyright © 2026, Oracle and/or its affiliates.

Figure 25-135 Entering Insurance Reimbursement Amount



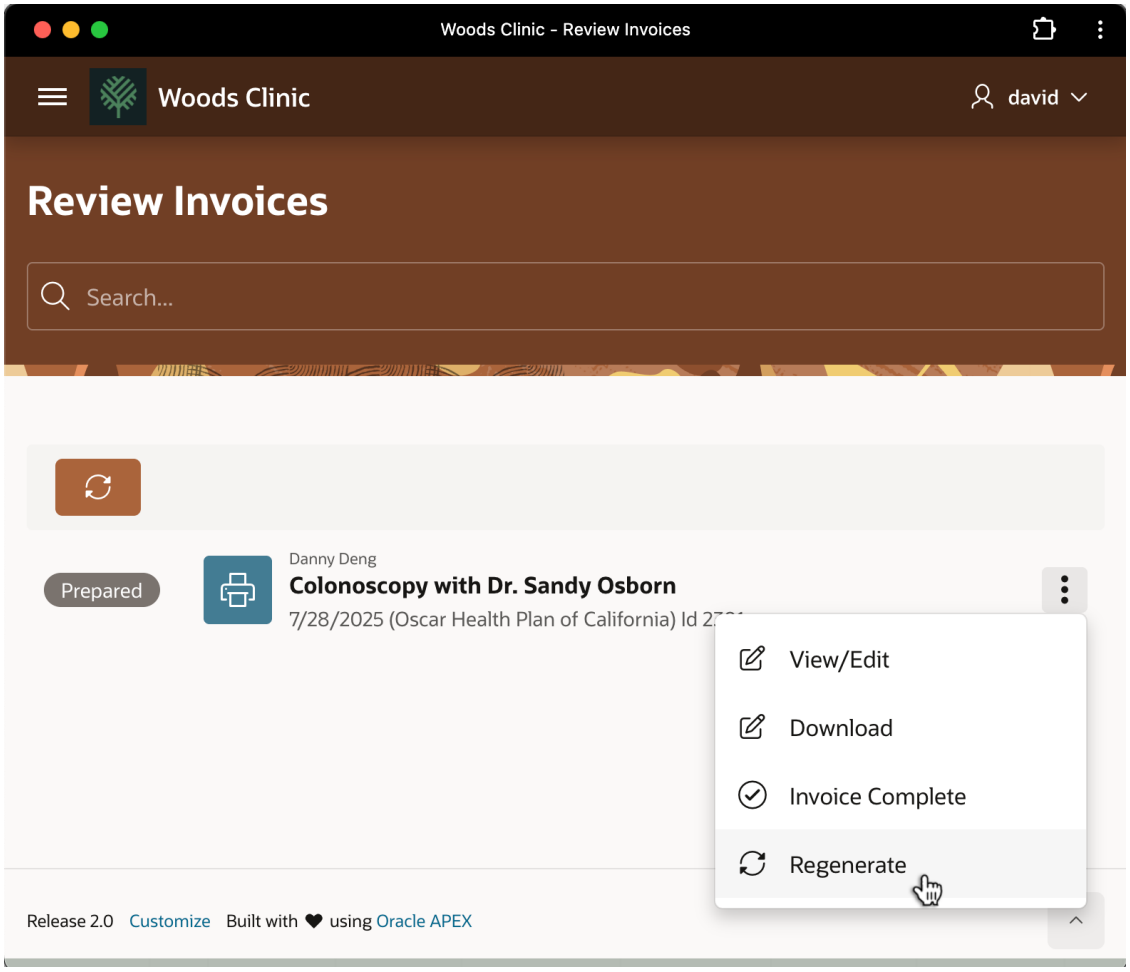
25.3.5.7 Regenerating Invoice in Background

Regenerate an invoice in the background while preventing duplicate runs for the same procedure.

After David enters the definitive amount Danny Deng's insurance provider will reimburse, he can regenerate the invoice to reflect the updated value. As shown below, he does this using the *Regenerate* option on the invoice three-vertical-dots dropdown menu.

Copyright © 2026, Oracle and/or its affiliates.

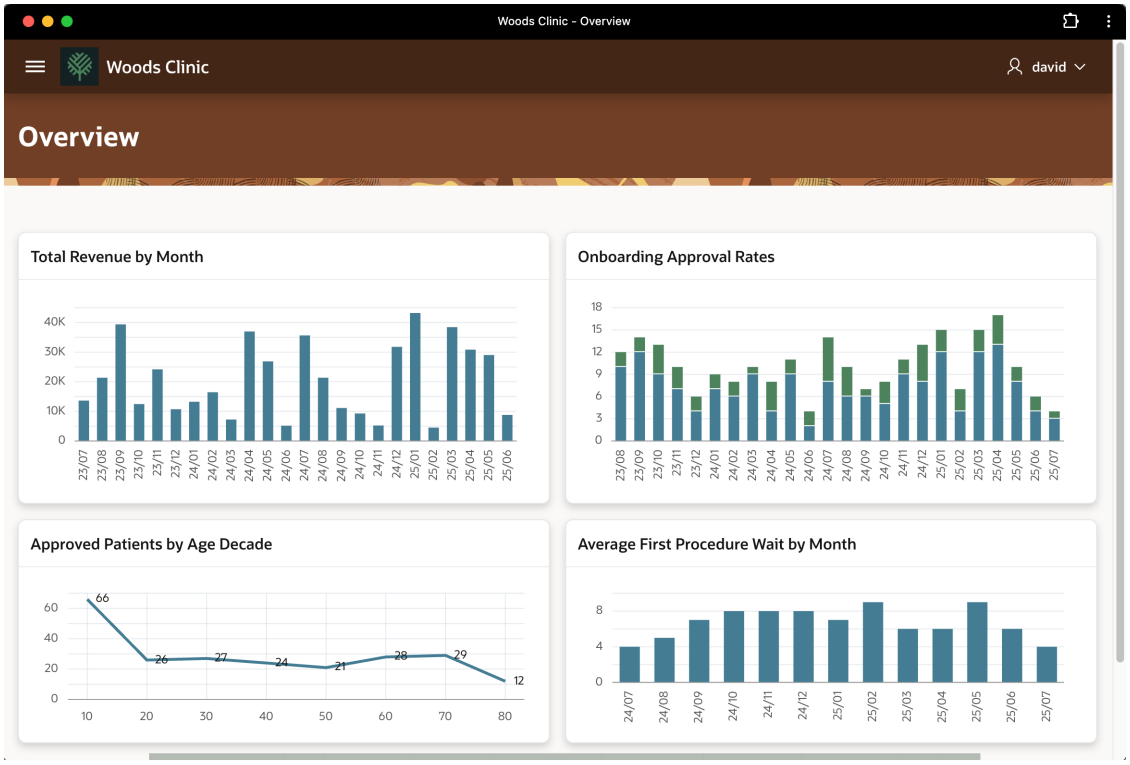
Figure 25-136 Regenerating Final Invoice After Entering Insurance Payment



Since the invoice generation happens in the background, David is free to continue using the *Woods Clinic* app in the meantime. For example, as shown below, he visits the *Overview* page for a glance at the dashboard of four charts showing various clinic trends.

Copyright © 2026, Oracle and/or its affiliates.

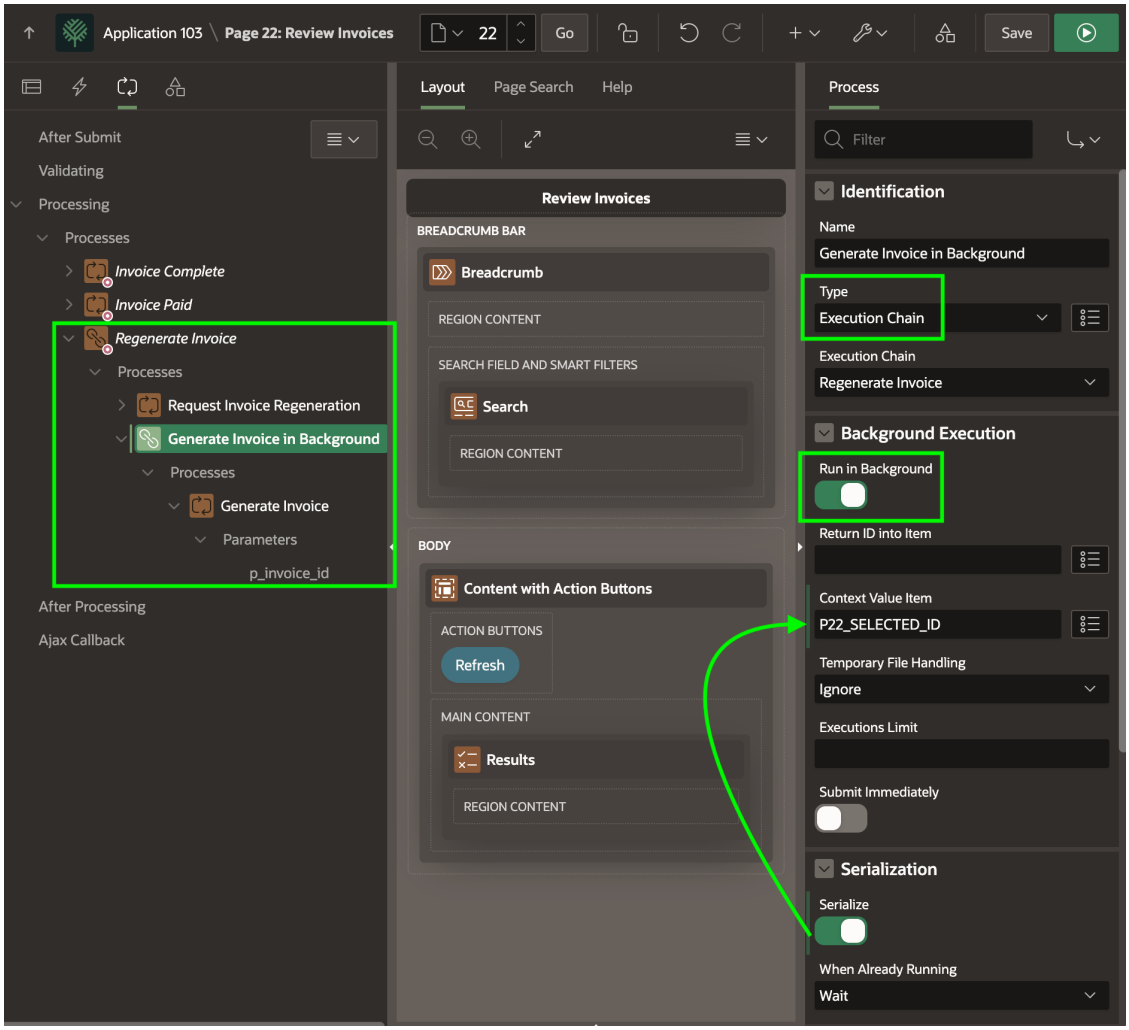
Figure 25-137 Reviewing Dashboard While Invoice Regenerates in Background



Your *Review Invoices* page makes quick work of offloading the invoice generation to the background using an *Execution Chain* page process type. It groups child processes, and they automatically run asynchronously when you enable its *Run in Background* switch. You enable its *Serialize* switch as well, and configure a *Context Value Item*. This combination ensures the APEX engine only lets one invoice generation per medical procedure ID happen simultaneously. The *Generate Invoice* child process invokes the same `GENERATE_INVOICE` procedure the workflow uses to prepare the original invoice draft.

Copyright © 2026, Oracle and/or its affiliates.

Figure 25-138 Execution Chain Simplifies Regenerating Invoice in the Background



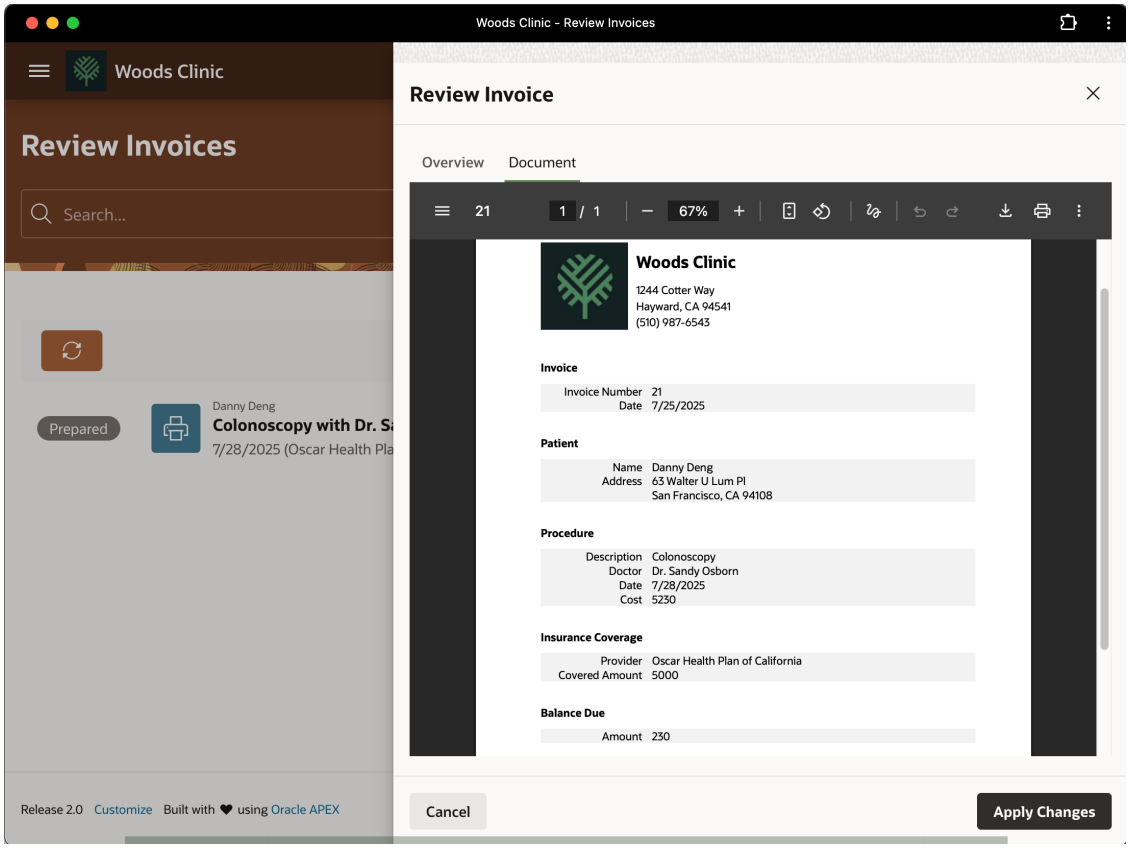
25.3.5.8 Invoicing Patient for the Balance

Mark the invoice complete so the workflow emails it to the patient and advances the procedure lifecycle.

