Oracle® JavaScript Extension Toolkit (JET)

Developing Applications with Oracle JET

4.1.0
E90638-01
November 2017

Oracle JavaScript Extension Toolkit (JET) Developing Applications with Oracle JET, 4.1.0

E90638-01

Copyright © 2014, 2017, Oracle and/or its affiliates. All rights reserved.

Primary Author: Michele Whittaker

Contributing Authors: Walter Egan, Krithika Gangadhar, Vaibhav Gupta, Sandhya Hombardi

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 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 installed on the hardware, and/or documentation, delivered to U.S. Government end users are "commercial computer software" pursuant to the applicable Federal Acquisition Regulation and agency-specific supplemental regulations. As such, use, duplication, disclosure, modification, and adaptation of the programs, including any operating system, integrated software, any programs installed on the hardware, and/or documentation, shall be subject to license terms and license restrictions applicable to the programs. 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 and Java are registered trademarks of Oracle and/or its affiliates. Other names may be trademarks of their respective owners.

Intel and Intel Xeon 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, Opteron, the AMD logo, and the AMD Opteron 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.

Contents

Preface

Audience xiii
Documentation Accessibility xiii
Related Resources xiii
Conventions xiv

What’s New in This Guide for Release 4.1.0

1 About Oracle JavaScript Extension Toolkit (JET)

The Oracle JET Architecture 1-1
What's Included in Oracle JET 1-4

2 Getting Started with Oracle JET Application Development

Typical Workflow for Getting Started with Oracle JET Application Development 2-1
Prerequisites for Developing Applications with Oracle JET 2-2
- Prerequisite Knowledge for Working with Oracle JET 2-2
- Choose a Development Environment 2-3
- Install the Prerequisite Packages 2-4
  - Install Node.js 2-4
  - Install the Oracle JET Command-Line Interface 2-4

Getting Started with Oracle JET Web Application Development 2-6
- Create a Web Application Using the Oracle JET Command-Line Interface 2-7
  - Scaffold a Web Application with the Oracle JET CLI 2-7
  - Build a Web Application with the Oracle JET CLI 2-13
  - Serve a Web Application with the Oracle JET CLI 2-15
- Create a Web Application Using the oraclejet.zip Download 2-18
  - Download the Oracle JET Zip File 2-19
  - Create an Oracle JET Application in the NetBeans IDE 2-19
  - Include References to CSS Files in Your Oracle JET Application 2-21
  - Use RequireJS to Manage Library, Link, and Script References 2-22

Specify Library, Link, and Script References Without RequireJS 2-22
Create a Web Application Using the Oracle JET Starter Templates 2-24
Downloading Oracle JET with a Starter Template 2-24
Loading the Oracle JET Starter Templates 2-24
Working with the Oracle JET Web Nav Drawer Starter Template 2-29
Add Oracle JET to an Existing JavaScript Application 2-34
Getting Started with Oracle JET Hybrid Mobile Application Development 2-35
Install the Mobile Tooling 2-36
Install Apache Cordova 2-36
Install Android Development Tools 2-37
Install iOS Development Tools 2-42
Install Windows Development Tools 2-42
Create a Hybrid Mobile Application 2-45
Scaffold a Hybrid Mobile Application with the Oracle JET CLI 2-46
Build a Hybrid Mobile Application with the Oracle JET CLI 2-50
Serve a Hybrid Mobile Application with the Oracle JET CLI 2-54
Review Your Application Settings in the config.xml File 2-59
Change the Splash Screen and App Launcher Icon 2-60
Accessing Mobile Device Services Using Cordova Plugins 2-62
About Apache Cordova and Cordova Plugins 2-62
Using a Plugin in Your App 2-63
Cordova Plugins Recommended by Oracle JET 2-64
Using a Different Web View in your JET Hybrid Mobile App 2-64
Using the Native Platform's Date Picker UI in your JET Hybrid Mobile App 2-65
Loading Oracle JET from CDN 2-65

3 Designing Responsive Applications

Typical Workflow for Designing Responsive Applications in Oracle JET 3-1
Oracle JET and Responsive Design 3-2
Media Queries 3-2
Oracle JET Flex, Grid, Form, and Responsive Helper Class Naming Convention 3-4
Oracle JET Flex Layouts 3-5
Oracle JET Grids 3-8
About the Grid System 3-8
The Grid System and Printing 3-9
Grid Convenience Classes 3-11
Responsive Layout and Content Design Patterns 3-13
Responsive Form Layouts 3-17
Adding Responsive Design to Your Application 3-19
Using Responsive JavaScript 3-19

The Responsive JavaScript Classes 3-20
Changing a Custom Element's Attribute Based on Screen Size 3-21
Conditionally Loading Content Based on Screen Size 3-22
Creating Responsive Images 3-23
Using the Responsive Helper Classes 3-23
Creating Responsive CSS Images 3-24
Changing Default Font Size 3-25
Changing Default Font Size Across the Application 3-25
Changing Default Font Size Based on Device Type 3-26
Controlling the Size and Generation of the CSS 3-26

4 Using RequireJS for Modular Development

Typical Workflow for Using RequireJS 4-1
Oracle JET and RequireJS 4-2
Oracle JET Module Organization 4-2
Using RequireJS in an Oracle JET Application 4-4
Adding Third-Party Tools or Libraries to Your Oracle JET Application 4-7
Troubleshooting RequireJS in an Oracle JET Application 4-9
Using JavaScript Partitions and RequireJS in an Oracle JET Application 4-9

5 Creating Single-Page Applications

Typical Workflow for Creating Single-Page Applications in Oracle JET 5-1
Designing Single-Page Applications Using Oracle JET 5-1
Understanding Oracle JET Support for Single-Page Applications 5-2
Creating a Single-Page Application in Oracle JET 5-3
Using ojModule 5-14
Working with ojModule's ViewModel Life Cycle 5-17

6 Understanding Oracle JET User Interface Basics

Typical Workflow for Working with the Oracle JET User Interface 6-1
About the Oracle JET User Interface 6-1
Identifying Oracle JET UI Components, Patterns, and Utilities 6-2
Common Functionality in Oracle JET Components 6-2
Oracle JET Reserved Namespaces and Prefixes 6-5
Adding an Oracle JET Custom Element to Your Page 6-5
Adding Animation Effects 6-6

7 Working with Oracle JET User Interface Components

Typical Workflow for Working with Oracle JET UI Components 7-1
Working with Collections 7-2
Choosing a Table or Data Grid 7-2
Working with Data Grids 7-4
Working with oj. CubeDataGridDataSource 7-6
Working with List Views 7-11
Understanding Data Requirements for List Views 7-11
Working with List Views and Knockout Templates 7-14
Working with Pagination 7-15
Working with Row Expanders 7-17
Working with Tables 7-20
Understanding oj-table and Sorting 7-22
Working with Tree Views 7-24
Working with Controls 7-29
Working with Buttons 7-29
Working with Button Sets 7-30
Working with Conveyor Belts 7-32
Working with File Picker 7-33
Working with Film Strips 7-35
Configuring Film Strips 7-36
Working with Menus 7-38
Working with oj-menu 7-38
Working with Menu Buttons 7-39
Working with Context Menus 7-41
Working with Progress Indicators 7-43
Working with Tags 7-44
Working with Toolbars 7-44
Working with Trains 7-45
Working with Forms 7-46
Working with Checkbox and Radio Sets 7-47
Working with Color Pickers 7-48
Working with oj-color-palette 7-48
Working with oj-color-spectrum 7-50
Working with Comboboxes 7-52
Working with Form Controls 7-55
Working with Form Layout Features 7-56
Working with Input Components 7-56
Working with Labels 7-58
Working with Select 7-59

Working with Sliders 7-63
About the oj-slider Component 7-63
Creating Sliders 7-64
Formatting Tips for oj-slider 7-65
Working with Switches 7-66
Working with Validation and User Assistance 7-67
Working with Layout and Navigation 7-68
Working with Accordions 7-68
Working with Collapsibles 7-69
Working with Dialogs 7-70
Working with Masonry Layouts 7-71
Configuring Masonry Layouts 7-72
Understanding the oj-masonry-layout Layout Process 7-74
oj-masonry-layout Size Style Classes 7-74
Working with Nav Lists 7-74
Understanding Data Requirements for Nav Lists 7-75
Working with Nav Lists and Knockout Templates 7-77
Working with offCanvasUtils 7-78
Configuring an Off-Canvas Partition 7-79
Working with Panels 7-79
Working with Popups 7-80
Working with oj-popup 7-81
Working with the Oracle JET Popup Framework 7-82
Working with Tab Bars 7-85
Working with Visualizations 7-89
Choosing a Data Visualization Component for Your Application 7-89
Using Attribute Groups With Data Visualization Components 7-95

8 Working with Oracle JET Composite Components

Typical Workflow for Working with Oracle JET Composite Components 8-1
About Composite Components 8-1
Creating Composite Components 8-6
Recommended Standard Patterns and Coding Practices 8-14
Testing Composite Components 8-18
Adding Composite Components to Your Page 8-18

9 Using the Common Model and Collection API

Typical Workflow for Binding Data in Oracle JET 9-1
About Oracle JET Data Binding 9-1

Using the Oracle JET Common Model and Collection Framework 9-2
About the Oracle JET Common Model and Collection API 9-2
About Oracle JET Data Binding and Knockout 9-3
Using the Oracle JET Common Model and Collection API 9-3
Integrating REST Services 9-5
About Oracle JET Support for Integrating REST Services 9-5
Passing Custom AJAX Options in Common Model CRUD API calls 9-5
Supplying a customURL Callback Function 9-6
Replacing oj.sync or oj.ajax Functions 9-7
Creating a CRUD Application Using Oracle JET 9-8
Defining the ViewModel 9-8
Reading Records 9-15
Creating Records 9-15
Updating Records 9-18
Deleting Records 9-22

10 Validating and Converting Input

Typical Workflow for Validating and Converting Input 10-1
About Oracle JET Validators and Converters 10-2
About Validators 10-2
Oracle JET Validators 10-3
Oracle JET Component Validation Attributes 10-3
Oracle JET Component Validation Methods 10-4
About Converters 10-4
Using Oracle JET Converters 10-5
Using Oracle JET Converters with Oracle JET Components 10-5
Understanding Oracle JET Converters Lenient Parsing 10-9
Understanding Time Zone Support in Oracle JET 10-9
Using Custom Converters in Oracle JET 10-12
Using Oracle JET Converters Without Oracle JET Components 10-16
Using Oracle JET Validators 10-17
Using Oracle JET Validators with Oracle JET components 10-18
Using Custom Validators in Oracle JET 10-21

11 Working with User Assistance

Typical Workflow for Working with User Assistance 11-1
Understanding Oracle JET's Messaging APIs on Editable Components 11-2
Oracle JET Editable Component Messaging Attributes 11-3
Oracle JET Component Messaging Methods 11-3

Understanding How Validation and Messaging Works in Oracle JET Editable Components 11-4
Understanding How an Oracle JET Editable Component Performs Normal Validation 11-5
Normal Validation Process When User Changes Value of an Editable Component 11-5
Normal Validation Process When Validate() is Called on Editable Component 11-6
Understanding How an Oracle JET Editable Component Performs Deferred Validation 11-7
Deferred Validation Process When an Oracle JET Editable Component is Created 11-7
Deferred Validation Process When value Property is Changed Programmatically 11-7
Using Oracle JET Messaging 11-8
Notifying an Oracle JET Editable Component of Business Validation Errors 11-8
Using the messages-custom Attribute 11-8
Using the showMessages() Method on Editable Components 11-10
Configuring an Editable Component's oj-label Help Attribute 11-10
Configuring an Editable Component's help.instruction Attribute 11-11
Controlling the Display of Hints, Help, and Messages 11-13

12 Developing Accessible Applications

Typical Workflow for Developing Accessible Oracle JET Applications 12-1
Oracle JET and Accessibility 12-1
Using the Accessibility Features of Oracle JET Components 12-2
Creating Accessible Oracle JET Pages 12-3
Configuring WAI-ARIA Landmarks 12-3
Configuring High Contrast Mode 12-5
Understanding Color and Background Image Limitations in High Contrast Mode 12-6
Adding High Contrast Mode to Your Oracle JET Application 12-6
Adding High Contrast Images or Icon Fonts 12-7
Testing High Contrast Mode 12-7
Hiding Screen Reader Content 12-8

13 Internationalizing and Localizing Applications

Typical Workflow for Internationalizing and Localizing Oracle JET Applications 13-1
About Internationalizing and Localizing Oracle JET Applications 13-1
Internationalizing and Localizing Oracle JET Applications 13-3
Using Oracle JET's Internationalization and Localization Support 13-3

Enabling Bidirectional (BiDi) Support in Oracle JET 13-5
Setting the Locale Dynamically 13-6
Working with Currency, Dates, Time, and Numbers 13-9
Working with Oracle JET Translation Bundles 13-9
About Oracle JET Translation Bundles 13-9
Adding Translation Bundles to Oracle JET 13-12

14 Theming Applications

Typical Workflow for Theming an Oracle JET Application 14-1
CSS Files Included With Oracle JET 14-2
DOCTYPE Requirement 14-3
ThemeUtils 14-3
Setting Text Direction 14-3
Customizing Themes Using the Tooling 14-4
Sass Files, Variables, and Tools 14-6
SCSS File Organization and Naming Convention 14-6
SCSS Variables 14-7
Using Variables to Control CSS Content 14-8
Understanding Right-to-Left Behavior 14-9
Understanding Oracle JET Theming For Compatibility 14-9
SCSS Tools 14-12
Oracle JET Selectors 14-12
Component and Pattern Selectors 14-13
Marker Classes 14-13
Working with Images 14-16
Image Considerations 14-16
Icon Fonts 14-16
Image Files 14-17

15 Securing Applications

Typical Workflow for Securing Oracle JET Applications 15-1
About Securing Oracle JET Applications 15-1
Oracle JET Components and Security 15-2
Oracle JET Security and Developer Responsibilities 15-2
Oracle JET Security Features 15-2
Using oj.OAuth in Your Oracle JET Application 15-4
Initializing oj.OAuth 15-4
Verifying oj.OAuth Initialization 15-5
Obtaining the OAuth Header 15-5

Using oj.OAuth with Oracle JET Common Model 15-6
Embedding oj.OAuth in Your Application's ViewModel 15-6
Adding oj.OAuth as a Plugin in Your ViewModel 15-7
Integrating oj.OAuth with Oracle Identity Management (iDM) Server 15-8
About Securing Hybrid Mobile Applications 15-9
Managing App Configuration for JET Hybrid Mobile Apps 15-9
Dealing With Cross-Origin Resource Sharing (CORS) 15-9

16 Optimizing Performance

Typical Workflow for Optimizing Performance of Oracle JET Applications 16-1
About Performance and Oracle JET Applications 16-1
Adding Performance Optimization to an Oracle JET Application 16-2

17 Testing and Debugging

Typical Workflow for Testing and Debugging an Oracle JET Application 17-1
Testing Oracle JET Applications 17-1
Testing Applications 17-2
Testing Hybrid Mobile Applications 17-2
Using oj.BusyContext API in Automated Testing 17-3
Debugging Oracle JET Applications 17-7
Debugging Web Applications 17-7
Debugging Hybrid Mobile Applications 17-8

18 Packaging and Deploying Applications

Typical Workflow for Packaging and Deploying Applications 18-1
Packaging and Deploying Web Applications 18-1
Packaging Web Applications 18-2
Deploying Web Applications 18-2
Packaging and Publishing Hybrid Mobile Applications 18-2
About Packaging and Publishing Hybrid Mobile Applications 18-3
Packaging a Hybrid Mobile App on Android 18-3
Packaging a Hybrid Mobile App on iOS 18-4
Packaging a Hybrid Mobile App on Windows 18-6
How to Create the Build Configuration File to Package Your App on Windows 18-6
How to Build Your App for Windows 18-7
Removing Build Output and Extraneous Files from Your JET Application’s Source 18-8

A Troubleshooting

B Oracle JET v4.1.0 Tooling Migration

Migrating a v3.x.0 Application to v4.1.0 B-1
Migrating a v4.0.0 Application to v4.1.0 B-3

C Oracle JET References

Oracle Libraries and Tools C-1
Third-Party Libraries and Tools C-1

Preface

Developing Applications with Oracle JET describes how to build responsive web and hybrid mobile applications using Oracle JET.

Topics:

- Audience
- Documentation Accessibility
- Related Resources
- Conventions

Audience

Developing Applications with Oracle JET is intended for intermediate to advanced JavaScript developers who want to create pure client-side, responsive web or hybrid mobile applications based on JavaScript, HTML5, and CSS3.

Documentation Accessibility

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.

Access to Oracle Support

Oracle customers that have purchased support have access to electronic support through My Oracle Support. For information, visit http://www.oracle.com/pls/topic/lookup?ctx=acc&id=info or visit http://www.oracle.com/pls/topic/lookup?ctx=acc&id=trs if you are hearing impaired.

Related Resources

For more information, see these Oracle resources:

- Oracle JET Web Site
- JavaScript API Reference for Oracle® JavaScript Extension Toolkit (JET)
- Oracle® JavaScript Extension Toolkit (JET) Keyboard and Touch Reference
- Oracle® JavaScript Extension Toolkit (JET) Styling Reference

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.

What's New in This Guide for Release 4.1.0

For Oracle JET Release 4.1.0 this guide has been updated in several ways. The following table lists the sections that are new or changed since Oracle JET Release 4.0.0.

For changes made to Oracle JavaScript Extension Toolkit (JET) for this release, see the product Release Notes.

Chapter	Link	Changes Made
Chapter 2 Getting Started with Oracle JET Application Development	Getting Started with Oracle JET Application Development	Chapter updated to reflect the following changes: • Loading Oracle JET from CDN updated to describe changes you must make to your RequireJS build configuration if you're building your application in release mode and use the `r.js` optimizer. • Install Android Development Tools updated to include a new sub-section that describes the environment variables you must set or review to develop a hybrid mobile application using Oracle JET.
Chapter 7 Working with Oracle JET User Interface Components	Working with Oracle JET User Interface Components	Chapter updated to reflect the following changes: • Working with Tab Bars updated to reflect new add, remove, and reorder capability. • Working with Tables updated to reflect removal of `summary` attribute and change to `aria-label` for accessibility.
Chapter 8 Working with Oracle JET Composite Components	Working with Oracle JET Composite Components	Chapter updated to reflect the following changes: • Creating Composite Components updated to expand the list of attribute, property, event listener, and method names that you shouldn't use in your composite components to avoid naming collisions. • About Composite Components updated to reflect name change of `oj-slot` to `oj-bind-slot`.
Chapter 14 Theming Applications	Theming Applications	Chapter updated to describe how to overwrite default SVG images using JET tooling. See Image Files.
Appendix Oracle JET v4.1.0 Tooling Migration	Oracle JET v4.1.0 Tooling Migration	Version update. New topic also added to describe steps to migrate from Oracle JET `v4.0.0` to `v4.1.0`.

1

About Oracle JavaScript Extension Toolkit (JET)

Oracle JET is a collection of Oracle and open source JavaScript libraries engineered to make it as simple and efficient as possible to build client-side web and hybrid mobile applications based on JavaScript, HTML5, and CSS.

Oracle JET is designed to meet the following application needs:

- Add interactivity to an existing page.
- Create a new end-to-end client-side web application using JavaScript, HTML5, CSS, and best practices for responsive design.
- Create a hybrid mobile application that looks and feels like a native iOS, Android or Windows application.

Topics:

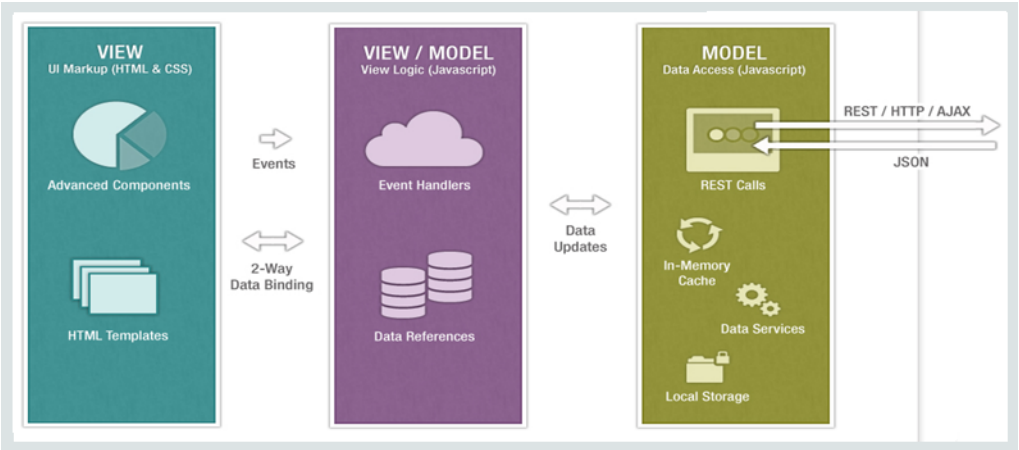
- The Oracle JET Architecture
- What's Included in Oracle JET

You can also view videos that provide an introduction to Oracle JET in the Oracle JET Videos collection.

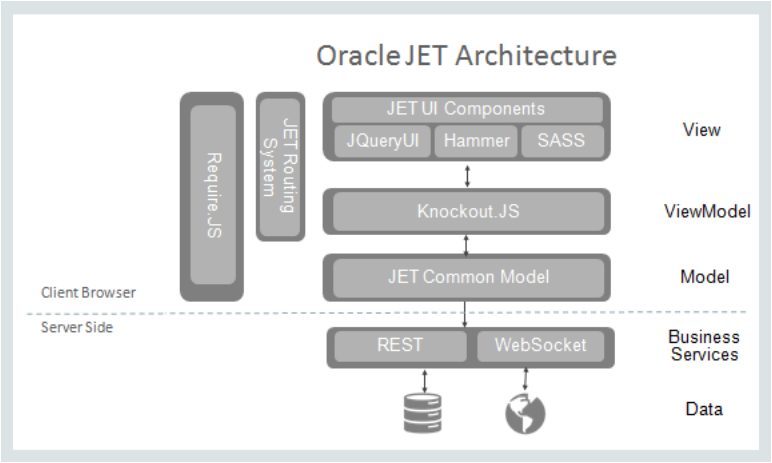
The Oracle JET Architecture

Oracle JET supports the Model-View-ViewModel (MVVM) architectural design pattern.

In MVVM, the Model represents the application data, and the View is the presentation of the data. The ViewModel exposes data from the Model to the view and maintains the application's state.



To support the MVVM design, Oracle JET is built upon a modular framework that includes a collection of third-party libraries and Oracle-provided files, scripts, and libraries.



The Oracle JET Common Model and Collection Application Programming Interface (API) implements the model layer. The API includes the following JavaScript objects:

- `oj.Model`: Represents a single record data from a data service such as a RESTful web service
- `oj.Collection`: Represents a set of data records and is a list of `oj.Model` objects of the same type
- `oj.Events`: Provides methods for event handling
- `oj.KnockoutUtils`: Provides methods for mapping the attributes in an `oj.Model` or `oj.Collection` object to Knockout observables for use with component view models.

To implement the View layer, Oracle JET provides a collection of UI components implemented as HTML5 custom elements, ranging from basic buttons to advanced data visualization components such as charts and data grids.

Knockout.js implements the ViewModel and provides two-way data binding between the view and model layers.

Oracle JET Features

Oracle JET features include:

- Messaging and event services for both Model and View layers
- Validation framework that provides UI element and component validation and data converters
- Caching services at the Model layer for performance optimization of pagination and virtual scrolling
- Filtering and sorting services provided at the Model layer
- Connection to data sources through Web services, such as Representational State Transfer (REST) or WebSocket

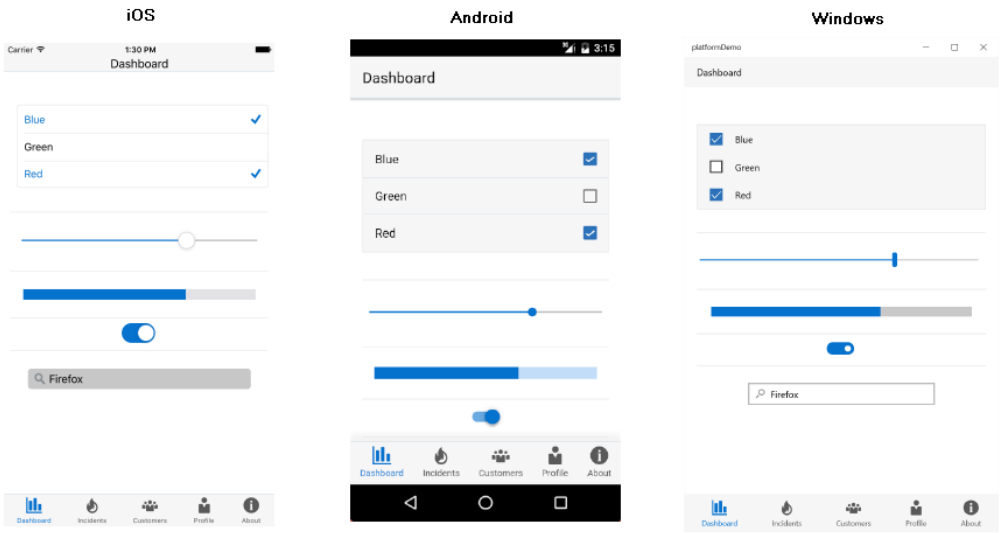
- Management of URL and browser history using Oracle JET `oj.Router` and `ojModule` components
- Integrated authorization through OAuth 2.0 for data models retrieved from REST Services
- Resource management provided by RequireJS
- API compatibility with Backbone.js Model, Collection, and Events classes, except for Backbone.js Underscore methods.
- JavaScript logging
- Popup UI handling

Hybrid Mobile Application Development Toolkit Features

Oracle JET includes support for hybrid mobile applications that run on iOS, Android, and Windows mobile devices within the Apache Cordova container.

Apache Cordova enables you to use web technologies such as HTML5, CSS, and JavaScript to develop applications that you can deploy to mobile devices. Using the Cordova JavaScript APIs to access native device services, major mobile platforms such as Android, iOS, and Windows can be supported from a common code base.

The following image shows the same application code rendered on an iOS, Android, and Windows mobile device. Oracle JET automatically applies the native theme when you configure the application for the desired platform.



To support hybrid mobile development, Oracle JET includes the following features:

- Native themes that you can use with Cordova to create hybrid mobile applications that emulate the look and feel of native iOS, Android, and Windows mobile devices
- Tooling that enables you to scaffold and build Cordova-based hybrid applications using Oracle JET directly from the command line
- Code samples, applications, and demos that you can use to create hybrid mobile applications using Oracle JET best practices

- Mobile UI behaviors and interactive gestures such as animation, sticky headers, pull to refresh, and swipe to reveal

Oracle JET Visual Component Features

Oracle JET visual components include the following features and standards compliance:

- Compliance with Oracle National Language Support (NLS) standards (i18n) for numeric, currency, and date/time formatting
- Built-in theming supporting the Oracle Alta UI style specifications
- Support for Oracle software localization standards, l10n, including:
  - Lazy loading of localized resource strings at run time
  - Oracle translation services formats
  - Bidirectional locales (left-to-right, right-to-left)
- Web Content Accessibility Guidelines (WCAG) 2.0. In addition, components provide support for high contrast and keyboard-only input environments.
- Gesture functionality by touch, mouse, and pointer events where appropriate
- Support for Oracle test automation tooling
- Responsive layout framework

What's Included in Oracle JET

The Oracle JET zip distribution includes Oracle JET libraries and all third party libraries that the toolkit uses.

Specifically, Oracle JET includes the following files and libraries:

- CSS and SCSS files for the Alta theme
- Minified and debug versions of the Oracle JET libraries
- Data Visualization Tools (DVT) CSS and JavaScript
- Knockout and Knockout Mapping libraries
- jQuery libraries
- RequireJS, RequireJS `text` plugin, and RequireJS CSS plugin
- js-signals
- `es6-promise` polyfill
- Hammer.js

  Oracle JET components use Hammer.js internally for gesture support. Do not add to Oracle JET components or their associated DOM nodes.
- Oracle JET `dnd-polyfill` HTML5 drag and drop polyfill
- proj4js library
- webcomponentsjs polyfill

2

Getting Started with Oracle JET Application Development

Use Oracle JET to create web and hybrid mobile applications. The toolkit provides customizable Alta themes for desktop platforms and mobile devices. The toolkit also includes Oracle JET components, Starter Templates, UI design patterns, tooling, and a mechanism to load content from the Oracle Content Delivery Network (CDN).

Topics

- Typical Workflow for Getting Started with Oracle JET Application Development
- Prerequisites for Developing Applications with Oracle JET
- Getting Started with Oracle JET Web Application Development
- Getting Started with Oracle JET Hybrid Mobile Application Development
- Loading Oracle JET from CDN

Typical Workflow for Getting Started with Oracle JET Application Development

Familiarize yourself with the third party tools that Oracle JET uses before you start development. Depending on your installation method, you may also need to install prerequisite packages. After you've created your application, you can customize your configuration or load Oracle JET from the Oracle Content Delivery Network (CDN).

To get started developing Oracle JET applications, refer to the typical workflow described in the following table. After you verify your prerequisites, you can choose to create a web or hybrid mobile application.

Task	Description	More Information
Verify prerequisites	Verify that you meet the prerequisite knowledge and choose a development environment. If you will be using the recommended tooling, install the prerequisite packages.	Prerequisites for Developing Applications with Oracle JET

Task	Description	More Information
Create a web application	Create a web application using the tooling, Starter Template, or base distribution. Alternatively, you can add the Oracle JET download to your existing JavaScript application and add the necessary links to your application's main page.	Create a Web Application Using the Oracle JET Command-Line Interface *or* Create a Web Application Using the Oracle JET Starter Templates *or* Create a Web Application Using the oraclejet.zip Download *or* Add Oracle JET to an Existing JavaScript Application
Create a hybrid mobile application	Install Cordova and optional Android, iOS, and Windows tools. Scaffold, build, and serve development versions of Android, iOS, and Windows hybrid mobile applications.	Getting Started with Oracle JET Hybrid Mobile Application Development
Load Oracle JET from CDN	Load Oracle JET files and libraries from the Oracle Content Delivery Network (CDN).	Loading Oracle JET from CDN

Prerequisites for Developing Applications with Oracle JET

Familiarize yourself with the third party libraries and technologies that Oracle JET uses and choose a development environment before you start developing Oracle JET applications. If you will use Oracle JET tooling to develop web or hybrid mobile applications, install the prerequisite packages.

Topics:

- Prerequisite Knowledge for Working with Oracle JET
- Choose a Development Environment
- (Optional) Install the Prerequisite Packages

Prerequisite Knowledge for Working with Oracle JET

Before you can successfully develop applications with Oracle JET, you should be familiar with the third party libraries and technologies that Oracle JET uses.

Name	Description	More Information
CSS	Cascading Style Sheets	http://www.w3.org/Style/CSS
HTML5	Hypertext Markup Language 5	http://www.w3.org/TR/html5
JavaScript	Programming language	https://developer.mozilla.org/en/About_JavaScript
jQuery	JavaScript library designed for HTML document traversal and manipulation, event handling, animation, and Ajax. jQuery includes an API that works across most browsers.	http://jquery.com

Name	Description	More Information
Knockout	JavaScript library that provides support for two-way data binding	http://www.knockoutjs.com
RequireJS	JavaScript file and module loader used for managing library references and lazy loading of resources. RequireJS implements the Asynchronous Module Definition (AMD) API.	RequireJS: http://www.requirejs.org AMD API: http://requirejs.org/docs/whyamd.html
SASS	SASS (Syntactically Awesome Style Sheets) extends CSS3 and enables you to use variables, nested rules, mixins, and inline imports to customize your application's themes. Oracle JET uses the SCSS (Sasy CSS) syntax of SASS.	http://www.sass-lang.com

If you will be using Oracle JET tooling to create web or hybrid mobile applications, you should also be familiar with the following technologies.

Name	Description	More Information
Apache Cordova (Hybrid only)	Open source mobile development framework that allows you to use HTML5, CSS3, and JavaScript for cross-platform development targeted to multiple platforms with one code base	http://cordova.apache.org/
Node.js	Open source, cross-platform runtime environment for developing server-side web applications, used by Oracle JET for package management. Node.js includes the `npm` command line tool.	https://nodejs.org

Choose a Development Environment

You can develop Oracle JET applications in virtually any integrated development environment (IDE) that supports JavaScript, HTML5, and CSS3.

However, an IDE is not required for developing Oracle JET applications, and you can use any text editor to develop your application. NetBeans IDE version 8.1 and higher includes support for creating HTML5 projects in JavaScript and an Oracle JET Support plugin that contains sample applications and support for code completion. For additional information about creating an Oracle JET application in the NetBeans IDE, see Create an Oracle JET Application in the NetBeans IDE.

If you want to develop hybrid mobile applications using Oracle JET tooling, you must install a platform-specific SDK for the platform (Android, iOS, or Windows) where you want to run the hybrid mobile application. For details about developing hybrid applications using Oracle JET tooling, see Getting Started with Oracle JET Hybrid Mobile Application Development.

Install the Prerequisite Packages

If you plan to use Oracle JET tooling to develop web or hybrid mobile applications, you must install Node.js and the Oracle JET command-line interface (CLI), `ojet-cli`.

> **Note:**
>
> If you already have some or all of the prerequisite packages installed on your development platform, check that you are using the minimum versions supported by Oracle JET tooling and upgrade as needed. For the list of minimum supported versions, see Oracle JET Support.

To install the prerequisite packages:

1. Install Node.js
2. Install the Oracle JET Command-Line Interface

Install Node.js

Install Node.js on your development machine.

From a web browser, download and install one of the installers appropriate for your OS from the Node.js download page. Oracle JET recommends that you install the latest LTS version. Node.js is pre-installed on macOS, but is likely an old version, so upgrade to the latest LTS version if necessary.

After you complete installation and setup, you can enter `npm` commands from a command prompt to verify that your installation succeeded. For example, to configure a proxy server, use `npm config`.

```
npm config set proxy http-proxy-server-URL:proxy-port
npm config set https-proxy https-proxy-server-URL:proxy-port
```

Include the complete URL in the command. For example:

```
npm config set proxy http://my.proxyserver.com:80
npm config set https-proxy http://my.proxyserver.com:80
```

Install the Oracle JET Command-Line Interface

Use npm to install the Oracle JET command-line interface (`ojet-cli`).

1. At the command prompt of your development machine, enter the following command as Administrator on Windows or use `sudo` on Macintosh and Linux machines:

   ```
   [sudo] npm install -g @oracle/ojet-cli
   ```

> **Note:**
>
> It may not be obvious that the installation succeeded. Enter `npm list -g ojet-cli` to verify that the command succeeded. If the package is not listed, scroll through the install command output to locate the source of the failure.
>
> - If you receive an error related to a network failure, verify that you have set up your proxy correctly if needed.
> - If you receive an error that your version of `npm` is outdated, type the following to update the version: `[sudo] npm install -g npm`.