When David sees the regenerated invoice badge shows *Prepared* again, he reviews it to make sure the *Covered Amount* and patient's *Balance Due* look right. He confirms the insurance company's 5000 payment shows as expected, and the patient's remaining balance to pay is correct.

Copyright © 2026, Oracle and/or its affiliates.

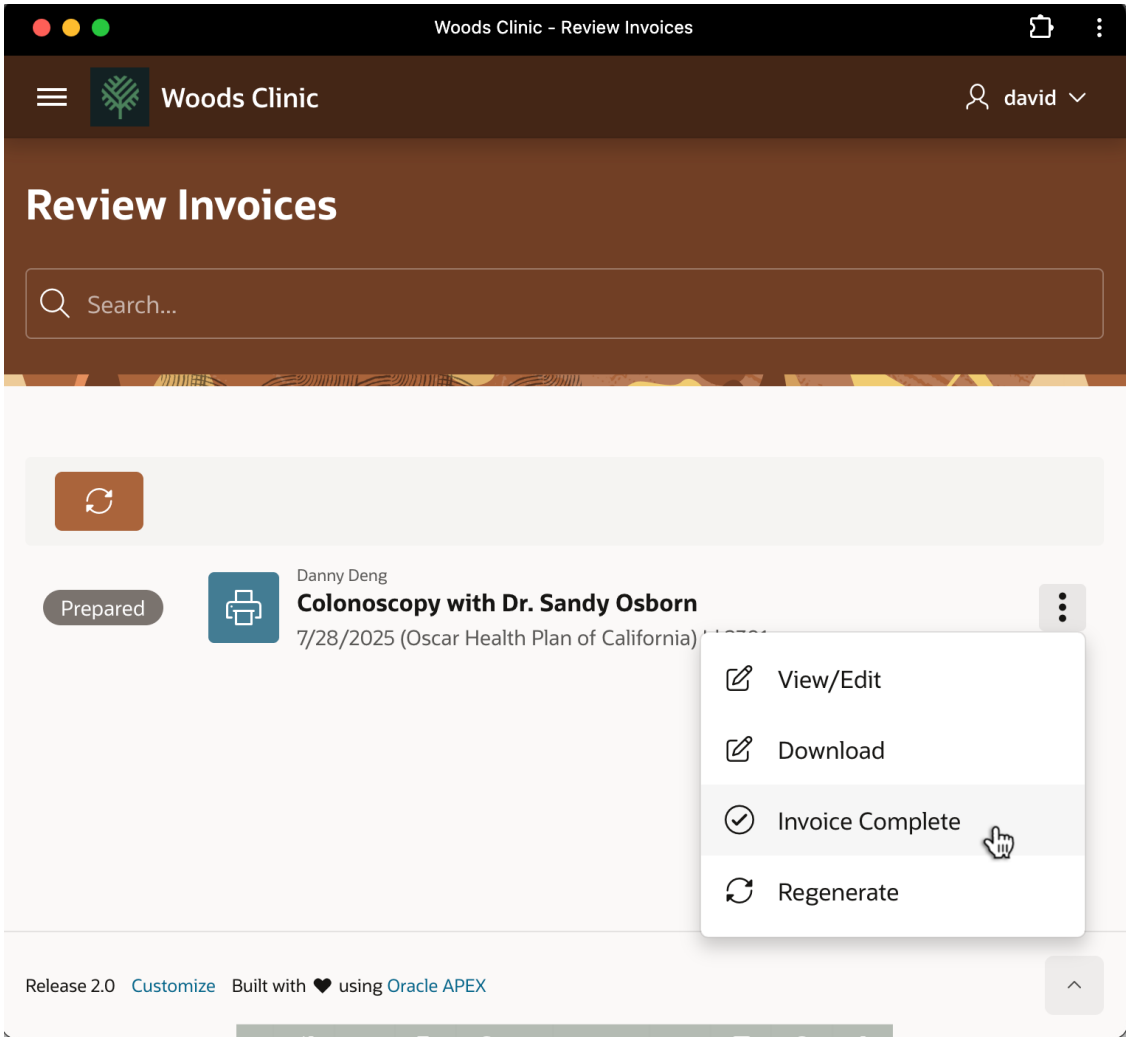
Figure 25-139 Reviewing Regenerated Invoice Before Sending to Patient



David marks the invoice as ready to send to the patient by choosing *Invoice Complete* from the three-vertical-dots dropdown menu as shown below.

Copyright © 2026, Oracle and/or its affiliates.

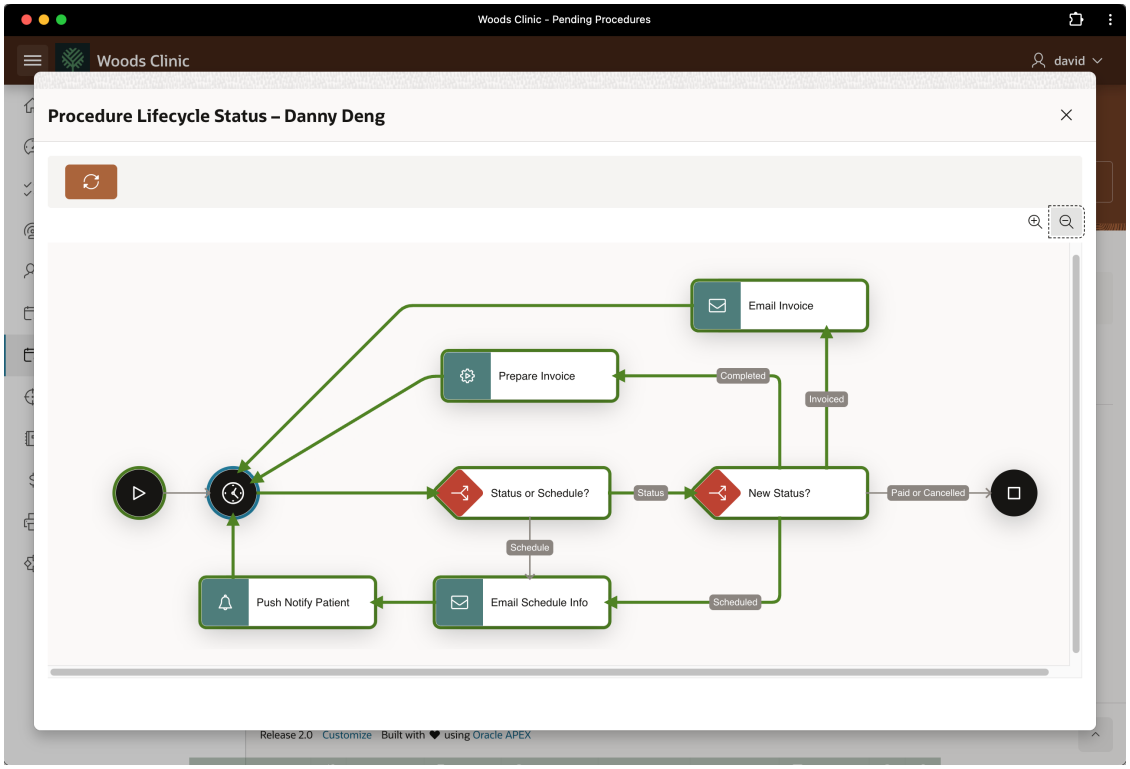
Figure 25-140 David Marks Danny Deng's Invoice as Ready to Send to Patient



When David reviews Danny Deng's procedure progress later, he notices, as shown below, that your workflow reacted to the completion of the invoice by emailing it to the patient.

Copyright © 2026, Oracle and/or its affiliates.

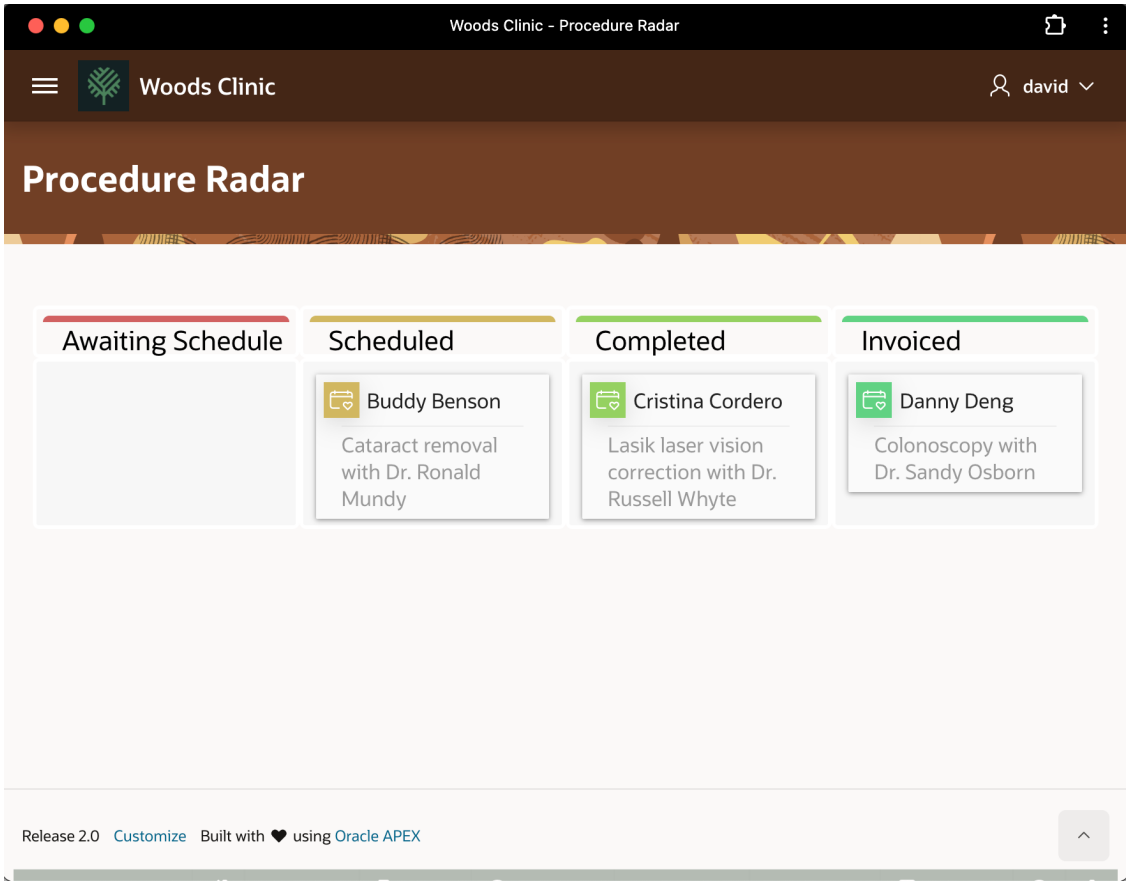
Figure 25-141 Danny Deng's Procedure Lifecycle Workflow Shows Invoice Emailed



He also sees Danny Deng's procedure shows in the *Invoiced* column in the Kanban board as shown below.

Copyright © 2026, Oracle and/or its affiliates.

Figure 25-142 Progress of Danny Deng's Procedure on Kanban Board



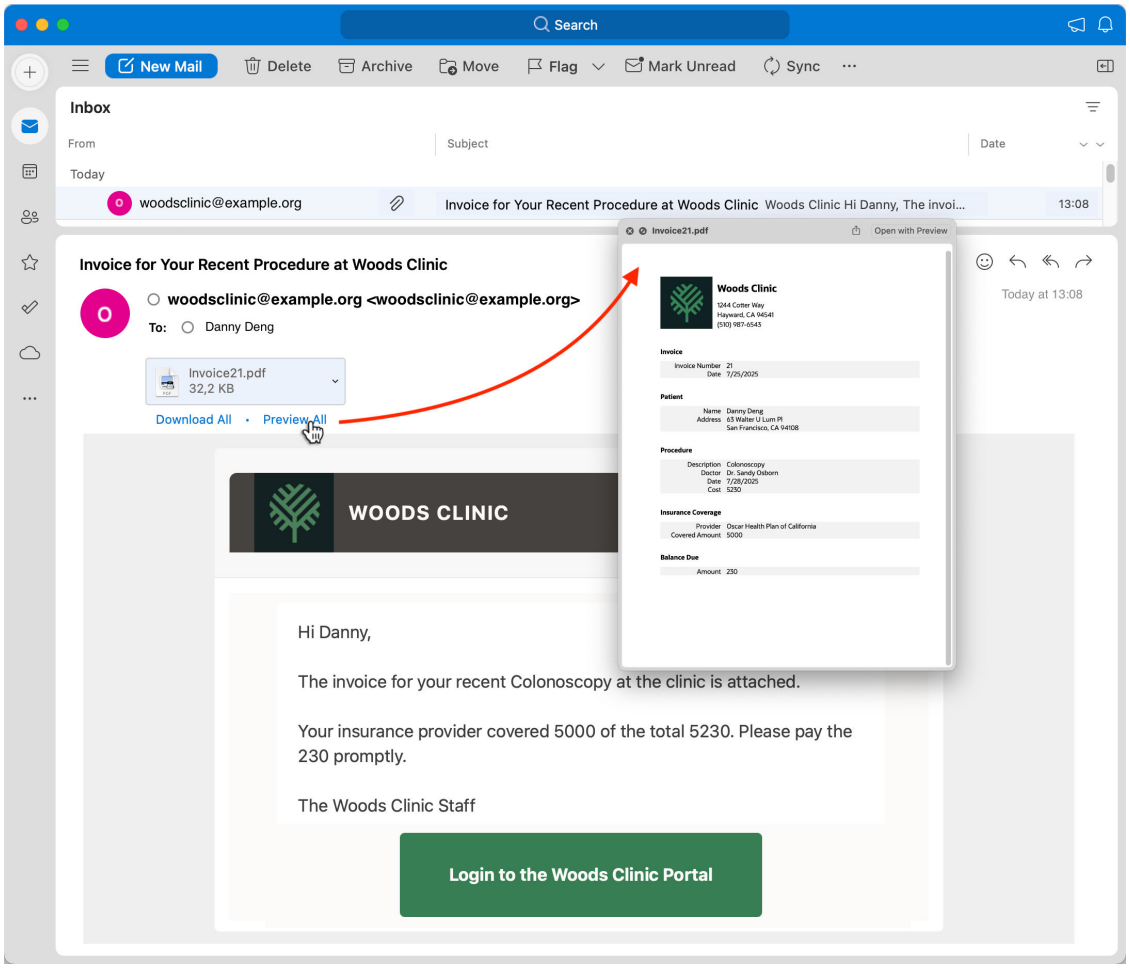
25.3.5.9 Acknowledging Receipt of Payment

Record the patient's payment so the procedure reaches its final status.

Patient Danny Deng gets the email, and reviews the invoice as shown below. He then calls *Woods Clinic* to pay the balance with his credit card.

Copyright © 2026, Oracle and/or its affiliates.

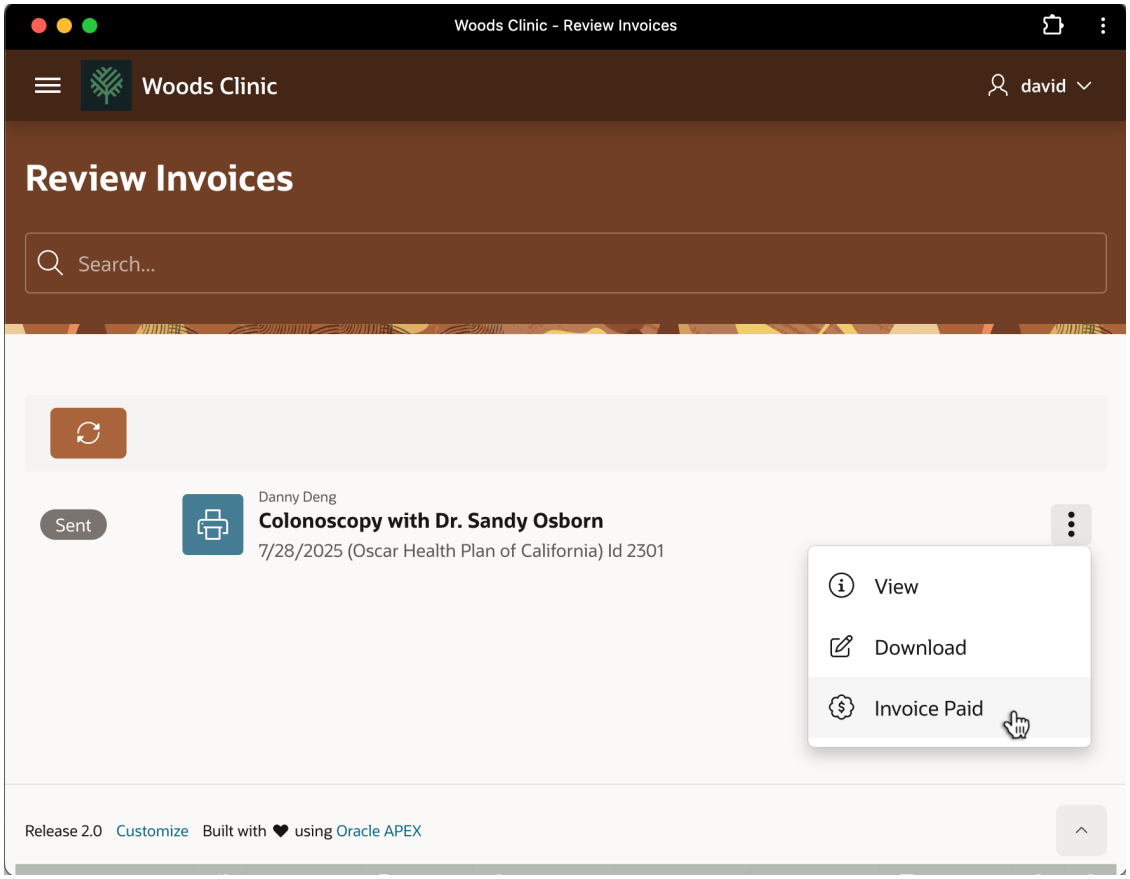
Figure 25-143 Danny Deng Receives Invoice for His Medical Procedure



David accepts Danny's payment on the phone, and, as shown below, he marks the invoice as settled using the *Invoice Paid* option on the three-vertical-dots dropdown menu.

Copyright © 2026, Oracle and/or its affiliates.

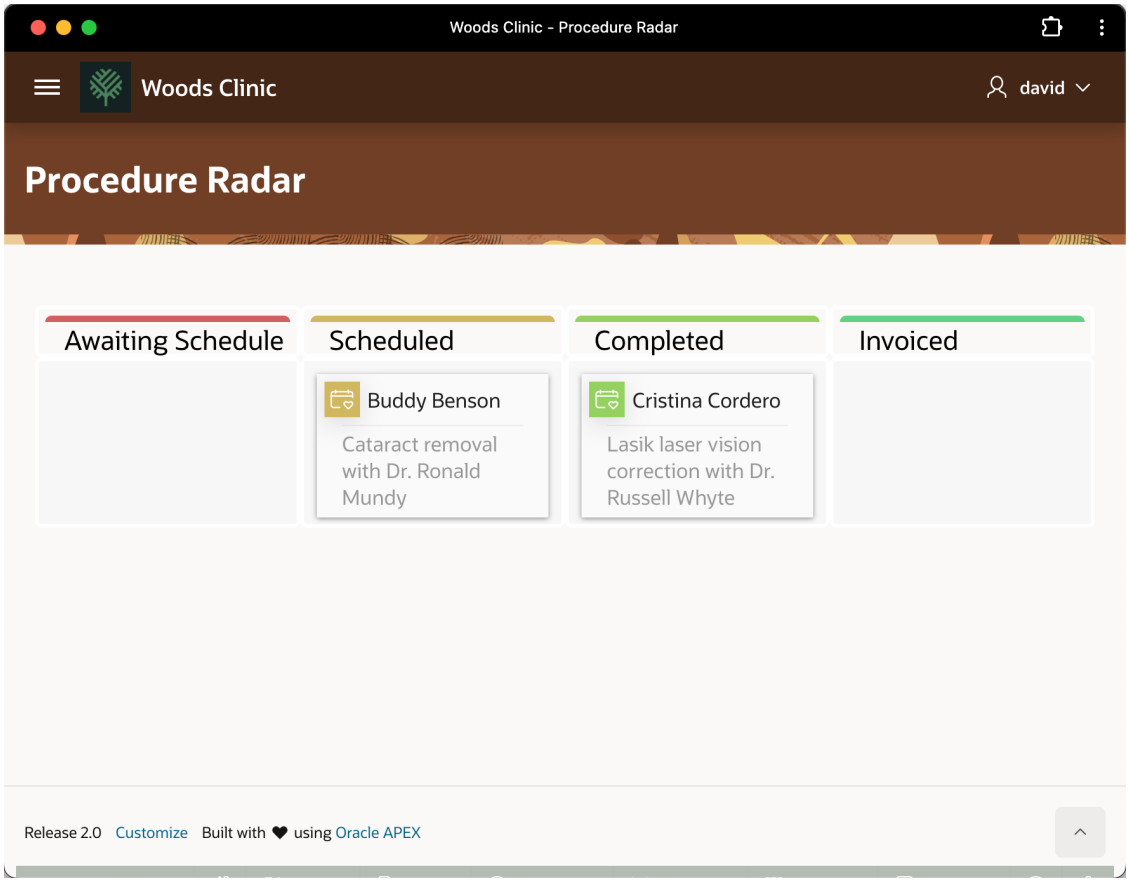
Figure 25-144 David Marks Danny Deng's Medical Procedure Invoice as Paid



This changes the status of Danny Deng's medical procedure row to "`Paid`" and, as shown below, David sees Danny's has moved off the Kanban board.

Copyright © 2026, Oracle and/or its affiliates.

Figure 25-145 Kanban No Longer Shows Danny Deng's Procedure After Invoice is Paid



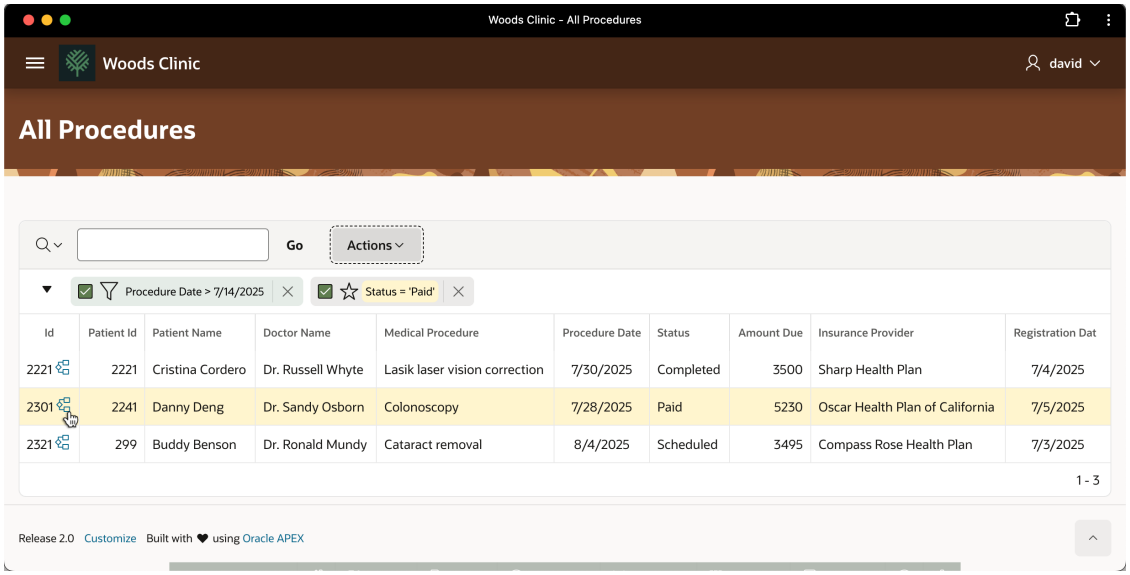
25.3.5.10 Confirming the Lifecycle is Complete

Confirm the paid procedure completed its workflow and full business lifecycle.

Reviewing patient procedures since mid-month on the *All Procedures* page, as shown below, David confirms that Danny Deng's medical procedure shows as *Paid*. He clicks the workflow icon next to the procedure to review the *Procedure Lifecycle* workflow progress one last time.