2. If you use a proxy server, configure `HTTP_PROXY` and `HTTPS_PROXY` environment variables.

   The process to set the environment variables depends upon your development environment:

   - Windows

     You can set the variables at a command prompt:

     ```
     set HTTP_PROXY=http-proxy-server-URL:proxy-port
     set HTTPS_PROXY=https-proxy-server-URL:proxy-port
     ```

     Include the complete URL in the command.

     ```
     set HTTP_PROXY=http://my.proxyserver.com:80
     set HTTPS_PROXY=http://my.proxyserver.com:80
     ```

     These settings will last only as long as your terminal session. You can set the variables permanently in **Control Panel > System and Security > System > Advanced System Settings > Environment Variables**.

   - Macintosh and Linux

     On Mac and Linux systems, the settings depend upon your default shell. For example, on a Mac system using the Bash shell, set the environment variables using the following format:

     ```
     export HTTP_PROXY=http-proxy-server-URL:proxy-port
     export HTTPS_PROXY=https-proxy-server-URL:proxy-port
     ```

     On a Linux system using C Shell, you would use the following format:

     ```
     setenv HTTP_PROXY http-proxy-server-URL:proxy-port
     setenv HTTPS_PROXY https-proxy-server-URL:proxy-port
     ```

     These settings will last only as long as your terminal session. To set the variables permanently on a Mac, add the same two lines to the `.bash_profile` file in your home directory. For a Linux system using C Shell, add the lines to the `.cshrc` file in your home directory.

Getting Started with Oracle JET Web Application Development

Developing client-side web applications with Oracle JET is designed to be simple and efficient using the development environment of your choice and Starter Templates to ease the development process.
Oracle JET includes several options for downloading, installing, and creating web applications. Depending upon your application requirements, you can:

- Use the Oracle JET command-line interface package (`ojet-cli`) to scaffold a web application containing either a blank template or a complete pre-configured sample application that you can modify as needed.

  With this method, you can use `ojet-cli` to build the application, serve it in a local web browser, and create a package ready for deployment. This is the preferred approach and recommended if you also plan on creating hybrid mobile applications.

> **Note:**
>
> For additional information about creating hybrid mobile applications, see Getting Started with Oracle JET Hybrid Mobile Application Development.

- Download Oracle JET with a Starter Template ready to run and display in your local browser.

  With this method, you can get up and running quickly, but it will be up to you to package and deploy your application, using whatever method is appropriate for your use case.

- Download the Oracle JET zip distribution which contains the Oracle JET and third-party libraries, CSS and SCSS files, and a RequireJS bootstrap file.

  With this method, you must manually create the `index.html` or main application file and pull in the appropriate libraries and CSS as needed. This method is best if you want to use your own application design pattern instead of the templates included with the tooling or Oracle JET Starter collection.

- Add Oracle JET to an existing JavaScript application.

  With this method, you overlay the pieces of Oracle JET that you need to your existing JavaScript application and modify your main application file and scripts as needed.

Topics:

- Create a Web Application Using the Oracle JET Command-Line Interface
- Create a Web Application Using the oraclejet.zip Download
- Create a Web Application Using the Oracle JET Starter Templates
- Add Oracle JET to an Existing JavaScript Application

Create a Web Application Using the Oracle JET Command-Line Interface

Use the Oracle JET command-line interface (CLI) to create, build, and serve web applications that run in desktop and mobile browsers.

Before you can create your first Oracle JET web application using the CLI, you must also install the prerequisite packages if you haven't already done so. For details, see Install the Prerequisite Packages.

Topics

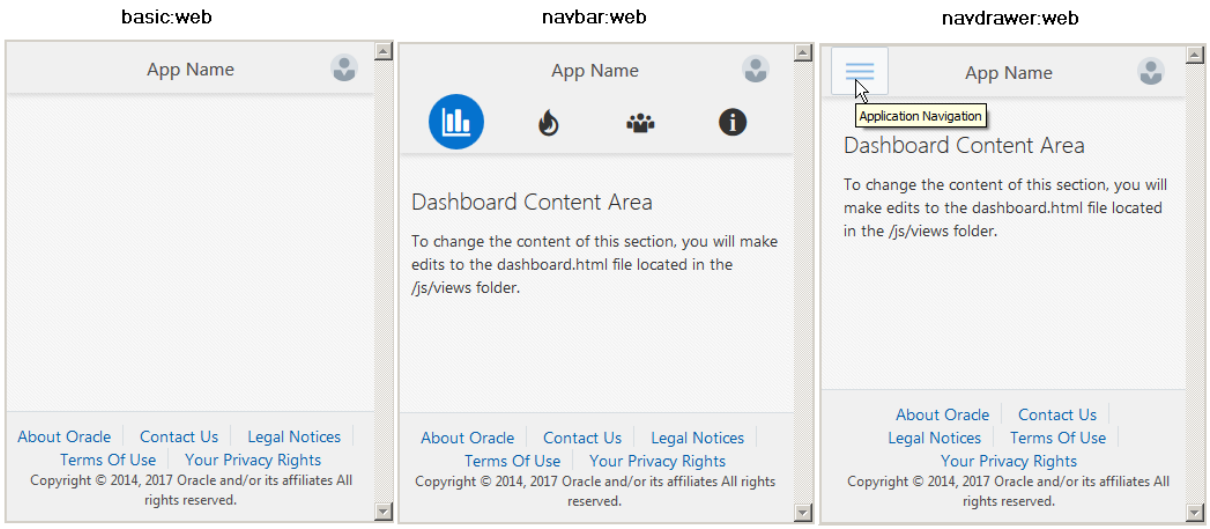
- Scaffold a Web Application with the Oracle JET CLI
- (Optional) Build a Web Application with the Oracle JET CLI
- Serve a Web Application with the Oracle JET CLI

> **Note:**
>
> You can also use the tooling to create hybrid mobile applications. For additional information, see Getting Started with Oracle JET Hybrid Mobile Application Development

Scaffold a Web Application with the Oracle JET CLI

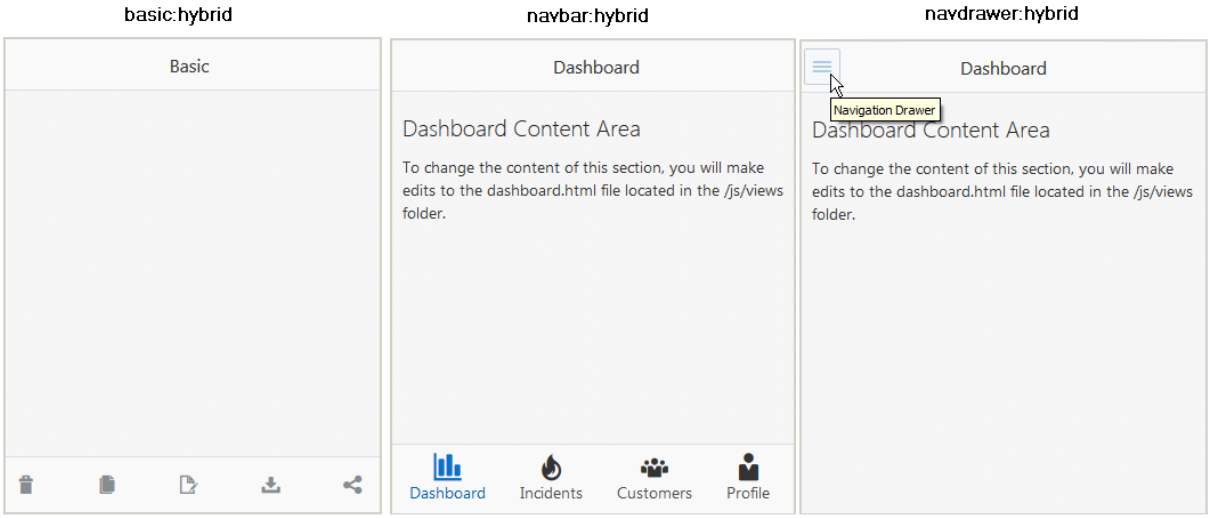
Use the Oracle JET command-line interface (CLI) to scaffold an application that contains a blank template or one pre-configured with a basic layout, navigation bar, or navigation drawer. Each pre-configured template is optimized for responsive web or hybrid mobile applications. After scaffolding, you can modify the application as needed.

The following image shows the differences between the pre-configured templates. The blank template contains an index.html but no UI features and is not shown. The `basic:web` template is similar to the blank template but adds responsive styling that will adjust the display when the screen size changes. The `navbar:web` and `navdrawer:web` templates contain sample content and follow best practices for layout, navigation, and styling that you can also modify as needed.



> **Note:**
>
> Do not confuse the basic template which contains no content with the Oracle JET QuickStart Basic application available in NetBeans 8.2 examples. The NetBeans application uses the `navdrawer:web` layout shown above and contains the same sample content.

If you want your web application to look more like a mobile application, you can scaffold your web application with a hybrid mobile version of the basic, navbar, or navdrawer template: `basic:hybrid`, `navbar:hybrid`, or `navdrawer:hybrid`.



1. At a command prompt, enter `ojet create` with optional arguments to create the Oracle JET application and scaffolding.

```
ojet create [directory]
            [--template={template-name:[web|hybrid]|template-url|template-file}]
            [--help]
```

The following table describes the available options and provides examples for their use.

> **Tip:**
>
> You can also enter `ojet help` at a terminal prompt to get additional help with the Oracle JET CLI.

Option	Description
directory	Application location. If not specified, the application is created in the current directory. The directory will be created during scaffolding if it doesn't already exist.

`template`	Template to use for the application. Specify one of the following:
	• *template-name*
	Predefined template. You can enter `blank`, `basic`, `navbar` or `navdrawer` . Defaults to `blank` if not specified.
	Optionally, add `:web` or `:hybrid` to the template name to specify web or hybrid mobile styling. By default template will use the web styling, but you can add `:hybrid` to change the styling to hybrid mobile, typically `basic:hybrid`, `navbar:hybrid` or `navdrawer:hybrid`.
	• *template-URL*
	URL to zip file containing the name of a zipped application: `http://path-to-app/app-name.zip`.
	• *template-file*
	Path to zip file on your local file system containing the name of a zipped application: "*path-to-app/app-name*`.zip`". For example:
	`--template="C:\Users\SomeUser\app.zip"`
	`--template="/home/users/SomeUser/app.zip"`
	`--template="~/projects/app.zip"`

`help`	Displays a man page for the `ojet create` command, including usage and options: `ojet create --help`.

For example, the following command will create a web application in the `web-app-navbar` directory using the web version of the `navbar` template:

```
ojet create web-app-navbar --template=navbar
```

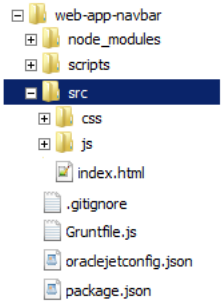
To scaffold the web application with the hybrid mobile version of the navbar template, enter the following command:

```
ojet create web-app-navbar --template=navbar:hybrid
```

The scaffolding will take some time to complete. When successful, the terminal will display:

```
Oracle JET: Your app is ready! Change to your new app directory web-app-navbar
and try ojet build and serve...
```

The new application will have a directory structure similar to the one shown in the following image.



The application folders contain the application and configuration files that you will modify as needed for your own application.

Directory	Description
`node_modules`	Contains the Node.js modules used by the tooling.
`scripts`	Contains the `oraclejet-build.js` and `oraclejet-serve.js` files that you can modify to customize your application's build and serve steps. See Customize a Web Application's Build Behavior and Customize a Web Application's Serve Behavior.
`src`	Site root for your application. Contains the application files that you can modify as needed for your own application and should be committed to source control. The content will vary, depending upon your choice of template. Each template, even the blank one, will contain an `index.html` file and a `main.js` RequireJS bootstrap file. Other templates may contain view templates and viewModel scripts pre-populated with content. For example, if you specified the navbar template during creation, the `js/views` and `js/viewModels` folders will contain the templates and scripts for a web application that uses a nav bar for navigation.

2. In your development environment, update the code for your application.

   For example, you can modify the content to contain samples from the Oracle JET Cookbook. For details about modifying content in pre-configured templates, see Working with the Oracle JET Web Nav Drawer Starter Template.

> **Tip:**
>
> If you selected the blank template during scaffolding, you can still follow the same process to add cookbook samples or other content to your application. However, it will be up to you to create the appropriate view templates or viewModel scripts.

After scaffolding, you can perform the following optional tasks to customize your application:

- Modify the Web Application's File Structure
- Add Hybrid Mobile Features to Web Applications

Modify the Web Application's File Structure

You can modify your scaffolded application's file structure if the default structure doesn't meet your needs.

The `oraclejetconfig.json` file in your application's top level directory contains the default source and staging file paths that you can modify.

```
{
  "paths": {
    "source": {
      "common": "src",
      "web": "src-web",
      "hybrid": "src-hybrid",
      "javascript": "js",
      "styles": "css",
      "themes": "themes"
    },
    "staging": {
      "web": "web",
      "hybrid": "hybrid",
      "themes": "themes"
    }
  },
  "generatorVersion": "x.x.x"
}
```

To change the web application's file structure:

1. In your application's top level directory, open `oraclejetconfig.json` for editing.
2. In `oraclejetconfig.json`, change the paths as needed and save the file.

   For example, if you want to change the default `styles` path from `css` to `app-css`, edit the following line in `oraclejetconfig.json`:

   ```
   "styles": "app-css"
   ```

3. Rename the directories as needed for your application, making sure to change only the paths listed in `oraclejetconfig.json`.

   For example, if you changed `styles` to `app-css` in `oraclejetconfig.json`, change the application's `css` directory to `app-css`.

4. Update your application files as needed to reference the changed path

   For example, if you modified the path to the CSS for your application to `app-css`, update the links appropriately in your application's `index.html`.

   ```
   <link rel="icon" href="app-css/images/favicon.ico" type="image/x-icon" />

       <!-- This is the main css file for the default Alta theme -->
       <!-- injector:theme -->
       <link rel="stylesheet" href="app-css/libs/oj/vx.x.x/alta/oj-alta-min.css"
   type="text/css"/>
       <!-- endinjector -->
   ```

```
    <!-- This contains icon fonts used by the starter template -->
    <link rel="stylesheet" href="app-css/demo-alta-site-min.css" type="text/
css"/>

    <!-- This is where you would add any app specific styling -->
    <link rel="stylesheet" href="app-css/override.css" type="text/css"/>
```

To use the new paths, in your application's top level directory, build your application as described in Build a Web Application with the Oracle JET CLI.

Add Hybrid Mobile Features to Web Applications

Add Cordova hybrid mobile platforms to your web application using the `ojet add hybrid` command. When you run this command, the tooling creates a Cordova project and new `hybrid-src` and `web-src` directories, enabling you to create both web and hybrid mobile applications from the same source.

Since mobile applications typically provide a very different user experience to web applications, taking this approach typically works best if your web application has been designed to suit a mobile device user experience.

To add hybrid mobile platforms to your web application:

If needed, install the mobile tooling packages as described in Getting Started with Oracle JET Hybrid Mobile Application Development.

1. At a terminal prompt, in your application's top level directory, enter the following command to add hybrid mobile features to your web application:

   ```
   ojet add hybrid [--platforms=android,ios,windows]
                   [--appid=application-id]
                   [--appname=application-name]
                   [--help]
   ```

   You should specify at least one platform for the `platforms` value. Add additional platforms as needed, separated by commas. If you don't specify a platform, the command will attempt to locate available platforms and prompt for your selection.

   ```
   --platforms=android
   --platforms=ios,android
   ```

   The `platforms`, `appid`, `appname`, and `help` options use the same values as the `ojet create --hybrid` command described in Scaffold a Hybrid Mobile Application with the Oracle JET CLI

   When you run the command, Oracle JET tooling creates the Cordova project and two new empty source directories that you can use for platform specific content: `src-hybrid` and `src-web`.

> **Note:**
>
> The file locations will vary for this step and the remaining steps in this procedure if you modified your directory structure as described in Modify the Web Application's File Structure or Modify the Hybrid Mobile Application's File Structure.

2. To make changes to content that apply to both web and hybrid platforms, edit content in `src`.
3. To make changes to content that apply only to the web or hybrid platform, add the new content to `src-web` or `src-hybrid` as needed.

   When you build your application, the content in the platform specific file takes precedence over content in `src`. For example, if you create a new `index.html` file to `src-hybrid`, content in that file will take precedence when you build the application as a hybrid mobile application.

Build a Web Application with the Oracle JET CLI

Use the Oracle JET command-line interface (CLI) to build a development version of your web application before serving it to a browser. This step is optional.

Change to the application's root directory and use the `ojet build` command to build your application.

```
ojet build [--theme=themename[:android|ios|web|windows] --themes=theme1,theme2,...
            --sass]
```

The following table describes the available options and provides examples for their use.

> **Tip:**
>
> You can also enter `ojet help` at a terminal prompt to get help for specific Oracle JET CLI options.

Option	Description
`theme`	Theme to use for the application. The theme defaults to `alta:web`, but you can specify a hybrid mobile theme (`alta:android`, `alta:ios`, `alta:windows`) if you want your application to have the look and feel of a mobile application. You can also enter a different *`themename`* with optional platform for a custom theme as described in Customizing Themes Using the Tooling.
`themes`	Themes to include in the application, separated by commas. If you don't specify the `--theme` flag as described above, Oracle JET will use the first element that you specify in `--themes` as the default theme. .
`sass`	Manages Sass compilation. If you add Sass and specify the `--theme` or `--themes` option, Sass compilation occurs by default and you can use `--sass=false` or `--no-sass` to turn it off. If you add Sass and do not specify a theme option, Sass compilation will not occur by default, and you must specify `--sass=true` or `--sass` to turn it on.

The command will take some time to complete. If it's successful, you'll see the following message:

```
Done.
```

The command will also create a web folder in your application's root to contain the built content.

> **Note:**
>
> You can also use the `ojet build` command with the `--release` option to build a release-ready version of your application. For information, see Packaging and Deploying Web Applications.

Customize a Web Application's Build Behavior

Edit `oraclejet-build.js` to modify default build behavior. For example, you can add additional libraries to your application or change the default markers used to inject links into the application's `index.html` during build time.

The Oracle JET tooling defines default build and serve behavior in `node_modules/@oracle/oraclejet-tooling/lib/defaultconfig.js`. The first part of the file contains the default build configuration, including entries for application paths and other options used during the build process. To modify defaults or add your own, edit `oraclejet-build.js` and remove comments from the options that you want to configure. The file contains detailed instructions for each configurable option.

Oracle JET will merge the options with the default configuration during the build. However, be aware that `fileList` options will completely replace the corresponding option in the default configuration and will not be merged.

To customize build behavior:

1. Open `scripts/config/oraclejet-build.js` for editing in your favorite text editor.
2. Identify the entry that you want to change and remove the comments (//) preceding each line.

   For example, to add a custom library to the build's staging directory, remove the comments from the `copyCustomLibsToStaging` entry in `oraclejet-build.js` as shown below:

```
/**
 * # copyCustomLibsToStaging
 * This task copies any custom libraries that are not provided by JET to staging
directory.
 * This task supports a single option: fileList. The fileList option defines an
array of file objects.
 * Each file object contains the following properties:
 *   cwd, current working directory
 *   dest, destination path
 *   src, array of source file patterns
 *   rename, function to return the full path of desired destination
 * If a fileList value is specified, it completely replaces the default fileList
value defined by JET
 * Example: {cwd: 'app', src: ['**', '!test.js'], dest: 'staging',  rename:
function (dest, file) {return renamed path}}
 */
```

```
copyCustomLibsToStaging: {
  fileList: [
    {
    cwd:'node_modules/oraclejet/',
    src: ['*'],
    dest: 'web/js/libs/oraclejet'
    }
  ]
}
```

3. Edit the options as desired and save the file.

   For example, to add a custom library named `my_custom_module`, place the module in `node_modules`, and edit the `copyCustomLibsToStaging` entry:

```
copyCustomLibsToStaging: {
    fileList: [
      {
        cwd:'node_modules/my_custom_module/',
        src: ['*'],
        dest: 'web/js/libs/my_custom_module'
      }
    ]
}
```

4. Repeat the previous step for each option that you want to modify.

Test your changes by running Build a Web Application with the Oracle JET CLI with appropriate options.

Serve a Web Application with the Oracle JET CLI

Use `ojet serve` to run your web application in a local web server for testing and debugging. By default, a live reload option is enabled which lets you make changes to your application code that are immediately reflected in the browser.

Enter the following commands at a terminal prompt:

1. Change to the application's root directory and use the `ojet serve` command with optional arguments to launch the application.

```
ojet serve [--server-port=server-port-number --livereload-port=live-reload-port-
number
            --livereload
            --sass
            --build
            --theme=themename[:android|ios|web|windows] --
themes=theme1,theme2,...
            --server-only
           ]
```

   The following table describes the available options and provides examples for their use.

> **Tip:**
>
> You can also enter `ojet help` at a terminal prompt to get help specific to the Oracle JET tooling commands and `ojet serve --help` to get additional help with serve options.

Option	Description
`server-port`	Server port number. If not specified, defaults to 8000.
`livereload-port`	Live reload port number. If not specified, defaults to 35729.
`livereload`	Enable the live reload feature. Live reload is enabled by default (`--livereload=true`). Use `--livereload=false` or `--no-livereload` to disable the live reload feature. Disabling live reload can be helpful if you're working in NetBeans or other IDE and want to use that IDE's mechanism for loading updated applications.
`sass`	Manages Sass compilation. If you add Sass and specify the `--theme` or `--themes` option, Sass compilation occurs by default and you can use `--sass=false` or `--no-sass` to turn it off. If you add Sass and do not specify a theme option, Sass compilation will not occur by default, and you must specify `--sass=true` or `--sass` to turn it on.
`build`	Build the app before you serve it. By default, an app is built before you serve it (`--build=true`). Use `--build=false` or `--no-build` to suppress the build if you've already built the application and just want to serve it.
`theme`	Theme to use for the application. The theme defaults to `alta:web`, but you can specify a hybrid mobile theme (`alta:android`, `alta:ios`, `alta:windows`) if you want your application to have the look and feel of a mobile application. You can also enter a different *`themename`* with optional platform for a custom theme as described in Customizing Themes Using the Tooling.
`themes`	Themes to use for the application, separated by commas. If you don't specify the `--theme` flag as described above, Oracle JET will use the first element that you specify in `--themes` as the default theme. Otherwise Oracle JET will serve the application with the theme specified in `--theme`.
`server-only`	Serves the application, as if to a browser, but does not launch a browser. Use this option in cloud-based development environments so that you can attach your browser to the app served by the development machine.

For example, the following command will launch your application in the default web browser with live reload enabled: `ojet serve`.

2. Make any desired code change in the `src` folder, and the browser will update automatically to reflect the change unless you set the `--no-livereload` flag.

   While the application is running, the terminal window remains open, and the watch task waits for any changes to the application. For example, if you change the content in `src/js/views/dashboard.html` , the watch task will reflect the change in the terminal as shown below on a Windows desktop.

```
Starting watcher...
Listening on port 35729...
Server ready: http://localhost:8000
Watching files....
Watcher: sass is ready...
Watcher: sourceFiles is ready...
Watcher: themes is ready...
Changed: c:\web-app-navbar\src\js\views\dashboard.html
Page reloaded resume watching...
```

3. To terminate the process, close the application and press Ctrl+C in the command window.

   You may need to enter Ctrl+C a few times before the process terminates.

> **Note:**
>
> You can also use the `ojet serve` command with the `--release` option to serve a release-ready version of your application. For information, see Packaging and Deploying Web Applications.

Customize a Web Application's Serve Behavior

Edit `oraclejet-serve.js` to modify default serve behavior. For example, you can add your own watch tasks or specify actions specific to web or hybrid applications.

The Oracle JET tooling defines default build and serve behavior in `node_modules/@oracle/oraclejet-tooling/lib/defaultconfig.js`. The second half of the file contains the default serve configuration, including entries for application paths and other options used during the serve process. To modify defaults or add your own, edit `scripts/config/oraclejet-serve.js` and remove comments from the options that you want to configure. The file contains detailed instructions for each configurable option.

Oracle JET will merge the options with the default configuration during the build. However, be aware that rewriting the three default watch targets (`sourceFiles`, `sass`, and `themes`) can result in unpredictable behavior.

To customize serve behavior:

1. Open `scripts/config/oraclejet-serve.js` for editing in your favorite text editor.
2. Identify the entry that you want to change and remove the comments (//) preceding each line that you want to include.

   For example, to define one or more shell commands that you want to execute after Oracle JET serves the application and you modify a source file, you can edit the `sourceFiles` option in the `watch` sub task. Remove the comments from the highlighted lines as shown below.

```
// // Sub task watch default options
     watch: {
       sourceFiles:
       {
    //       files: [],
    //       options: {
    //       livereload: true
    //      },
     },
```

```
    //   sass: {
    //     files: [],
    //     commands: ['compileSass']
    //   },

    //   themes: {
    //     files: [],
    //     options: {
    //       livereload: true
    //     },
    //     commands: ['copyThemes']
    //   },
},
```

3. Edit the options as desired and save the file.

   For command options, add the commands option and one or more commands, each surrounded by single quotes and separated by commas. For example, the following code snippet adds `node --version` and `npm --version` to the `commands` option in the `sourceFiles` sub task:

```
 watch: {
   sourceFiles:
   {
//     files: [],
//     options: {
//       livereload: true
//     },
       commands: ['node --version', 'npm --version']
   },
   ... contents omitted
 },
```

Test your changes by running Serve a Web Application with the Oracle JET CLI with appropriate options. When you change one of the source files, the terminal output will display the output of the `node --version` and `npm --version` commands and reload the page.

Create a Web Application Using the oraclejet.zip Download

You can create a new Oracle JET application in your favorite IDE or text editor using files extracted from `oraclejet.zip`. After you create your application, you can modify it to include references to Oracle JET CSS and libraries.

Topics

- Download the Oracle JET Zip File
- (Optional) Create an Oracle JET Application in the NetBeans IDE
- Include References to CSS Files in Your Oracle JET Application
- Use RequireJS to Manage Library, Link, and Script References
- Specify Library, Link, and Script References Without RequireJS

Download the Oracle JET Zip File

You can download Oracle JET from the Oracle Technology Network (OTN) as a single zip file. The zip file contain the libraries, scripts, styles, and files needed to develop applications using Oracle JET.

To download Oracle JET from the Oracle Technology Network (OTN):

1. Navigate to the Oracle JET download location at: Oracle JET Downloads.
2. Choose **Accept License Agreement**.
3. Download **Oracle JavaScript Extension Toolkit: Base Distribution**.

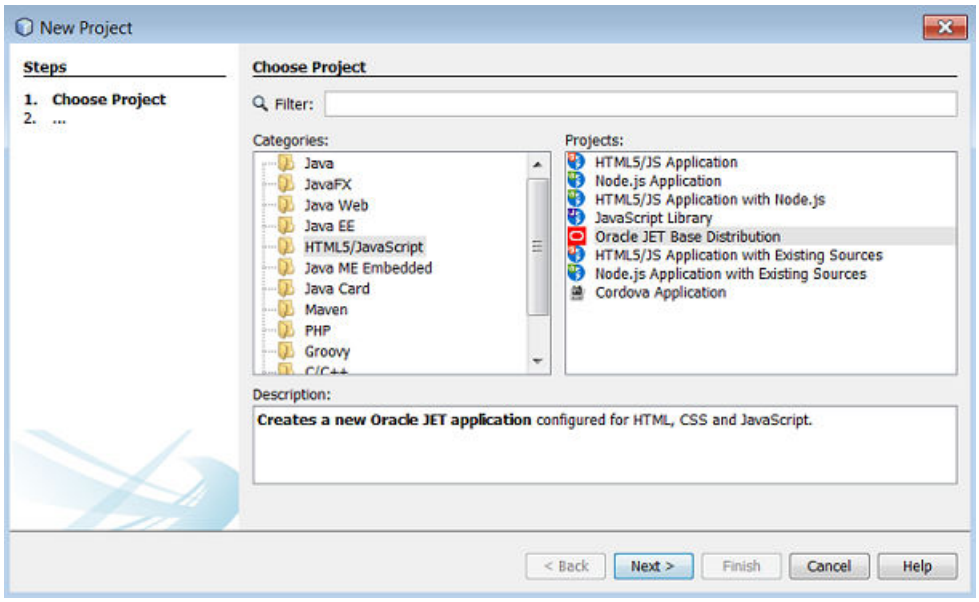
> **Tip:**
>
> If you plan to use NetBeans 8.2 or higher for developing your Oracle JET application, you can skip this step. The NetBeans IDE's Oracle JET Base Distribution Sample already includes `oraclejet.zip`. For details, see Create an Oracle JET Application in the NetBeans IDE

Create an Oracle JET Application in the NetBeans IDE

You can use NetBeans 8.2 or higher to create an Oracle JET application using the NetBeans Oracle JET Base Distribution, which includes the `oraclejet.zip` download.

To create an Oracle JET application in the NetBeans IDE:

1. If needed, download and install NetBeans. Select a download bundle that includes support for HTML5. For additional detail, see https://netbeans.org/downloads/index.html.
2. In the NetBeans IDE, choose **File** > **New Project** from the main menu.
3. In the **New Project** dialog, create a new HTML5/JavaScript application.
   a. In the Categories area, click **HTML5/JavaScript**.
   b. In the Projects area, click **Oracle JET Base Distribution**.



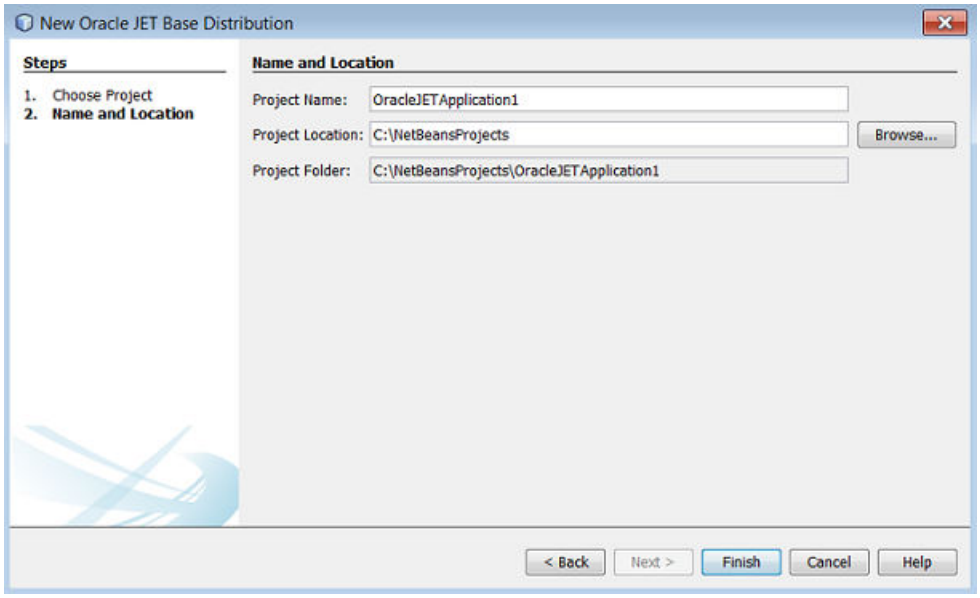
In this example, the `All` download bundle is used for the initial NetBeans 8.2 download. If you downloaded a different NetBeans download bundle or are using a different version, the dialog may include different categories.

c. Click **Next**.

4. In the **New Oracle JET Base Distribution** dialog, configure the project name and project location.

   a. In the **Project Name** field, enter a name for your project.

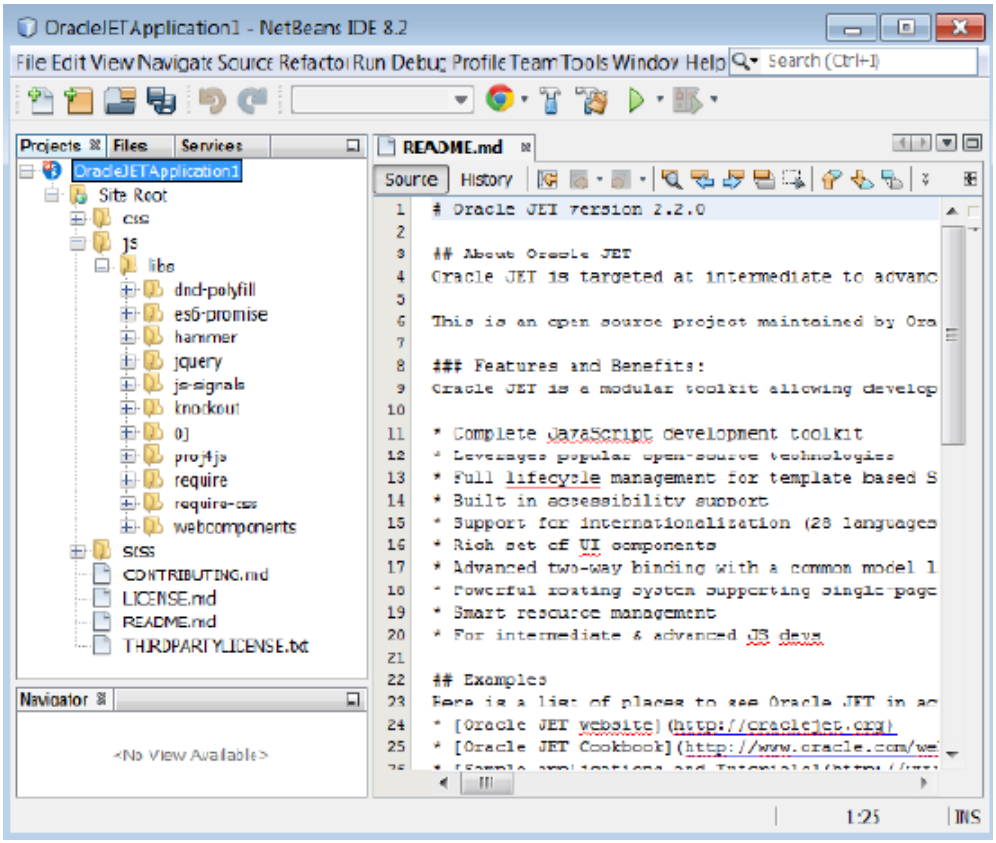
   b. In the **Project Location** field, enter a location to store the project files.



In this example, the project is named `OracleJETApplication1`, and the project files will be stored in the `C:\NetBeansProjects\` directory.

5. Click **Finish** to create your application.

   When you click **Finish**, NetBeans creates the project from the Oracle JET Base Distribution. You can expand the `js/libs/` directory to see the JavaScript libraries that constitute Oracle JET. This figure shows the project files and `README.md` file that NetBeans 8.2 generated for a project named `OracleJETApplication1` with no optional tools installed.



> **Note:**
>
> If you decide you want to use one of the tools after you create your application, you can add it in the NetBeans IDE. Choose **Tools**, then **Options**, and **HTML/JS**. Select the tab containing the tool you wish to add, and select the option to install the package.

Include References to CSS Files in Your Oracle JET Application

Specify the reference to the CSS files directly in your application's main page, typically `index.html`.

If needed, download Oracle JET as described in Download the Oracle JET Zip File.

1. Create a new `index.html` file in the project root.
2. In the application's main page, `index.html`, add the HTML `link` element and point it to the CSS theme that you will use.

For example, to use the Oracle JET Alta web theme:

```
<!-- Oracle JET CSS files -->
<link rel="stylesheet" href="css/libs/oj/version/alta/oj-alta-min.css"
type="text/css"/>
```

For additional information about Oracle JET and the themes available for your use, see Theming Applications.

3. If you will be using the CSS included with Oracle JET, add the following line to the top of your application's main HTML page:

```
<!DOCTYPE html>
```

This line is needed for the Oracle JET CSS to function as expected.

Use RequireJS to Manage Library, Link, and Script References

RequireJS is a JavaScript file and module loader that makes it easier to manage library references and is designed to improve the speed and quality of your code.

Oracle JET uses RequireJS by default for the Oracle JET Starter Templates and is required if you plan to use Oracle JET's internationalization, data visualization components, or the `oj.OAuth` plugin in your application.

To use RequireJS to manage references:

If needed, download Oracle JET as described in Download the Oracle JET Zip File.

1. Copy `js/libs/oj/version/main-template.js` to the `js` folder.
2. In the `js` folder, rename `main-template.js` to `main.js`.
3. Add the following script reference to your `index.html` file:

```
<script data-main="js/main" src="js/libs/require/require.js"></script>
```

4. Update `main.js` as needed to reference Oracle JET modules or your own scripts.

   For a list of Oracle JET modules and additional details about using RequireJS in your Oracle JET application, see Using RequireJS for Modular Development.

For more information about RequireJS, see http://requirejs.org.

Specify Library, Link, and Script References Without RequireJS

You can specify the Oracle JET script and link references directly in your `index.html` or main application file without using RequireJS.

If you know that your application will not need RequireJS, you can add script references specifying the library links. However, Oracle JET requires its use for major pieces of the toolkit, such as internationalization and data visualization components.

To specify library, link, and script references without using RequireJS to manage references:

If needed, download Oracle JET as described in Download the Oracle JET Zip File.

1. If your application will run on devices that support touch, add the following entry to your `index.html` inside the `head` element to eliminate tap delay:

```
<head>
...
```

```
<meta name="viewport" content="width=device-width">
...
</head>
```

This entry has the effect of disabling double tap to zoom on supported browsers. If you don't add this entry, browsers will wait approximately 300 to 350 milliseconds to see if the first tap is followed by another before firing the click event on the first tap.

> **Note:**
>
> Due to lack of universal browser support, you should also add the following entry to your CSS if you're not using one of the themes included with Oracle JET:
>
> ```
> body {
>  touch-action: manipulation;
> }
> ```

2. Add the following script references as needed, inside the `head` element in your `index.html` file:

```
<head>
...
<!-- 3rd Party JavaScript files -->
<script type="text/javascript" src="js/libs/knockout/knockout-3.4.0.js"></script>
<script type="text/javascript" src="js/libs/jquery/jquery-3.1.1.min.js"></script>
<script type="text/javascript" src="js/libs/jquery/jquery-
ui-1.12.0.custom.min.js"></script>
<script type="text/javascript" src="js/libs/knockout/knockout.mapping-
latest.js"></script>
<script type="text/javascript" src="js/libs/js-signals/signals.min.js"></script>
<script type="text/javascript" src="js/libs/es6-promise/es6-promise.min.js"></
script>
<script type="text/javascript" src="js/libs/hammer/hammer-2.0.8.min.js"></script>
<script type="text/javascript" src="js/libs/require-css/css.min.js"></script>
<script type="text/javascript" src="js/libs/proj4js/dist/proj4.js"></script>

<!-- Oracle JavaScript files -->
<script type="text/javascript" src="js/libs/dnd-polyfill/dnd-
polyfill-1.0.0.min.js"></script>
<script type="text/javascript" src="js/libs/oj/v4.0.0/min/oj.js"></script>
<script type="text/javascript" src="js/libs/webcomponents/custom-
elements.min.js"></script>
...
</head>
```

If you want to use the design time Oracle JET libraries for debugging, replace the `min/oj.js` reference with:

```
<script type="text/javascript" src="js/libs/oj/v4.0.0/debug/oj.js"></script>
```

Create a Web Application Using the Oracle JET Starter Templates

You can load the Oracle JET Starter Templates in NetBeans or your favorite development environment. After you load one of the templates, you can customize it for your application by modifying the template's `index.html` and application files.

Topics:

- Downloading Oracle JET with a Starter Template
- Loading the Oracle JET Starter Templates
- Working with the Oracle JET Web Nav Drawer Starter Template

Downloading Oracle JET with a Starter Template

You can download any of the applications in the Oracle JET Starter Template collection to get Oracle JET. The Web Nav Drawer Starter Template contains the Oracle JET bundle, an `index.html` file already configured for Oracle JET and is a good choice for getting started with Oracle JET.

> **Tip:**
>
> If you use the Oracle JET CLI r to scaffold a web application with a `navdrawer` template as described in Scaffold a Web Application with the Oracle JET CLI, then you already have a Starter Template and can skip this step.
>
> Also, if you use NetBeans IDE version 8.2, it includes the Oracle JET QuickStart Basic project which contains the same content as the Web Nav Drawer Starter Template.

To download Oracle JET with a Starter Template:

1. Navigate to the Oracle JET download location at: Oracle JET Downloads.
2. Choose **Accept License Agreement**.
3. Download the desired template.

   For example, to download the Web Nav Drawer template, select **Web Nav Drawer Starter Template with Oracle JavaScript Extension Toolkit Pre-configured**.

Loading the Oracle JET Starter Templates

The Oracle JET Starter Templates include all of the Oracle JET libraries and an `index.html` file already configured for your use. To load the Oracle JET Starter Templates, extract the zip files and copy them as needed for your development environment.

You can also load the Oracle JET Starter Templates in the NetBeans IDE.

Topics:

- Create an Application in the NetBeans IDE with a Site Template

- Run the Oracle JET Starter Templates in NetBeans

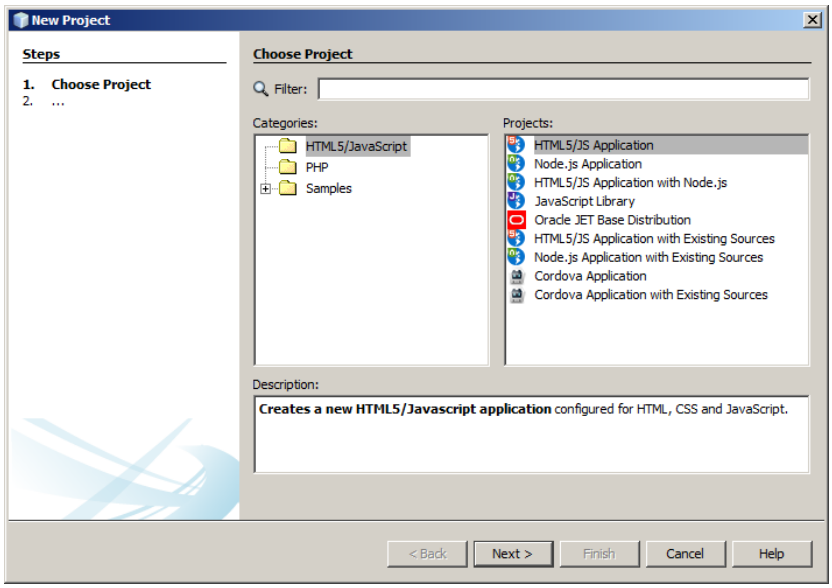
Create an Application in the NetBeans IDE with a Site Template

You can create a new Oracle JET application in the NetBeans IDE using files extracted from `oraclejet.zip` or any of the Oracle JET Starter Templates. NetBeans 8.1 and higher includes support for Oracle JET applications using the Oracle JET zip files as site templates in the NetBeans IDE.

To create an Oracle JET application in the NetBeans IDE using a site template:

> **Tip:**
>
> If you're specifically interested in the Web Nav Drawer Starter Template, you can skip all steps following Step 1. In the NetBeans IDE, choose **File** > **New Project**, then in the Categories area, select **Samples** > **HTML5/JavaScript**, and select **Oracle JET QuickStart Basic** from the Projects list.

1. If needed, download and install NetBeans. Select a download bundle that includes support for HTML5. For additional detail, see https://netbeans.org/downloads/index.html
2. In the NetBeans IDE, click **New Project**.
3. In the New Project dialog, create a new HTML5 application.
   a. In the Categories area, click **HTML5/JavaScript**.
   b. In the Projects area, click **HTML5/JS Application**.



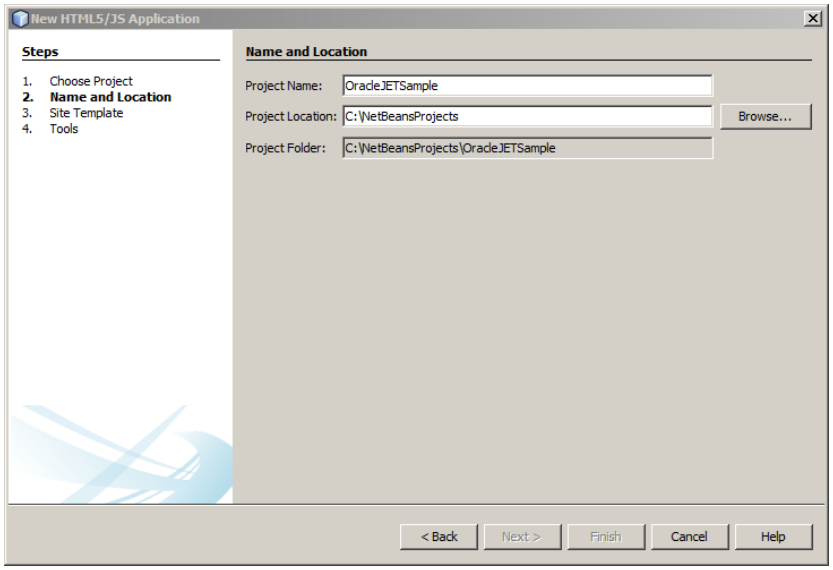
In this example, the `HTML5 & PHP` download bundle is used for the initial NetBeans 8.2 download, and the categories are limited to **HTML5** and **PHP**. If you downloaded a different NetBeans download bundle or are using a different version, the dialog may include additional or different categories.

c. Click **Next**.

4. In the New HTML5/JS Application dialog, configure the project name and project location.

   a. In the **Project Name** field, enter a name for your project.

   b. In the **Project Location** field, enter a location to store the project files.



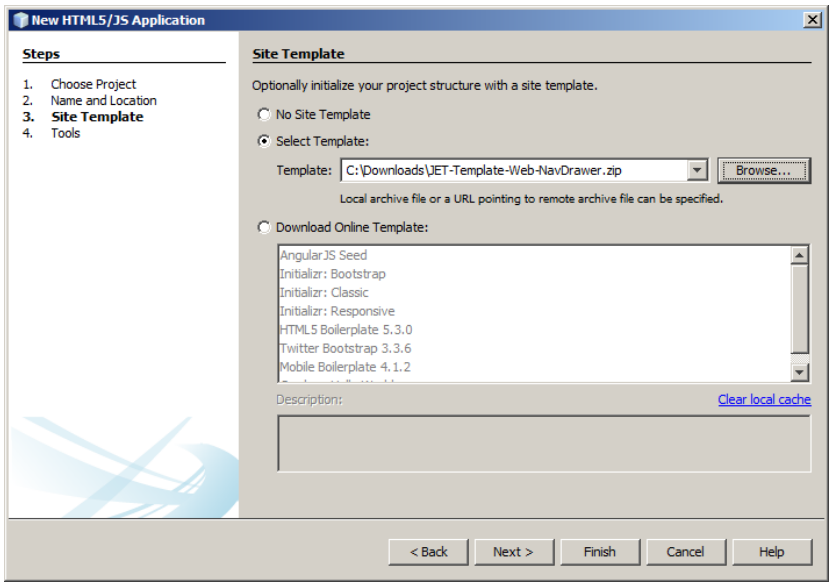
   In this example, the project is named `OracleJETSample`, and the project files will be stored in the `C:\NetBeansProject` directory.

   c. Click **Next**.

5. In the New HTML5/JS Application dialog, specify the name of the site template.

   a. Choose **Select Template**.

   b. In the Template field, choose **Browse** and select the location containing the download bundle.



In this example, the completed HTML5 Application dialog specifies `C:\Downloads` as the location for the `JET-Template-Web-NavDrawer.zip` file.

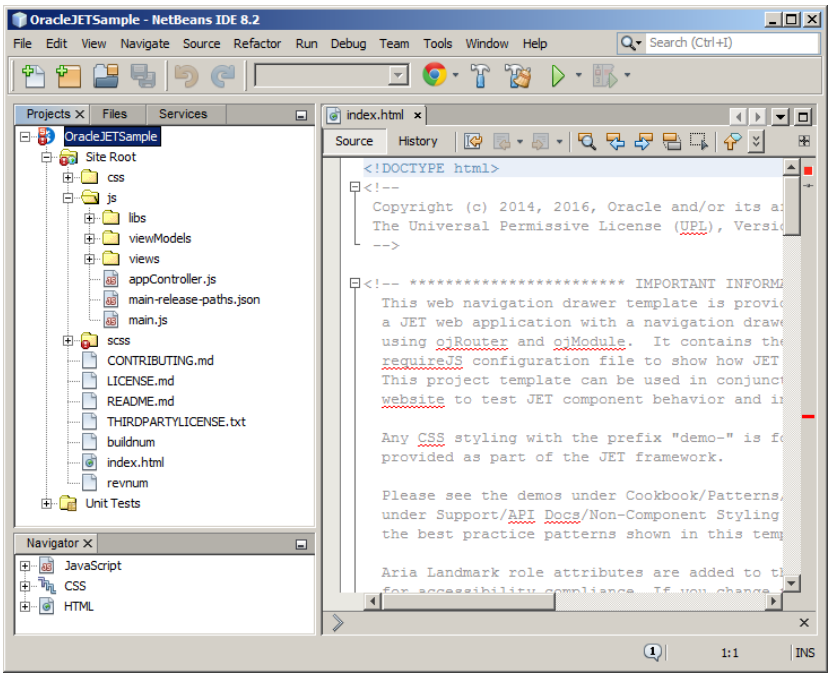
c. Click **Next**.

6. Optionally, in the New HTML5 Application Dialog, remove the check mark from any tools your application won't need. By default, all tools are checked.

> **Tip:**
>
> If you decide you want to use one of the tools after you create your application, you can add it in the NetBeans IDE. Choose **Tools**, then **Options**, and **HTML/JS**. Select the tab containing the tool you wish to add, and select the option to install the package.

7. Click **Finish** to create your application.

   When you click **Finish**, NetBeans creates the project and adds an `index.html` file to your project. This figure shows the project files and `index.html` file that NetBeans 8.2 generated for a project named `OracleJETSample` with no optional tools installed.



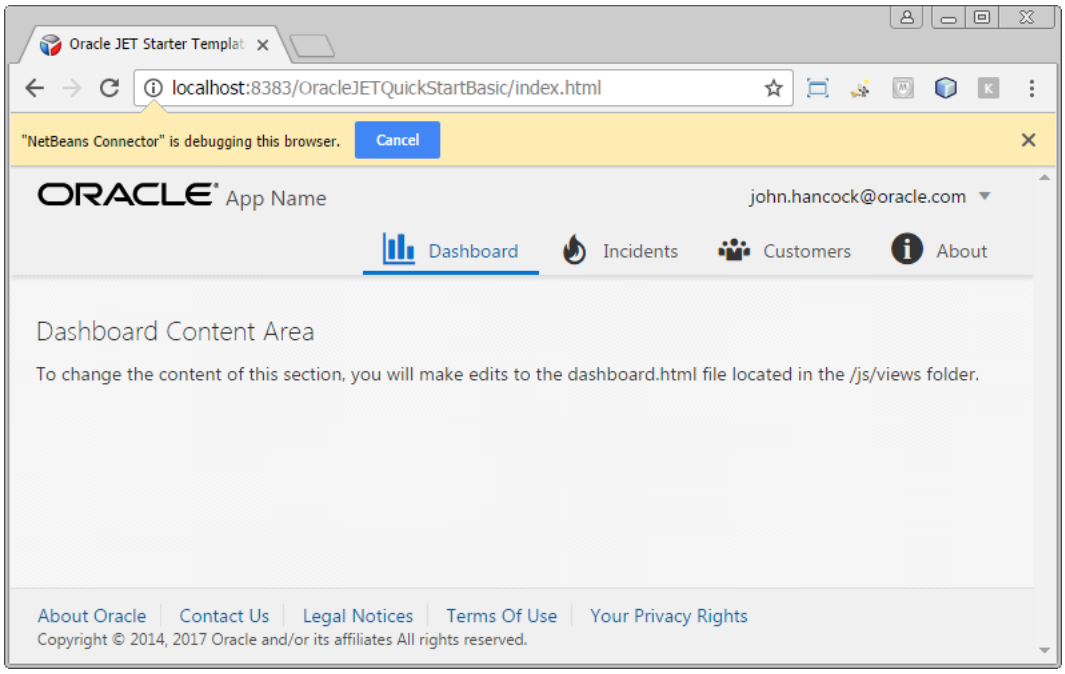
Run the Oracle JET Starter Templates in NetBeans

To run the Oracle JET Starter Templates in the NetBeans IDE:

In the Projects panel, right-click the **index.html** file and choose **Run File** to display the page in your browser.

> **Note:**
>
> To take full advantage of the NetBeans integrated tools, the NetBeans team recommends that you install the NetBeans Connector extension for the Chrome browser. For additional info, see `https://netbeans.org/kb/docs/webclient/html5-gettingstarted.html`.

The graphic below shows the Oracle JET Web Nav Drawer Starter Template displayed in the Chrome browser at run time. In this example, the NetBeans Connector is enabled.



Working with the Oracle JET Web Nav Drawer Starter Template

The Oracle JET Web Nav Drawer Starter Template in the Starter Template Collection provides everything you need to start working with code immediately. It is designed to work with the Oracle JET Cookbook examples, and you can use it to familiarize yourself with the Oracle JET components and basic structure of an Oracle JET application.

Topics:

- About the Web Nav Drawer Starter Template
- Modifying the Web Nav Drawer Starter Template Content

You can also view a video that shows how to work with the Oracle JET Starter Templates in the Oracle JET Videos collection.

About the Web Nav Drawer Starter Template

The Web Nav Drawer Starter Template in the Starter Template Collection is a single page application that is structured for modular development using RequireJS, Knockout templates, and the Oracle JET `ojModule` binding. Instead of storing all the application markup in the `index.html` file, the application uses Knockout's `data-bind` syntax and the `ojModule` binding to bind either a view template containing the HTML markup for the section or both the view template and JavaScript file that contains the viewModel for any components defined in the section.

The following code shows a portion of `index.html` highlighting the `ojModule` binding definition. For the sake of brevity, most of the code and comments are omitted. Comments describe the purpose of each section, and you should review the full source code for accessibility and usage tips.

```
<!DOCTYPE html>
<html lang="en-us">
```

```
  <head>
    <title>Oracle JET Starter Template - Web Nav Drawer</title>
       ... contents omitted
  </head>
  <body class="oj-web-applayout-body">

    ... contents omitted

     <div role="main" class="oj-web-applayout-max-width oj-web-applayout-content"
data-bind="ojModule: router.moduleConfig">
     </div>

     ... contents omitted

         <script type="text/javascript" src="js/libs/require/require.js"></script>
         <script type="text/javascript" src="js/main.js"></script>
  </body>
</html>
```

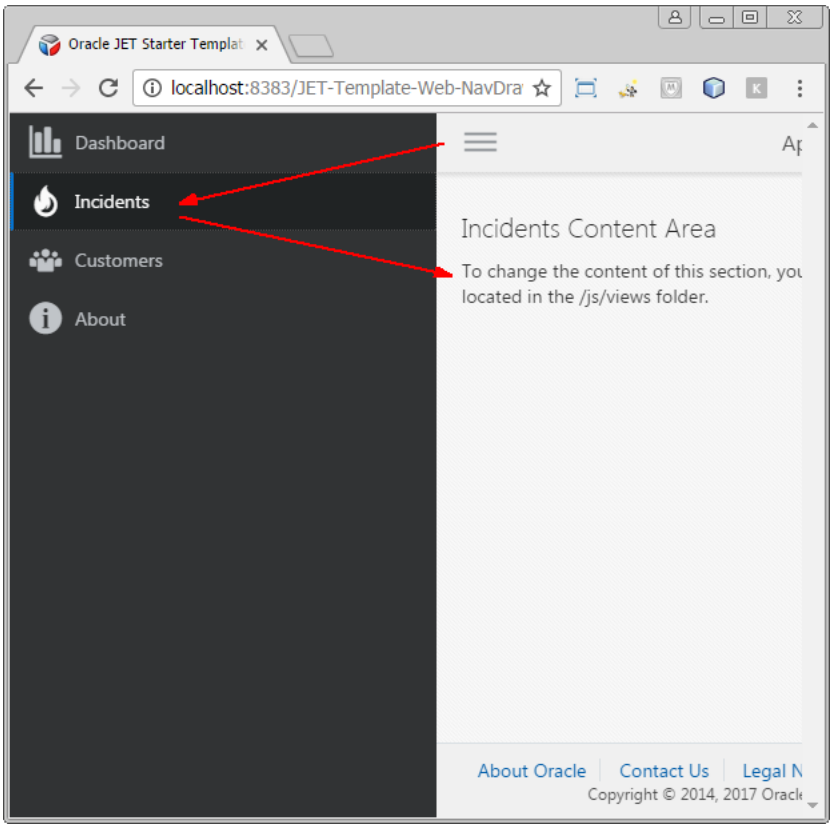
The main page's content area uses the Oracle JET `oj-web-applayout-*` CSS classes to manage the responsive layout. The main page's content uses the HTML `div` element with its role defined as main (`role="main"`) for accessibility.

The `ojModule` binding's `viewName` option tells Oracle JET that the section is only defining a view template, and the view will be bound to the existing viewModel. When the `ojModule` binding's `name` option is defined instead, the application will load both the viewModel and view template with the name corresponding to the value of the `name` option.

When the `ojModule` binding's `name` and `viewName` options are missing, as in this example, the behavior will depend on the parameter specified in the binding's definition.

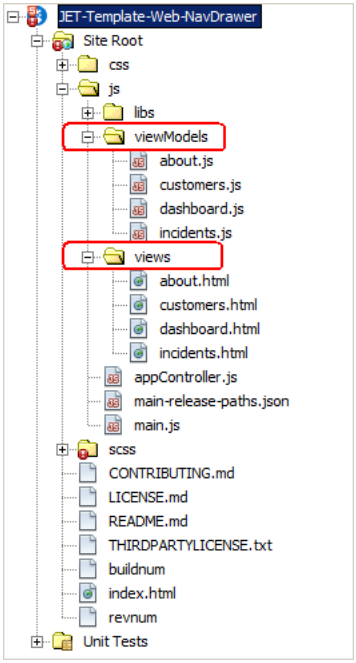
- If the parameter is an Oracle JET router's `moduleConfig` object as in the above example, then `ojModule` will automatically load and render the content of the viewModel script and view template corresponding to the router's current state.

  The Web Nav Drawer Starter Template uses `oj.Router` to manage navigation when the user clicks one of the application's navigation items. The router states include `dashboard`, `incidents`. `customers`, , and `about`. If the user clicks **Incidents**, for example, the main content area changes to display the content in the `incidents` view template.



- If the parameter is a Knockout observable containing the name of the viewModel, the application will load both the viewModel and view template with the indicated name.

The `site_root/js/views` folder contains the view templates for the application. `site_root/js/viewModels` contains the viewModel scripts. The image below shows the Web Nav Drawer Starter Template file structure in the NetBeans IDE.



For additional information about working with single page applications, `ojModule`, `oj.Router`, and Knockout templates, see Creating Single-Page Applications.

For details about the `oj-web-applayout-*` CSS classes, see Application Shell: Weband Web Application Patterns. For additional information about working with responsive design, see Designing Responsive Applications.

Modifying the Web Nav Drawer Starter Template Content

You can add your own content to the main content area or anywhere on the page.

To add content, modify the appropriate view template and ViewModel script (if it exists) for the section that you want to update. Add any needed RequireJS modules to the ViewModel's `define()` definition, along with functions to define your ViewModel.

If you add content to a section that changes its role, then be sure to change the role associated with that section. Oracle JET uses the role definitions for accessibility, specifically WAI-ARIA landmarks. For additional information about Oracle JET and accessibility, see Developing Accessible Applications.

Before you begin:

1. Download the Web Nav Drawer Starter Template as described in Downloading Oracle JET with a Starter Template.
2. Load the Starter Template as described in Create a Web Application Using the Oracle JET Starter Templates.

To modify the Web Nav Drawer Starter Template content:

1. In the `index.html` file, locate the `ojModule` binding for the section you want to modify and identify the template and optional ViewModel script.

   In the Web Nav Drawer Starter Template, the `ojModule` binding is using the `router.moduleConfig` parameter. The following code sample shows the `mainContent` HTML `div` definition in `index.html` .

   ```
   <div role="main" class="oj-web-applayout-max-width oj-web-applayout-content"
        data-bind="ojModule: router.moduleConfig">
   </div>
   ```

   The return value of `router.moduleConfig` is set to the current state of the `ojRouter` object. The `ojRouter` object is defined with an initial value of `dashboard` in the application's `appController.js` script, shown below.

   ```
   // Router setup
   self.router = oj.Router.rootInstance;
   self.router.configure({
     'dashboard': {label: 'Dashboard', isDefault: true},
     'incidents': {label: 'Incidents'},
     'customers': {label: 'Customers'},
     'about': {label: 'About'}
   });
   oj.Router.defaults['urlAdapter'] = new oj.Router.urlParamAdapter();
   ```

   To modify the Dashboard Content Area, for example, you will modify both `dashboard.html` and `dashboard.js`.

2. To modify the view template, remove unneeded content, and add the new content to the view template file.

For example, if you are working with an Oracle JET Cookbook sample, you can copy the markup into the view template you identified in the previous step (`dashboard.html`). Replace everything after the `<h1>Dashboard Content Area</h1>` markup in the template with the markup from the sample.

The following code shows the modified markup if you replace the existing content with a portion of the content from the Date and Time Pickers demo.

```
<div id="div1">
  <oj-label for="dateTime">Default</oj-label>
  <oj-input-date-time id="dateTime" value={{value}}>
  </oj-input-date-time>
  <br/><br/>
  <span class="oj-label">Current component value is:</span>
  <span data-bind="text: value"></span>
</div>
```

3. To modify the ViewModel, remove unneeded content, and add the new content as needed. Include any additional RequireJS modules that your new content may need.

   The application's `main.js` file contains the list of Require modules currently included in the application. Compare the list of libraries with the list you need for your application, and add any missing modules to your `define()` function in the ViewModel script. For example, to use the `oj-input-date-time` elements shown in the demo, add `ojs/ojdatetimepicker` to the `dashboard.js` ViewModel script since it's not already defined in `dashboard.js`.

   The sample below shows a portion of the modified `dashboard.js` file, with the changes highlighted in bold.

```
define(['ojs/ojcore' ,'knockout', 'ojs/ojdatetimepicker'
   ], function(oj, ko) {
   /**
    * The view model for the main content view template
    */
  function DashboardViewModel() {
    var self = this;
    self.value = ko.observable(oj.IntlConverterUtils.dateToLocalIso(new
Date(2013, 0, 1)));
  }

  return DashboardViewModel;
});
```

> **Important:**
>
> Notice that with this example, you are not copying the entire code section. The Cookbook uses a `require()` call to load and use the needed libraries in a single bootstrap file. The Starter Template that you are pasting uses `define()` to create a RequireJS module that can be used by other parts of your application.

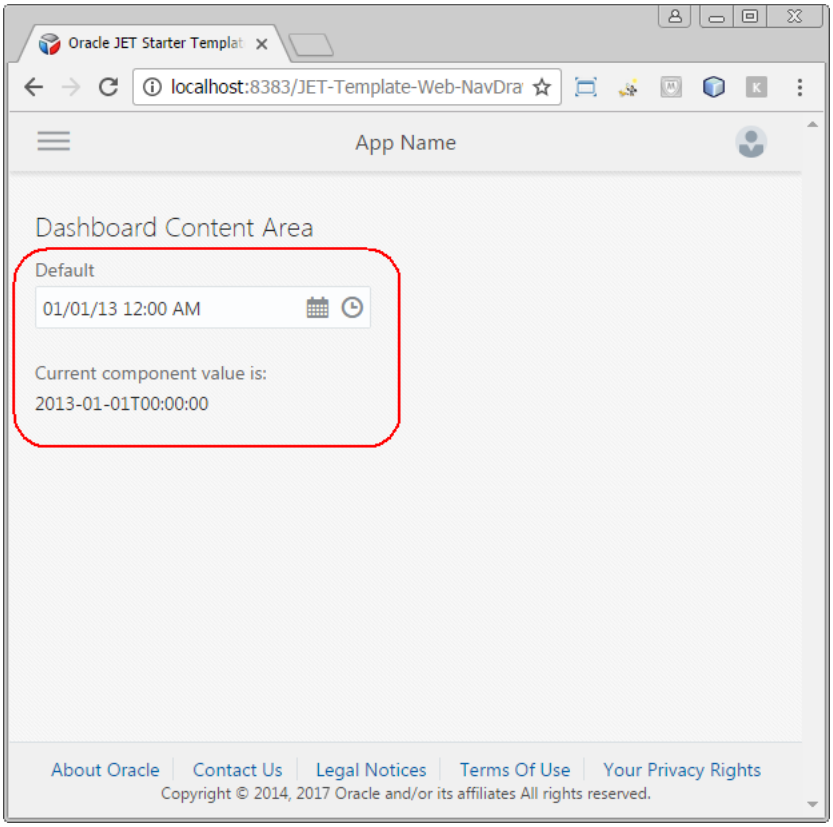
   If you're not using a Cookbook example and are not sure which RequireJS module to include, see the table at Oracle JET Module Organization.

4. If you want to add, change, or delete modules or templates to the application, modify the `main.js` RequireJS bootstrap file as needed.

The file also contains the event handler that responds when a user clicks one of the navigation buttons. Depending upon your modifications, you may need to update this method as well.

5. Run the `index.html` file in your favorite browser and verify the change.

   The following image shows the runtime view of the Web Nav Drawer Starter Template with the new Dashboard Content Area content showing `oj-input-date-time` with its current value.



Add Oracle JET to an Existing JavaScript Application

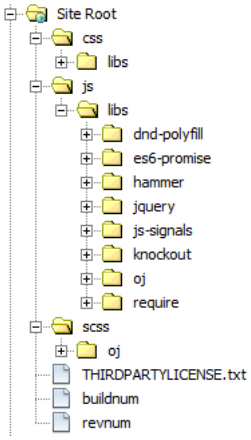
You can add Oracle JET to your existing JavaScript application by extracting the `oraclejet.zip` file into the site root of your application and including references to the Oracle JET libraries and CSS as needed.

To add Oracle JET to an existing JavaScript application:

If needed, download `oraclejet.zip` as described in Download the Oracle JET Zip File.

1. Extract `oraclejet.zip` into the site root of your application.

   For example, if your application's site root is `Site Root`, extract `oraclejet.zip` into `Site Root`. After extraction, your site root folder will contain the following Oracle JET files and folders as shown in the NetBeans IDE.



The `css` folder contains the themes included with Oracle JET, and the `scss` folder contains the SASS source files for the Oracle JET themes. For additional information about Oracle JET theming, see Theming Applications.

The `js` folder contains the Oracle JET libraries and third party libraries, and the `buildnum` and `revnum` files identify the Oracle JET build number and version. For additional information about the libraries included in Oracle JET, see What's Included in Oracle JET.

2. If you want to use one of the themes included with Oracle JET, add the appropriate link to the CSS. For details, see Include References to CSS Files in Your Oracle JET Application.
3. If you want to use the Oracle JET libraries, you can either specify the links directly in your main application file or use RequireJS to manage the Oracle JET library, link, and script references.
   - For information about specifying the links directly in your main application file, see: Specify Library, Link, and Script References Without RequireJS.
   - For information about configuring your application to use RequireJS, see Use RequireJS to Manage Library, Link, and Script References.

Getting Started with Oracle JET Hybrid Mobile Application Development

Oracle JET includes support for hybrid mobile application development using Cordova. The toolkit provides iOS, Android, and Windows Alta themes and UI behavior on Oracle JET components, starter applications, design patterns, and tooling support.

Before you can create your first hybrid mobile application, you should become familiar with the Oracle JET mobile features and third-party technologies. You must also install the prerequisite packages and Oracle JET mobile tools.

Topics

- Install the Mobile Tooling
- Create a Hybrid Mobile Application
- Accessing Mobile Device Services Using Cordova Plugins

Once you have created a hybrid mobile application, added Cordova plugins, and familiarized yourself with Oracle JET's mobile features, refer to the following information for details about other tasks that you need to do to complete your hybrid mobile application.

- About Securing Hybrid Mobile Applications
- Testing Hybrid Mobile Applications
- Debugging Hybrid Mobile Applications
- Packaging and Publishing Hybrid Mobile Applications

> **Tip:**
>
> If you're strictly interested in developing web applications that run in desktop and mobile browsers, you don't need to install the mobile tooling. For information on developing web applications, see Getting Started with Oracle JET Web Application Development

Install the Mobile Tooling

To create Oracle JET hybrid mobile applications, you must first install Cordova on your development machine. To develop hybrid mobile applications for Android, iOS, or Windows, you must also install the Android, iOS, or Windows development tools.

To install the mobile tooling:

If needed, install the tooling prerequisites as described in Install the Prerequisite Packages.

1. Install Apache Cordova
2. (Optional) Install Android Development Tools
3. (Optional) Install iOS Development Tools
4. (Optional) Install Windows Development Tools

Install Apache Cordova

Install Apache Cordova on your development machine.

Cordova is an open-source cross-platform development framework that enables application developers to use standard web technologies (HTML, CSS, and JavaScript) to develop hybrid mobile applications. JET uses plugins developed with the Apache Cordova framework to access the capabilities of the devices on which your JET mobile application is installed such as, for example, camera and GPS. JET also uses Cordova to build and serve hybrid mobile applications by invoking the Cordova command-line interface that you install with Cordova.

As Administrator on Windows or using sudo on Macintosh and Linux systems, enter the following command to install Cordova:

```
[sudo] npm install -g cordova
```

This installs the Cordova command-line interface and supporting tools that Oracle JET's CLI invokes when you issue commands such as `ojet create` and `ojet serve`.

Oracle JET recommends that you keep the Cordova command-line interface up-to-date as newer releases of Cordova provide support for recent versions of mobile operating systems in addition to bug fixes and other enhancements.