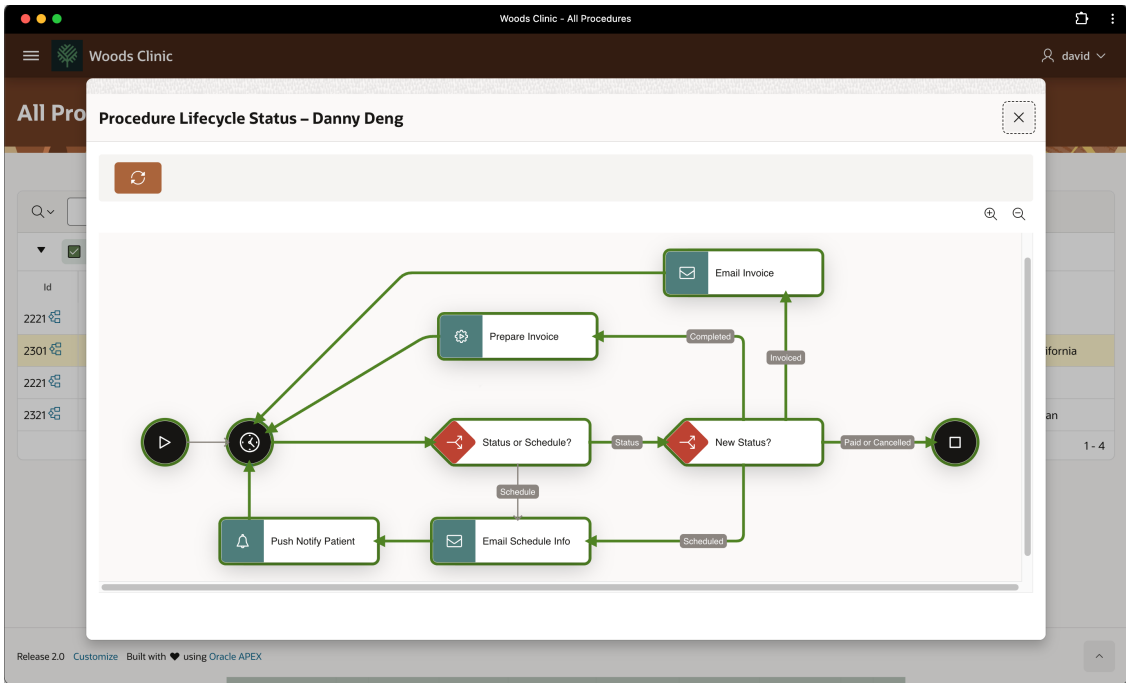
Copyright © 2026, Oracle and/or its affiliates.

Figure 25-146 David Confirms Danny Deng's Medical Procedure is Paid



David sees that all lifecycle steps are complete for Danny Deng's medical procedure, so he moves on to help other patients. The *Woods Clinic* staff gives you positive feedback on how much more effective the clinic is running now that you've automated so many of the manual steps in their patient onboarding and medical procedure business processes.

Figure 25-147 David Confirms Danny Deng's Procedure Lifecycle is Done



Copyright © 2026, Oracle and/or its affiliates.

25.3.6 Notifying Workflow of Data Changes

Notify a waiting workflow about data changes with events an automation processes periodically.

After their initial approval, your *Procedure Lifecycle* workflow automates previously manual steps performed as medical procedures progress through the clinic. It accomplishes this by reacting to changes in the procedure status or schedule.

To notify the workflow of data changes, you use a combination of a database trigger to log patient procedure lifecycle events, and an automation to periodically process the logged events. As the automation processes any events, it notifies the workflow of data changes by signaling the *Wait* activity to continue.

- Understanding Lifecycle Event Strategy
  Log procedure lifecycle events in a trigger and process them on a schedule with an automation.
- Simplifying Event Management
  Define a package API that lets the trigger add events and the automation process them.
- Adding Events Using a Trigger
  Use a trigger to log status or schedule changes as lifecycle events.
- Processing Events with an Automation
  Use an automation to process queued lifecycle events on a schedule.
- Continuing a Waiting Lifecycle Workflow
  Continue the waiting workflow activity and pass the event type into a workflow variable.

25.3.6.1 Understanding Lifecycle Event Strategy

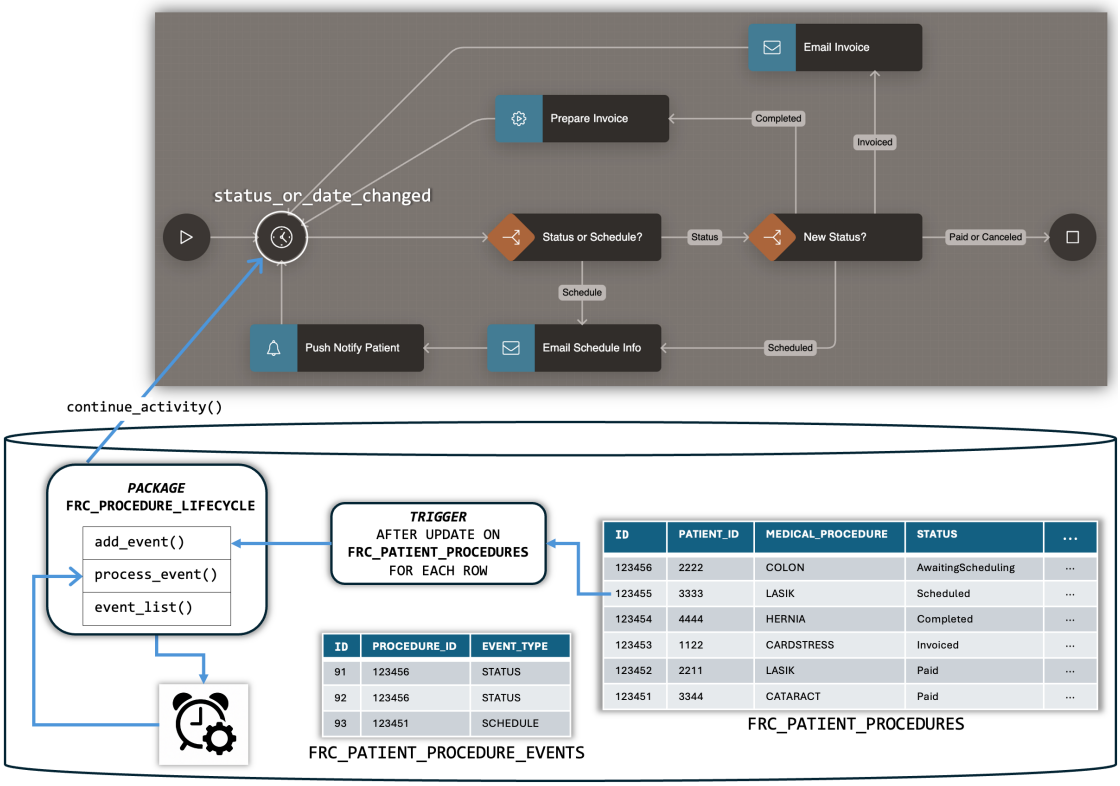
Log procedure lifecycle events in a trigger and process them on a schedule with an automation.

To notify the workflow of data changes, you adopt a strategy shown below that *decouples* the lifecycle event for a status change from the work it entails. This separation keeps the user experience fast by offloading work to the background. Using an `AFTER UPDATE FOR EACH ROW` trigger on the `FRC_PATIENT_PROCEDURES` table, you call the `ADD_EVENT` procedure in the `FRC_PROCEDURE_LIFECYCLE` application package to register the event. That procedure inserts a row in the `FRC_PATIENT_PROCEDURE_EVENTS` table. The row contains the primary key of the patient procedure, and an event type indicating whether the procedure is new, or whether its schedule or status changed.

On a recurring schedule, an *Automation* shared component "wakes up" and queries the list of events to process in the current run using the `EVENTS_LIST` function in the same package. For each lifecycle event to process, the automation's action calls the `PROCESS_EVENT` procedure. The package calls `CONTINUE_ACTIVITY` in the `APEX_WORKFLOW` package, passing the event type value back to the workflow.

Copyright © 2026, Oracle and/or its affiliates.

Figure 25-148 Strategy for Using Workflow to Orchestrate Patient Procedure Lifecycle



25.3.6.2 Simplifying Event Management

Define a package API that lets the trigger add events and the automation process them.

The `FRC_PROCEDURE_LIFECYCLE` package offers a simple API for managing lifecycle events. Its package specification shown below has procedures to add and process an event, and a function that returns the list of events.

```
package frc_procedure_lifecycle is
    type t_lifecycle_event is record (id
frc_patient_procedure_events.id%type);
    type t_lifecycle_events is table of t_lifecycle_event;
    ---------------------------------------------------------------------
    -- Add procedure lifecycle event to table processed by automation
    ---------------------------------------------------------------------
    procedure add_event(
        p_procedure_id in number,
        p_type         in varchar2);
    ---------------------------------------------------------------------
    -- Process procedure lifecycle event (called by automation)
    ---------------------------------------------------------------------
    procedure process_event(
        p_event_id in number);
    ---------------------------------------------------------------------
    -- Return list of event ids to process (used in automation query)
    ---------------------------------------------------------------------
    function event_list
```


Copyright © 2026, Oracle and/or its affiliates.

```
    return t_lifecycle_events
    pipelined;
end frc_procedure_lifecycle;
```

The ADD_EVENT procedure simply inserts a row into the FRC_PATIENT_PROCEDURE_EVENTS table. The table includes a primary key ID that is defaulted when null from an identity sequence, and a CREATED timestamp column that defaults to SYSTIMESTAMP. So here you only need to insert the patient procedure ID and the event type passed in (e.g. NEW, STATUS, or SCHEDULE).

```
-- external in package frc_procedure_lifecycle
procedure add_event(
    p_procedure_id in number,
    p_type         in varchar2)
is
begin
    insert into frc_patient_procedure_events(procedure_id, event_type)
    values (p_procedure_id, p_type);
end add_event;
```

The PROCESS_EVENT accepts the unique ID of a lifecycle event and signals a status or schedule change to a medical procedure's workflow. It calls the NOTIFY_DATA_CHANGED helper procedure to do so. If the process is successful, it deletes the event row it just processed from the FRC_PATIENT_PROCEDURE_LIFECYCLE table.

```
-- external in package frc_procedure_lifecycle
procedure process_event(
    p_event_id in number)
is
    l_success boolean := false;
begin
    for j in (select id, procedure_id, event_type
                from frc_patient_procedure_events
               where id = p_event_id )
    loop
        if j.event_type in ('STATUS','SCHEDULE') then
            l_success := notify_data_changed(j.procedure_id,j.event_type);
            if l_success then
                delete from frc_patient_procedure_events
                where id = p_event_id;
            end if;
        end if;
    end loop;
end process_event;
```

The EVENT_LIST function returns a table of T_LIFECYCLE_EVENT records. Each record contains a single ID field. Notice the PIPELINED modifier and corresponding PIPE ROW statement inside the loop. This combination defines a function whose result you can use in a SELECT statement as a row source by "wrapping" the function call with the TABLE() operator.

```
function event_list
return t_lifecycle_events
pipelined
is
begin
```


Copyright © 2026, Oracle and/or its affiliates.

```
    for j in (select id
                from (select id,
                             row_number() over
                             (partition by procedure_id
                              order by created) as rn
                        from frc_patient_procedure_events
                  )
                 where rn = 1)
    loop
        pipe row (t_lifecycle_event(j.id));
    end loop;
exception
    when no_data_needed then
        null;
end event_list;
```

> **Tip**
>
> `EVENT_LIST`'s cursor for loop uses the `ROW_NUMBER()` analytic function – aliased to `RN` – along with the `WHERE RN = 1` clause to ensure only a single event per distinct `PROCEDURE_ID` value is returned. Using the `ORDER BY CREATED` in the windowing clause guarantees that the single event returned for each procedure ID is the one that occurred earliest in time. This simplifies the event processing so that only one event per patient procedure ID row is processed in a given automation job run.

25.3.6.3 Adding Events Using a Trigger

Use a trigger to log status or schedule changes as lifecycle events.

When an `FRC_PATIENT_PROCEDURES` row is updated, it compares the `:NEW.STATUS` and `:OLD.STATUS` to decide if the status has changed. If so, it uses "`STATUS`" for the event type. If instead the procedure's date, doctor, or duration in hours changes, then the event type is "`SCHEDULE`". If either case occurs, then it calls `ADD_EVENT` in the `FRC_PROCEDURE_LIFECYCLE` package to do that.

```
trigger frc_patient_procedures_au
after update on frc_patient_procedures
for each row
declare
    l_event_type varchar2(80);
begin
    if :new.status != :old.status then
        l_event_type := 'STATUS';
    elsif    :new.procedure_date != :old.procedure_date
          or :new.doctor         != :old.doctor
          or :new.duration_hours != :old.duration_hours then
        l_event_type := 'SCHEDULE';
    end if;
    if l_event_type is not null then
        frc_procedure_lifecycle.add_event(:new.id, l_event_type);
    end if;
end;
```


Copyright © 2026, Oracle and/or its affiliates.

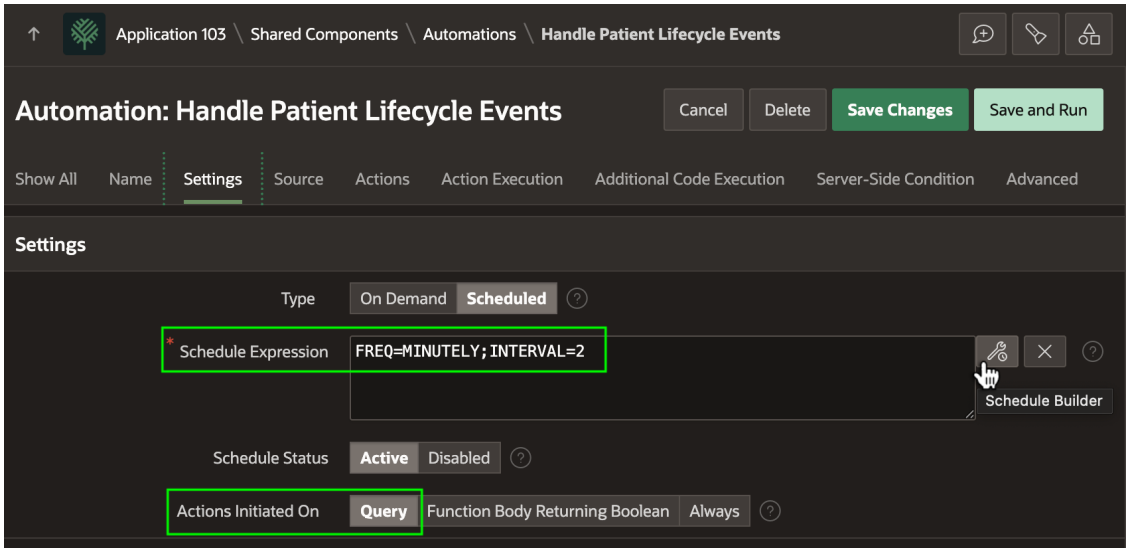
25.3.6.4 Processing Events with an Automation

Use an automation to process queued lifecycle events on a schedule.