Use the following command to update to the latest version of the Cordova command-line interface:

```
npm install -g cordova@latest
```

Use the following npm command to view the currently-installed version of the Cordova command-line interface on your system:

```
cordova --version
```

Each time that you scaffold a hybrid mobile application for one or more mobile platforms, a platform-specific Cordova module is installed for each platform (Android, iOS, Windows, and so on) that you specify when you scaffold the hybrid mobile application. These platform-specific module versions are separate from each other and separate from the version of the Cordova command-line interface. The following output shows the terminal window output where commands have been issued to retrieve the version of the Cordova command-line interface (7.0.1) and to list the installed and available versions of Cordova platform (android 6.2.3 and so on).

```
D:\JETapps\apps\JETDocExample\hybrid>cordova --version
7.0.1
...
D:\JETapps\apps\JETDocExample\hybrid>cordova platform
Installed platforms:
  android 6.2.3
  browser 4.1.0
Available platforms:
  blackberry10 ~3.8.0 (deprecated)
  webos ~3.7.0
  windows ~5.0.0
```

For additional information about managing Cordova platforms, see Platforms and Plugins Version Management and for an overview of Cordova, see Overview in Cordova's documentation.

Install Android Development Tools

Install the Android SDK to deploy a JET hybrid mobile application to Android devices.

The Android SDK provides the tools that build and package your application into an .APK file (the file type that installs applications on Android devices), an emulator to create Android Virtual Devices (AVD) where you can test your application if you do not have access to a physical Android device, and an OEM USB driver to connect your development machine to a physical Android device through a USB cable if you do have a device. This last option enables you to deploy an application from your development machine to the Android device.

Android Studio, Google's IDE for Android development, includes the Android SDK in its installation and provides wizard options that simplify the management of the SDK platforms and tools that you need.

Install Android Studio, and the Android SDK that it includes, by downloading the installation file from https://developer.android.com/studio/index.html. The Android Developer's website provides installation instructions for Windows, Mac, and Linux. See https://developer.android.com/studio/install.html.

Once you have installed Android Studio, perform the tasks described in the following topics:

- Install an Emulator Accelerator
- Create an Android Virtual Device
- Set Up Your Android Device to Install an App from Your Development Machine
- Configure Gradle Proxy Settings
- Configure Environment Variables to Reference JDK and Android SDK Installations

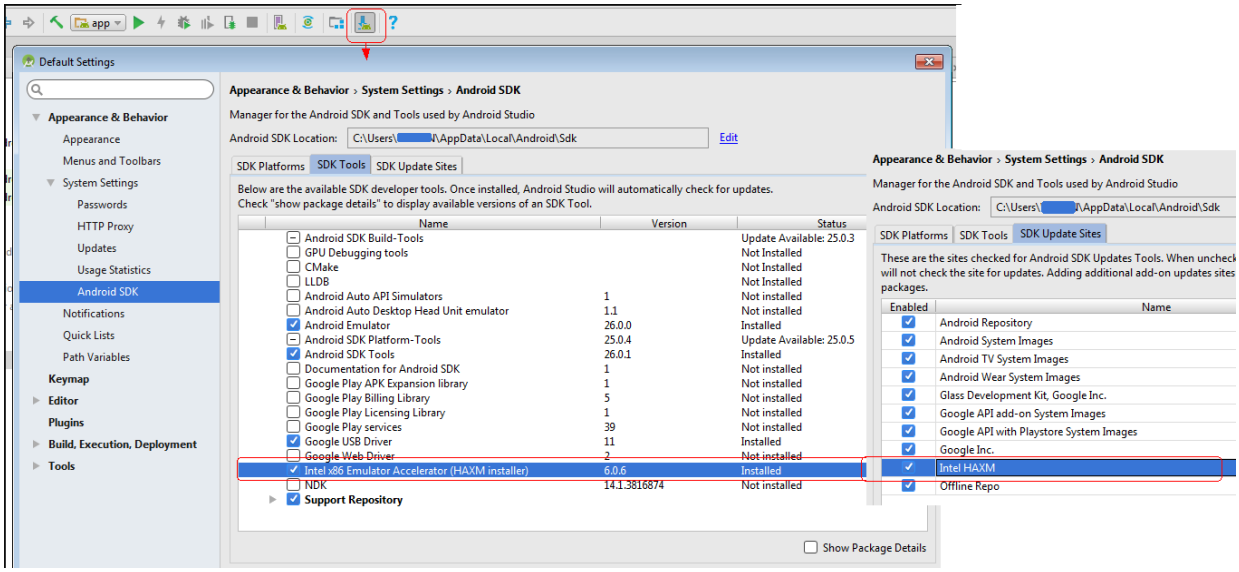
Android requires that all .APK files be digitally signed with a certificate before they can be installed. For apps in development, the Android SDK automatically creates a debug keystore and certificate and sets the keystore and key passwords the first time that you build an .APK file. On macOs and Unix, it creates these resources in `$HOME/.android/debug.keystore`. On Windows, they are in the directory referenced by the `%USERPROFILE%` variable (for example, `C:\Users\JDOE\.android`).

When you have completed development and want to sign and package the Android app for distribution, see Packaging a Hybrid Mobile App on Android.

Install an Emulator Accelerator

You can accelerate the performance of the emulator on Intel x86–based development machines by installing an emulator accelerator.

The emulator accelerator speeds up the performance of the emulator and the AVDs that it emulates by allocating additional resources from your development machine. Once installed, the accelerator appears in the SDK Tools list of the SDK Manager, as shown in the image below. Use Android Studio's **Tools**, **Android**, and **SDK Manager** menu options to open the SDK Manager or the icon that appears on the Android Studio toolbar.



Make sure that the update site for the emulator accelerator that you want to download is selected in the **SDK Update Sites** tab as shown. Once downloaded, execute the

installer. See https://developer.android.com/studio/run/emulator-acceleration.html#accel-vm.

Create an Android Virtual Device

An Android Virtual Device (AVD) replicates an Android device on your development computer. It is a useful option for testing, especially if you only have access to one or a limited range of physical Android devices.

The AVD Manager that you launch from Android Studio by clicking **Tools**, **Android**, and **AVD Manager** has a range of ready-to-use virtual devices, mostly of those devices developed by Google itself, such as the Nexus and Pixel XL range. Other Android device vendors, such as Samsung, provide specifications on their websites that you can use to create the AVD yourself.

Google maintains documentation describing how to manage AVDs. See https://developer.android.com/studio/run/managing-avds.html.

To create an AVD:

1. In Android Studio, launch the Android Virtual Device Manager by selecting **Tools** > **Android** > **AVD Manager**.
2. In the Your Virtual Devices screen, click **Create Virtual Device**.
3. In the Select Hardware screen, select a phone device, such as Pixel, and then click **Next**.
4. In the System Image screen, click **Download** for one of the recommended system images. Agree to the terms to complete the download.
5. After the download completes, select the system image from the list and click **Next**.
6. On the next screen, leave all the configuration settings unchanged and click **Finish**.
7. In the Your Virtual Devices screen, select the device you just created and click **Launch this AVD in the emulator**.

Set Up Your Android Device to Install an App from Your Development Machine

You can install your app directly from your development machine to your Android device by configuring the Android device and connecting it to your development machine using a USB cable.

To set up your Android device:

1. Connect your device to your development machine with a USB cable. If you are developing on Windows, you might need to install the appropriate USB driver for your device. For help installing drivers, see the OEM USB Drivers document.
2. Enable USB debugging on your device by going to **Settings** > **Developer options**.

> **Note:**
>
> On Android 4.2 and newer, Developer options is hidden by default. To make it available, go to **Settings** > **About phone** and tap **Build number** seven times. Return to the previous screen to find **Developer options**.

Configure Gradle Proxy Settings

Gradle is the tool that the Android SDK invokes to build the apps that you deploy to your AVD or Android device.

If you work inside a corporate network, create a `gradle.properties` file in your `~/.gradle` directory that includes the following properties so that you can successfully serve your app to the AVD or physical device.

```
systemProp.http.proxyHost=www-proxy.server.url.com
systemProp.http.proxyPort=proxy-port-number
systemProp.https.proxyHost=www-proxy.server.url.com
systemProp.https.proxyPort=proxy-port-number
```

If you are a Windows user, you locate your `~/.gradle` directory by navigating to `%USERPROFILE%\.gradle`. If you are a macOS or Linux user, navigate to `~/.grade`.

Configure Environment Variables to Reference JDK and Android SDK Installations

To function correctly, the Cordova CLI that the Oracle JET CLI communicates with requires that you configure certain environment variables.

The Cordova CLI attempts to set these environment variables for you, but in certain cases you may need to set them manually. In any event, it is a good idea to review and confirm the values of the following environment variables before you start development:

- Set `JAVA_HOME` to the location of your JDK installation
- Set `ANDROID_HOME` to the location of your Android SDK installation

We also recommend that you add the Android SDK's `tools`, `tools/bin`, and `platform-tools` directories to your `PATH` environment variable.

The process to set the environment variables depends upon your development environment:

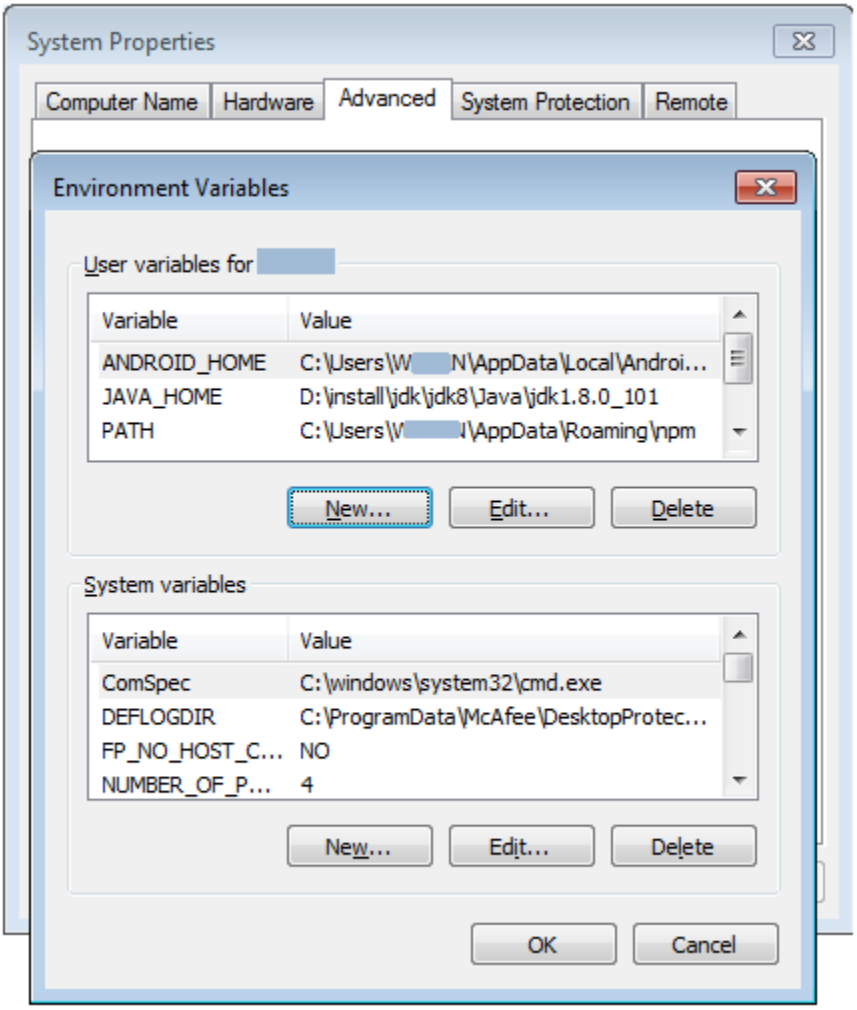
- Windows

  Set the variables permanently in **Control Panel** > **System and Security** > **System** > **Advanced System Settings** > **Environment Variables**.

  For example, to include the Android SDK's tools and platform-tools directories in your `PATH` environment variable, modify the entry that the Environment Variables dialog displays to include entries similar to the following:

  ```
  Path=C:\Users\JET\AppData\Local\Android\sdk\tools;C:\Users\JET\AppData\Local
  \Android\sdk\platform-tools;C:\Users\JET\AppData\Roaming\npm
  ```

  To set the `JAVA_HOME` and `ANDROID_HOME` environment variables, click **New** in the Environment Variables dialog and set the values of the variables to the location of the JDK and Android SDK locations, as shown in the following image.



If you have opened a command window, close it and open a new one for the change to take effect.

- Macintosh and Linux

  On Mac and Linux systems, the settings depend upon your default shell. For example, on a Mac system using the Bash shell, add the following lines in `~/.bash_profile`:

  ```
  export JAVA_HOME=/Development/jdkInstallDir/
  export ANDROID_HOME=/Development/android-sdk/
  ```

  To update your PATH environment variable, add a line resembling the following where you substitute the paths with your local Android SDK installation's location:

  ```
  export PATH=${PATH}:/Development/android-sdk/platform-tools:/Development/android-sdk/tools
  ```

  Reload your terminal to see this change reflected or run the following command:

  ```
  $ source ~/.bash_profile
  ```

  On a Linux system using C Shell, you would use the following format:

```
setenv JAVA_HOME=/Development/jdkInstallDir/
setenv ANDROID_HOME=/Development/android-sdk/
```

Reload your terminal to see this change reflected or run the following command:

```
~/.cshrc
```

Install iOS Development Tools

To develop applications for the iOS platform, you must install the Xcode development environment from the App Store which is only available on macOS.

After installing Xcode, execute the following command in a terminal window to install the Xcode command line tools:

```
xcode-select –install
```

These steps are sufficient for developing iOS applications and testing on iOS simulators. If, however, you want to use an actual iOS device for testing or to publish your application to the App Store, you must join the Apple iOS Developer program and create an appropriate provisioning profile. For additional information, see Maintaining Identifiers, Devices, and Profiles in Apple's documentation. For details about using the Oracle JET tooling to package and deploy your hybrid mobile application, see Packaging a Hybrid Mobile App on iOS.

In addition you must install a tool that is used by the Oracle JET CLI to deploy apps to iOS devices by executing the following command in a terminal window:

```
sudo npm install -g ios-deploy --unsafe-perm=true --allow-root
```

Once you have installed the iOS development tools, you can create a hybrid mobile application, as described in Create a Hybrid Mobile Application.

Install Windows Development Tools

JET hybrid mobile apps for Windows devices must be developed on computers that run the Windows 10 operating system.

JET supports the creation of hybrid mobile apps that are Universal Windows Platform (UWP) apps. UWP is the platform for Windows 10. For more information, see What's a Universal Windows Platform (UWP) app? in Microsoft's documentation.

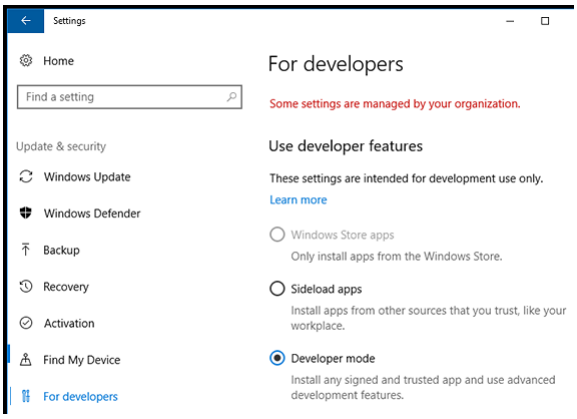
Enable Developer Mode on Windows 10

Developer mode allows you to install any signed and trusted apps to your Windows development machine, whereas the default Windows 10 setting only permits the installation of apps from the Windows Store.

1. From the **For developers** settings dialog, choose **Developer mode**.
2. Read the disclaimer in the dialog that appears and click Yes to accept the change.

> **Note:**
>
> If your device is owned by an organization, some options might be disabled by your organization. Contact your system administrator to ensure you have the options enabled that you require.

Figure 2-1 Enable Developer Mode on a Windows 10 Computer



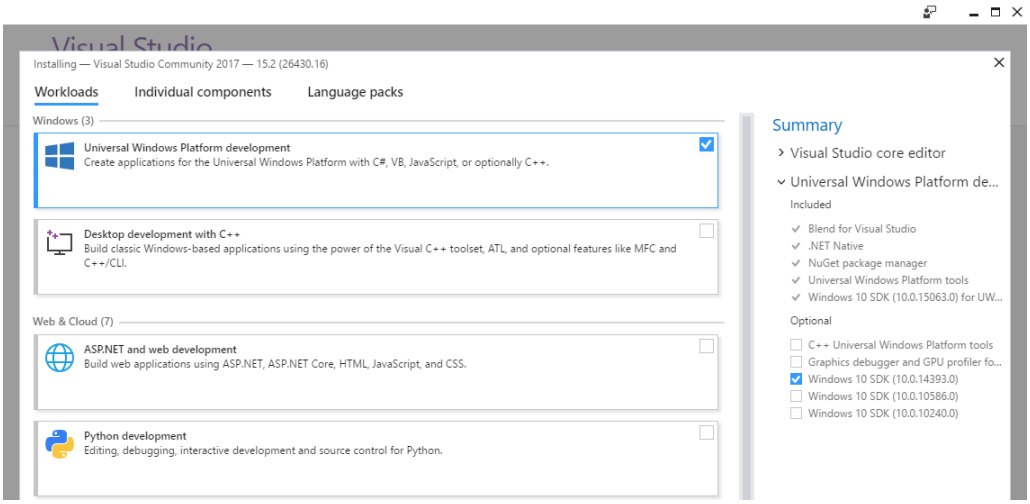
For information about Developer mode, and other settings that configure development on Windows 10 computers, see https://docs.microsoft.com/en-us/windows/uwp/get-started/enable-your-device-for-development.

Install Visual Studio

Install Visual Studio as it includes the Windows 10 SDK and the Universal Windows App Development Tools that you require to develop and deploy a JET hybrid mobile application on your computer.

Microsoft offers a number of editions of Visual Studio. The Visual Studio Community edition is a free, fully-featured IDE for students, open-source and individual developers. Visual Studio can be downloaded from https://www.visualstudio.com/vs/. During download and installation, select the **Universal Windows Platform development** workload and the latest version of the Windows 10 SDK that appears under the Optional list of items that you can install with the workload, as shown in the following image.

Figure 2-2 Installing Visual Studio



Run Visual Studio after you install it to execute any required setup tasks. This prompts you to enable Developer mode, if you have not already enabled this mode on your computer.

Install a Personal Information Exchange File in Your Computer's Certificate Store

Install a personal information exchange file (.PFX) in certificate stores on your Windows 10 computer. The PFX file packages a private key file and a certificate file that you use to sign your app.

This is a one-time task that, once complete, enables you to build and serve JET hybrid mobile applications in debug mode on your Windows 10 computer. You also require a PFX file to sign your JET hybrid mobile application when you complete development and want to distribute it.

The following procedure describes how to create PVK and CER files without password protection that you package into a PVK file that is also without password protection. For more information about the switches and arguments supported by the Microsoft tools that you use to create and package a PFX file, see the following Microsoft documentation:

- MakeCert
- Pvk2Pfx

For more information about packaging a completed JET hybrid app for Windows, see Packaging a Hybrid Mobile App on Windows.

To create and install a PFX file:

1. At a command prompt with Administrator privileges, enter the following commands:
   a. Change to the directory that contains the Windows SDK. For example:

      ```
      chdir "C:\Program Files (x86)\Windows Kits\10\bin\x64"
      ```

   b. Use the MakeCert tool to create certificate files with the specified parameters. For example:

      ```
      makecert.exe -sv C:\aDirectory\doc.pvk -n "CN=Doc
      Example,OU=JET,O=Oracle,C=US" -r -h 0 -eku
      "1.3.6.1.5.5.7.3.3,1.3.6.1.4.1.311.10.3.13" C:\aDirectory
      \doc.cer
      ```

   c. Package the resulting certificate files into a PFX file. For example:

      ```
      pvk2pfx.exe -pvk C:\aDirectory\doc.pvk -spc C:\aDirectory
      \doc.cer -pfx C:\aDirectory\doc.pfx
      ```

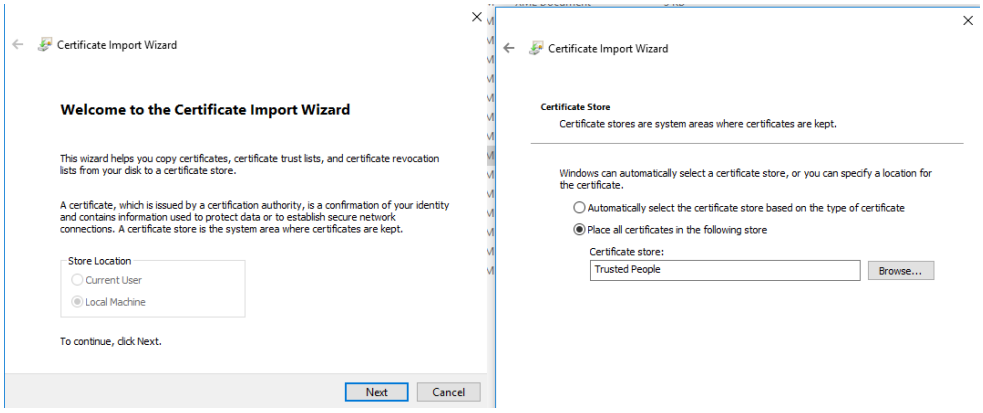
   This creates a PFX file named `doc.pfx` in the `C:\aDirectory\` directory.
2. Install the PFX file in the following Certificate Stores on your computer

Store Location	Certificate Store
Current User	Personal
Current User	Trusted People
Local Machine	Trusted People

3. To do this, double-click the PFX file to invoke the Certificate Import Wizard, choose the appropriate store location (for example, Current User) and click **Next**

to proceed to the page where you choose the Certificate Store, as shown by the following figure where the Trusted People certificate store is selected.

> **Note:**
>
> The file created in our previous example is not password-protected, so leave the Password input field blank in the Private key protection page of the wizard.



4. Click **Finish** to complete the installation of the certificate in the certificate store and repeat the process to install the PFX file in the remaining certificate stores.

Create a Hybrid Mobile Application

Use the Oracle JET mobile tooling commands to create, build, run, and customize hybrid mobile applications for Android, iOS, and Windows mobile devices. You can create an application that contains a blank template or one pre-configured with layouts and styling for the desired platform.

Before you use the mobile tooling, verify that you have installed all the prerequisite packages and configured your target platforms if needed. For additional information, see Install the Mobile Tooling.

1. Scaffold a Hybrid Mobile Application with the Oracle JET CLI
2. (Optional) Build a Hybrid Mobile Application with the Oracle JET CLI
3. Serve a Hybrid Mobile Application with the Oracle JET CLI
4. Review Your Application Settings in the config.xml File
5. Change the Splash Screen and App Launcher Icon

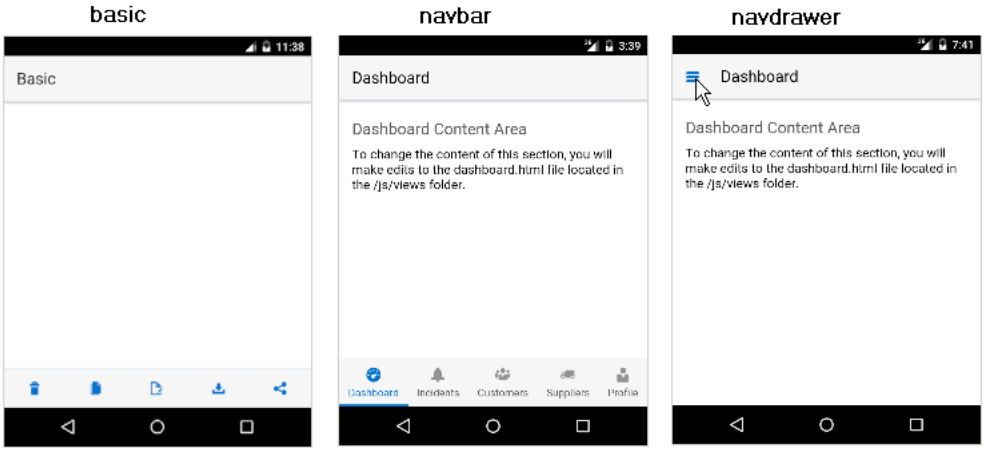
> **Note:**
>
> You can add hybrid platforms (Android, iOS, and Windows) to an existing web application. See Add Hybrid Mobile Features to Web Applications.

Scaffold a Hybrid Mobile Application with the Oracle JET CLI

Use the Oracle JET CLI to scaffold a hybrid mobile application for iOS, Android, and Windows mobile devices. You can scaffold the application to contain a blank template or one pre-configured with layout styles, a navigation bar, or a navigation drawer. After scaffolding, you can modify the app as desired.

Oracle JET provides a set of templates designed for hybrid mobile apps in addition to the templates for web apps. JET uses the hybrid version of the template by default when scaffolding a hybrid mobile app.

The following image shows the differences between the templates as shown on an Android Nexus 4 emulator using the Android Alta theme. The blank template contains `index.html` but no UI features, and is not shown in the image that follows. The basic template adds styling but no sample content, and can be used as the basis for your app if you have no navigational requirements or if you want to design the entire app yourself. The navbar and navdrawer templates contain sample content and best practices for layouts and styling that you can also modify as needed. The same app will run on any Android, iOS, or Windows emulator or device, using the appropriate Alta theme for the platform.



> **Note:**
>
> Do not confuse the basic template which contains no content with the QuickStart Basic sample application in the NetBeans JET Support plugin. The QuickStart Basic sample application uses a navdrawer layout and contains sample content.

At a command prompt, enter `ojet create --hybrid` with optional arguments to create the Oracle JET application and scaffolding.

```
ojet create [directory] --hybrid
                    [--appid=application-id] [--appname=application-name]
                    [--template={template-name:[web|hybrid]|template-url|template-
file}]
                    [--platforms=android,ios,windows|--platform=android|ios|windows]
```

The following table describes the available options and provides examples for their use.

> **Tip:**
>
> You can also enter `ojet help` at a terminal prompt to get additional help about the Oracle JET CLI.

Option	Description
`directory`	Application location. If not specified, the application is created in the current directory.
`appid`	Application ID entered in reverse domain style: `com.mydomain.myappname`. The `appid` is equivalent to `Package Name` on Android and `Bundle ID` on iOS. If not specified, the `appid` defaults to `org.oraclejet.`*`directory`*, using the directory you specified in the scaffolding command. For example, if you specified `MyApp` for the directory, the default `appid` will be `org.oraclejet.MyApp`.
`appname`	Application name displayed on device. To include spaces in the title, use quotes: `--appname="My Sample Application"`. If not specified, the `appid` defaults to the directory you specified in the scaffolding command.

`template`	Template to use for the application. Specify one of the following:
	• *`template-name`*
	Predefined template. You can specify `basic`, `blank`, `navbar` , or `navdrawer`. If you don't specify a template, your application will be configured with the `blank` template.
	• *`template-URL`*
	URL to zip file containing the name of the zipped application: `http://`*`path-to-app/app-name`*`.zip`.
	• *`template-file`*
	Path to zip file on your local file system containing the name of a zipped application: `"`*`path-to-app/app-name`*`.zip"`. For example:
	`--template="C:\Users\SomeUser\app.zip"`
	`--template="/home/users/SomeUser/app.zip"`
	`--template="~/projects/app.zip"`

`platforms`	Comma-separated list of platform names. You can specify one or more of the following: `ios`, `android`, or `windows` without spaces. For example: `ios,android`. If you don't specify a platform, the command will prompt you for your choice of platform after verifying that the necessary prerequisites are available.
`platform`	Platform name. You can specify one of the following: `ios`, `android`, or `windows`. If you don't specify a platform, the command will prompt you for your choice of platform after verifying that the necessary prerequisites are available.

WARNING:

Do not use `internal` in the `appid`. This is one of a number of undocumented Cordova reserved words, and the scaffolding will fail with the following error: `App id contains a reserved word, or is not a valid identifier`. To work around this limitation:

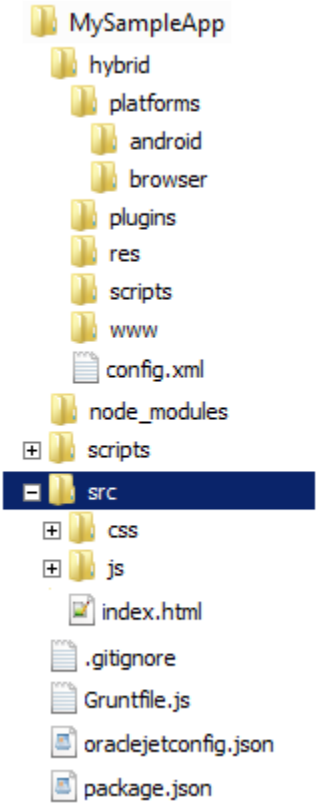
1. Scaffold your application with a temporary `appid`: `appid=com.mycompany.temp`.
2. Change to the application's `hybrid` directory and edit `config.xml` to replace the temporary `appid` with the new `appid`: `id="com.mycompany.internal"`.

For example, the following command creates a hybrid mobile application named `Sample NavBar` in the `MySampleApp` directory using the `navbar` template and targeted for Android devices:

```
ojet create MySampleApp --hybrid --appname="Sample NavBar" --template=navbar --
platform=android
```

The scaffolding will take some time to complete. When successful, the terminal will display: `Your app is ready! Change to your new app directory MySampleApp...`

The new application will have a directory structure similar to the one shown in the following image.



The application folders contain the application and configuration files that you will modify as needed for your own application.

Directory	Description
`hybrid`	Contains platform-specific files that are merged in with a copy of the application files at build time. Most of the files in this directory are staging files that you should not commit to your source control system. One exception is the `hybrid/config.xml` file.

`hybrid/config.xml`	Contains the Cordova global configuration settings. You can edit the `config.xml` file to specify core Cordova API features, plugins, and platform-specific settings.
	For example, the following entry sets the application's orientation to landscape-only on all platforms.
	`<?xml version='1.0' encoding='utf-8'?>`
	`<widget`
	`... contents omitted`
	`<preference name="Orientation" value="landscape" />`
	`</widget>`
	Commit the `config.xml` file to your source control system, if you use one. For more information about the `config.xml` file, see Review Your Application Settings in the config.xml File.

`scripts`	Contains the `oraclejet-build.js` and `oraclejet-serve.js` files that you can modify to customize your application's build and serve steps. See Customize a Web Application's Build Behavior and Customize a Web Application's Serve Behavior.
`node_modules`	Contains the Node.js modules used by JET tooling.
`src`	Site root for your application. Contains the application files that you can modify as needed for your own application and should be committed to source control. The content will vary, depending upon your choice of template. Each template, even the blank one, will contain an `index.html` file and a `main.js` RequireJS bootstrap file. Other templates may contain view templates and viewModel scripts pre-populated with content. For example, if you specified the navbar template during creation, the `js/views` and `js/viewModels` folders will contain the templates and scripts for a hybrid mobile application that uses a nav bar for navigation.

Modify the Hybrid Mobile Application's File Structure

You can modify your scaffolded application's file structure if the default structure doesn't meet your needs.

The `oraclejetconfig.json` file in your application's top level directory contains the default source and staging file paths that you can modify.

```
{
  "paths": {
    "source": {
      "common": "src",
      "web": "src-web",
      "hybrid": "src-hybrid",
      "javascript": "js",
      "styles": "css",
```

```
      "themes": "themes"
    },
    "staging": {
      "web": "web",
      "hybrid": "hybrid",
      "themes": "themes"
    }
  },
"serveToBrowser": "chrome",
"generatorVersion": "x.x.x"
}
```

To change the hybrid mobile application's file structure:

1. In your application's top level directory, open `oraclejetconfig.json` for editing.
2. In `oraclejetconfig.json`, change the paths as needed and save the file.

   For example, if you want to change the default `styles` path from `css` to `app-css`, edit the following line in `oraclejetconfig.json`:

   ```
   "styles": "app-css"
   ```

3. Rename the directories as needed for your application, making sure to change only the paths listed in `oraclejetconfig.json`. For example, if you changed `styles` to `app-css` in `oraclejetconfig.json`, change the application's `css` directory to `app-css`.
4. Update your application files as needed to reference the changed path

   For example, if you modify the path to the CSS for your application, update the links appropriately in your application's index.html.

   ```
   <link rel="icon" href="app-css/images/favicon.ico" type="image/x-icon" />

       <!-- This is the main css file for the default Alta theme -->
       <!-- injector:theme -->
       <link rel="stylesheet" href="app-css/libs/oj/vx.x.x/alta/oj-alta-min.css"
   type="text/css"/>
       <!-- endinjector -->

       <!-- This contains icon fonts used by the starter template -->
       <link rel="stylesheet" href="app-css/demo-alta-site-min.css" type="text/
   css"/>

       <!-- This is where you would add any app specific styling -->
       <link rel="stylesheet" href="app-css/override.css" type="text/css"/>
   ```

To use the new paths, in your application's top level directory, build your application as described in Build a Hybrid Mobile Application with the Oracle JET CLI.

Build a Hybrid Mobile Application with the Oracle JET CLI

Use Oracle JET CLI to build your Android, iOS, or Windows hybrid mobile application for testing and debugging.

Change to the application's root directory and use the `ojet build` command for each platform that your hybrid mobile application will support.

```
ojet build [android|ios|windows]
           [--build-config=path/buildConfig.json --destination=device|emulator
            --theme=themename[:android|ios|web|windows] --themes=theme1,theme2,...
```

```
--sass
--platform-options="string"]
```

The following table describes the commonly-used options and provides examples for their use.

> **Tip:**
>
> You can also enter `ojet help` at a terminal prompt to get help for specific Oracle JET CLI options.

Option	Description
`[android|ios|windows]`	Desired platform. Enter `android`, `ios`, or `windows`.
`build-config`	Path to `buildConfig.json`. The `buildConfig.json` file contains details that Cordova can use to sign the application. You do not need this file when building a debug version of your app for Android or Windows, or if you are building your app for deployment to an iOS simulator. However, you must configure one for testing on an iOS device and when you're ready to release your Android, iOS, or Windows application. For additional information, see Packaging and Publishing Hybrid Mobile Applications.
`destination`	Required for iOS applications. Specify one of the following: • `emulator`: builds your application for deployment to the iOS Simulator. • `device`: builds your application for deployment to an iOS device. Be aware, though, if you want to deploy your application to an iOS device, you must take additional steps as described in Packaging a Hybrid Mobile App on iOS. Typically, you can develop and test your iOS application in the simulator or web browser until you're ready to build your release.
`device`	Equivalent to `destination=device`.
`emulator`	Equivalent to `destination=emulator`.
`theme`	Theme to use for the application. The theme defaults to `alta:platform`, but you can specify an alternate theme, such as a custom theme as described in Customizing Themes Using the Tooling.
`themes`	Themes to include in the application, separated by commas without spaces. If you don't specify the `--theme` flag as described above, Oracle JET will use the first element that you specify in `--themes` as the default theme.
`sass`	Manages Sass compilation. If you add Sass and specify the `--theme` or `--themes` option, Sass compilation occurs by default and you can use `--sass=false` or `--no-sass` to turn it off. If you add Sass and do not specify a theme option, Sass compilation will not occur by default, and you must specify `--sass=true` or `--sass` to turn it on. For more information, see Customizing Themes Using the Tooling.

Option	Description

`platform-options`	Platform-specific options that will pass verbatim to the Cordova CLI.
	This option is typically required for Windows device deployments or if a Cordova plugin contains multiple static libraries for different CPUs. If the value passed contains a quoted string, the quotation marks must be escaped.
	For example, you can use platform-options to specify Windows architectures. By default, the architecture defaults to `anycpu`.
	To specify a single architecture, use the `--arch` option and specify `arm`, `x86`, `x64`, or `anycpu`.
	`--platform-options="--arch arm|x86|x64|anycpu"`
	To specify multiple architectures, use the `--archs` option with a space-separated list passed in as a quoted string. Note that you must escape the quotation marks as shown below.
	`--platform-options="--archs=\"arm x86 x64\""`

For example, the following command will build the sample Android application shown in Scaffold a Hybrid Mobile Application with the Oracle JET CLI

```
ojet build android
```

The command will take some time to complete. If it's successful, you'll see the following message: `Done`. The command will also output the name and location of the built application in `hybrid/platforms/android` , `hybrid/platforms/ios`, or `hybrid/platforms/windows`.

By default `ojet build` creates a debug version of your application. You can also use the `ojet build` command with the `--release` option to build a release-ready version of your application. For information, see Packaging and Publishing Hybrid Mobile Applications.

WARNING:

If you use a proxy server and specify the Android platform, the build command will fail the first time you issue it. To resolve this, create a `gradle.properties` file in your *`HOME`*`/.gradle` directory and rerun the build command. The file should contain the following:

```
systemProp.http.proxyHost=proxy-server-URL
systemProp.http.proxyPort=80
systemProp.https.proxyHost=proxy-server-URL
systemProp.https.proxyPort=80
```

Customize a Hybrid Mobile Application's Build Behavior

Edit `oraclejet-build.js` to modify default build behavior. For example, you can add additional libraries to your application or change the default markers used to inject links into the application's `index.html` during build time.

The Oracle JET tooling defines default build and serve behavior in `node_modules/@oracle/oraclejet-tooling/lib/defaultconfig.js`. The first part of the file contains the default build configuration, including entries for application paths and other options used during the build process. To modify defaults or add your own, edit `oraclejet-build.js` and remove comments from the options that you want to configure. The file contains detailed instructions for each configurable option.

Oracle JET will merge the options with the default configuration during the build. However, be aware that `fileList` options will completely replace the corresponding option in the default configuration and will not be merged.

To customize build behavior:

1. Open `scripts/config/oraclejet-build.js` for editing in your favorite text editor.
2. Identify the entry that you want to change and remove the comments (//) preceding each line.

   For example, to add a custom library to the build's staging directory for hybrid applications, remove the comments from the following section in `oraclejet-build.js` as shown below:

```
/**
 * This is the hybrid specific configuration. You can specify configurations
targeted only hybrid apps.
 * The hybrid specific configurations will override the general configuration.
 */
hybrid: {
  copyCustomLibsToStaging: {
    fileList: [
      {
        cwd:'node_modules/oraclejet/',
        src: ['*'],
        dest: 'hybrid/www/js/libs/oraclejet'
      }
    ]
  }
}
```

3. Edit the options as desired and save the file.

   For example, to add a custom library named `my_custom_module`, place the module in `node_modules`, and edit the `copyCustomLibsToStaging` entry:

```
hybrid: {
  copyCustomLibsToStaging: {
    fileList: [
      {
         cwd:'node_modules/my_custom_module/',
         src: ['*'],
         dest: 'hybrid/js/libs/my_custom_module'
      }
    ]
```

```
    }
  }
```

4. Repeat the previous step for each option that you want to modify.

Test your changes by running Build a Hybrid Mobile Application with the Oracle JET CLI with appropriate options.

Serve a Hybrid Mobile Application with the Oracle JET CLI

Use Oracle JET CLI to launch your hybrid mobile application in a browser, simulator, or mobile device for testing and debugging. When you serve your application to a browser or emulator, a live reload option is enabled, and changes you make to the code are immediately reflected in the running application.

At a command prompt, change to the application's top level directory and use the `ojet serve` command with options to launch the application.

```
ojet serve [ios|android|windows]
           [--build-config=path/buildConfig.json
            --server-port=server-port-number --livereload-port=live-reload-port-
number
            --destination=emulator[:emulator-name]|browser[:browser-name]|
device[:device-name]|server-only
            --livereload --build
            --theme=themename[:android|ios|windows|web]] --themes=theme1,theme2,...
            --sass
            --platform-options="string"]
```

The following table describes the commonly-used options and provides examples for their use.

> **Tip:**
>
> You can also enter `ojet help` at a terminal prompt to get help for specific Oracle JET CLI commands.

Option	Description
`[ios|android|windows]`	Desired platform. Enter `android`, `ios`, or `windows`.
`build-config`	Specify the path to `buildConfig.json`. The `buildConfig.json` file contains details that Cordova can use to sign the application. You do not need this file when building a debug version of your application for Android or Windows, or if you are building your app for deployment to an iOS simulator. However, you must configure one for testing on an iOS device and for pre-release testing of your Android, iOS or Windows application. For additional information, see Packaging and Publishing Hybrid Mobile Applications.
`server-port`	Server port number. If not specified, defaults to 8000.
`livereload-port`	Live reload port number. If not specified, defaults to 35729.

Option	Description

`destination`	Specify one of the following:
	• `emulator`: Displays your application in the default Android AVD, iOS Simulator, or Windows Emulator.
	To use a different emulator, append its name to the `emulator` option: `--destination=emulator:`*`emulator-name`*.
	Tip:
	You can view the list of available emulators for each platform by invoking the following from your app's top-level folder:
	`hybrid/platforms/{platform}/cordova/lib/list-emulator-images`
	where `{platform}` is one of android, ios, or windows.
	• `browser`: Displays your application in the Chrome browser on your local machine.
	If you don't have Chrome installed or want to use a different browser, append the name of the desired browser to the `browser` option:
	`--destination=browser:firefox|edge|ie|opera|safari|chrome`
	Tip:
	To change your application's default browser from Chrome, open `oraclejetconfig.json` in the application's top level directory and change the name of the browser in `defaultBrowser`. For example, to change the default browser to Firefox, edit `oraclejetconfig.json` as shown below.
	`"defaultBrowser": "firefox"`
	• `device`: Sends the application to an attached device.
	Optionally, append the name of the device to `device` option: `--destination=device:myDevice`.
	Note:
	If you want to send your application to an iOS device, you must take additional steps as described in Packaging a Hybrid Mobile App on iOS.
	• `server-only`: Serves the application, as if to a browser, but does not launch a browser. Use this option in cloud-based development environments so that you can attach your browser to the app served by the development machine.

`browser[=`*`browser-name`*`]`	Equivalent to `destination=browser[:`*`browser-name`*`]`.
`emulator[=`*`emulator-name`*`]`	Equivalent to `destination=emulator[:`*`emulator-name`*`]`.

Option	Description
`device[=device-name]`	Equivalent to `destination=device[:device-name]`.
`server-only`	Equivalent to `destination=server-only`.
`livereload`	Enable the live reload feature. Live reload is enabled by default (`--livereload=true`). Use `--livereload=false` or `--no-livereload` to disable the live reload feature. Disabling live reload can be helpful if you're working in NetBeans or another IDE and want to use that IDE's mechanism for loading updated applications.
`build`	Build the app before you serve it. By default, an app is built before you serve it (`--build=true`). Use `--build=false` or `--no-build` to suppress the build if you've already built the application and just want to serve it.
`theme`	Theme to use for the application. The theme defaults to `alta:platform`, but you can specify an alternate theme. You can also enter a different *themename* with optional platform for a custom theme as described in Customizing Themes Using the Tooling .
`themes`	Themes to use for the application, separated by commas. If you don't specify the `--theme` flag as described above, Oracle JET will use the first element that you specify in `--themes` as the default theme. Otherwise Oracle JET will build the application with the theme specified in `--theme`.
`sass`	Manages Sass compilation. If you add Sass and specify the `--theme` or `--themes` option, Sass compilation occurs by default and you can use `--sass=false` or `--no-sass` to turn it off. If you add Sass and do not specify a theme option, Sass compilation will not occur by default, and you must specify `--sass=true` or `--sass` to turn it on. For more information, see Customizing Themes Using the Tooling . **Note:** The option that you choose controls both Sass compilation in the build step and Sass watch in the serve step.
`platform-options`	Platform-specific options that will pass verbatim to the Cordova CLI. This option is typically required for Windows device deployments or if a Cordova plugin contains multiple static libraries for different CPUs. If the value passed contains a quoted string, the quotation marks must be escaped. For example, you can use platform-options to specify Windows architectures. By default, the architecture defaults to `anycpu`. To specify a single architecture, use the `--arch` option and specify `arm`, `x86`, `x64`, or `anycpu`. `--platform-options="--arch arm"`

The application will launch in a local browser, emulator/simulator, or device depending upon the options you specify. The following table shows examples.

Command	Description
`ojet serve windows --browser=firefox`	Launches a Windows version of the application in the Firefox browser.
`ojet serve ios`	Launches the application in the iOS Simulator using the Alta iOS theme.
`ojet serve android --destination=emulator:MyEmulator`	Launches the application in the Android emulator using the AVD named “MyEmulator”. The emulator name is case-sensitive.
`ojet serve android --device`	Launches the application on the attached Android mobile device.

The terminal will also output the names of the files as they are loaded. If your application contains multiple views, the output will reflect the names of the views and associated files as you navigate through the application.

WARNING:

If you specify the Android platform, use a proxy server and skipped the `ojet build` step, the `serve` command will fail the first time you issue it. To resolve this, create a `gradle.properties` file in your *`HOME`*`/.gradle` directory and rerun the serve command. The file should contain the following:

```
systemProp.http.proxyHost=proxy-server-URL
systemProp.http.proxyPort=80
systemProp.https.proxyHost=proxy-server-URL
systemProp.https.proxyPort=80
```

Behavior of Live Reload When Enabled

If you left live reload enabled, the terminal window updates to reflect that the code has changed. For example, if you save a change to `dashboard.html` in an application scaffolded with the navbar or navdrawer template, the terminal window outputs the name of the changed file, and the browser or emulator/simulator updates with the change. Live reload is disabled when you serve an application to a device.

To terminate the batch job when using live reload, press Ctrl+C in the command window and then enter `y` if prompted to terminate the batch job.

Note:

You can also use the `ojet serve` command with the `--release` option to serve a release-ready version of your application. For information, see Packaging and Publishing Hybrid Mobile Applications.

Customize a Hybrid Mobile Application's Serve Behavior

Edit `oraclejet-serve.js` to modify default serve behavior. For example, you can add your own watch tasks or specify actions specific to web or hybrid applications.

Oracle JET tooling defines default build and serve behavior in `node_modules/oraclejet-tooling/lib/defaultconfig.js`. The second half of the file contains the default serve configuration, including entries for application paths and other options used during the serve process. To modify defaults or add your own, edit `oraclejet-serve.js` and remove comments from the options that you want to configure. The file contains detailed instructions for each configurable option.

Oracle JET will merge the options with the default configuration during the build. However, be aware that rewriting the three default watch targets (`sourceFiles`, `sass`, and `themes`) can result in unpredictable behavior.

To customize serve behavior:

1. Open `scripts/config/oraclejet-serve.js` for editing in your favorite text editor.
2. Identify the entry that you want to change and remove the comments (//) preceding each line that you want to include.

   For example, to define one or more shell commands that you want to execute after the application is served and you modify a source file, you can edit the `sourceFiles` option in the `watch` sub task. Remove the comments from the highlighted lines as shown below.

```
// // Sub task watch default options
       watch: {
         sourceFiles:
         {
    //     files: [],
    //     options: {
    //       livereload: true
    //     }
         },

    //   sass: {
    //     files: [],
    //     commands: ['compileSass']
    //   },

    //   themes: {
    //     files: [],
    //     options: {
    //       livereload: true
    //     },
    //     commands: ['copyThemes']
    //   },
      }
```

3. Edit the options as desired and save the file.

   For command options, add one or more commands, each surrounded by single quotes and separated by commas. For example, the following code snippet adds `node --version` and `npm --version` to the `commands` option in the `sourceFiles` sub task. Add a comma to the closing braces.

```
 watch: {
   sourceFiles:
   {
//     files: [],
//     options: {
//       livereload: true
//     },
       commands: ['node --version', 'npm --version']
   },
   ... contents omitted
 },
```

Test your changes by running Serve a Hybrid Mobile Application with the Oracle JET CLI with appropriate options. When you change one of the source files, the terminal output will display the output of the `node --version` and `npm --version` commands and reload the page.

Review Your Application Settings in the config.xml File

The `AppRootDir/hybrid/config.xml` file contains a number of default settings that you may want to review and modify before you package and publish your app for distribution to end users.

You can configure entries that affect the behavior of your app across all platforms and entries that are applied only to a specific platform.

Examples of entries for all platforms that you may want to modify in your file before you publish your app include the following:

- Attribute values of the `widget` element. Specifically, the values for the `id` and `version` attributes.

  The value of the `id` attribute determines the unique identifier for this app. By default, JET apps use a combination of reverse domain notation with the `oraclejet.org` domain name as input and the app's short name. Change this to use, for example, the reverse of your company's domain name.
- The value of the `version` attribute identifies the version number of your app to end users. It appears, for example, on the App Info screen on Android devices. Change it to an appropriate value.
- Value for the `name` element. This is the name of the app displayed on the springboard of your user's mobile device and in the app stores.

Apart from these generic settings that affect your app irrespective of the platform where it is deployed (Android, iOS, or Windows), you can configure a range of other entries in the `config.xml` file that set preferences for your app on specific platforms.

The following example `AppRootDir/hybrid/config.xml` file displays a number of illustrative examples.

```
<?xml version='1.0' encoding='utf-8'?>
<widget id="org.oraclejet.docexample" version="1.0.0" xmlns="http://www.w3.org/ns/
widgets"
        xmlns:cdv="http://cordova.apache.org/ns/1.0">
  <name>docexample</name>
  <description>A sample Oracle JavaScript Extension Toolkit (JET) mobile app based
on Cordova</description>
  <author email="undefined" href="http://www.oraclejet.org">Oracle Jet Team</author>
  <content src="index.html"/>
  <plugin name="cordova-plugin-whitelist" spec="1"/>
```

```
  <access origin="*"/>
  <allow-intent href="http://*/*"/>
  ...
  <allow-intent href="geo:*"/>
  <platform name="windows">
    <preference name="windows-target-version" value="10.0"/>
    <icon src="res/icon/windows/Square30x30Logo.scale-100.png" width="30"
height="30"/>
    ...
    <splash src="res/screen/windows/SplashScreenPhone.scale-240.png" width="1152"
height="1920"/>
  </platform>
  <platform name="android">
    <allow-intent href="market:*"/>
    <preference name="DisallowOverscroll" value="true"/>
<!-- The following entry displays your app using the full screen of the Android
device and thus hides the Android
                                                                    device's status
bar and menu button.-->
<preference name="Fullscreen" value="true" />
    <icon src="res/icon/android/icon-ldpi.png" width="36" height="36"/>
    ...
    <splash src="res/screen/android/splash-port-xxxhdpi.9.png" density="port-
xxxhdpi"/>
  </platform>
  <platform name="ios">
    <allow-intent href="itms:*"/>
    <allow-intent href="itms-apps:*"/>
    <preference name="Orientation" value="all"/>
    <icon src="res/icon/ios/icon-small.png" width="29" height="29"/>
    ...
    <splash src="res/screen/ios/Default-Portrait~ipad.png" width="768"
height="1024"/>
  </platform>
  ...
</widget>
```

For more information about the `AppRootDir/hybrid/config.xml` file, see https://cordova.apache.org/docs/en/latest/config_ref/.

Change the Splash Screen and App Launcher Icon

Replace the JET-provided images with those that you want your app to use as a splash screen or an app launcher icon.

The default splash screen behavior of your app depends on the platform where you run the app. On Android, for example, the default behavior is to display a white screen. JET provides a set of splash screen images for each platform that can be used in your app. These images are stored in the following sub-directories of your app's hybrid directory.

```
AppRootDirectory/hybrid/res/screen
+---android
|       splash-land-hdpi.9.png
|      ...
|       splash-port-xxxhdpi.9.png
|
+---ios
|       Default-568h@2x~iphone.png
|      ...
```

```
|       Default~iphone.png
|
\---windows
        SplashScreen.scale-100.png
        SplashScreenPhone.scale-240.png
```

To use these splash screens, install the `cordova-plugin-splashscreen` plugin by executing the following command:

```
ojet add plugin cordova-plugin-splashscreen
```

You can replace the JET-provided splash screen images with your own images, matching the names and sizes. You can change the behavior of the splash screen by configuring your app's `AppRootDir/hybrid/config.xml` file. The following example shows how you display the splash screen for 4000 milliseconds.

```
  <preference name="SplashScreenDelay" value="4000" />
```

For more information about the plugin, including how to configure its behavior plus platform-specific information, see https://cordova.apache.org/docs/en/latest/reference/cordova-plugin-splashscreen/.

JET provides a set of app launcher icons for each platform that can be used in your app. To use an alternative app launcher icon to the JET-provided icons, replace the images in the following directories.

```
AppRootDir/hybrid/res/icon
+---android
|       icon-hdpi.png
|       ...
|       icon-xxxhdpi.png
|
+---ios
|       icon-40.png
|       ...
|       icon@2x.png
|
\---windows
        Square150x150Logo.scale-100.png
        ...
        Wide310x150Logo.scale-240.png
```

Both the splash screen and app launcher icons that your app uses are referenced from your app's `AppRootDir/hybrid/config.xml` file, as shown by the following example excerpts.

```
<platform name="windows">
    ...
    <icon src="res/icon/windows/Square30x30Logo.scale-100.png" width="30"
height="30"/>
    ...
    <splash src="res/screen/windows/SplashScreen.scale-100.png" width="620"
height="300"/>

<platform name="android">
    ...
    <icon src="res/icon/android/icon-ldpi.png" width="36" height="36"/>
    ...
    <splash src="res/screen/android/splash-land-ldpi.9.png" density="land-ldpi"/>
```

```
    ...
<platform name="ios">
    ...
    <icon src="res/icon/ios/icon-small.png" width="29" height="29"/>
    ...
    <splash src="res/screen/ios/Default@2x~iphone.png" width="640" height="960"/>
    ...
```

For more information about icons and their entries in the `config.xml` file, see https://cordova.apache.org/docs/en/latest/config_ref/images.html.

Accessing Mobile Device Services Using Cordova Plugins

You can enable user access to device features, such as camera, geolocation and the local file system, by including Cordova plugins into your JET app.

Topics

- About Apache Cordova and Cordova Plugins
- Using a Plugin in Your App
- Cordova Plugins Recommended by Oracle JET
- Using a Different Web View in your JET Hybrid Mobile App
- Using the Native Platform's Date Picker UI in your JET Hybrid Mobile App

About Apache Cordova and Cordova Plugins

JET uses plugins developed with the Apache Cordova framework to access the capabilities of the devices on which your hybrid mobile application is installed.

Apache Cordova is an open-source cross-platform development framework that enables application developers to use standard web technologies (HTML, CSS, and JavaScript) to develop hybrid mobile applications. A hybrid mobile application refers to mobile applications that are native applications which are installed onto mobile devices in the usual manner, but which use a web view to render the UI rather than using the platform's native UI components. For additional information about Apache Cordova, see the Overview page at http://cordova.apache.org/docs/en/latest/guide/overview/index.html.

Since a hybrid mobile application is developed using standard web technologies, the same code can be reused across the supported platforms (such as Android, iOS and Windows). Apache Cordova provides tools that enable you to add support for the platforms that the application will run on independent of this code.

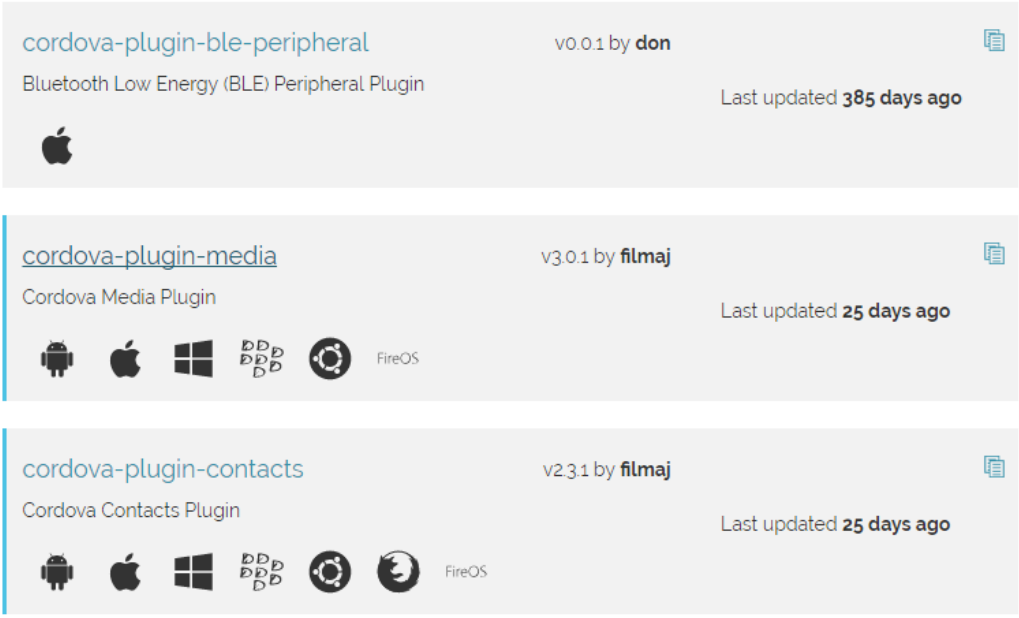
JET uses Apache Cordova to develop web technology-based applications for some of the mobile platforms supported by Apache Cordova. So, one could say that the JET hybrid mobile application that you develop is an Apache Cordova application where JET provides features such as components and themes that determine the look and feel of your application.

A plugin is a package of code that allows the Cordova web view within which your application renders to communicate with the native platform on which it runs. The plugin does this by providing a JavaScript interface to native components that allows your application to use native device capabilities, such as camera, geolocation, and so on.

So, assume for example, that you want your application to use a camera. In that case, you look for an existing plugin to provide access to the camera on the platform(s) where your application will be installed as there is a good possibility that someone has already developed a plugin to address this requirement.

If you cannot find a plugin that meets your requirements, you can develop your own plugin. Although this blog post makes reference to another mobile development framework (MAF), it provides a suitable introduction to creating a Cordova plugin that could be used in a JET application or any Cordova-based application.

To find a plugin to use in your application, go to the Plugins page at https://cordova.apache.org/plugins/ that provides a registry of core and third-party Cordova plugins. Core plugins are plugins provided by Apache Cordova while third-party plugins are those developed by independent developers. Core plugins display a blue vertical strip to the left of the card that describes the plugin, as shown in the following figure, where two core plugins (`cordova-plugin-media` and `cordova-plugin-contacts`) appear after a third-party plugin named `cordova-plugin-ble-peripheral`.



The Plugins page categorizes plugins according to the platform on which the plugins will run. Some plugins run on a wide variety of mobile platforms (Android, iOS, Windows, and so on) while other plugins may only support one platform. Ensure that the plugins you install in your application support the platforms where your application will run.

Each entry generally provides information such as a brief description of the plugin's functionality, the platforms it supports, and the number of days since it was updated. It then links to a web page where you can access more detail, such as how to install the plugin in your application and use it once installed.

Using a Plugin in Your App

You add the plugin and then write code that invokes the plugin's JavaScript interface to access the device APIs.

To add a plugin to your app, change to the application's root directory and use the `ojet add` plugin command. The following example illustrates how you install the phonegap-plugin-barcodescanner plugin.

```
ojet add plugin phonegap-plugin-barcodescanner
```

Once you have added the plugin to your app, you need to write code in your app to use the plugin. The following code excerpts illustrate how you might use the `phonegap-plugin-barcodescanner` plugin by exposing a button in the `appRootDir/src/js/views/incidents.html` page of a JET app built using the `navbar` template that invokes the barcode scanner on the device.

```
...
<oj-button id='b1Scan' on-click='{{buttonClick}}' label='Scan'></oj-button>
...
```

The following code excerpt shows the entries to add to the `appRootDir/src/js/viewModels/incidents.js` file which invokes the barcode scanner when an end user clicks the button that the `incidents.html` page renders.

```
self.buttonClick = function(data, event){
      cordova.plugins.barcodeScanner.scan(
          function (result) {
              alert("We got a barcode\n" +
                    "Result: " + result.text + "\n" +
                    "Format: " + result.format + "\n" +
                    "Cancelled: " + result.cancelled);
          },
          function (error) {
              alert("Scanning failed: " + error);
          }
      );
    }
```

Cordova Plugins Recommended by Oracle JET

JET recommends a number of Cordova plugins that you can use in your app to provide native device functionality.

The Cordova Plugins page on the Oracle JET web site lists a number of plugins that have been successfully used in the verification testing of JET sample or demo apps. Although Oracle JET recommends these plugins, it does not support them.

Using a Different Web View in your JET Hybrid Mobile App

JET hybrid mobile apps use the default web view supplied by each mobile operating system.

On the Android platform, this is the Android platform's WebView. A number of Cordova plugins exist that enable you to use a different web view in your hybrid mobile app. One example for the Android platform is the `cordova-plugin-crosswalk-webview` plugin that configures your app to use the Crosswalk web view to bring performance improvements and compatibility improvements across older Android versions. Install this plugin in your app using the following command:

```
ojet add plugin cordova-plugin-crosswalk-webview
```

For more information about the `cordova-plugin-crosswalk-webview` plugin, read its documentation.

For apps that run on the iOS platform, consider adding the `cordova-plugin-wkwebview-file-xhr` plugin to your app so that your app uses the more performant WKWebView instead of the default UIWebView used on iOS devices.

```
ojet add plugin cordova-plugin-wkwebview-file-xhr
```

For more information about the `cordova-plugin-wkwebview-file-xhr` plugin, read its documentation.

Using the Native Platform's Date Picker UI in your JET Hybrid Mobile App

Include the `cordova-plugin-datepicker plugin` in your JET hybrid mobile app if you want to present end users with the date picker UI of the underlying platform (Android, iOS, or Windows) when your app renders JET date or time components such as `oj-input-date` or `oj-input-date-time`.

JET hybrid mobile apps that do not include the `cordova-plugin-datepicker` plugin render the aforementioned JET components using the default UI provided by Oracle JET.

Install the `cordova-plugin-datepicker` plugin in your app using the following command:

```
ojet add plugin cordova-plugin-datepicker
```

For more information about the `cordova-plugin-datepicker` plugin, read its documentation.

Loading Oracle JET from CDN

After you create your Oracle JET application, you can configure it to load Oracle JET files and libraries from the Oracle Content Delivery Network (CDN).

You can significantly improve your application's performance by using distributed servers that can deliver cached web pages and other web content to users based on their geographic locations.

To load Oracle JET from CDN, you must update your RequireJS bootstrap file, typically `js/main.js`, to reference the CDN location hosting the Oracle JET libraries and files. You must also update your index.html file to reference the Oracle JET CSS files and RequireJS library, also hosted on the Oracle CDN. Finally, if you used the Oracle JET tooling, update the paths specified in the application's `main-release-paths.json`.

If needed, create your Oracle JET application as described in Getting Started with Oracle JET Web Application Development or Getting Started with Oracle JET Hybrid Mobile Application Development.

1. In your application's `js` directory, open your application's RequireJS bootstrap file, typically `main.js`, for editing.
2. Update the path mapping to point to the CDN locations for the Oracle JET libraries and files.

   In the code sample below, the updates you must make are highlighted in bold.

   ```
   requirejs.config({
     // Path mappings for the logical module names
   ```

```
  paths: {
    'knockout': 'https://static.oracle.com/cdn/jet/v4.1.0/3rdparty/knockout/
knockout-3.4.0',
    'jquery': 'https://static.oracle.com/cdn/jet/v4.1.0/3rdparty/jquery/
jquery-3.1.1.min',
    'jqueryui-amd': 'https://static.oracle.com/cdn/jet/v4.1.0/3rdparty/jquery/
jqueryui-amd-1.12.0.min',
    'ojs': 'https://static.oracle.com/cdn/jet/v4.1.0/default/js/min',
    'ojL10n': 'https://static.oracle.com/cdn/jet/v4.1.0/default/js/ojL10n',
    'ojtranslations': 'https://static.oracle.com/cdn/jet/v4.1.0/default/js/
resources',
    'text': 'https://static.oracle.com/cdn/jet/v4.1.0/3rdparty/require/text',
    'promise': 'https://static.oracle.com/cdn/jet/v4.1.0/3rdparty/es6-promise/
es6-promise.min',
    'hammerjs': 'https://static.oracle.com/cdn/jet/v4.1.0/3rdparty/hammer/
hammer-2.0.8.min',
    'signals': 'https://static.oracle.com/cdn/jet/v4.1.0/3rdparty/js-signals/
signals.min',
    'ojdnd': 'https://static.oracle.com/cdn/jet/v4.1.0/3rdparty/dnd-polyfill/dnd-
polyfill-1.0.0.min',
    'css': 'https://static.oracle.com/cdn/jet/v4.1.0/3rdparty/require-css/
css.min',
    'customElements': 'https://static.oracle.com/cdn/jet/v4.1.0/3rdparty/
webcomponents/custom-elements.min',
    'proj4js': 'https://static.oracle.com/cdn/jet/v4.1.0/3rdparty/proj4js/dist/
proj4'
  },

  // Shim configuration
  shim: {
      'jquery': {
          exports: ['jQuery', '$']
      },
  ... remaining contents omitted
);
```

3. If you used Oracle JET tooling to create your application, in your application's `src/js` directory, open `main-release-paths.json` for editing and update the release paths to point to the same CDN locations you referenced in the previous step.

   The highlighted code shows the updates to `main-release-paths.json`.

```
{
  "knockout": "https://static.oracle.com/cdn/jet/v4.1.0/3rdparty/knockout/
knockout-3.4.0",
  "jquery": "https://static.oracle.com/cdn/jet/v4.1.0/3rdparty/jquery/
jquery-3.1.1.min",
  "jqueryui-amd": "https://static.oracle.com/cdn/jet/v4.1.0/3rdparty/jquery/
jqueryui-amd-1.12.0.min",
  "ojs": "https://static.oracle.com/cdn/jet/v4.1.0/default/js/min",
  "ojL10n": "https://static.oracle.com/cdn/jet/v4.1.0/default/js/ojL10n",
  "ojtranslations": "https://static.oracle.com/cdn/jet/v4.1.0/default/js/
resources",
  "text": "https://static.oracle.com/cdn/jet/v4.1.0/3rdparty/require/text",
  "promise": "https://static.oracle.com/cdn/jet/v4.1.0/3rdparty/es6-promise/es6-
promise.min",
  "hammerjs": "https://static.oracle.com/cdn/jet/v4.1.0/3rdparty/hammer/
hammer-2.0.8.min",
  "signals": "https://static.oracle.com/cdn/jet/v4.1.0/3rdparty/js-signals/
signals.min",
  "ojdnd": "https://static.oracle.com/cdn/jet/v4.1.0/3rdparty/dnd-polyfill/dnd-
polyfill-1.0.0.min",
```

```
  "css": "https://static.oracle.com/cdn/jet/v4.1.0/3rdparty/require-css/css.min",
  "customElements": "https://static.oracle.com/cdn/jet/v4.1.0/3rdparty/
webcomponents/custom-elements.min",
  "proj4js": "https://static.oracle.com/cdn/jet/v4.1.0/3rdparty/proj4js/dist/
proj4"
}
```

4. Open `index.html` for editing and update the references to `require.js` and the Oracle JET CSS file.

   The highlighted code below shows the necessary modifications.

```
<!-- This is the main css file for the default Alta theme -->
<link id="css" rel="stylesheet" href="https://static.oracle.com/cdn/jet/v4.1.0/
default/css/alta/oj-alta-min.css" type="text/css"/>
...
<!-- RequireJS bootstrap file -->
<script type="text/javascript" src="https://static.oracle.com/cdn/jet/
v4.1.0/3rdparty/require/require.js"></script>
```

5. If you want to build your application in release mode and use the RequireJS Optimizer and Oracle JET tooling, modify the Oracle JET build configuration script to skip loading CDN resources.

   The RequireJS Optimizer, `r.js`, cannot load network resources like CDN libraries and files. However, you can specify an `empty` path in your application's `scripts/config/oraclejet-build.js` for each CDN library or file that you want to skip.

   a. In your application's top level directory, open `scripts/config/oraclejet-build.js` for editing in your favorite text editor.

   b. Remove the comments from the `requireJs` section and add an entry for paths as shown below.

```
/**
 * # requireJs
 * This task runs requirejs optimizer to bundle all scripts in to a large
minified main.js for release.
 * It runs only when build in release mode.
 * The task mirrors the configuration in this link https://github.com/
gruntjs/grunt-contrib-requirejs
 */
requireJs: {
       baseUrl: 'path to the js directory in staging area',
       name: 'the main.js file name',
       mainConfigFile: `the main configuration file`,
       optimize: 'option for optimize',
       out: 'output file path',
       paths: {
       }
     },
```

   c. Edit the existing entries as needed for your application, and add each CDN resource to the `paths` section.

      The code sample below shows a completed `oraclejet-build.js` file for a web application.

```
requireJs: {
      baseUrl: 'web/js',
      name: 'main-temp',
      mainConfigFile: 'web/js/main-temp.js',
      optimize: 'none',
      out: 'web/js/main.js',
```

```
        paths: {
          jquery: "empty:",
          knockout: "empty:",
          jqueryui-amd: "empty:",
          ojs: "empty:",
          ojL10n: "empty:",
          ojtranslations: "empty:",
          text: "empty:",
          promise: "empty:",
          hammerjs: "empty:",
          signals: "empty:",
          ojdnd: "empty:",
          css: "empty:",
          customElements: "empty:",
          proj4js: "empty:"
        }
},
```

For a hybrid mobile application, update the `baseURL`, `mainConfigFile`, and `out` entries as shown below.

```
requireJs: {
      baseUrl: 'hybrid/www/js',
      name: 'main-temp',
      mainConfigFile: 'hybrid/www/js/main-temp.js',
      optimize: 'none',
      out: 'hybrid/www/js/main.js',
      paths: {
        jquery: "empty:",
        knockout: "empty:",
        ... contents omitted
      }
},
```

> **Tip:**
>
> If your application doesn't use Oracle JET tooling, see the RequireJS Optimizer documentation for other ways to use the `empty` paths scheme for skipping CDN resources during optimization.

For additional information about using RequireJS and Oracle JET, see Using RequireJS for Modular Development.

3
Designing Responsive Applications

Oracle JET includes classes that support a flexbox–based layout, 12-column responsive grid system, design patterns, responsive form layout, and responsive JavaScript that you can use to design responsive web and hybrid mobile applications.