The *Handle Patient Lifecycle Event* automation shown below is a shared component that defines a recurring background job. It runs periodically based on the *Schedule Expression* you create with the *Schedule Builder* dialog, or type in directly. This particular job runs in the background every two minutes.

The *Query* choice for *Actions Initiated On* means the automation's actions execute once for each row in the query you define on the *Source* tab.

Figure 25-149 Lifecycle Events Handler Automation – Settings

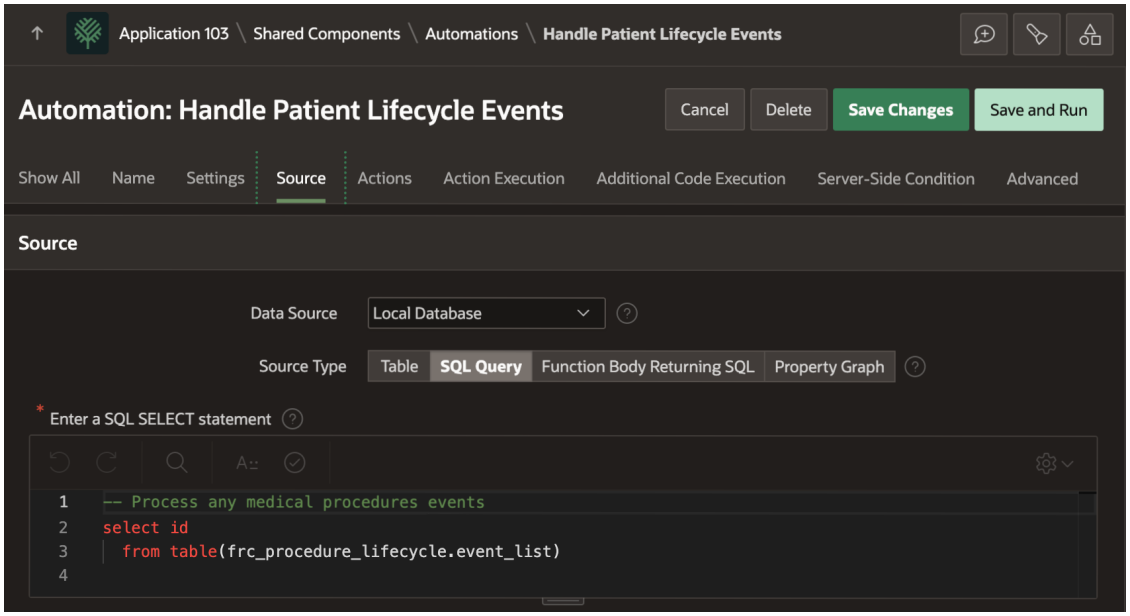


On the *Source* tab you specify the simple query that retrieves the patient procedure lifecycle events to process during the current automation job run. The query uses the `TABLE()` operator to select the event `ID` values to process from the `EVENT_LIST` function in the `FRC_PROCEDURE_LIFECYCLE` package. The simple query is:

```
select id
  from table(frc_procedure_lifecycle.event_list)
```


Copyright © 2026, Oracle and/or its affiliates.

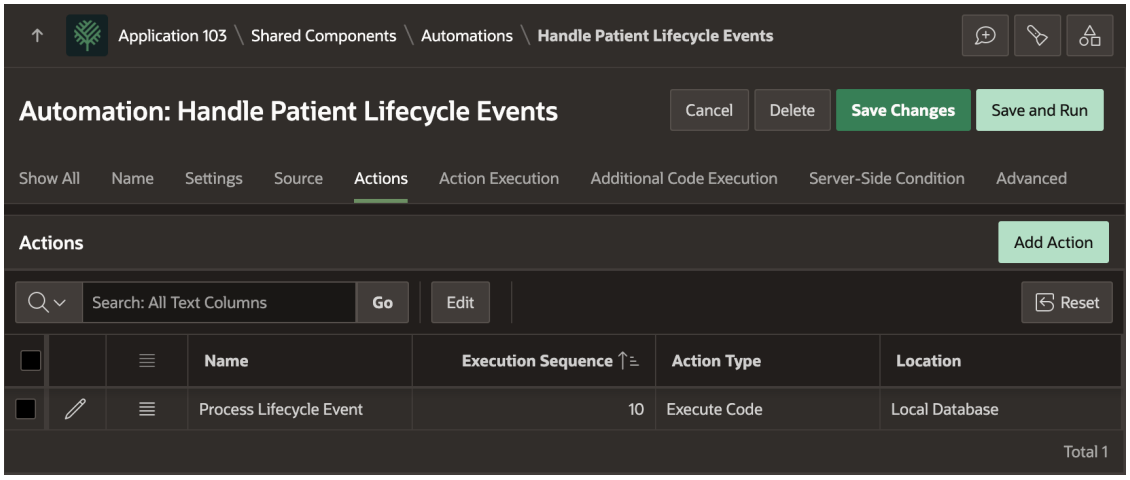
Figure 25-150 Lifecycle Events Handler Automation – Source



An automation can have multiple actions, each of which can contain a server-side condition to control whether it executes or not for any given row. However, in this simple use case, you only need a single *Process Lifecycle Event* action shown below. Like any action, it can reference the source query column values for the current row as bind variables. This action uses the *Execute Code* type to run the single line below, passing the value of the current event `ID`:

```
frc_procedure_lifecycle.process_event(:ID);
```

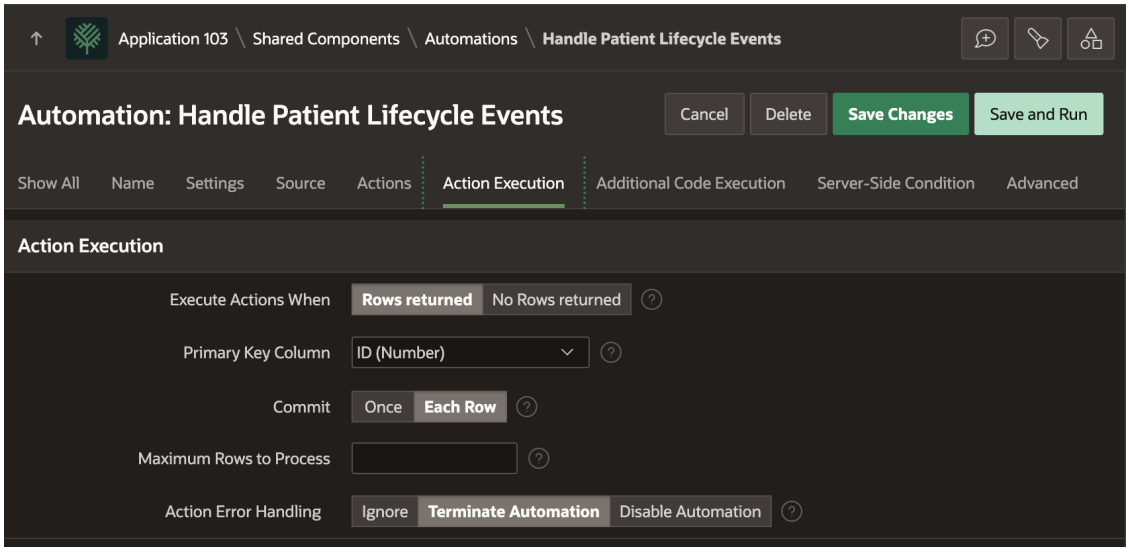
Figure 25-151 Lifecycle Events Handler Automation – Actions



On the *Action Execution* tab, as shown below, you configure error handling and commit behavior for the job. By committing after each event row is processed, you ensure that a later event's error does not rollback the successful processing of earlier events.

Copyright © 2026, Oracle and/or its affiliates.

Figure 25-152 Lifecycle Events Handler Automation – Action Execution



25.3.6.5 Continuing a Waiting Lifecycle Workflow

Continue the waiting workflow activity and pass the event type into a workflow variable.

When the `PROCESS_EVENT` procedure in the `FRC_PROCEDURE_LIFECYCLE` package executes, it calls `NOTIFY_DATA_CHANGED` shown below. This calls a local helper function `WORKFLOW_ID_FOR_WAITING_ACTIVITY` to look up the `procedure_lifecycle` workflow instance ID for the current medical procedure row.

In the process, it ensures the workflow is active and has an activity with *Static ID* `status_or_date_changed` that is waiting to continue. If such a workflow instance is found, it passes its ID to `CONTINUE_ACTIVITY` in the `APEX_WORKFLOW` package to signal the waiting activity to continue.

It passes a map of name/value pairs back to the activity including the key `V_WHAT_CHANGED` whose value is the event type passed in. Any workflow variable whose name matches a map key value is automatically assigned the corresponding value in the process. Consequently, the `V_WHAT_CHANGED` workflow variable then contains the event type and the workflow continues to the next activity.

```
-- internal in package frc_procedure_lifecycle
-------------------------------------------------------------------
-- Notify an instance of the procedure_lifecycle workflow associated
-- with the supplied patient procedure id that
-------------------------------------------------------------------
function notify_data_changed(
    p_procedure_id in number,
    p_event_type   in varchar2)
    return            boolean
is
    l_workflow_id number := workflow_id_for_waiting_activity(
                                'procedure_lifecycle',
                                'status_or_date_changed',
                                p_procedure_id);
begin
```


Copyright © 2026, Oracle and/or its affiliates.

```
    if l_workflow_id is not null then
        apex_workflow.continue_activity(
            p_instance_id     => l_workflow_id,
            p_static_id       => 'status_or_date_changed',
            p_activity_params => apex_global_application.vc_map(
                                    'V_WHAT_CHANGED' => p_event_type));
        return true;
    end if;
    return false;
end notify_data_changed;
```

The `WORKFLOW_ID_FOR_WAITING_ACTIVITY` helper function ensures the procedure lifecycle workflow for the *Details Primary Key* value passed in is active and that the desired activity is waiting. If both of these conditions hold, then it returns the ID of that workflow instance. Otherwise, it returns null.

```
-- internal in package frc_procedure_lifecycle
-------------------------------------------------------------------
-- Return the active workflow id of static def id type passed in
-- associated with provided PK and having a waiting activity with
-- supplied static id that can be continued. Otherwise, null
-------------------------------------------------------------------
function workflow_id_for_waiting_activity(
    p_workflow_static_id         in varchar2,
    p_waiting_activity_static_id in varchar2,
    p_details_pk                 in varchar2)
    return                          number
is
    l_ret number;
begin
    -- Only return the workflow id if the workflow is in ACTIVE
    -- state and the indicated continue activity is in WAITING state
    for wf in ( select workflow_id
                  from apex_workflow_activities ac
                  join apex_workflows wf using (workflow_id)
                 where wf.detail_pk              = p_details_pk
                   and wf.workflow_def_static_id = p_workflow_static_id
                   and ac.static_id              =
p_waiting_activity_static_id
                   and ac.state                  = 'WAITING'
                   and wf.state_code             = 'ACTIVE')
    loop
        l_ret := wf.workflow_id;
    end loop;
    if l_ret is null then
        apex_debug.warn('workflow_id_for_waiting_activity: No active workflow
(%s)'
                        |' with activity (%s) waiting for pk (%s)',
            p_workflow_static_id,
            p_waiting_activity_static_id,
            p_workflow_static_id);
    end if;
    return l_ret;
end workflow_id_for_waiting_activity;
```


Copyright © 2026, Oracle and/or its affiliates.

25.4 Maintaining Workflow Processes

Maintain workflows and task definitions through import, debugging, deployment, and cleanup.

As you create and enhance your workflows and task definitions to automate business processes, it's important you can:

- upgrade an existing app correctly to retain workflows and tasks,
- recognize when you are running in development mode,
- debug problems that arise during development,
- navigate the workflow and task data dictionary views,
- anticipate periodic purging of completed workflow and task instances,
- activate an *In Development* version for use in production,
- verify a workflow version is no longer used before deleting it, and
- remove development instances created during testing.

- Importing vs. Deleting an Application
  Import upgrades over the existing app to preserve production workflow and task instances.
- Recognizing Development Mode
  Recognize Development Mode by the Developer Toolbar or `APP_BUILDER_SESSION` value.
- Debugging Business Process Logic
  Use APEX debug tracing to investigate workflow and task errors.
- Accessing Workflow and Task Data
  Understand APEX dictionary views and packages for workflows, tasks, and their definitions.
- Understanding Task and Workflow Purging
  Plan for workflow and task purging by storing any permanent history in application tables.
- Activating an In Development Workflow
  Activate a workflow version before running it outside Development Mode.
- Deploying Workflows to Production
  Promote active workflow versions from development so production stays in sync.
- Deleting a Workflow with No Instances
  Check whether any workflow instances still use a version before deleting it.
- Removing Development Instances
  Remove Development Mode workflow instances and their related tasks after testing.

25.4.1 Importing vs. Deleting an Application

Import upgrades over the existing app to preserve production workflow and task instances.

Once you deploy your application using workflows and tasks into production, end users begin exercising its functionality to get their jobs done. The workflow and task instances your application creates are business critical, so they are as important as any other rows of data in your application data model. As you enhance your app and fix issues users report, you periodically *import* a new version of your application into production. When you do this, it is critical that you perform the upgrade ***without deleting the existing application first***. There is clear motivation for this guidance. When you import a new application version *over* an existing

Copyright © 2026, Oracle and/or its affiliates.

one, any existing workflow and task instances ***remain as they are***. This is ***not the case*** when you delete an application.

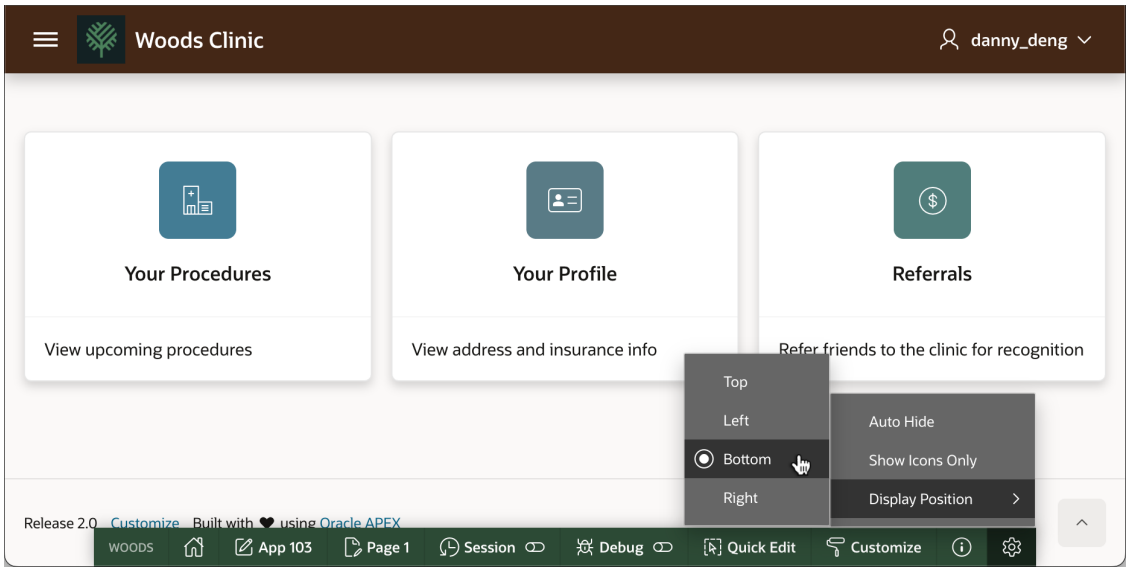
> ⚠ **Caution**
>
> When you delete an app, all its workflow and task instances are deleted, too.

25.4.2 Recognizing Development Mode

Recognize Development Mode by the Developer Toolbar or `APP_BUILDER_SESSION` value.

During development, when you run your app from App Builder, it's in "Development Mode". The *Developer Toolbar* appears when in this mode. Using its settings menu, as shown below, you can change its default display position, auto-hide behavior, and button display style.

Figure 25-153 Development Toolbar Appears When Running in Development Mode

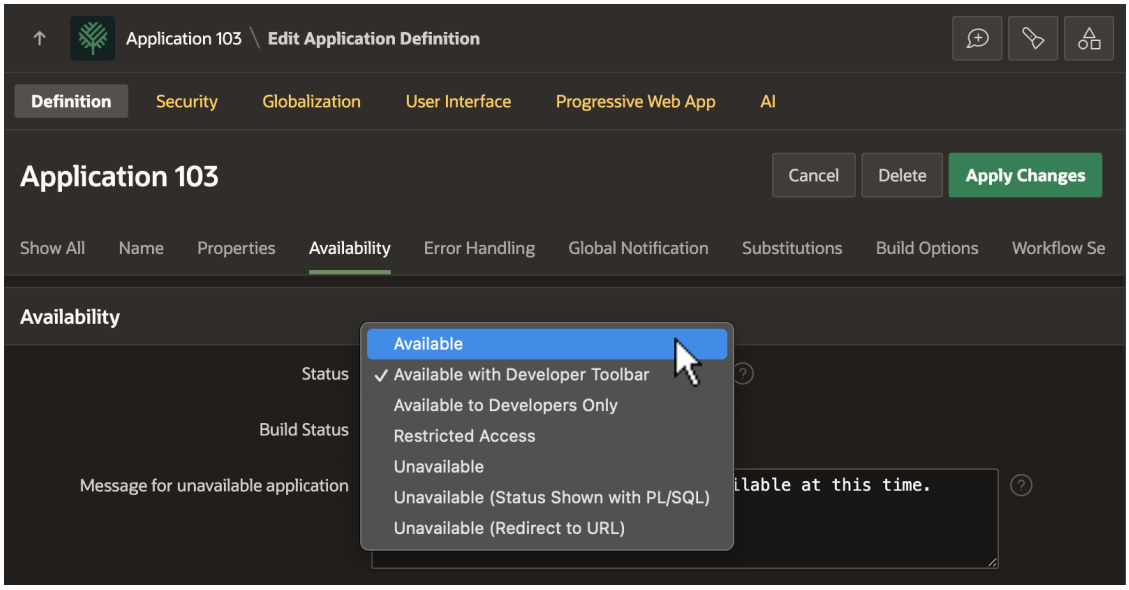


> ✓ **Tip**
>
> To test at runtime if your app is running in Development Mode, you can use the condition `V('APP_BUILDER_SESSION') IS NOT NULL`, which returns true in this mode.

You can *disable* the Developer Toolbar in Development Mode session if necessary. As shown below, change the *Availability* > *Status* setting in *Shared Components* > *Application Definition* to *Available* instead of the default *Available with Developer Toolbar*.

Copyright © 2026, Oracle and/or its affiliates.

Figure 25-154 Your Availability Setting Can Disable the Developer Toolbar



> **Tip**
>
> Check your app's *Availability* > *Status* setting if you *don't* see the Developer Toolbar when running from App Builder, but *do* want it to display. Make sure it's set to *Available with Developer Toolbar*.

25.4.3 Debugging Business Process Logic

Use APEX debug tracing to investigate workflow and task errors.

When a business process encounters an error, use APEX debug tracing to learn more information about what went wrong.

- Querying Recent Debug Messages
  Query recent debug messages to find workflow errors and related trace details.
- Enabling Debug Tracing Proactively
  Enable debug tracing from the Developer Toolbar or directly on a workflow version.
- Viewing Debug Trace Information
  View workflow debug traces by clearing page filters in the Debug Message Data window.
- Adding Custom Debug Messages
  Add custom debug messages with `APEX_DEBUG.INFO` when needed.

25.4.3.1 Querying Recent Debug Messages

Query recent debug messages to find workflow errors and related trace details.

The APEX engine logs unexpected errors. Query them using the `APEX_DEBUG_MESSAGES` view. Run the query below in the *SQL Workshop* > *SQL Commands* window to see the most recent

Copyright © 2026, Oracle and/or its affiliates.

debug messages in your workspace. These may include messages related to business process logic, or messages from page processing.

```
select apex_util.get_since(message_timestamp) as when, message
  from apex_debug_messages
 order by message_timestamp desc
```

A more detailed query against this view to see only *workflow* related messages is the following. Notice it also includes the `WORKFLOW_INSTANCE_ID` that encountered the problem:

```
select message_timestamp,
       elapsed_time,
       execution_time,
       message,
       page_id,
       workflow_instance_id,
       session_id,
       message_level,
       trim(replace(call_stack,'%')) as call_stack
 from apex_debug_messages
where application_id = :APP_ID
and apex_user = 'nobody'
and call_stack like '%\%WORKFLOW.%' escape '\'
and message_level < 9
order by message_timestamp desc
```

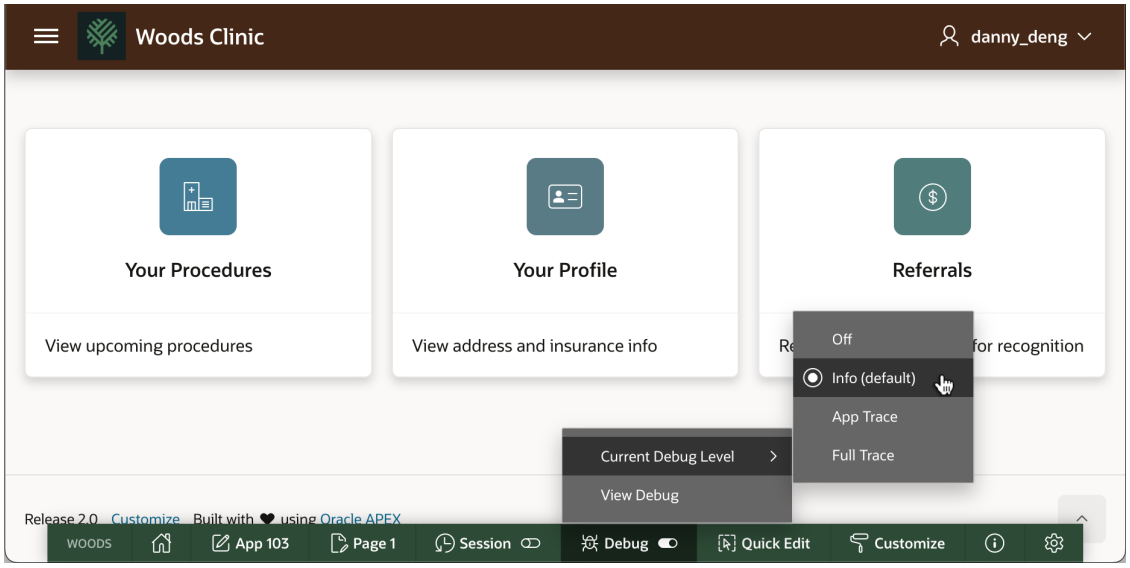
25.4.3.2 Enabling Debug Tracing Proactively

Enable debug tracing from the Developer Toolbar or directly on a workflow version.

Using the Developer Toolbar's *Debug > Current Debug Level* menu, as shown below, you can also enable debug tracing proactively. Start with the default *Info* level of detail. Other levels are increasingly verbose, but typically not required to triage most problems. This refreshes the current page and keeps your session in debug tracing mode until you turn if off by using the same menu.

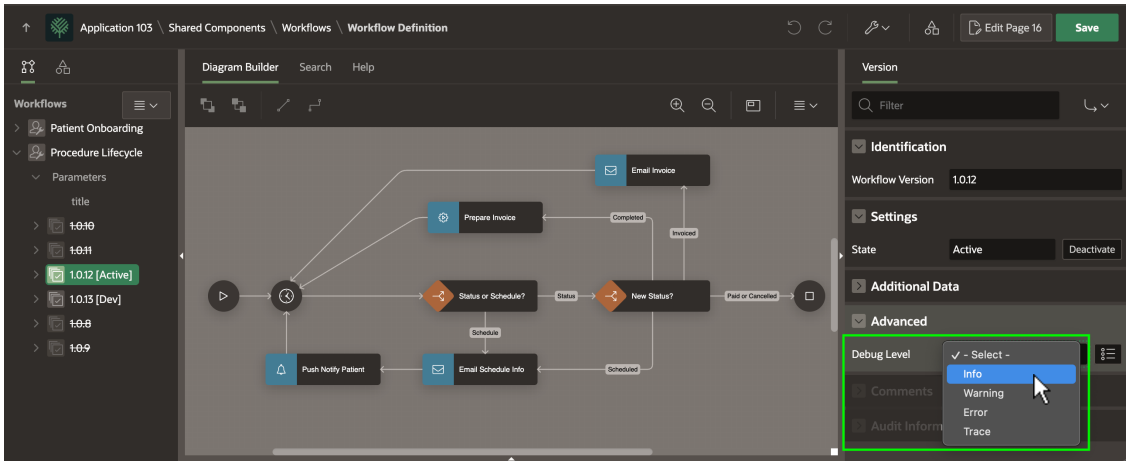
Copyright © 2026, Oracle and/or its affiliates.

Figure 25-155 Enabling Info-Level Debug Tracing with the Developer Toolbar



Since your workflows run asynchronously in the background, the error may occur without any page interaction. In this case, as shown below, you can enable the debug tracing level on a workflow version in the *Advanced* section of the Property Editor.

Figure 25-156 Enabling Debug Tracing for Background Workflow Execution



25.4.3.3 Viewing Debug Trace Information

View workflow debug traces by clearing page filters in the Debug Message Data window.