Topics:

- Typical Workflow for Designing Responsive Applications in Oracle JET
- Oracle JET and Responsive Design
- Media Queries
- Oracle JET Flex, Grid, Form, and Responsive Helper Class Naming Convention
- Oracle JET Flex Layouts
- Oracle JET Grids
- Responsive Layout and Content Design Patterns
- Responsive Form Layouts
- Adding Responsive Design to Your Application
- Using Responsive JavaScript
- Using the Responsive Helper Classes
- Creating Responsive CSS Images
- Changing Default Font Size
- Controlling the Size and Generation of the CSS

Typical Workflow for Designing Responsive Applications in Oracle JET

Oracle JET includes classes for creating responsive applications. After you create your application, you can fine tune your design using Oracle JET responsive JavaScript and helper classes.

To design responsive applications in Oracle JET, refer to the typical workflow described in the following table:

Task	Description	More Information
Understand Oracle JET's support for responsive design	Understand the Oracle JET flex layout, 12–column mobile-first grid, and form layout framework classes, and the responsive class naming conventions.	Oracle JET and Responsive Design Media Queries Oracle JET Flex, Grid, Form, and Responsive Helper Class Naming Convention Oracle JET Flex Layouts Oracle JET Grids Responsive Layout and Content Design Patterns Responsive Form Layouts
Create a responsive application using the Oracle JET responsive classes.	Add application, flex, grid, and form layout classes to your application.	Adding Responsive Design to Your Application
Fine tune your responsive design	Use responsive JavaScript, helper classes, and responsive CSS images to complete the responsive design. Change default font size across your application.	Using Responsive JavaScript Using the Responsive Helper Classes Creating Responsive Images Changing Default Font Size

Oracle JET and Responsive Design

Responsive design describes a design concept that uses fluid grids, scalable images, and media queries to present alternative layouts based on the media type. With responsive design, you can configure Oracle JET applications to be visually appealing on a wide range of devices, ranging from small phones to wide-screen desktops.

Oracle JET includes classes that support a flexible box layout. In a flex layout, you can lay out the children of a flex container in any direction, and the children will grow to fill unused space or shrink to avoid overflowing the parent. You can also nest boxes (for example, horizontal inside vertical or vertical inside horizontal) to build layouts in two dimensions.

Oracle JET also provides a 12-column grid system and form layout classes that include styles for small, medium, large, and extra large screens or devices that you can use in conjunction with the flex layout classes to achieve finer control of your application’s layout. The grid system and form classes use media queries to set the style based on the width of the screen or device, and you can use them to customize your page layout based on your users' needs.

In addition, media queries form the basis for responsive helper classes that show or hide content, align text, or float content based on screen width. They are also the basis for responsive JavaScript that loads content conditionally or sets a component's option based on screen width.

Media Queries

CSS3 media queries use the `@media` at-rule, media type, and expressions that evaluate to true or false to define the cases for which the corresponding style rules will be applied. Media queries form the basis for Oracle JET’s responsive classes.

```
<style>
@media media_types (expression){
  /* media-specific rules */
}
</style>
```

The CSS3 specification defines several media types, but specific browsers may not implement all media types. The media type is optional and applies to all types if not specified. The following media query will display a sidebar only when the screen is wider than 767 pixels.

```
@media (max-width: 767px){
 .facet_sidebar {
    display: none;
  }
}
```

Oracle JET defines CSS3 media queries for the screen widths listed in the following table.

Width	Default Range in Pixels	Device Examples
small	0-767	phones
medium	768-1023	tablet portrait
large	1024-1280	tablet landscape, desktop
extra large	1281+	large desktop

> **Note:**
>
> For printing, Oracle JET uses the large screen layout for printing in landscape mode and the medium screen layout for printing in portrait mode.

Oracle JET's size defaults and media queries are defined in the Sass variables contained in *site_root*`/scss/oj/`*vx.x.x*`/common/_oj.common.variables.scss` and are used in the grid, form, and responsive helper style classes. The following code sample shows the responsive screen width variables and a subset of the responsive media queries. In most cases the defaults are sufficient, but be sure to check the file for additional comments that show how you might modify the variables for your application if needed.

```
// responsive screen widths
$screenSmallRange:  0, 767px !default;
$screenMediumRange: 768px, 1023px !default;
$screenLargeRange:  1024px, 1280px !default;
$screenXlargeRange: 1281px, null !default;

// responsive media queries
$responsiveQuerySmallUp:    "print, screen" !default;
$responsiveQuerySmallOnly:  "screen and (max-width: #{upper-
bound($screenSmallRange)})" !default;

$responsiveQueryMediumUp:   "print, screen and (min-width: #{lower-
bound($screenMediumRange)})" !default;
$responsiveQueryMediumOnly: "print and (orientation: portrait), screen and (min-
width: #{lower-bound($screenMediumRange)}) and (max-width: #{upper-
```

```
bound($screenMediumRange)})" !default;
$responsiveQueryMediumDown: "print and (orientation: portrait), screen and (max-
width: #{upper-bound($screenMediumRange)})" !default;

$responsiveQueryLargeUp:    "print and (orientation: landscape), screen and (min-
width: #{lower-bound($screenLargeRange)})" !default;
$responsiveQueryLargeOnly:  "print and (orientation: landscape), screen and (min-
width: #{lower-bound($screenLargeRange)}) and (max-width: #{upper-
bound($screenLargeRange)})" !default;
$responsiveQueryLargeDown:  "print and (orientation: landscape), screen and (max-
width: #{upper-bound($screenLargeRange)})" !default;

$responsiveQueryXlargeUp:   "screen and (min-width: #{lower-
bound($screenXlargeRange)})" !default;
$responsiveQueryXlargeOnly: null !default;
$responsiveQueryXlargeDown: null !default;

$responsiveQueryXXlargeUp:  null !default;

$responsiveQueryPrint:      null !default;
```

Responsive media queries are based on the screen widths defined in the `$screen{size}Range` variables and a range qualifier. For example:

- `$responsiveQuerySmallUp` applies to all screens in the `$screenSmallRange` or wider.
- `$responsiveQuerySmallOnly` applies only to screens in the `$screenSmallRange`.
- `$responsiveQueryXlargeDown` applies to all screens in the `$screenXlargeRange` and narrower.

For additional information about Oracle JET's use of Sass and theming, see Theming Applications.

For additional information about CSS3 media queries, see https://developer.mozilla.org/en-US/docs/Web/Guide/CSS/Media_queries and http://www.w3.org/TR/css3-mediaqueries.

Oracle JET Flex, Grid, Form, and Responsive Helper Class Naming Convention

The Oracle JET flex, form, grid, and responsive style classes use the same naming convention which can help you identify the style size, function, and number of columns the class represents.

Each class follows the same format as shown below:

`oj-size-function-[1-12]columns`

Size can be one of `sm`, `md`, `lg`, `xl`, and `print` and are based on the media queries described in Media Queries. Oracle JET will apply the style to the size specified and any larger sizes unless *function* is defined as `only`. For example:

- `oj-lg-hide` hides content on large and extra-large screens.
- `oj-md-only-hide` hides content on medium screens. The style has no effect on other screen sizes.

You can find a summary of the classes available to you for responsive design in Oracle JET in the Oracle® JavaScript Extension Toolkit (JET) Styling Reference.

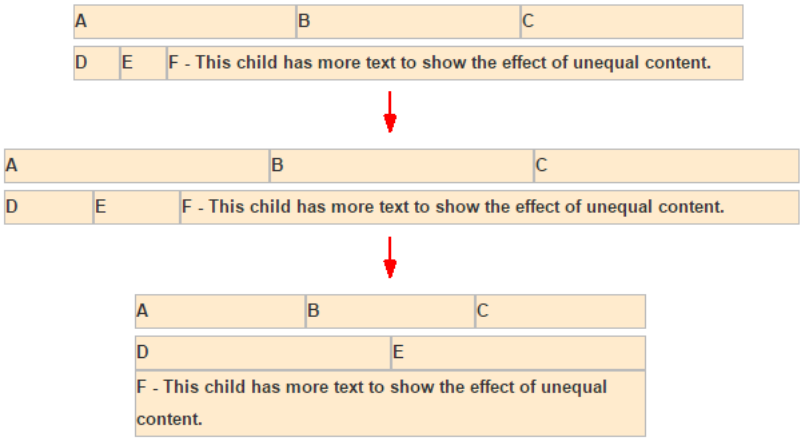
Oracle JET Flex Layouts

Use the Oracle JET `oj-flex` and `oj-flex-item` classes to create flexible box layouts that are based on the CSS flexible box layout model.

In the flex layout model, you create flex containers with children that you can lay out in any direction or order. As the available unused space grows or shrinks, the children grow to fill the unused space or shrink to avoid overflowing the parent.

To create a basic flex layout, add the `oj-flex` class to a container element (HTML `div`, for example) and then add the `oj-flex-item` class to each of the container's children.

The following image shows an example of a default flex layout using the Oracle JET flex box styles. The sample contains two flex containers, each with three children. As the screen size widens, the flex container allocates unused space to each of the children. As the screen size shrinks below the width of one of the flex items, the flex container will wrap the content in that item as needed to no wider than the maximum display width. In this example, this has the effect of causing the F child to wrap to the next row.



The markup for this flex layout is shown below, with the flex layout classes highlighted in bold. The demo-flex-display class sets the color, font weight, height, and border around each flex item in the layout.

```
<div id="container">
  <div class="demo-flex-display">
    <div class="oj-flex">
      <div class="oj-flex-item">A</div>
      <div class="oj-flex-item">B</div>
      <div class="oj-flex-item">C</div>
    </div>

    <div class="oj-flex">
      <div class="oj-flex-item">D</div>
      <div class="oj-flex-item">E</div>
      <div class="oj-flex-item">F - This child has more text to show the effect of
unequal content.</div>
    </div>
```

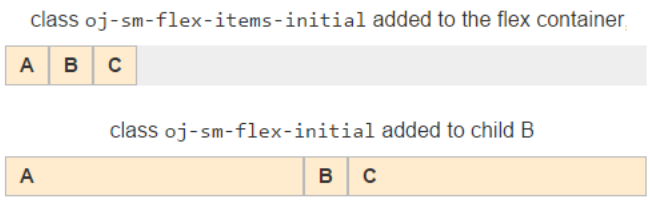
```
  </div>
</div>
```

You can customize the flex layout using styles detailed in Flex Layout Styling and described below.

Modifying the `flex` Property

The Oracle JET layout classes are based on the CSS Flexible Box Layout Module which defines CSS flex common values for flex item sizing. By default, Oracle JET's flex layout defaults to the `auto` CSS flex property which allows a flex item to shrink or grow as needed for responsive layouts. However, the CSS model sets the default `flex` property to `initial` , which allows a flex item to shrink but will not allow it to grow.

You can achieve the same effect by adding the `oj-sm-flex-items-initial` class to the flex container to set the flex property to `initial` for all child flex items, or add the `oj-sm-flex-initial` class to an individual flex item to set its property to `initial`. The following image shows the effect.



The code sample below shows the markup. In this example, padding is also added to the content using the `oj-flex-items-pad` class on the parent container.

```
<div id="container">
  <div class="demo-flex-display oj-flex-items-pad">
    <div class="oj-flex oj-sm-flex-items-initial">
      <div class="oj-flex-item">A</div>
      <div class="oj-flex-item">B</div>
      <div class="oj-flex-item">C</div>
    </div>

    <div class="oj-flex">
      <div class="oj-flex-item">A</div>
      <div class="oj-sm-flex-initial oj-flex-item">B</div>
      <div class="oj-flex-item">C</div>
    </div>
  </div>
</div>
```

You can also override the default `auto` flex property by using the `oj-size-flex-items-1` class on the flex container. This class sets the `flex` property to `1`, and all flex items in the flex container with a screen size of *size* or higher will have the same width, regardless of the items' content.

class `oj-sm-flex-items-1` added to the flex container, which sets '`flex: 1`' on all children

A	B	C
A	B	C - This child has more text to show the effect of unequal content.

To set the flex property to 1 on an individual flex item, add `oj-sm-flex-1` to the flex item. The Flex Layouts section in the Oracle JET Cookbook includes the examples used in this section that you can use and modify to observe the flex layout's responsive behavior.

Wrapping Content

By default, Oracle JET sets the CSS flex-wrap property to `wrap`, which sets the flex container to multi-line. Child flex items will wrap content to additional lines when the screen width shrinks to less than the width of the flex item's content. However, the CSS model sets the `flex-wrap` property to `nowrap`, which sets the flex container to single-line. When a child item's content is too wide to fit on the screen, the content will wrap within the child.

You can set the `flex-wrap` property to `nowrap` by adding `oj-sm-flex-nowrap` to the `oj-flex` container. The following image shows the effect of changing the `flex-wrap` property to `nowrap`.

A	B
C - This child has more text to show the effect of unequal content.	

add class `'oj-sm-flex-wrap-nowrap'`

A	B	C - This child has more text to show the effect of unequal content.

Customizing Flex Layouts

You can customize an Oracle JET flex layout by adding the appropriate style to the flex container or child. The flex layout classes support some of the more commonly-used values for:

- `flex-direction`
- `align-items`
- `align-self`
- `justify-content`
- `order`

The Oracle JET Cookbook includes examples for customizing your flex layout at: Flex Layouts.

Oracle JET Grids

Use the Oracle JET grid classes with flex layouts to create grids that vary the number and width of columns based on the width of the user's screen.

Topics:

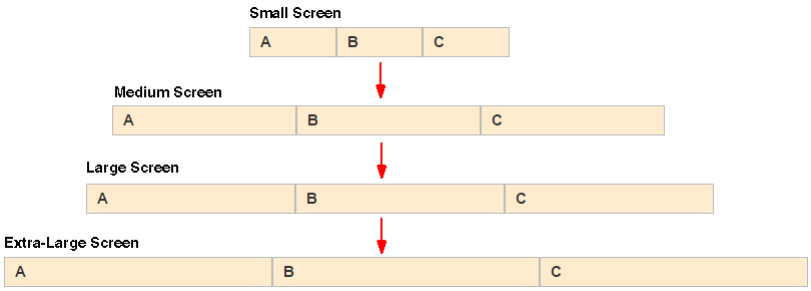
- About the Grid System
- The Grid System and Printing
- Grid Convenience Classes

The Responsive Grids section in the Oracle JET Cookbook provides several examples and recipes for using the Oracle JET grid system, and you should review them to get accustomed to the grid system.

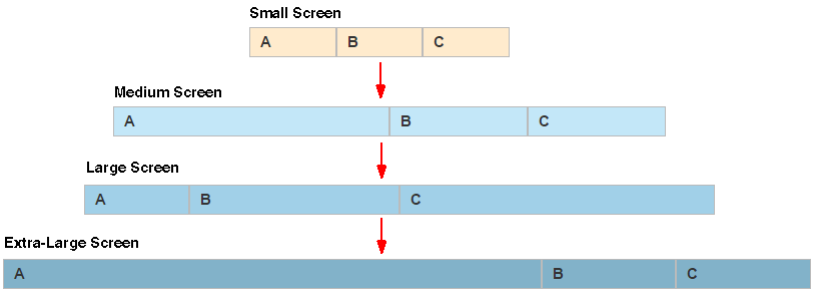
About the Grid System

Oracle JET provides a 12-column responsive mobile-first grid system that you can use for responsive design. The grid builds upon the Oracle JET flex layout and allows you to specify the widths of each flex item using sizing classes for small, medium, large, and extra-large screens.

For example, you can use the grid classes to change the default display in the Flex Layouts Auto example to use different widths for the flex items when the screen size changes. As shown in the image below, the flex layout by default will allocate the unused space evenly to the three flex items regardless of the screen size.



When the defaults are not sufficient, you can specify relative widths for the flex items when the screen size changes. In the following image, the flex layout is using grid classes to define different widths when the screen size changes to medium, large, and extra large.



The grid classes follow the Oracle JET Flex, Grid, Form, and Responsive Helper Class Naming Convention. Use `oj-`*size*`-`*numberofcolumns* to set the width to the specified *numberofcolumns* when the screen is the specified *size* or larger. For example:

- `oj-sm-6` works on all screen sizes and sets the width to 6 columns.
- `oj-lg-3` sets the width to 3 columns on large and extra-large screens.
- `oj-sm-6` and `oj-lg-3` on the same flex item sets the width to 6 columns wide on small and medium screens and 3 columns wide on large and extra-large screens.

Design for the smallest screen size first and then customize for larger screens as needed. You can further customize the grid by adding one of the Grid Convenience Classes or by using one of the responsive helper classes described in Using the Responsive Helper Classes.

The following code sample shows the markup for the modified Flex Auto Layout display, with grid classes defined for medium, large, and extra-large screens.

```
<div class="oj-flex">
  <div class="oj-md-6 oj-lg-2 oj-xl-8 oj-flex-item">A</div>
  <div class="oj-md-3 oj-lg-4 oj-xl-2 oj-flex-item">B</div>
  <div class="oj-md-3 oj-lg-6 oj-xl-2 oj-flex-item">C</div>
</div>
```

When the screen size is small, the flex layout default styles are used, and each item uses the same amount of space. When the screen size is medium, the A flex item will use 6 columns, and the B and C flex items will each use 3 columns. When the screen size is large, The A flex item will use 2 columns, the B flex item will use 4 columns, and the C flex item will use 6 columns. Finally, when the screen size is extra large, the A flex item will use 8 columns, and the B and C flex items will each use 2 columns.

For a complete example that illustrates working with the grid system, see Responsive Grids.

The Grid System and Printing

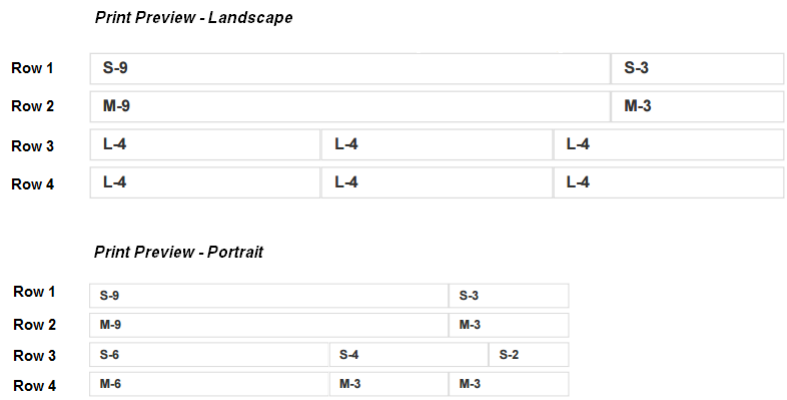
The Oracle JET grid system applies the large styles for printing in landscape mode and the medium style for printing in portrait mode if they are defined. You can use the defaults or customize printing using the print style classes.

In the grid example below, Row 2 and Row 4 include the `oj-md-*` style classes. Row 3 and Row 4 include the `oj-lg-4` style for all columns in the row.

```
<div class="demo-grid-sizes demo-flex-display">
  <div class="oj-flex oj-flex-items-pad">
    <div class="oj-sm-9 oj-flex-item"></div>
```

```
    <div class="oj-sm-3 oj-flex-item"></div>
  </div>
  <div class="oj-flex oj-flex-items-pad">
    <div class="oj-sm-6 oj-md-9 oj-flex-item"></div>
    <div class="oj-sm-6 oj-md-3 oj-flex-item"></div>
  </div>
  <div class="oj-flex oj-flex-items-pad">
    <div class="oj-sm-6 oj-lg-4 oj-flex-item"></div>
    <div class="oj-sm-4 oj-lg-4 oj-flex-item"></div>
    <div class="oj-sm-2 oj-lg-4 oj-flex-item"></div>
  </div>
  <div class="oj-flex oj-flex-items-pad ">
    <div class="oj-sm-8 oj-md-6 oj-lg-4 oj-xl-2 oj-flex-item"></div>
    <div class="oj-sm-2 oj-md-3 oj-lg-4 oj-xl-8 oj-flex-item"></div>
    <div class="oj-sm-2 oj-md-3 oj-lg-4 oj-xl-2 oj-flex-item"></div>
  </div>
</div>
```

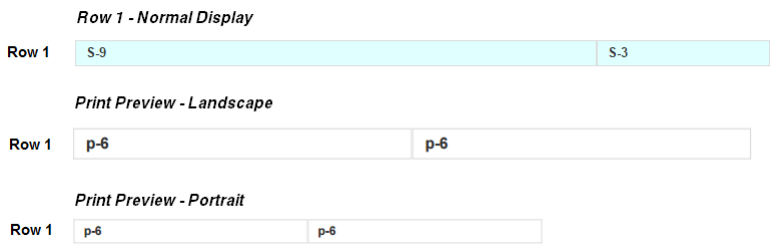
As shown in the following print preview, when you print this grid in landscape mode, the `oj-lg-4` style classes will be applied on Row 3 and Row 4. When you print the grid in portrait mode, the `oj-md-*` style classes apply on Row 2 and Row 4.



If you want to change the printing default, you can set the Sass `$responsiveQueryPrint` variable to `print` in a custom settings file. After you enable the print classes, you can add the `oj-print-`*`numberofcolumns`* style class to the column definition. This has the effect of changing the column sizes for printing purposes only. In the following example, Row 1 includes the `oj-print-6` class for each column in the row.

```
<div class="oj-flex oj-flex-items-pad">
  <div class="oj-sm-9 oj-print-6 oj-flex-item"></div>
  <div class="oj-sm-3 oj-print-6 oj-flex-item"></div>
</div>
```

In normal mode, Row 1 contains two columns, one column with a size of 9 and one column with a size of 3, regardless of screen size. If you do a print preview, however, you'll see that Row 1 will print with two columns in portrait and landscape mode, both with a size of 6.



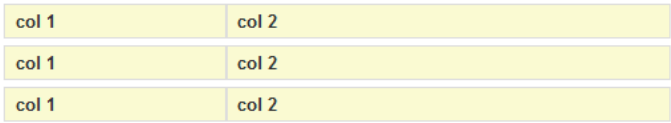
For information about setting Sass variables in a custom settings file, see Customizing Themes Using the Tooling.

Grid Convenience Classes

Oracle JET's grid system includes convenience classes that make it easier to create two- and four- column layouts with specified widths.

- `oj-`*`size`*`-odd-cols-`*`numberofcolumns`*: Use this in a 2-column layout. Instead of putting sizing classes on every column, you can put a single class on the flex parent. The number of columns specifies how many of the 12 columns the odd-numbered columns can use. In a 2-column layout, the even-numbered columns will take up the remainder of the columns.

  For example, setting `oj-md-odd-cols-4` on the flex parent will have the effect of setting the odd column (`col1`) width to 4 and the even column (`col2`) width to 8 for all rows in the grid on medium-size screens and higher.



  The code sample below shows the grid configuration used to render the figure. The example also sets `oj-sm-odd-cols-12` which will set the odd column width to 12 on small screens, displaying `col2` on a new row.

```
<div class="oj-md-odd-cols-4 oj-flex-items-pad">
  <div class="oj-flex">
    <div class="oj-flex-item">col 1</div>
    <div class="oj-flex-item">col 2</div>
  </div>
  <div class="oj-flex">
    <div class="oj-flex-item">col 1</div>
    <div class="oj-flex-item">col 2</div>
  </div>
  <div class="oj-flex">
    <div class="oj-flex-item">col 1</div>
    <div class="oj-flex-item">col 2</div>
  </div>
</div>
```

  You could achieve the same effect by defining `oj-md-4` for the first column's width and `oj-md-8` for the second column's width on each flex item.

```
<div class="oj-flex-items-pad"
  <div class="oj-flex">
    <div class="oj-sm-12 oj-md-4 oj-flex-item">col 1</div>
    <div class="oj-sm-12 oj-md-8 oj-flex-item">col 2</div>
  </div>
  <div class="oj-flex">
    <div class="oj-sm-12 oj-md-4 oj-flex-item">col 1</div>
    <div class="oj-sm-12 oj-md-8 oj-flex-item">col 2</div>
  </div>
  <div class="oj-flex">
    <div class="oj-sm-12 oj-md-4 oj-flex-item">col 1</div>
    <div class="oj-sm-12 oj-md-8 oj-flex-item">col 2</div>
  </div>
</div>
```

`oj-size-even-cols-numberofcolumns`: Use in a 4-column layout. In this layout, you must use both the `odd-cols` class to control the width of odd-numbered columns and the `even-cols` class to control the width of the even columns.

For example, setting `oj-md-odd-cols-2` and `oj-md-even-cols-4` on the flex parent has the effect of setting the first and third column widths to 2, and the second and fourth column widths to 4.

col 1	col 2	col 3	col 4
col 1	col 2	col 3	col 4

The code sample below shows the grid configuration used to render the figure.

```
<div class="oj-sm-odd-cols-12 oj-md-odd-cols-2 oj-md-even-cols-4 oj-flex-items-
pad">
  <div class="oj-flex">
    <div class="oj-flex-item">col 1</div>
    <div class="oj-flex-item">col 2</div>
    <div class="oj-flex-item">col 3</div>
    <div class="oj-flex-item">col 4</div>
  </div>
  <div class="oj-flex">
    <div class="oj-flex-item">col 1</div>
    <div class="oj-flex-item">col 2</div>
    <div class="oj-flex-item">col 3</div>
    <div class="oj-flex-item">col 4</div>
  </div>
</div>
```

If you don't use the convenience classes, you must define the size classes on every column in every row as shown below.

```
<div class="oj-flex-items-pad">
  <div class="oj-flex">
    <div class="oj-sm-odd-cols-12 oj-md-2 oj-flex-item">col 1</div>
    <div class="oj-sm-odd-cols-12 oj-md-4 oj-flex-item">col 2</div>
    <div class="oj-sm-odd-cols-12 oj-md-2 oj-flex-item">col 3</div>
    <div class="oj-sm-odd-cols-12 oj-md-4 oj-flex-item">col 4</div>
  </div>
  <div class="oj-flex">
    <div class="oj-sm-odd-cols-12 oj-md-2 oj-flex-item">col 1</div>
    <div class="oj-sm-odd-cols-12 oj-md-4 oj-flex-item">col 2</div>
    <div class="oj-sm-odd-cols-12 oj-md-2 oj-flex-item">col 3</div>
```

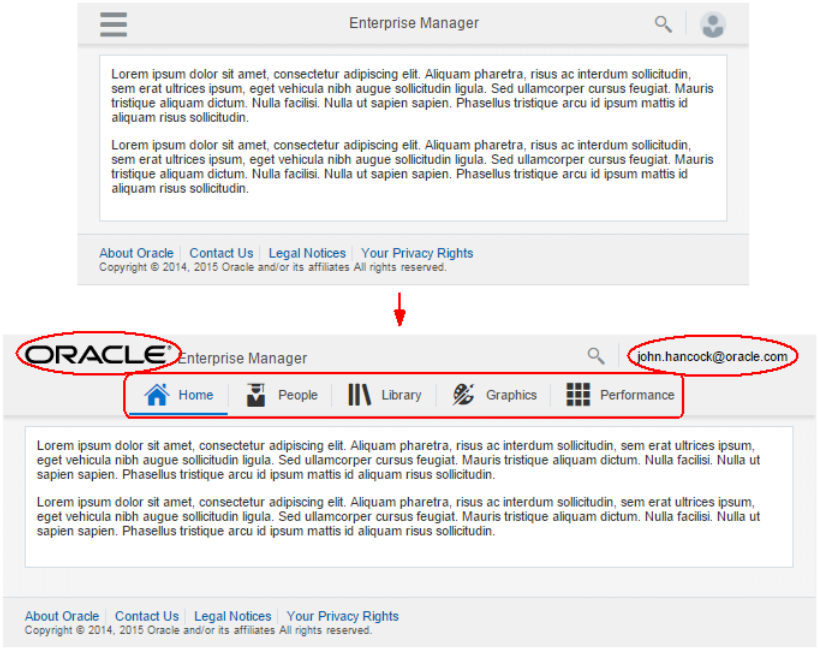
```
    <div class="oj-sm-odd-cols-12 oj-md-4 oj-flex-item">col 4</div>
  </div>
</div>
```

Responsive Layout and Content Design Patterns

Oracle JET provides application layout classes to use on your page to create responsive header, content, and footer sections. In addition, Oracle JET includes support for the column drop, layout shift, view switcher, and off-canvas responsive page content design patterns.

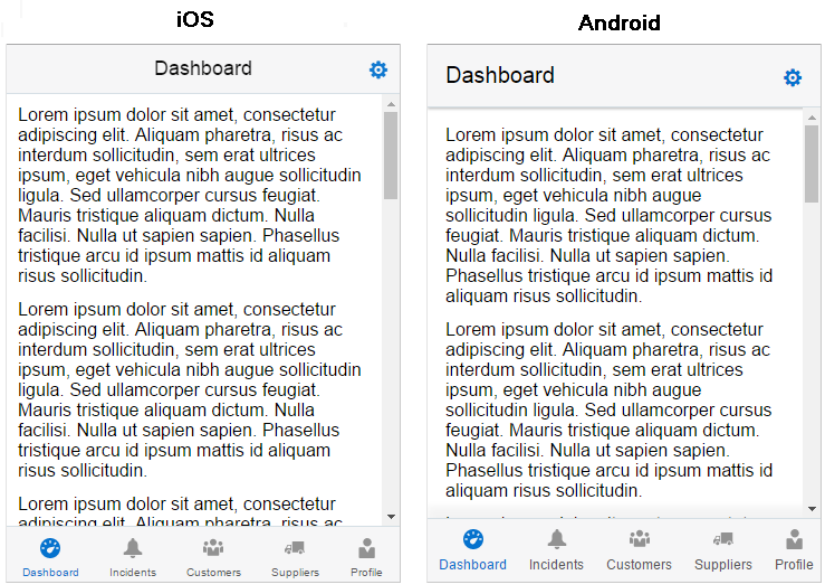
- Web Application Patterns

  Oracle JET provides the `oj-web-applayout-*` responsive classes that you can use in conjunction with a flex layout to configure your web page for responsive behavior. The following image shows the same page displayed on a small and large screen. When the screen size increases, the page displays the Oracle logo, user's email address, and navigation bar.

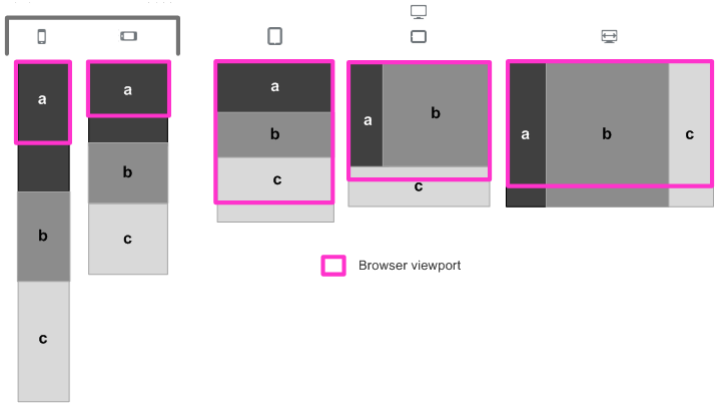


- Oracle JET provides the `oj-hybrid-applayout-*` responsive classes that you can use in conjunction with a flex layout and native theming to create hybrid mobile applications for iOS and Android mobile devices.

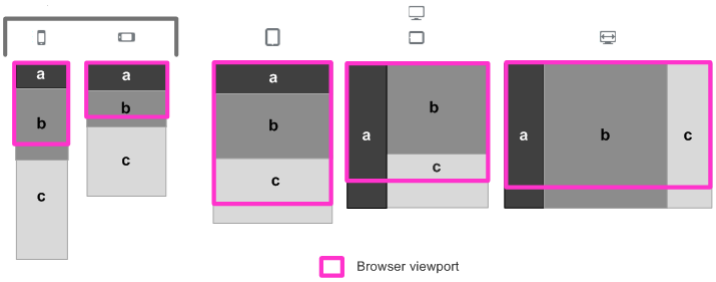
  The image below shows the hybrid mobile header navBar sample pattern displayed in a desktop browser using the Alta iOS and Android themes.



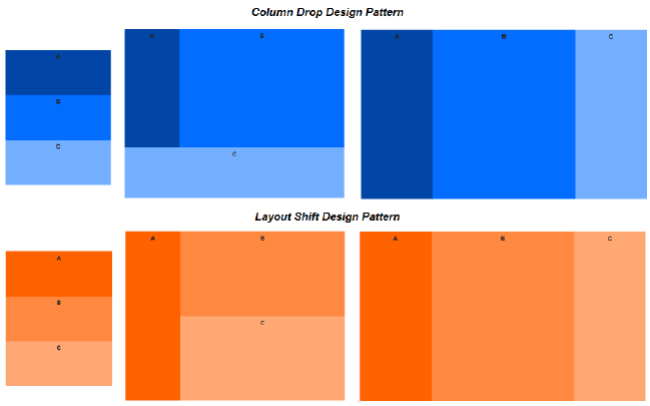
- Column Drop: Columns drop below each other for smaller screens. At the widest breakpoint, content displays in columns. As the display size is reduced, columns drop from the rightmost side of the display and add as rows below the remaining column(s).



- Layout Shift: Column layout changes for smaller screens. Layout shape may differ at each breakpoint as content is repositioned. Rather than simply adding or removing columns, for example, the leftmost column could transition to a row at the top of the display.

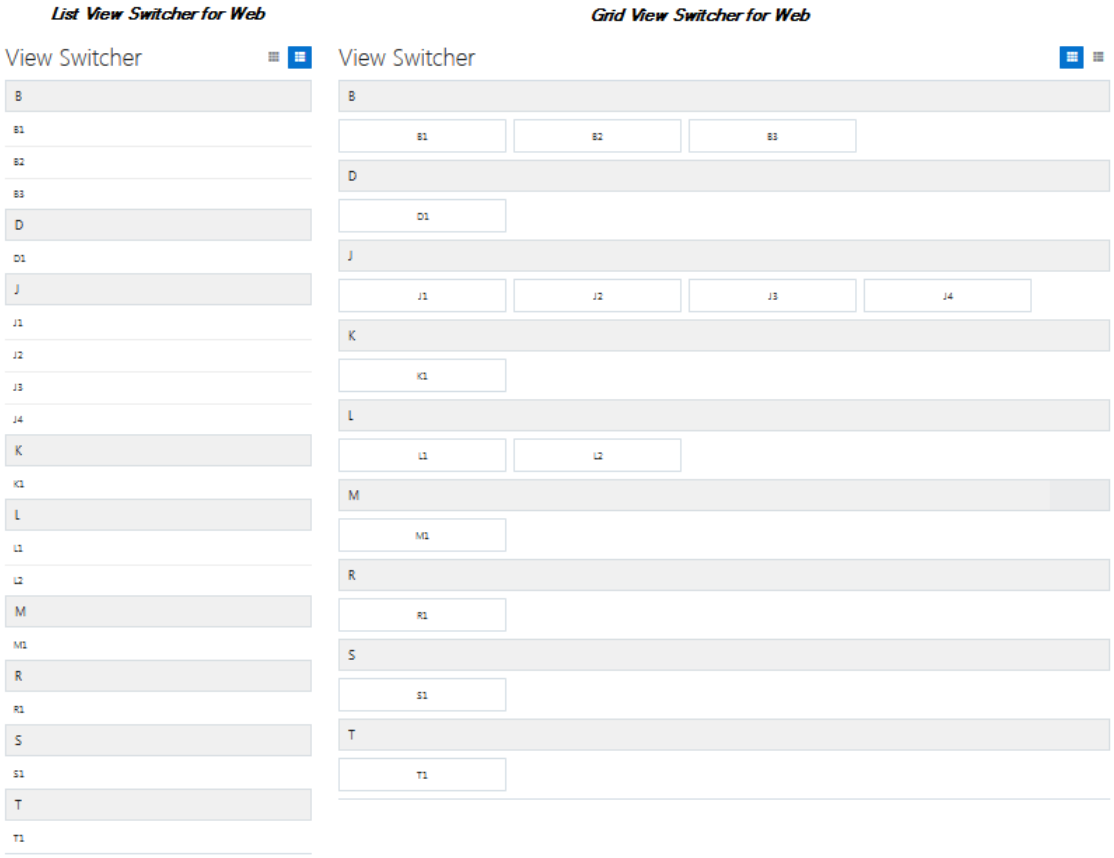


The difference between the two design patterns is subtle, and you can get a better feel for the difference by looking at the Oracle JET Cookbook examples at differing widths or on different devices. In the figure below, the column drop and layout shift patterns are shown at three different screen widths on a desktop display. In this example, the difference is noticeable in the middle figure for each pattern.

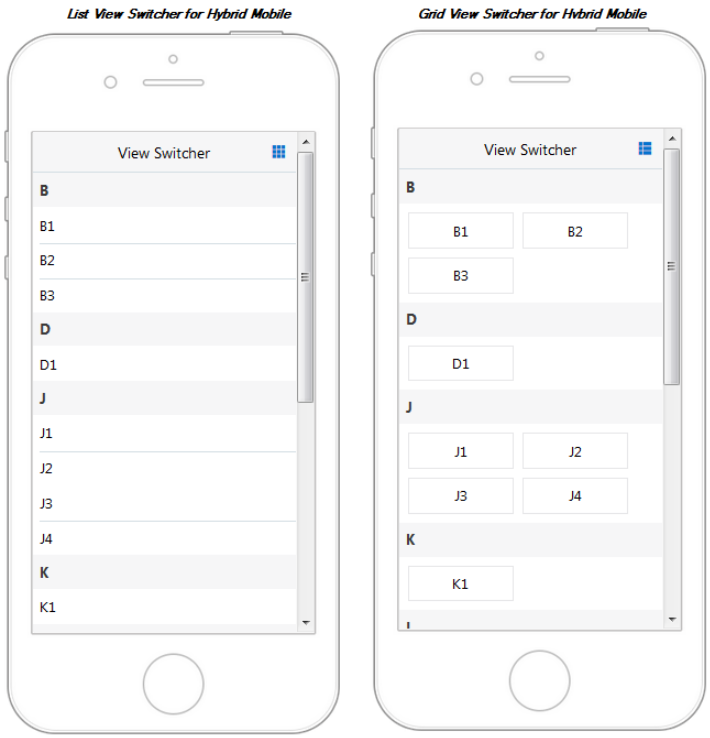


- List - View Switcher The `View Switcher` allows you to switch between list view and grid view.

  The image below shows the `View Switcher` web layout displayed in a desktop browser. The `View Switcher` switches between list view and grid view as follows.

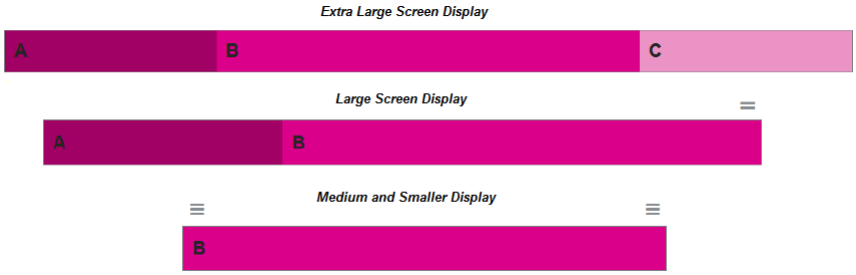


The image below shows the list view and grid view patterns of the `View Switcher` hybrid layout on a hybrid mobile application with a fixed header.



- Off-Canvas Push and Off-Canvas Overlay: You can use Oracle JET's oj.OffcanvasUtils to make an off-canvas partition that responsively change its position from off-screen to fixed inside the viewport when the browser width is changed.

  In the image below, when the screen size is extra large, the off-canvas design shows three columns: the off-canvas partition on the start edge labeled `A`, the main content labeled `B`, and the off-canvas partition on the end edge labeled `C`. When the screen size is large, the off-canvas position on the end edge (`C`) is hidden off-screen, and the launch button indicates its availability. For medium screens and smaller, both the start edge and end edge partitions are hidden off-screen, and two launch buttons indicate their availability.



For additional information about using `oj.offCanvasUtils` in your application layout, see Working with offCanvasUtils.

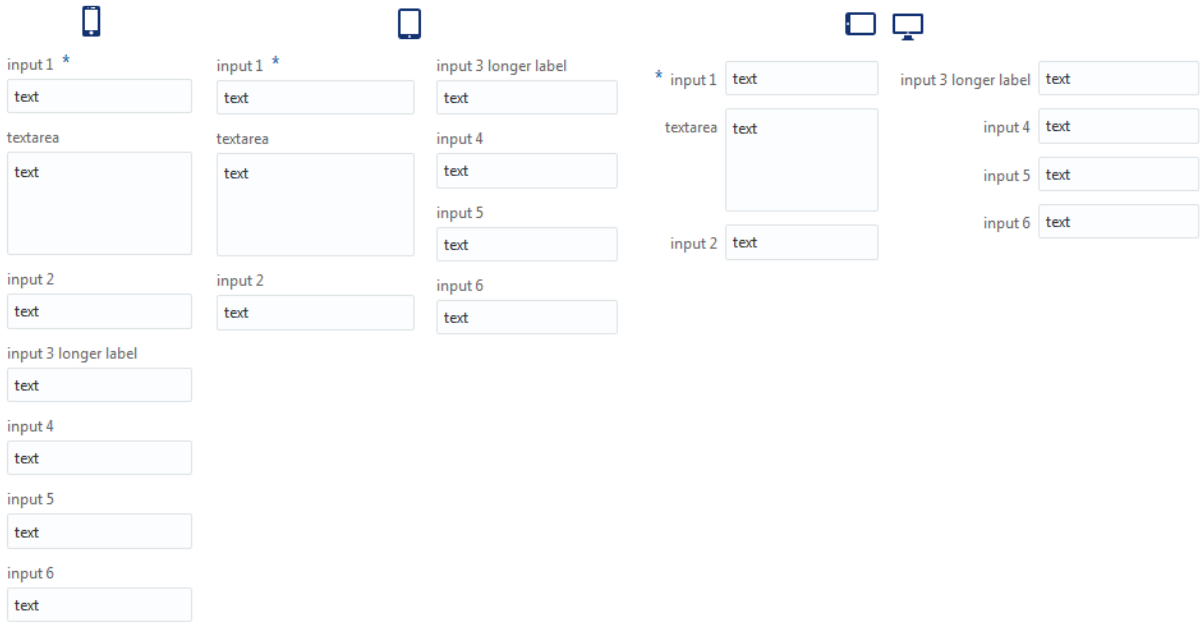
Responsive Form Layouts

Use the Oracle JET form-specific style classes to create responsive form layouts. The classes include styles to control label placement and wrapping, column display, nesting, and grouping with headers and separators. The classes also use the responsive media query variables, allowing you to specify behavior based on screen size.

See Form Styling for the complete list of form classes. The following list shows some commonly-used styles.

- `oj-form`: Place on the parent of form controls to ensure proper spacing and padding between the controls.
- `oj-form-control-group`: Place around a group of form controls. The style class ensures that the correct amount of space is maintained from the last form control in the group to the next form control in the stacked layout, even if the form controls have wrapped within their container.
- `oj-form-cols` and `oj-form-cols-labels-inline`: Display a single column on small screens, and create additional columns when there is enough space. Place on the parent of the rows, the same element where you would place `oj-form`.

  Use these classes for forms where you expect the user to enter input down a column. For larger screens, the form will display two or more columns, and the user will enter the input down the first column and then down the remaining columns. Use `oj-form-cols` to display labels in their default location above the field. Use `oj-form-cols-labels-inline` to display labels inline for larger screens, as shown in the following image.



When you specify `oj-form-cols`, Oracle JET calculates the width of the field to determine if another column will fit in the display. When you specify `oj-form-cols-`

`labels-inline`, Oracle JET totals the field and inline label width to determine if another column will fit in the display.

> **Tip:**
>
> If you want to restrict the display to two columns, add the `oj-form-cols-max2` class to the same element where you placed `oj-form-cols` or `oj-form-cols-label-inline`.

The `oj-form-cols` and `oj-form-cols-labels-inline` classes use the CSS multi-column layout technique. With this method, you can display two or more columns whenever there's enough room on the page, not just at the responsive break points. You don't have to calculate programmatically where to break the fields into columns because the CSS automatically determines how to break the content into columns. Even if the individual elements have varying heights, the CSS breaks the elements into columns of fairly even heights. For additional information about the multi-column layout technique, see `http://dev.w3.org/csswg/css-multicol`.

> **Note:**
>
> The CSS multi-column layout technique is not supported by all browsers. If the browser doesn't support the technique, the display will default to single column for all screen sizes.

- Label placement and wrapping
  - `oj-`*`size`*`-label-inline`: Show the label inline, which is on the left side of the field in left-to-right languages and the right side of the field in right-to-left languages. By default, Oracle JET is designed to be mobile first and places labels above the field.
  - `oj-`*`size`*`-labels-inline`: Use to show *all* labels inline. Place on the parent of the rows, the same element where you would place `oj-form`.
  - `oj-`*`size`*`-label-nowrap`: Use to set `nowrap` on a label. The label text will be truncated if it's too wide.
  - `oj-`*`size`*`-labels-nowrap`: Use to set `nowrap` on *all* labels. Place on the parent of the rows, the same element where you would place `oj-form`.

To see examples of the classes in use, consult the Oracle JET Cookbook at Form Layouts for:

- Stacking and nesting forms
- Inline responsive forms
- Grouping forms
- Column spanning

Adding Responsive Design to Your Application

To create your responsive application using Oracle JET, design for the smallest device first and then customize as needed for larger devices. Add the applicable application, flex, grid, form, and responsive classes to implement the design.

To design a responsive application using Oracle JET classes:

1. Determine whether you want to use the Oracle JET Responsive Layout and Content Design Patterns or provide your own to lay out your page.
2. Design the application content's flex layout.

   For help, see Oracle JET Flex Layouts
3. If the flex layout defaults are not sufficient and you need to specify column widths when the screen size increases, add the appropriate responsive grid classes to your flex items.

   For help, see Oracle JET Grids.
4. If you're adding a form to your page, add the appropriate form style classes.

   For help, see Responsive Form Layouts
5. Customize your design as needed.

   For additional information, see:

   - Using Responsive JavaScript
   - Using the Responsive Helper Classes
   - Creating Responsive Images
   - Changing Default Font Size

For the list of responsive design classes and their behavior, see the Responsive* classes listed in the Oracle® JavaScript Extension Toolkit (JET) Styling Reference.

For Oracle JET Cookbook examples that implement responsive design, see:

- Application Patterns
- Flex Layouts
- Responsive Grids
- Form Layouts
- Responsive Helpers
- Responsive JavaScript Framework Queries

Using Responsive JavaScript

Oracle JET includes the `ResponsiveUtils` and `ResponsiveKnockoutUtils` utility classes that leverage media queries to change a component's `value` option or load content and images based on the user's screen size or device type.

Topics:

- The Responsive JavaScript Classes

- Changing a Custom Element's Attribute Based on Screen Size
- Conditionally Loading Content Based on Screen Size
- Creating Responsive Images

The Responsive JavaScript Classes

The `ResponsiveUtils` and `ResponsiveKnockoutUtils` responsive JavaScript classes provide methods that you can use in your application's JavaScript to obtain the current screen size and use the results to perform actions based on that screen size. In addition, the `ResponsiveUtils` provides a method that you can use to compare two screen sizes, useful for performing actions when the screen size changes.

JavaScript Class	Methods	Description
`responsiveUtils`	`compare(size1, size2)`	Compares two screen size constants. Returns a negative integer if the first argument is less than the second, a zero if the two are equal, and a positive integer if the first argument is more than the second. The screen size constants identify the screen size range media queries. For example, the `oj.ResponsiveUtils.SCREEN_RANGE.SM` constant corresponds to the Sass `$screenSmallRange` variable and applies to screen sizes smaller than 768 pixels in width.
`responsiveUtils`	`getFrameworkQuery(frameworkQueryKey)`	Returns the media query to use for the framework query key parameter. The framework query key constant corresponds to a Sass responsive query variable. For example, the `oj.ResponsiveUtils.FRAMEWORK_QUERY_KEY.SM_UP` constant corresponds to the `$responsiveQuerySmallUp` responsive query which returns a match when the screen size is small and up.

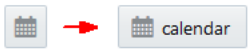
`responsiveKnockoutUtils`	`createMediaQueryObservable(queryString)`	Creates a Knockout observable that returns true or false based on a media query string. For example, the following code will return true if the screen size is 400 pixels wide or larger.
		`var customQuery =`
		`oj.ResponsiveKnockoutUtils.createMediaQueryObservable( '(min-width: 400px)');`

`responsiveKnockoutUtils`	`createScreenRangeObservable()`	Creates a computed Knockout observable, the value of which is one of the `oj.ResponsiveUtils.SCREEN_RANGE` constants.
		For example, on a small screen (0 - 767 pixels), the following code will create a Knockout observable that returns `oj.ResponsiveUtils.SCREEN_RANGE.SM`.
		`self.screenRange =`
		`oj.ResponsiveKnockoutUtils.createScreenRangeObservable();`

For additional detail about `responsiveUtils`, see the oj.ResponsiveUtils API documentation. For more information about `responsiveKnockoutUtils`, see oj.ResponsiveKnockoutUtils.

Changing a Custom Element's Attribute Based on Screen Size

You can set the value for a custom element's attribute based on screen size using the responsive JavaScript classes. For example, you may want to add text to a button label when the screen size increases using the `oj-button` element's `display` attribute.



In this example, the `oj-button` element's `display` attribute is defined for icons. The code sample below shows the markup for the button.

```
<div id="optioncontainer">
  <oj-button display="[[large() ? 'all' : 'icons']]">
    <span slot='startIcon' class="oj-fwk-icon oj-fwk-icon-calendar"></span>
    calendar
  </oj-button>
</div>
```

The code sample also sets the `oj-button` display attribute to `all`, which displays both the label and icon when the `large()` method returns `true`, and icons only when the `large()` method returns false.

The code sample below shows the code that sets the value for `large()` and completes the knockout binding. In this example, `lgQuery` is set to the `LG_UP` framework query key which applies when the screen size is large or up. `self.large` is initially set to `true` as the result of the call to `oj.ResponsiveKnockoutUtils.createMediaQueryObservable(lgQuery)`. When the screen changes to a smaller size, the `self.large` value changes to `false`, and the `display` attribute value becomes `icons`.

```
require(['ojs/ojcore', 'knockout', 'jquery', 'ojs/ojknockout', 'ojs/ojbutton'],
  function(oj, ko, $)
  {
  function MyModel(){
    var self = this;

    // observable for large screens
    var lgQuery = oj.ResponsiveUtils.getFrameworkQuery(
                            oj.ResponsiveUtils.FRAMEWORK_QUERY_KEY.LG_UP);

    self.large = oj.ResponsiveKnockoutUtils.createMediaQueryObservable(lgQuery);

  }

  $(function() {
    ko.applyBindings(new MyModel(), document.getElementById('optioncontainer'));

  });
});
```

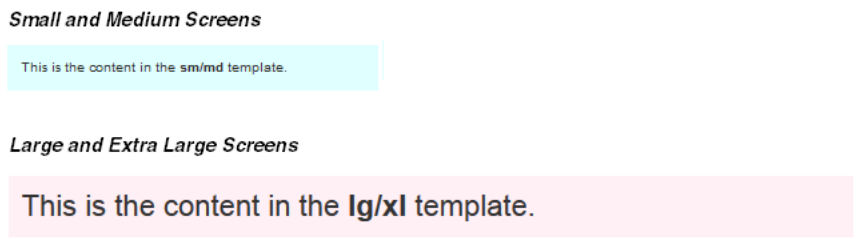
The Oracle JET Cookbook contains the complete code for this example which includes a demo that shows a computed observable used to change the button label's text

depending on the screen size. You can also find examples that show how to use custom media queries and Knockout computed observables. For details, see Responsive JavaScript Framework Queries.

For additional information about working with `oj-button`, see Working with Buttons.

Conditionally Loading Content Based on Screen Size

You can change the HTML content based on screen size using the responsive JavaScript classes. For example, you might want to use a larger font or a different background color when the screen size is large.



In this example, the HTML content is defined in Knockout templates. The markup uses Knockout's data-bind utility to display a template whose name depends on the value returned by the `large()` call. If the screen is small or medium, the application will use the `sm_md_template`. If the screen is large or larger, the application will use the `lg_xl_template`.

```
<div id="sample_container">

  <!-- large template -->
  <script type="text/html" id="lg_xl_template">
    <div id="lg_xl"
         style="background-color:lavenderblush;
                padding: 10px; font-size: 22px" >
     This is the content in the <strong>lg/xl</strong> template.
    </div>
  </script>

  <!-- small template -->
  <script type="text/html" id="sm_md_template">
    <div id="sm_md"
         style="background-color:lightcyan;
                padding: 10px; font-size: 10px" >
      This is the content in the <strong>sm/md</strong> template.
    </div>
  </script>


  <!-- display template -->
  <div data-bind="template: {name: large() ? 'lg_xl_template' :
                                             'sm_md_template'}"></div>
</div>
```

The code that sets the value for `large()` is identical to the code used for setting component option changes. For details, see Changing a Custom Element's Attribute Based on Screen Size.

For the complete code used in this example, see the Responsive Loading with JavaScript demo in the Oracle JET Cookbook.

Creating Responsive Images

You can use the responsive JavaScript classes to load a different image when the screen size changes. For example, you may want to load a larger image when the screen size changes from small and medium to large and up.

In this example, the image is defined in a HTML `img` element. The markup uses Knockout's data-bind utility to display a larger image when the `large()` call returns `true`.

```
<div id="samplecontainer">
  <img alt="puzzle" id="puzzle"
       data-bind="attr: { src: large() ? 'images/responsive/puzzle.png' :
                                         'images/responsive/puzzle_small.png'}" >

</div>
```

The code that set the value for `large()` is identical to the code used for setting component option changes. For details, see Changing a Custom Element's Attribute Based on Screen Size.

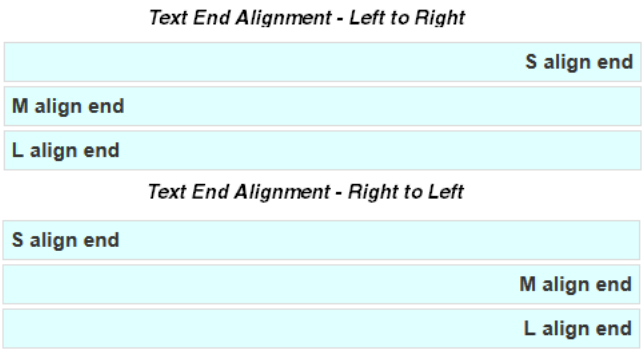
> **Note:**
>
> The image will not begin to load until the JavaScript is loaded. This could be an issue on devices with slower connections. If performance is an issue, you can use responsive CSS as described in Creating Responsive CSS Images. You could also use the HTML `picture` element which supports responsive images without CSS or JavaScript. However, browser support is limited and may not be an option for your environment.

For the complete code used in this example, see the Oracle JET Cookbook Responsive Images demos.

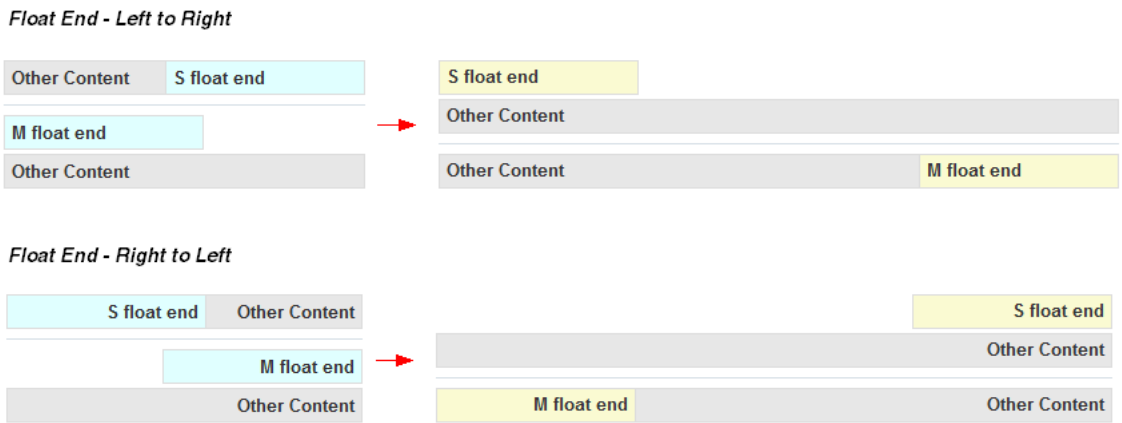
Using the Responsive Helper Classes

Use the Oracle JET generic responsive utility classes to hide content, end align text, and set float in your grid.

- `oj-`*`size`*`-hide`: Hide content at the specified *`size`*.
- `oj-`*`size`*`-text-align-end`: In left-to-right languages, set `text-align` to right. In right-to-left languages, set `text-align` to left.



- `oj-size-float-end`: In left-to-right languages, set `float` to right. In right-to-left languages, set `float` to left.



- `oj-size-float-start`: In left-to-right languages, set `float` to left. In right-to-left languages, set `float` to right.

To see examples that implement the responsive helper classes, consult the Oracle JET Cookbook at Responsive Helpers.

Creating Responsive CSS Images

Use CSS generated from Sass media query variables and responsive queries to use a different, larger image when the screen width changes from small to large.

The code below shows the markup that defines the image. In this example, `bulletlist` is a CSS class generated from the Sass responsive variables and media queries.

```
<div role="img" class="oj-icon bulletlist" title="bulleted list image"></div>
```

The following image shows the `bulletlist` CSS class. When the screen is small or medium size, the `icon_small.png` image loads. When the screen increases to large or larger, or to print, the `icon.png` loads instead.

```
.bulletlist {
  width: 24px;
  height: 24px; }
.bulletlist:before {
  content: url("images/hiResContrast/icon_small.png"); }
@media print and (orientation: landscape), screen and (min-width: 1024px) {
  .bulletlist {
    width: 48px;
    height: 48px; }
  .bulletlist:before {
    content: url("images/hiResContrast/icon.png"); } }
```

The Oracle JET Cookbook includes the Sass variables and queries used to generate the `bulletlist` CSS class. You can also find a Sass mixin that makes generating the CSS easier, but you are not required to use SCSS to create responsive CSS images.

In addition, the Oracle JET Cookbook includes examples that show high resolution images, sprites, button images, and more. For details, see Responsive CSS Images.

> **Note:**
>
> You can also use responsive JavaScript to change the images based on screen size. For details, see Creating Responsive Images.

Changing Default Font Size

By default, Oracle JET includes themes such as Alta that set a default font size on the root (`html`) element. This font size is optimized for visibility and touchability on mobile devices, but you can customize the size as needed for your application.

Topics

- Changing Default Font Size Across the Application
- Changing Default Font Size Based on Device Type

Changing Default Font Size Across the Application

The browser's font size is defined in the Sass `$rootFontSize` variable and included in the generated CSS `html` class. You can use Sass to change the variable or override the generated CSS.

To change the browser default font size across your application, do *one* of the following:

- In a custom Sass settings file, modify the Sass `$rootFontSize` variable, and regenerate the CSS.

  For details about customizing Oracle JET themes, see Customizing Themes Using the Tooling.

- In your application-specific CSS, override the font-size setting for the `html` class.

  For example, to set the browser's default font size to 12 pixels, add the following to your application-specific CSS:

```
html {
  font-size: 12px;
}
```

Changing Default Font Size Based on Device Type

You can change the default font size based on device type by detecting the device type used to access the browser and then setting the appropriate style on the `html` element.

To change the browser default font size based the user's device type:

1. Use whatever means you like to detect that the browser is running on the specified device.

   For example, you may want to change the browser's default font size on a desktop device. Use your own code or a third party tool to detect the device type.

2. When your application detects that the user is accessing the browser with the specified device, on the `html` element in your markup, set `style="font-size: xxpx"`. Substitute the desired pixel size for *xx*.

   For example, to set the font size to 12 pixels when the application detects the specified device, add logic to your application to add the highlighted code to your markup.

   ```
   <html style="font-size: 12px">
      ... contents omitted
   </html>
   ```

> **Note:**
>
> Be sure to perform this step before initializing components since some Oracle JET components measure themselves.

Controlling the Size and Generation of the CSS

You can change the size of the CSS content automatically generated by Oracle JET so that unused classes or particular types of classes are compressed, removed, excluded, or not generated.

When you use the responsive framework classes, Oracle JET generates a large number of classes that you may not need. Here are some steps you can take to control the size and generation of the CSS.

- Use compression.

  The responsive classes are often repetitive and compress well. For details about compressing your CSS, see Optimizing Performance.

- Remove unused classes.

  By default, Oracle JET generates responsive classes small, medium, large, and xlarge screens. If you know that your application will not use some of these classes, you can set the associated `$responsiveQuery*` variables to `none`.

  ```
  // If you don't want xlarge classes, you could set:
  $screenXlargeRange:         none;
  ```

```
$responsiveQueryLargeOnly:  none;
$responsiveQueryXlargeUp:   none;
```

- Exclude unused classes from the application layout, flex, grid, form layout, and responsive helper groups.

  You can use the following variables to exclude classes from these groups altogether:

  - `$includeAppLayoutClasses`
  - `$includeAppLayoutWebClasses`
  - `$includeFlexClasses`
  - `$includeGridClasses`
  - `$includeFormLayoutClasses`
  - `$includeResponsiveHelperClasses`

  For additional information about using the `$include*` variables, see Using Variables to Control CSS Content.

- Stop generation of a particular responsive helper class.

  For finer-grained control, there are additional variables that you can set to `false` to generation of a particular type of class.

Variable	Description
`$responsiveGenerateHide`	Generate hide classes like `.oj-md-hide`.
`$responsiveGenerateTextAlignEnd`	Generate text-align end classes like `.oj-md-text-align-end`.
`$responsiveGenerateFloatStart`	Generate float start classes like `.oj-md-float-start`.
`$responsiveGenerateFloatEnd`	Generate float end classes like `.oj-md-float-end`.

4

Using RequireJS for Modular Development

Oracle JET includes RequireJS, a third party JavaScript library that you can use in your application to load only the Oracle JET libraries you need. Using RequireJS, you can also implement lazy loading of modules or create JavaScript partitions that contain more than one module.

Topics:

- Typical Workflow for Using RequireJS
- Oracle JET and RequireJS
- Using RequireJS in an Oracle JET Application
- Adding Third-Party Tools or Libraries to Your Oracle JET Application
- Troubleshooting RequireJS in an Oracle JET Application
- Using JavaScript Partitions and RequireJS in an Oracle JET Application

Typical Workflow for Using RequireJS

Understand Oracle JET's module organization before using RequireJS in an Oracle JET application. You should also understand why you might use RequireJS in your application. For some features, you must use RequireJS. In other cases, its use is optional.

To start using RequireJS for modular development in Oracle JET, refer to the typical workflow described in the following table:

Task	Description	More Information
Understand Oracle JET's use of RequireJS	Understand Oracle JET's module organization and why you might use RequireJS in your Oracle JET application.	Oracle JET and RequireJS
Configure RequireJS	Configure RequireJS in your Oracle JET application to load Oracle JET modules as needed.	Using RequireJS in an Oracle JET Application
(Optional) Add third-party tools or libraries to your Oracle JET application	Understand the process to add third-party tools or libraries to your Oracle JET application.	Adding Third-Party Tools or Libraries to Your Oracle JET Application
Troubleshoot RequireJS	Troubleshoot issues with RequireJS.	Troubleshooting RequireJS in an Oracle JET Application
(Optional) Use JavaScript partitions and RequireJS in an Oracle JET application.	Configure RequireJS bundles.	Using JavaScript Partitions and RequireJS in an Oracle JET Application

Oracle JET and RequireJS

RequireJS is a JavaScript file and module loader that simplifies managing library references and is designed to improve the speed and quality of your code.

RequireJS implements the Asynchronous Module Definition (AMD) API which provides a mechanism for asynchronously loading a module and its dependencies.

Oracle JET's modular organization enables application developers to load a subset of needed features without having to execute `require()` calls for each referenced object. Each Oracle JET module represents one functional area of the toolkit, and it typically defines more than one JavaScript object. The `ojs/ojcore` module defines the `oj` namespace object, and all subsequently loaded modules place their exported objects within the `oj` namespace.

> **Note:**
>
> Oracle JET reserves the `oj` namespace, and you may not use `oj` as a namespace in your Oracle JET application.

You do not have to use RequireJS to reference Oracle JET libraries, but it is required if you plan to use Oracle JET's internationalization or data visualization components in your application. The Oracle JET download includes the RequireJS library, and it is used by default in the Oracle JET Starter Templates and Cookbook examples.

For more information about RequireJS, see http://requirejs.org.

Oracle JET Module Organization

The Oracle JET modules are listed in the following table with description and usage tips. Use this table to determine which modules you must load in your application.

Oracle JET Module	Description	When to Use?
`ojs/ojcore`	Core toolkit module that defines a base Oracle JET object. Includes support for prototype-based JavaScript object inheritance and extending JavaScript objects by copying their properties. The module returns `oj` namespace. You must include this module in any Oracle JET application.	Always
`ojs/ojmodel`	Oracle JET's Common Model	Use if your application uses the Oracle JET Common Model.
`ojs/ojknockout-model`	Utilities for integrating Oracle JET's Common Model into Knockout.js	Use if your application uses the Oracle JET Common Model, and you want to integrate with Knockout.js.

Oracle JET Module	Description	When to Use?
`ojs/`***`ojcomponent`***:	Oracle JET component modules. Examples include • `ojs/ojbutton` • `ojs/ojtoolbar` • `ojs/ojtabs` Most Oracle JET components have their own module with the same name in lowercase and without hyphens as shown above, except for the following components: • `oj-buttonset-*`: `ojs/ojbutton` • `oj-input-password`: `ojs/ojinputtext` • `oj-text-area`: `ojs/ojinputtext` • `oj-combobox-*`: `ojs/ojselectcombobox` • `oj-select-*`: `ojs/ojselectcombobox` • `oj-spark-chart`: `ojs/ojchart` • `oj-*-gauge`: `ojs/ojgauge`	Use component modules that correspond to any Oracle JET component in your application.
`ojs/ojknockout`	Oracle JET `ojComponent` binding and services for Knockout.js	Use if your application includes Oracle JET components and you want to use `ojComponent` binding for these components in Knockout.js.
`ojs/ojrouter`	Class for managing routing in single page applications	Use if your single page application uses `oj.Router` for routing.
`ojs/ojmodule`	Binding that implements navigation within a region of a single page application	Use if your single page application uses `ojModule` for managing content replacement within a page region.
`ojs/ojmoduleanimations`	Used in conjunction with `ojs/ojmodule`. Adds animation support via CSS animation effects.	Use if your application adds animation effects.
`ojs/ojoffcanvas`	Methods for controlling off-canvas regions	Use if your application uses `oj.offCanvasUtils` for managing off-canvas regions.
`ojs/ojcube`	Class for aggregating data values in `ojDataGrid`	Use if your application renders aggregated cubic data in an `ojDataGrid` component.
`ojs/oj`***`type`***`datasource`	Data source modules. Examples include: • `ojs/ojarraydatagriddatasource` • `ojs/ojcollectionpagingdatasource` • `ojs/ojjsontreedatasource`	Use if your application includes an Oracle JET component, and its data source is defined in one of the `oj.*DataSource` classes.

Oracle JET Module	Description	When to Use?
`basemaps/ojthematicmap-basemap-layer`	Built-in basemap to be loaded for `oj-thematic-map` via RequireJS. Examples include: • `basemaps/ojthematicmap-world-continents` • `basemaps/ojthematicmap-africa-countries` • `basemaps/ojthematicmap-usa-states`	Use if your application shows data with geographic locations and has several built-in maps.
`ojs/ojtimezonedata`	Time zone data	Use if you want to add time zone support to `oj-input-date-time`, `oj-input-time`, or converters.
`ojs/ojvalidaton-datetime`	Date and time validation and conversion services. Includes `ojs/ojvalidation-base`.	Use if your application does not have `ojInputDateTime` but uses a date and time converter or validator.
`ojs/ojvalidation-number`	Number validation and conversion services. Includes `ojs/ojvalidation-base`.	Use if your application does not have `ojInputNumber` but uses a number converter or validator.
`ojs/ojvalidation-base`	Some of the base classes for converters and validators. Smaller data validation and conversion services compared to `ojs/ojvalidation-datetime` or `ojs/ojvalidation-number` (for example, `oj.RequiredValidator`, `oj.LengthValidator`, `oj.RegExpValidator`).	Use if your application uses converters or validators that are not date, time or number outside of Oracle JET editable components.

Note:

Oracle JET includes several modules that aren't listed in this table because they are dependencies of other modules. When you load an Oracle JET component module, the module's dependencies are also loaded by default, and you do not need to include them explicitly. For example, `oj-dialog` requires the `ojs/ojcomponentcore` module, and the `ojs/ojcomponentcore` module will load automatically when you specify the `ojs/ojdialog` module in your RequireJS bootstrap file.

Using RequireJS in an Oracle JET Application

Oracle JET includes the RequireJS library and sample bootstrap file in the Oracle JET download. To use RequireJS in your application, edit the bootstrap file to add the

Oracle JET modules you need. You can also add your own modules as needed for your application code.