Use the Developer Toolbar's *Debug* > *View Debug* menu option to open the *Debug Message Data* window shown below. By default, the Interactive Report showing debug trace requests is filtered by the current application id and the current page. Since workflow activities execute in the background, they are not associated to a page. They also have no logged-in username. To see the workflow debug messages, as shown below, either:

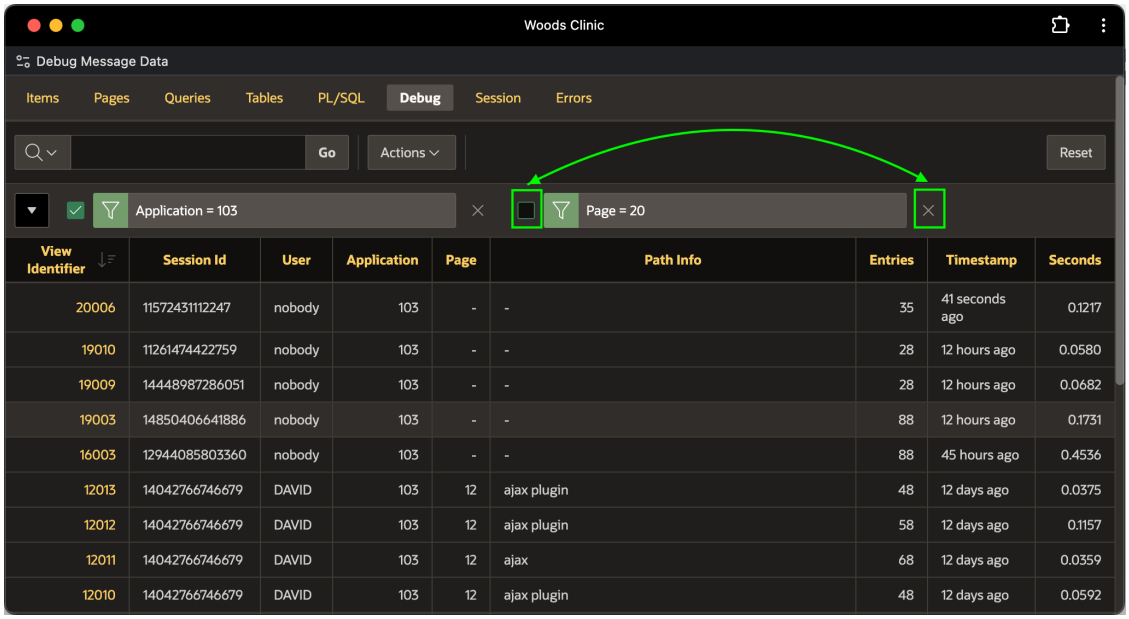
- Disable the *Page* filter by unchecking its checkbox, or

Copyright © 2026, Oracle and/or its affiliates.

- Clear the *Page* filter by clicking its "x" icon.

Then workflow messages for user `nobody` with no associated page appear.

Figure 25-157 Disabling Page Filter to View Recent Workflow Debug Requests



View Identifier	Session Id	User	Application	Page	Path Info	Entries	Timestamp	Seconds
20006	11572431112247	nobody	103	-	-	35	41 seconds ago	0.1217
19010	11261474422759	nobody	103	-	-	28	12 hours ago	0.0580
19009	14448987286051	nobody	103	-	-	28	12 hours ago	0.0682
19003	14850406641886	nobody	103	-	-	88	12 hours ago	0.1731
16003	12944085803360	nobody	103	-	-	88	45 hours ago	0.4536
12013	14042766746679	DAVID	103	12	ajax plugin	48	12 days ago	0.0375
12012	14042766746679	DAVID	103	12	ajax plugin	58	12 days ago	0.1157
12011	14042766746679	DAVID	103	12	ajax	68	12 days ago	0.0359
12010	14042766746679	DAVID	103	12	ajax plugin	48	12 days ago	0.0592

Clicking on a view identifier that occurred around the time the error happened, you can view all debug messages related to that APEX engine unit of work. As shown below, while investigating different approaches for notifying your workflow of data changes, one *earlier* approach attempted to *start* a workflow from an `AFTER INSERT` trigger. The call to `APEX_WORKFLOW.START_WORKFLOW` encountered an `ORA-04091` exception while running its *Additional Details* query. As shown below, the debug log includes:

- The query executing when the error occurred
- The `ORA-04091` exception with call stack
- A related mention of:

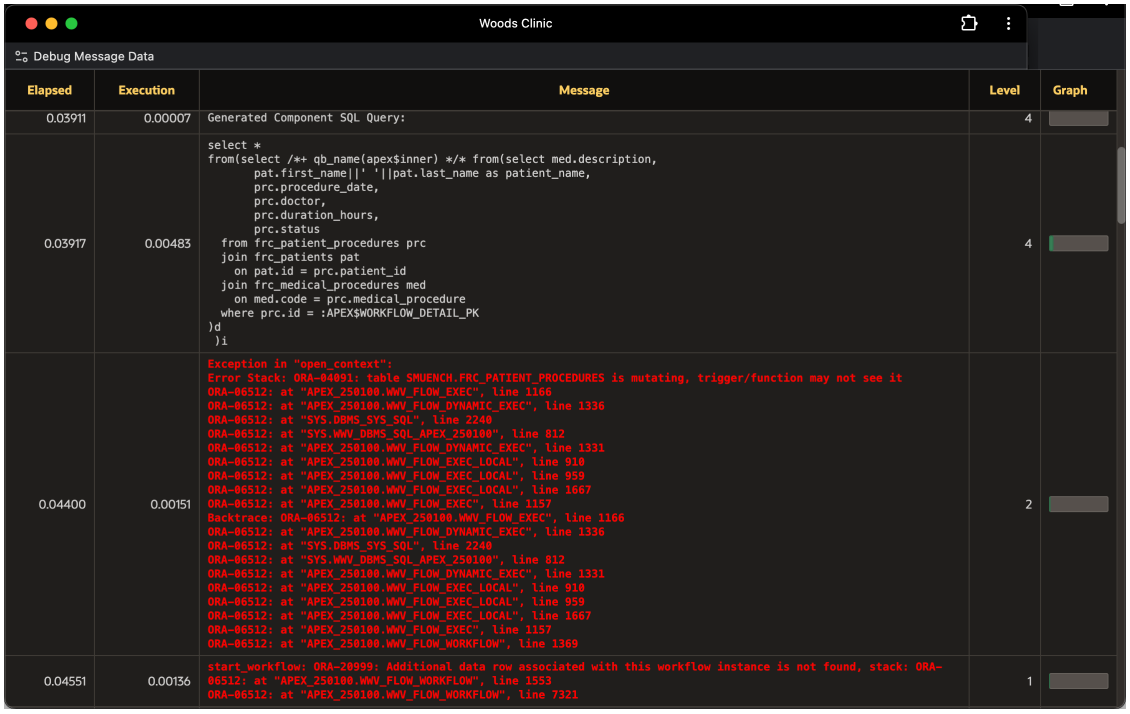
```
start_workflow: ORA-20999: Additional data row … not found
```

The debug trace log usually includes enough information to provide clues on where the issue is happening.

> **Tip**
>
> The `ORA-04091` error is known as a "Mutating Table" exception. It occurs when a database trigger initiates logic that queries the table that is in the process of being changed.

Copyright © 2026, Oracle and/or its affiliates.

Figure 25-158 Examining Debug Exception Call Stack in Debug Viewer



25.4.3.4 Adding Custom Debug Messages

Add custom debug messages with `APEX_DEBUG.INFO` when needed.

When natively available debug information does not suffice, you can add your own by calling the `INFO` procedure in the `APEX_DEBUG` package. Use it in an *Execute Code* type:

- page process in a page,
- action in a task definition or automation, or
- activity in a workflow.

To log a message, use `INFO` as shown below. Consider starting your debug message with a prefix like `###` that's easy to search for in the debug trace log. By surrounding the `%s` token with parentheses, a null value is easier to see.

```
apex_debug.info('### somevar=(%s), someexpr=(%s), anotherexpr=(%s)',
                :SOMEVAR,
                some_expression,
                another_expression);
```

The message can contain up to ten `%s` replacement tokens that are substituted sequentially by the corresponding additional values you pass as additional parameters. Alternatively the message can contain `%0`, `%1`, …, `%9` to reference the additional parameter values. For example, the following produces the same message as above:

```
apex_debug.info('### somevar=(%2), someexpr=(%0), anotherexpr=(%1)',
                some_expression,
```


Copyright © 2026, Oracle and/or its affiliates.

```
    another_expression,
    :SOMEVAR);
```

The message, including substituted values, can be up to 32727 characters. However, messages exceeding 4000 characters appear "chunked" in the debug viewer. By default, the parameter values are truncated at 1000 characters each. Pass the optional `p_max_length` parameter to specify longer values.

> **Tip**
>
> If your message substitution token appear to be ignored, check the message string more closely. It's common to accidentally type `%` alone instead of the required `%s` or `%0`, `%1`, etc.

25.4.4 Accessing Workflow and Task Data

Understand APEX dictionary views and packages for workflows, tasks, and their definitions.

APEX dictionary views shown below contain complete information about your workflow definitions and instances. The `APEX_WORKFLOW` package is your API to work programmatically with workflows.

Table 25-7 APEX Dictionary Views for Workflows

Workflow	View	Parent View
Definitions	APEX_APPL_WORKFLOWS	APEX_APPLICATIONS
Parameter Definitions	APEX_APPL_WORKFLOW_PARAMS	APEX_APPL_WORKFLOWS
Process Parameter Values	APEX_APPL_WORKFLOW_COMP_PARAMS	APEX_APPLICATION_PAGE_PROC
Versions	APEX_APPL_WORKFLOW_VERSIONS	APEX_APPL_WORKFLOWS
Variable Definitions	APEX_APPL_WORKFLOW_VARIABLES	APEX_APPL_WORKFLOW_VERSIONS
Participant Definitions	APEX_APPL_WORKFLOW_PARTICIPANTS	APEX_APPL_WORKFLOW_VERSIONS
Activity Definitions	APEX_APPL_WORKFLOW_ACTIVITIES	APEX_APPL_WORKFLOW_VERSIONS
Activity Variable Definitions	APEX_APPL_WORKFLOW_ACT_VARS	APEX_APPL_WORKFLOW_ACTIVITIES
Branches	APEX_APPL_WORKFLOW_BRANCHES	APEX_APPL_WORKFLOW_ACTIVITIES
Transitions	APEX_APPL_WORKFLOW_TRANSITIONS	APEX_APPL_WORKFLOW_ACTIVITIES
Instances	APEX_WORKFLOWS	APEX_APPL_WORKFLOW_VERSIONS
Activities	APEX_WORKFLOW_ACTIVITIES	APEX_WORKFLOWS
Activity Variables	APEX_WORKFLOW_ACTIVITY_VARS	APEX_WORKFLOW_ACTIVITIES
Variables	APEX_WORKFLOW_VARIABLES	APEX_WORKFLOWS
Parameters	APEX_WORKFLOW_PARMETERS	APEX_WORKFLOWS
Participants	APEX_WORKFLOW_PARTICIPANTS	APEX_WORKFLOWS

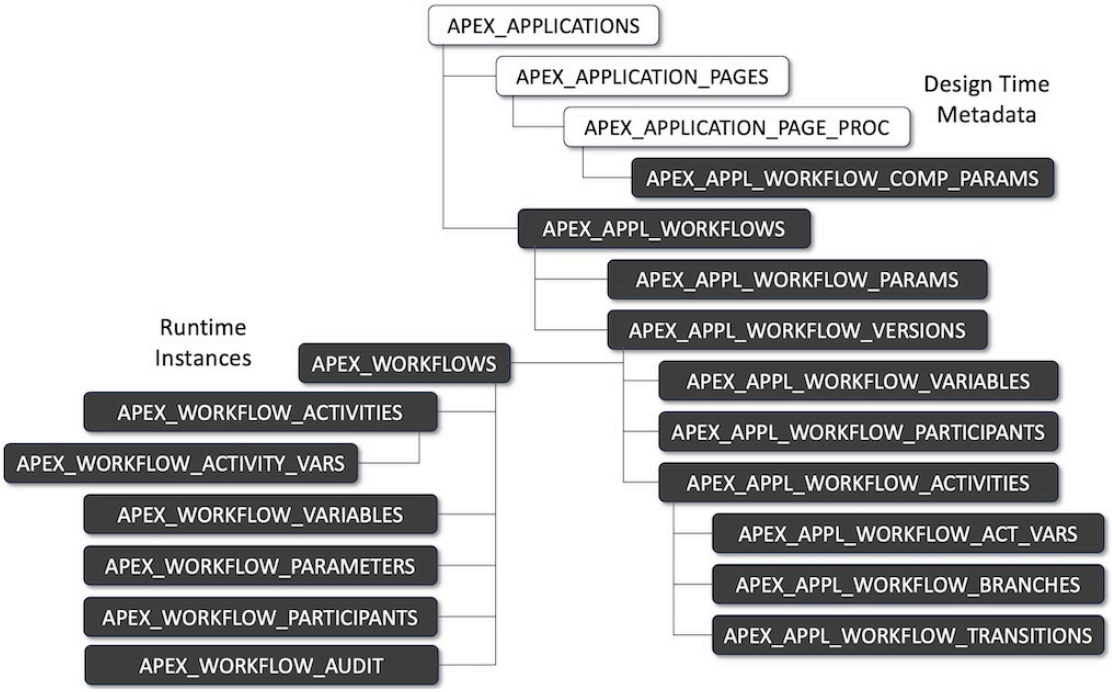
Copyright © 2026, Oracle and/or its affiliates.

Table 25-7 (Cont.) APEX Dictionary Views for Workflows

Workflow	View	Parent View
Audit	APEX_WORKFLOW_AUDIT	APEX_WORKFLOWS

The diagram shows the relationship between dictionary views related to workflow definition metadata and workflow instance data.

Figure 25-159 APEX Dictionary Views Related to Workflow Instances



The additional views shown below contain all relevant information about your task definitions and instances. The APEX_HUMAN_TASK package is your API to work programmatically with tasks.

Table 25-8 APEX Dictionary Views for Workflows

Task	View	Parent View
Definitions	APEX_APPL_TASKDEFS	APEX_APPLICATIONS
Parameter Definitions	APEX_APPL_TASKDEF_PARAMS	APEX_APPL_TASKDEFS
Participant Definitions	APEX_APPL_TASKDEF_PARTICIPANTS	APEX_APPL_TASKDEFS
Action Definitions	APEX_APPL_TASKDEF_ACTIONS	APEX_APPL_TASKDEFS
Instances	APEX_TASKS	APEX_APPL_TASKDEFS
Parameters	APEX_TASK_PARMETERS	APEX_TASKS
Participants	APEX_TASK_PARTICIPANTS	APEX_TASKS
History	APEX_TASK_HISTORY	APEX_TASKS

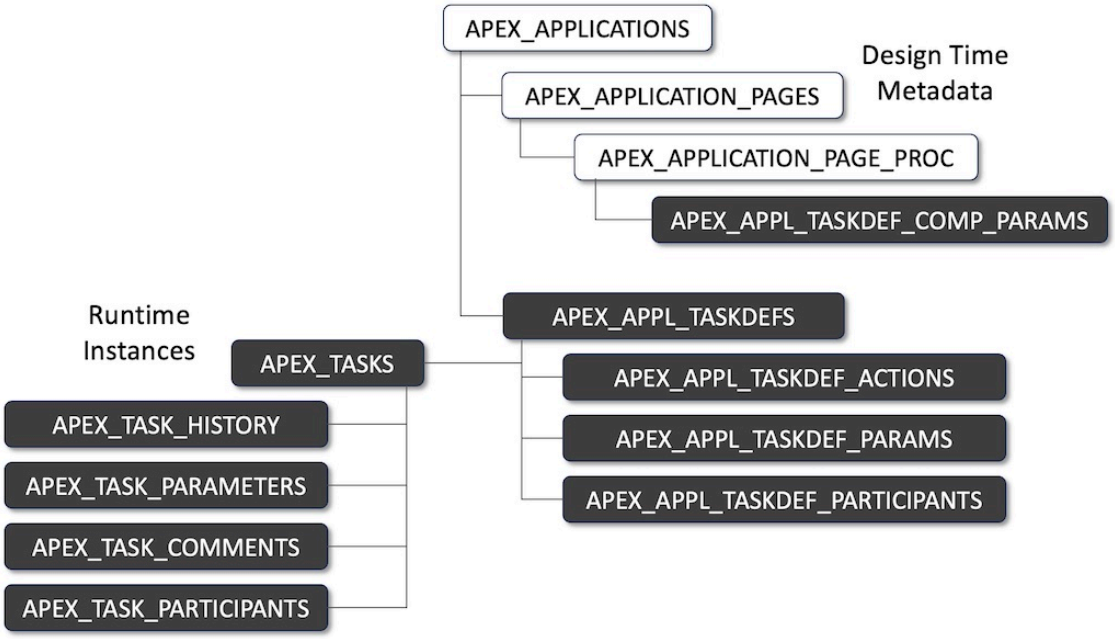
Copyright © 2026, Oracle and/or its affiliates.

Table 25-8 (Cont.) APEX Dictionary Views for Workflows

Task	View	Parent View
Comments	APEX_TASK_COMMENTS	APEX_TASKS

The diagram shows the relationship between dictionary views related to task definition metadata and task instance data.

Figure 25-160 APEX Dictionary Views Related to Task Instances



25.4.5 Understanding Task and Workflow Purging

Plan for workflow and task purging by storing any permanent history in application tables.

The APEX platform manages the workflow and task instances for all workspaces. To avoid unbounded storage requirements, a daily job enforces the retention period your instance administrator's configures for completed tasks and workflows.

The default retention period defaults to 7 days for tasks and 30 days for workflows. The administrator can extend these up to 30 days for tasks and up to 100 days for workflows. Any completed instances beyond the retention period get archived, then purged.

> **Tip**
>
> Any errored or canceled tasks are archived and removed daily. They don't have a retention period.

Given this platform policy, evaluate if any aspect of your workflow and task instance information needs to be recorded permanently in a table in your application data model. If so, some approaches to consider include:

Copyright © 2026, Oracle and/or its affiliates.

- A task action on the *Complete* event can store key info in application tables
- A workflow activity can store key info in application tables at any time
- An automation can transfer key info to application tables by querying appropriate dictionary views.

25.4.6 Activating an In Development Workflow

Activate a workflow version before running it outside Development Mode.

When you run you application in Development Mode, starting a workflow like *Procedure Lifecycle* shown below uses the *In Development* version if it exists. Otherwise, it uses the *Active* version. However an attempt to start a workflow with *only* an *In Development* version when outside of a Development Mode session produces the runtime error message:

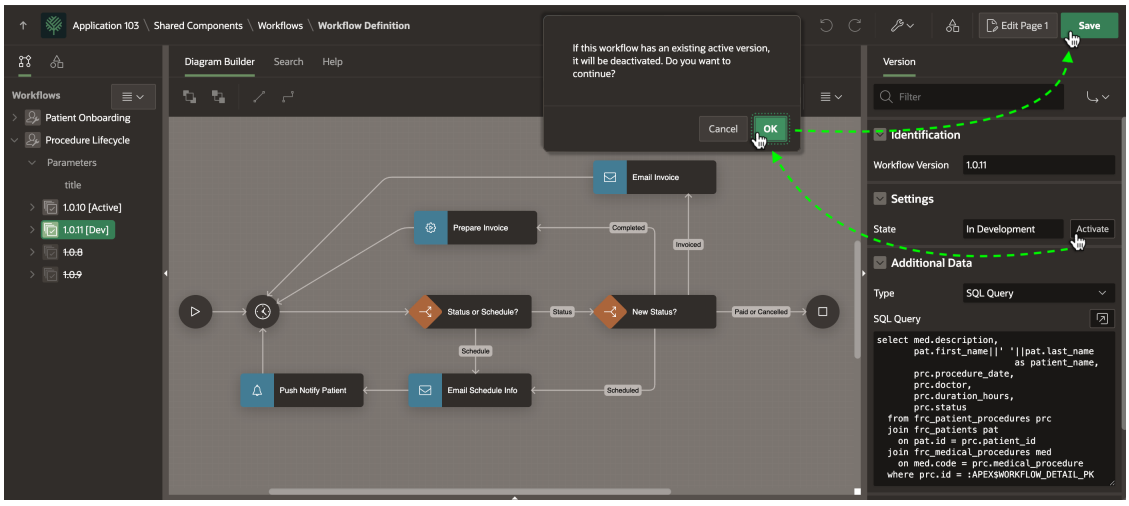
- **Workflow *your_static_id* has no Active version**

> **⚠ Caution**
>
> To use a workflow outside of a Development Mode session, you must activate a workflow version first.

To activate a version, select it in the *Workflows* tree. First, ensure it has a *Workflow Owner* participant configured. It's recommended to configure a *Workflow Administrator*, too. Then, use the *Activate* option on the context menu. Alternatively, as shown below, click the *(Activate)* button next to the *State* setting in the Property Editor. A confirmation dialog appears informing you any *currently* active version will be deactivated. This means the state of the existing *Active* workflow changes to *Inactive*.

Workflow versions in the *Inactive* state display with a strikethrough font in the *Workflows* tree like `1.0.8` and `1.0.9` below. Make any activations and deactivations permanent by clicking *(Save)* in the Diagram Builder.

Figure 25-161 Activating an In Development Workflow



Copyright © 2026, Oracle and/or its affiliates.

25.4.7 Deploying Workflows to Production

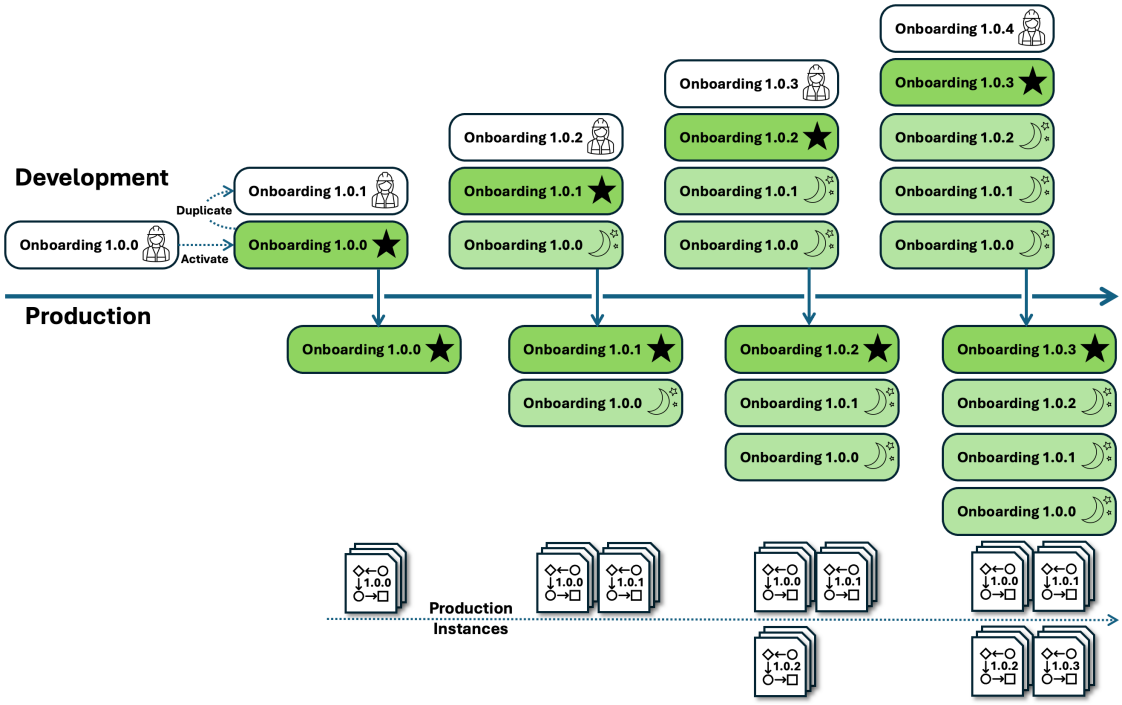
Promote active workflow versions from development so production stays in sync.

Once your workflow has an *Active* version – for example, `1.0.0` – you can promote your application to the production environment. Any *new* workflow instances created in production, use the *Active* version at their time of creation. Every workflow instance tracks the version used to create it, and ***continues*** using that version for its entire lifecycle.

Business processes change over time. So naturally, at some point new requirements compel you to duplicate *Active* version `1.0.0` to create the *In Development* version `1.0.1`. Always do this in your Development environment. You make any necessary changes to the process, testing the *In Development* version in Development Mode by running your app from App Builder. When it's ready, you activate version `1.0.1` in development.

When you next promote your application to production, new workflow instances will now use the *Active* version `1.0.1`, while previously created flows continue using the *Inactive* version `1.0.0` they were created with. As time goes on, your application accumulates new workflow versions by repeating this process over and over. You only modify *In Development* workflow versions in your Development environment, promote the application to Production, and *never* modify workflow versions in Production. This ensures your production environment is always in sync with development. The diagram illustrates how workflow versions accumulate in production over an application's lifecycle. The starred workflow version is the single active one. The versions with a moon icon are inactive, but might still be used by existing workflow instances.

Figure 25-162 Workflow Versions Accumulate Over Time



Copyright © 2026, Oracle and/or its affiliates.

> **⚠ Caution**
>
> To keep development and production environments in sync, never duplicate *Active* versions in production. Since your APEX application export contains *all* component definitions, the application you import into production technically contains the *In Development* versions, too. However, since these are only usable in Development Mode, the diagram omits them in production for clarity.

25.4.8 Deleting a Workflow with No Instances

Check whether any workflow instances still use a version before deleting it.

Workflow versions let your business process evolve over time. Every workflow instance created outside of Development Mode uses the *Active* version at the time of creation. The `WORKFLOW_VERSION` column in the `APEX_WORKFLOWS` dictionary view reflects the version each instance uses. A given workflow instance continues using the workflow version it was created with throughout its lifecycle.

If you activate a new version, this only affects new workflow instances. An *Inactive* version does not mean an *Unused* version. It only means no new workflow instances will use that inactive version.

In addition to deactivating previous workflow versions, the Workflow Builder also lets you delete workflow versions. However, you must ensure no workflow instances are using that version any more before you delete it. Before deleting a workflow, run the following query in the *SQL Workshop* > *SQL Commands* page to verify that the version you are considering deleting is no longer in use.

```
-- list procedure_lifecycle workflow versions in use
select distinct workflow_version
 from apex_workflows
where application_id = 103
  and workflow_def_static_id = 'procedure_lifecycle'
order by workflow_version
```

> **⚠ Caution**
>
> Avoid accidentally deleting a workflow version that is still in use by running the above query first!

25.4.9 Removing Development Instances

Remove Development Mode workflow instances and their related tasks after testing.

In Development Mode, starting a workflow uses the *In Development* version if available. Otherwise, the *Active* version is used. During development, you typically create many workflow instances for testing purposes. When you want to clean up these testing artifacts, call the `REMOVE_DEVELOPMENT_INSTANCES` procedure in the `APEX_WORKFLOW` package. It immediately purges any workflow instances created in Development Mode *without* archiving them. Since task instances created by workflows are dependent artifacts, this procedure also immediately purges any tasks related to these deleted workflow instances without archiving them.

Copyright © 2026, Oracle and/or its affiliates.

Task instances created outside of Development Mode cannot be deleted. The workflow administrator can cancel them, and the daily job that purges canceled workflows and tasks removes them within 24 hours.

Tip

Since task and workflow instances are removed when an application is deleted, you can use a *Working Copy* of an application to perform tests that might result in creating new workflow and task instances. When you delete the working copy, any workflow and task instances it created are deleted, too.

Copyright © 2026, Oracle and/or its affiliates.

Glossary

Copyright © 2026, Oracle and/or its affiliates.

Index

Copyright © 2026, Oracle and/or its affiliates.