The code below shows the `main-template.js` bootstrap file distributed with the Oracle JET base distribution. Typically, you place the bootstrap file in your application's `js` directory and rename it to `main.js` . The comments in the code describe the purpose of each section. The sections that you normally edit are highlighted in bold.

```
requirejs.config({
  // Path mappings for the logical module names
  paths: {
  'knockout': 'libs/knockout/knockout-3.4.0',
  'jquery': 'libs/jquery/jquery-3.1.1.min',
  'jqueryui-amd': 'libs/jquery/jqueryui-amd-1.12.0.min',
  'ojs': 'libs/oj/v4.0.0/min',
  'ojL10n': 'libs/oj/v4.0.0/ojL10n',
  'ojtranslations': 'libs/oj/v4.0.0/resources',
  'text': 'libs/require/text',
  'promise': 'libs/es6-promise/es6-promise.min',
  'hammerjs': 'libs/hammer/hammer-2.0.8.min',
  'signals': 'libs/js-signals/signals.min',
  'ojdnd': 'libs/dnd-polyfill/dnd-polyfill-1.0.0.min'
  'css': 'libs/require-css/css.min',
  'customElements': 'libs/webcomponents/custom-elements.min',
  'proj4': 'libs/proj4js/dist/proj4'
},
  // Shim configurations for modules that do not expose AMD
  shim: {
    'jquery': {
       exports: ['jQuery', '$']
    }
},

// This section configures the i18n plugin. It is merging the Oracle JET built-in
translation
// resources with a custom translation file.
// Any resource file added, must be placed under a directory named "nls". You can
use a path mapping or you can define
// a path that is relative to the location of this main.js file.
    config: {
        ojL10n: {
            merge: {
                //'ojtranslations/nls/ojtranslations': 'resources/nls/
myTranslations'
            }
        }
    }
});

/**
 * A top-level require call executed by the Application.
 * Although 'ojcore' and 'knockout' would be loaded in any case (they are specified
as dependencies
 * by the modules themselves), we are listing them explicitly to get the references
to the 'oj' and 'ko'
 * objects in the callback
 *
 * For a listing of which JET component modules are required for each component,
 * see the specific component demo pages in the JET cookbook.
 */
require(['ojs/ojcore', 'knockout', 'jquery', 'ojs/ojknockout', 'ojs/ojbutton',
```

```
'ojs/ojtoolbar', 'ojs/ojmenu'],
  // add additional JET component libraries as needed
  function(oj, ko, $) // this callback gets executed when all required modules
are loaded
  {
      // add any startup code that you want here
  }
);
```

To use RequireJS in an Oracle JET Application:

If needed, install Oracle JET and RequireJS as described in Use RequireJS to Manage Library, Link, and Script References.

1. In the bootstrap file or your application scripts, in the `require()` definition, add additional Oracle JET modules as needed using the modules listed in Oracle JET Module Organization.

   For example, if your application uses the Oracle JET Common Model integrated with Knockout and includes an `oj-dialog`, you would add the highlighted code to your bootstrap file or application script.

   ```
   require(['ojs/ojcore', 'knockout', 'jquery', 'ojs/ojmodel', 'ojs/ojknockout-
   model','ojs/ojdialog'],
     function(oj, ko) // obtaining a reference to the oj namespace
     {
     }
   );
   ```

2. Add any scripts that your application uses to the `require()` definition and update the `function(oj, ko)` definition to include the script.

   For example, if your script is named `app.js`, add the highlighted code to your `require()` definition.

   ```
   require(['app', 'ojs/ojcore', 'knockout', 'jquery', 'ojs/ojmodel', 'ojs/
   ojknockout-model', 'ojs/ojdialog'],
     function(app, oj, ko) // obtaining a reference to the oj namespace
     {
     }
   );
   ```

   Note:

   You can also use RequireJS with `ojModule` to define view templates and viewModels for page sections in a single-page application. For example, the Oracle JET Starter Templates use the `ojModule` binding with RequireJS to use a different view and viewModel when the user clicks one of the navigation buttons.
   For additional information about the Oracle JET Starter Templates, see Create a Web Application Using the Oracle JET Starter Templates. For more information about using `ojModule` and templates, see Creating Single-Page Applications.

3. Add any application startup code to the callback function.

   For example, the following code adds a Knockout binding call to the callback function for an element named `dialogWrapper`.

```
require(['app', 'ojs/ojcore', 'knockout', 'jquery', 'ojs/ojmodel', 'ojs/
ojknockout-model', 'ojs/ojdialog'],
  function(app, oj, ko) // obtaining a reference to the oj namespace
  {
    ko.applyBindings(new app()/*View Model instance*/,
                     document.getElementById('dialogWrapper'));
  }
);
```

4. If your application includes resource bundles, enter the path to the bundle in the `merge` section.

```
config: {
  ojL10n: {
    merge: {
      'ojtranslations/nls/ojtranslations': 'resources/nls/myTranslations'
    }
  }
}
```

Adding Third-Party Tools or Libraries to Your Oracle JET Application

You can add third-party tools or libraries to your Oracle JET application. If you're using RequireJS, you must update the bootstrap file. If you use the tooling, you must also install the library and make modifications to the `scripts/config/oraclejet-build.js` and `js/main-release-paths.json` files.

> **Note:**
>
> This process is provided as a convenience for Oracle JET developers. Oracle JET will not support the additional tools or libraries and cannot guarantee that they will work correctly with other Oracle JET components or toolkit features.

To add a third-party tool or library to your Oracle JET application:

1. If you scaffolded your application using the tooling, do the following:

   a. In your main project directory, enter the following command to install the library using npm:

   ```
   npm install library-name --save
   ```

   For example, enter the following command to install a library named `my-library`:

   ```
   npm install my-library --save
   ```

   b. In the `scripts/config` directory, update the `copyCustomLibsToStaging` section in `oraclejet-build.js` file so that it copies your library to the application when you build the application.

   Open the file in your favorite text editor and remove the comments (`//`) from the `copyCustomLibsToStaging` section. Add the new libraries as shown below.

```
/**
 * # copyCustomLibsToStaging
 * This task copies any custom libraries that are not provided by JET to
staging directory.
 * This task supports a single option: fileList. The fileList option defines
an array of file objects.
 * Each file object contains the following properties:
 *   cwd, current working directory
 *   dest, destination path
 *   src, array of source file patterns
 *   rename, function to return the full path of desired destination
 * If a fileList value is specified, it completely replaces the default
fileList value defined by JET
 * Example: {cwd: 'app', src: ['**', '!test.js'], dest: 'staging',  rename:
function (dest, file) {return renamed path}}
 */
copyCustomLibsToStaging: {
    fileList: [
               {cwd:'node_modules/my-library/',
                src: ['*'],
                dest: 'web/js/libs/my-library'}
               ]
},
```

c. In the `js` directory update the `js/main-release-paths.json` file to include the new library.

   Use the minified version if available as shown below. The tooling uses this file when you build your release version.

```
{
    "knockout": "libs/knockout/knockout-x.x.x",
    "jquery": "libs/jquery/jquery-x.x.x.min",
    ... contents omitted
    "my-library": "libs/my-library/my-library.min"
}
```

2. If you didn't use the tooling to create your application, in the application's `js/libs` directory, create a new directory and add the new library and any accompanying files to it.

   For example, for a library named `my-library`, create the `my-library` directory and add the `my-library.js` file and any needed files to it. Be sure to add the minified version if available.

3. In your RequireJS bootstrap file, typically `main.js`, add a link to the new file in the path mapping section and include the new library in the `require()` definition.

   For example, add the highlighted code below to your bootstrap file to use a library named `my-library`.

```
requirejs.config({
  // Path mappings for the logical module names
  paths:
  {
    'knockout': 'libs/knockout/knockout-3.4.0',
    'jquery': 'libs/jquery/jquery-3.1.0.min',
    ... contents omitted
    'text': 'libs/require/text',
    'my-library': 'libs/my-library/my-library
  }
  ,
```

```
  // Shim configurations for modules that do not expose AMD
  // Updated as needed for your library.
  shim: {
    'jquery': {
      exports: ['jQuery', '$']
    }
  },
  require(['ojs/ojcore', 'knockout', 'jquery', 'my-library'],
  function(oj, ko, $) // this callback gets executed when all required modules
are loaded
  {
      // add any startup code that you want here
  }
);
```

For additional information about using RequireJS to manage your application's modules, see Using RequireJS for Modular Development.

Troubleshooting RequireJS in an Oracle JET Application

RequireJS issues are often related to modules used but not defined.

Use the following tips when troubleshooting issues with your Oracle JET application that you suspect may be due to RequireJS:

- Check the JavaScript console for errors and warnings. If a certain object in the `oj` namespace is undefined, locate the module that contains it based on the information in Oracle JET Module Organization or the Oracle JET Cookbook and add it to your application.
- If the components you specified using Knockout.js binding are not displayed and you are not seeing any errors or warnings, verify that you have added the `ojs/ojknockout` module to your application.

Using JavaScript Partitions and RequireJS in an Oracle JET Application

RequireJS supports JavaScript partitions that contain more than one module.

You must name all modules using the RequireJS `bundles` option and supply a path mapping with the configuration options.

```
requirejs.config(
  {
    bundles:
    {
      'commonComponents': ['ojL10n', 'ojtranslations/nls/ojtranslations',
                           'ojs/ojcore', 'ojs/ojknockout', 'ojs/ojcomponentcore',
                           'ojs/ojbutton', 'ojs/ojpopup'],
      'tabs': ['ojs/ojtabs', 'ojs/ojconveyorbelt']    }
  }
);
```

In this example, two partition bundles are defined: `commonComponents` and `tabs`.

RequireJS ships with its own Optimizer tool for creating partitions and minifying Javascript code. The tool is designed to be used at build time with a complete project

that is already configured to use RequireJS. It analyzes all static dependencies and creates partitions out of modules that are always loaded together. The Oracle JET team recommends that you use an optimizer to minimize the number of HTTP requests needed to download the modules.

For additional information about the RequireJS Optimizer tool, see http://requirejs.org/docs/optimization.html.

For additional information about optimizing performance of Oracle JET applications, see Optimizing Performance.

5
Creating Single-Page Applications

Oracle JET includes the `ojModule` binding and `oj.Router` framework class that you can use to create single-page applications that simulate the look and feel of desktop applications.

The `ojModule` binding can work in conjunction with `oj.Router` , which provides routing support for Oracle JET applications.

Topics:

- Typical Workflow for Creating Single-Page Applications in Oracle JET
- Designing Single-Page Applications Using Oracle JET
- Using ojModule

Typical Workflow for Creating Single-Page Applications in Oracle JET

Understand Oracle JET's support for designing single-page applications. Learn how to use `ojModule` binding and `ojModule`'s life cycle listeners in your application.

To create a single-page application (SPA) in Oracle JET, refer to the typical workflow described in the following table:

Task	Description	More Information
Create a single-page Oracle JET application	Identify Oracle JET's support for single-page applications and how to use it to design your Oracle JET application.	Designing Single-Page Applications Using Oracle JET
Create view templates and viewModels	Identify the features and benefits of `ojModule` and how to use it in your Oracle JET application.	Using ojModule

Designing Single-Page Applications Using Oracle JET

Oracle JET includes Knockout for separating the model layer from the view layer and managing the interaction between them. Using Knockout, the Oracle JET `ojModule` binding, and the Oracle JET `oj.Router` framework class, you can create single-page applications that look and feel like a standalone desktop application.

Topics:

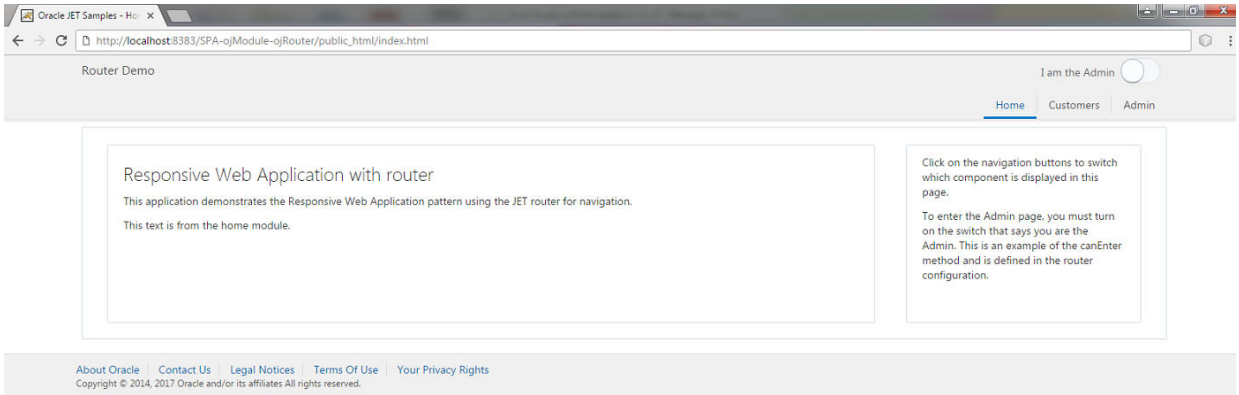
- Understanding Oracle JET Support for Single-Page Applications
- Creating a Single-Page Application in Oracle JET

Understanding Oracle JET Support for Single-Page Applications

Single-page applications (SPAs) are typically used to simulate the look and feel of a standalone desktop application. Rather than using multiple web pages with links between them for navigation, the application uses a single web page that is loaded only once. If the page changes because of the user's interaction, only the portion of the page that changed is redrawn.

Oracle JET includes support for single page applications using the `oj.Router` class for virtual navigation in the page, the `ojModule` binding for managing view templates and viewModel scripts, and Knockout for separating the model layer from the view layer and managing the binding between them.

In the Oracle JET SPA-ojModule-ojRouter application shown in the image below, the `index.html` page displays the application shell with a toolbar for choosing the **Home**, **Customers**, and **Admin** content.



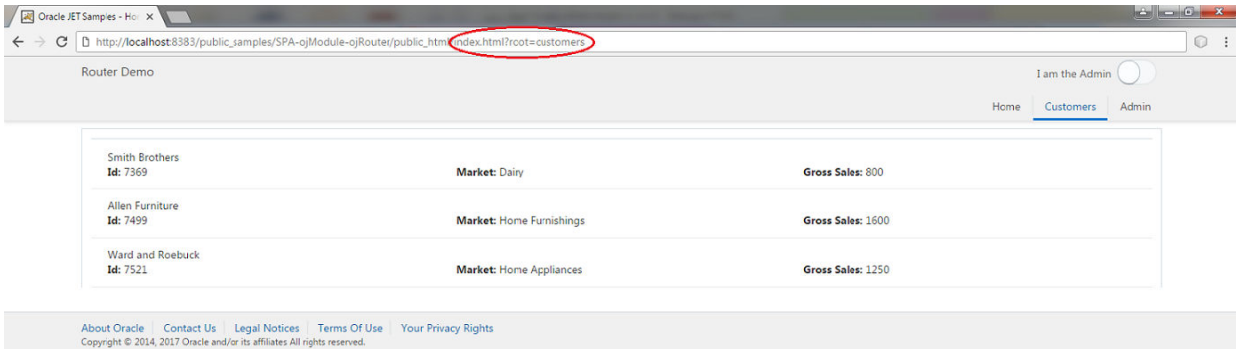
When the user selects a toolbar item such as the **Customers** button, the new content displays, and the URL changes to reflect the user's current location on the page.

If the user chooses several chapters to display, the browser remembers the previous selections. When the user selects the browser's back function, the URL and page content change to reflect the user's previous selection.

The router comes with two URL adapters. Each adapter defines how the URL is formatted to represent the application state:

- `urlPathAdapter` formats the URL in path segments. Each segment is the possessive router's current state id separated by a '/' , like `/customers`.
- `urlParamAdapter` formats the URL using query parameters. Each parameter is the router name and its current state id like `?root=customers`.

In this example, routing is handled using query parameters (*`web_site`*`/public_html/index.html`**`?root=customers`**).



You can also configure routing to use path segments (*web_site*/`public_html/customers`). To configure routing to use path segments, you must have

- Access to a server
- URL rewrite rules configured properly on the server
- Commented out the line in the `main.js` file: `//oj.Router.defaults['urlAdapter'] = new oj.Router.urlParamAdapter();`

The default adapter for the router is the urlPathAdapter adapter. However, the Oracle JET SPA-ojModule-ojRouter application uses the urlParamAdapter to ensure that it can run in any server environment without modification to the server.

When routing a single-page application, the page doesn't reload from scratch but the content of the page changes dynamically. In order to be part of the browser history and provide bookmarkable content, the Oracle JET router emulates the act of navigating using the HTML5 history push state feature. The router also controls the URL to look like traditional page URLs. However, there are no resources at those URLs, and you must set up the HTML server. This is done using a simple rule for a rewrite engine, like mod rewrite module for Apache HTTP server or a rewrite filter like UrlRewriteFilter for servlets.

In general, use query parameters when your application contains only a few views that the user will bookmark and that are not associated with a complex state. Use path segments to display simpler URLs, especially for nested paths such as `customers/cust/orders`.

The Oracle JET Cookbook and Oracle JET sample applications use the Oracle JET `ojModule` feature to manage the Knockout binding. With `ojModule`, you can store your HTML content for a page section in an HTML fragment or template file and the JavaScript functions that contain your viewModel in a viewModel file.

When `ojModule` and `oj.Router` are used in conjunction, you can configure an `ojModule` object where the module name is the router state. When the router changes state, `ojModule` will automatically load and render the content of the module name specified in the value of the current `RouterState` object.

Creating a Single-Page Application in Oracle JET

The Oracle JET Cookbook includes complete examples and recipes for creating a single-page application using path segments and query parameters for routing and

examples that use routing with the `ojModule` binding. Regardless of the routing method you use, the process to create the application is similar.

To create a single-page application in Oracle JET:

If needed, create the application that will house your main HTML5 page and supporting JavaScript. For additional information, see Getting Started with Oracle JET Web Application Development or Getting Started with Oracle JET Hybrid Mobile Application Development.

1. Design the application's structure.

   Identify the templates and ViewModels that your application will require. For example, the SPA-ojModule-ojRouter single-page sample application defines the HTML template for header and footer navigation items and uses `ojModule` in conjunction with `oj.Router` to navigate through the content. The code sample below shows the `index.html` file, with the router definition highlighted.

```
<!DOCTYPE html>
<!--
   Copyright (c) 2014, 2017, Oracle and/or its affiliates.
   The Universal Permissive License (UPL), Version 1.0
   -->
<html lang="en-us">
   <head>
      ... contents omitted
   </head>
   <body class="oj-web-applayout-body">
      <!-- Template for rendering navigation items shared between nav bar and
nav list -->
      <script type="text/html" id="navTemplate">
         <li><a href="#">
           <span data-bind="css: $data['iconClass']"></span>
           <!-- ko text: $data['name'] --> <!--/ko-->
         </a></li>
      </script>
      <div id="globalBody" class="oj-offcanvas-outer-wrapper oj-offcanvas-page">
         <!--
            ** Oracle JET V4.0.0 web application navigation drawer pattern.
            ** Please see the demos under Cookbook/Patterns/App Shell: Web
            ** and the CSS documentation under Support/API Docs/Non-Component
Styling
            ** on the JET website for more information on how to use this
pattern.
            ** The off-canvas section is used when the browser is resized to a
smaller media
            ** query size for a phone format and hidden until a user clicks on
            ** the header hamburger icon.
            -->
         <div id="navDrawer" role="navigation" class="oj-contrast-marker oj-web-
applayout-offcanvas oj-offcanvas-start">
            <oj-navigation-list data="[[navDataSource]]"
               edge="start"

item.renderer="[[oj.KnockoutTemplateUtils.getRenderer('navTemplate', true)]]"
               on-click="[[toggleDrawer]]"
               selection="{{router.stateId}}">
            </oj-navigation-list>
         </div>
         <div id="pageContent" class="oj-web-applayout-page">
            <!--
```

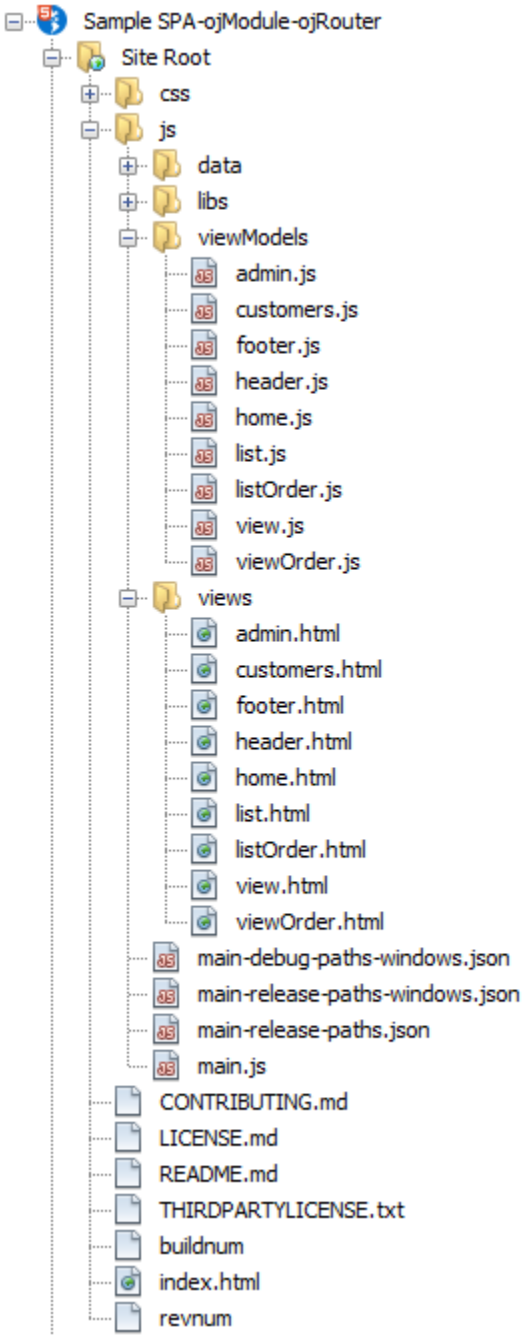
```
                ** Oracle JET V4.0.0 web application header pattern.
                ** Please see the demos under Cookbook/Patterns/App Shell: Web
                ** and the CSS documentation under Support/API Docs/Non-Component
Styling
                ** on the JET website for more information on how to use this
pattern.
                -->
            <header role="banner" class="oj-web-applayout-header">
               <div class="oj-web-applayout-max-width oj-flex-bar oj-sm-align-
items-center">
                  <!-- Offcanvas toggle button -->
                  <div class="oj-flex-bar-start oj-md-hide">
                     <oj-button id="drawerToggleButton" class="oj-button-lg" on-
click="[[toggleDrawer]]" chroming="half" display="icons">
                        <span slot="startIcon" class="oj-web-applayout-offcanvas-
icon"></span>
                        <span>Application Navigation</span>
                     </oj-button>
                  </div>
                  <div class="oj-flex-bar-middle oj-sm-align-items-baseline">
                     <span role="img" class="oj-icon demo-oracle-icon"
title="Oracle Logo" alt="Oracle Logo"></span>
                     <h1 class="oj-sm-only-hide oj-web-applayout-header-title"
title="Application Name" data-bind="text: appName"></h1>
                  </div>
                  <div class="oj-flex-bar-end">
                     <oj-label id="switchLabel" class="oj-label oj-label-inline"
for="switch">I am the Admin </oj-label>
                     <oj-switch id="switch" value="{{isAdmin}}"></oj-switch>
                  </div>
               </div>
               <div role="navigation" class="oj-web-applayout-max-width oj-web-
applayout-navbar">
                  <oj-navigation-list class="oj-sm-only-hide oj-md-condense oj-
md-justify-content-flex-end"
                     data="[[navDataSource]]"
                     edge="top"

item.renderer="[[oj.KnockoutTemplateUtils.getRenderer('navTemplate', true)]]"
                     selection="{{router.stateId}}">
                  </oj-navigation-list>
               </div>
            </header>
            <div class="oj-web-applayout-content oj-web-applayout-max-width">
               <div class="oj-flex oj-flex-items-pad">
                  <div class="oj-flex-item">
                     <div id="content" role="main" class="oj-panel"
style="padding-bottom:30px"
                     data-bind="ojModule: router.moduleConfig"></div>
                  </div>
               </div>
            </div>
            <footer class="oj-web-applayout-footer" role="contentinfo">
               <div class="oj-web-applayout-footer-item oj-web-applayout-max-
width">
                  <ul>
                     <!-- ko foreach: footerLinks -->
                     <li><a data-bind="text : name, attr : {id: linkId, href :
linkTarget}"></a></li>
                     <!-- /ko -->
                  </ul>
```

```
                </div>
                <div class="oj-web-applayout-footer-item oj-web-applayout-max-
width oj-text-secondary-color oj-text-sm">
                   Copyright © 2014, 2017 Oracle and/or its affiliates All rights
reserved.
                </div>
            </footer>
         </div>
      </div>
   </body>
</html>
```

In this example, `ojModule` manages content replacement for the header, main content, and footer sections of the body, and `oj.Router` provides routing support. For additional information about `ojModule`, see Using ojModule.

The application also contains content that displays when the user clicks **Customers** and **Admin** in the application header. To manage the content for those sections, the application includes the `customers.html` and `admin.html` view templates with corresponding `customers.js` and `admin.js` ViewModels.

The image below shows a portion of the directory structure for the SPA-ojModule-ojRouter single-page sample application. The ViewModel definitions are contained in JavaScript files in the `js/viewModels` folder, and the view templates are contained in HTML files in the `js/views` folder.



2. Add code to your application's main script that defines the states that the router can take, and add the `ojs/ojrouter` module to your `require()` list.

   For example, in the SPA-ojModule-ojRouter sample application, the router can be in the `home`, `customers`, or `admin` states. The code sample below shows the code that configures the router.

```
require(['ojs/ojcore', 'knockout', 'jquery',
        'ojs/ojknockout', 'ojs/ojbutton', 'ojs/ojmenu', 'ojs/ojtoolbar', 'ojs/
ojnavigationlist',
        'ojs/ojoffcanvas', 'ojs/ojarraytabledatasource', 'ojs/ojmodule', 'ojs/
ojrouter', 'text', 'ojs/ojcheckboxset', 'ojs/ojswitch'
    ],
    function(oj, ko, $) {
```

```
        'use strict';
        // Set debug mode and log level
        // oj.Assert.forceDebug();
        // oj.Logger.option('level',  oj.Logger.LEVEL_INFO);

        // Router setup
        oj.Router.defaults['urlAdapter'] = new oj.Router.urlParamAdapter();
        var router = oj.Router.rootInstance;
        router.configure({
            'home': {
                label: 'Home',
                isDefault: true,
                enter: function() {
                    console.log('entered Home');
                }
            },
            'customers': {
                label: 'Customers',
                exit: function() {
                    console.log('exited customers');
                }
            },
            'admin': {
                label: 'Admin',
                canEnter: function() {
                    return
ko.dataFor(document.getElementById('switch')).isAdmin();
                }
            }
        });

        function ViewModel() {
            var self = this;

            // Media queries for repsonsive layouts
            var smQuery =
oj.ResponsiveUtils.getFrameworkQuery(oj.ResponsiveUtils.FRAMEWORK_QUERY_KEY.SM_ON
LY);
            self.smScreen =
oj.ResponsiveKnockoutUtils.createMediaQueryObservable(smQuery);
            var mdQuery =
oj.ResponsiveUtils.getFrameworkQuery(oj.ResponsiveUtils.FRAMEWORK_QUERY_KEY.MD_UP
);
            self.mdScreen =
oj.ResponsiveKnockoutUtils.createMediaQueryObservable(mdQuery);

            self.router = router;

            // Navigation setup
            var navData = [{
                    name: 'Home',
                    id: 'home',
                    iconClass: 'oj-navigationlist-item-icon demo-icon-font-24
demo-chart-icon-24'
                },
                {
                    name: 'Customers',
                    id: 'customers',
                    iconClass: 'oj-navigationlist-item-icon demo-icon-font-24
demo-people-icon-24'
                },
```

```
                {
                    name: 'Admin',
                    id: 'admin',
                    iconClass: 'oj-navigationlist-item-icon demo-icon-font-24
demo-info-icon-24'
                }
            ];
            self.navDataSource = new oj.ArrayTableDataSource(navData, {
                idAttribute: 'id'
            });

            // Drawer
            // Close offcanvas on medium and larger screens
            self.mdScreen.subscribe(function() {
                oj.OffcanvasUtils.close(self.drawerParams);
            });
            self.drawerParams = {
                displayMode: 'push',
                selector: '#navDrawer',
                content: '#pageContent'
            };
            // Called by navigation drawer toggle button and after selection of
nav drawer item
            self.toggleDrawer = function() {
                return oj.OffcanvasUtils.toggle(self.drawerParams);
            }
            // Add a close listener so we can move focus back to the toggle
button when the drawer closes
            $("#navDrawer").on("ojclose", function() {
                $('#drawerToggleButton').focus();
            });

            // Header
            // Application Name used in Branding Area
            self.appName = ko.observable("Router Demo");
            // User Info used in Global Navigation area
            self.userLogin = ko.observable("john.hancock@oracle.com");

            self.selectHandler = function(event, ui) {
                if (('ojBeforeSelect' === event.type) &&
event.detail.originalEvent) {
                    // router takes care of changing the selection
                    event.preventDefault();
                    // Invoke go() with the selected item.
                    self.router.go(event.detail.key);
                }
            };

            self.isAdmin = ko.observable(false);

            // Footer
            function footerLink(name, id, linkTarget) {
                this.name = name;
                this.linkId = id;
                this.linkTarget = linkTarget;
            }
            self.footerLinks = ko.observableArray([
                new footerLink('About Oracle', 'aboutOracle', 'http://
www.oracle.com/us/corporate/index.html#menu-about'),
                new footerLink('Contact Us', 'contactUs', 'http://
www.oracle.com/us/corporate/contact/index.html'),
```

```
                new footerLink('Legal Notices', 'legalNotices', 'http://
www.oracle.com/us/legal/index.html'),
                new footerLink('Terms Of Use', 'termsOfUse', 'http://
www.oracle.com/us/legal/terms/index.html'),
                new footerLink('Your Privacy Rights', 'yourPrivacyRights',
'http://www.oracle.com/us/legal/privacy/index.html')
            ]);

        }


        oj.Router.sync().then(
            function() {
                ko.applyBindings(new ViewModel(),
document.getElementById('globalBody'));
            },
            function(error) {
                oj.Logger.error('Error when starting router: ' + error.message);
            }
        );
    }
);
```

In this example, the router is configured for query parameters. To configure the router for path segments, comment out the following line as shown below:

```
//oj.Router.defaults['urlAdapter'] = new oj.Router.urlParamAdapter();
```

To configure the router for path segments, you must also set the `baseUrl` property of the `requirejs.config`. This provides the router with a base to calculate which part belongs to the router state and which part is the existing URL.

```
requirejs.config({
   // Need to set baseUrl or nested view won't work because module location
relative to current url.
   // Change to the correct baseUrl when deployed to site like: http://host/myApp
   baseUrl: window.location.href.split('#')[0].substring(0,
window.location.href.split('#')[0].lastIndexOf('/')) + '/js',
// Path mappings for the logical module names
   paths: {
   ... contents omitted
});
```

> **Note:**
>
> Routing with path segments also requires that the web server recognizes the paths as existing resources and routes the requests to the root page. If you're using the Apache web server, you can use the mod_rewrite module to rewrite requested URLs on demand. For other web servers, you can create a servlet filter or use one of the publicly available URL rewrite filters. In addition to rewriting the URL, the base URL needs to be set so that the router can retrieve the part of the URL representing the state. If your application start at the context root and is using `index.html`, nothing needs to be done because the default value of `baseUrl` is `'/'`. If your application is nested in a directory named `myApp`, then the base URL should be changed:
>
> ```
> oj.Router.defaults['baseUrl'] = '/myApp/';
> ```
>
> If the `<base href>` tag is set in your starting page, it is usually the same value used for the router base URL.

3. Add code to the markup that triggers the state transition and displays the content of the current state.

   When the user clicks one of the buttons in the header, the content is loaded according to the router's current state.

   For additional information about creating templates and ViewModels, see Using ojModule.

4. To manage routing within a module, add a child router using `oj.Router.createChildRouter()`.

   For example, when the user clicks **Customers** , the page displays a list of customer names with IDs that the user can select. The user can move back and forth between the customer details, and the child router maintains the state. The HTML markup is stored in the `Customers.html` template and is shown below.

   ```
   <div id="customers" data-bind="ojModule: moduleConfig"></div>
   ```

   The template defines the Oracle JET `oj-navigation-list` component and uses the router's state IDs as an argument to the `oj-navigation-list`'s `selection` attribute. To display the customer list, the template iterates through each `RouterState` object in the `router.states` property.

   The code sample below shows the `customer.js` file that defines the child routers. There are two child routers in this code sample, one to display customers (either a list or details of a single customer) and another one to display orders of a specific customer. `custRouter` is a child of the parent router (`params`) and `orderRouter` is a child router of `custRouter`. The state of both the child routers along with the parent router determines the content of the page region.

   ```
   define(['ojs/ojcore', 'knockout', 'jquery', 'ojs/ojknockout', 'ojs/ojrouter'],
   function(oj, ko, $)
   {
      'use strict';

      var custArray = [
               {id:7369, name: 'Smith Brothers', market: 'Dairy', sales: 800,
   deptno: 20, orders: [
                     {id:100, name:'A'},
                     {id:110, name:'B'},
   ```

```
                  {id:120, name:'C'}] },
            {id:7499, name: 'Allen Furniture', market: 'Home Furnishings',
sales: 1600, deptno: 30, orders: [
                  {id:101, name:'A'},
                  {id:102, name:'C'}] },
            {id:7521, name: 'Ward and Roebuck', market: 'Home Appliances',
sales: 1250, deptno: 30, orders: [
                  {id:120, name:'A'},
                  {id:121, name:'F'},
                  {id:122, name:'G'}] },
            {id:7566, name: 'Jones Brothers', market: 'Reality', sales: 2975,
deptno: 20, orders: [
                  {id:130, name:'B'},
                  {id:131, name:'C'},
                  {id:132, name:'D'}] },
            {id:7654, name: 'Martin Marina', market: 'Water Sports', sales:
1250, deptno: 30, orders: [
                  {id:140, name:'C'},
                  {id:141, name:'B'}] },
            {id:7698, name: 'Blake and Sons', market: 'Accounting', sales: 2850,
deptno: 30, orders: [
                  {id:150, name:'E'},
                  {id:151, name:'G'}] },
            {id:7782, name: 'Clark Candies', market: 'Confectionaries', sales:
2450, deptno: 10, orders: [
                  {id:160, name:'B'},
                  {id:161, name:'E'}] },
            {id:7788, name: 'Scott Lawn Service', market: 'Gardening Supplies',
sales: 3000, deptno: 20, orders: [
                  {id:170, name:'C'},
                  {id:171, name:'D'}] },
            {id:7839, name: 'King Power', market: 'Champion Builders', sales:
5000, deptno: 10, orders: [
                  {id:180, name:'A'},
                  {id:181, name:'F'},
                  {id:182, name:'C'}] }];

  function viewModel(params)
  {
    var parentRouter = params.ojRouter.parentRouter;
    var routerConfig =
    {
     'list': { isDefault: true }
    };

    // Populate the router config object with all the items from the table
    custArray.forEach(function(item)
    {
      var id = item.id.toString();
      routerConfig[id] = { label: item.name, value: item };
    });

    // Create and configure the router
    this.custRouter =
parentRouter.createChildRouter('cust').configure(routerConfig);

    // Create and configure the order router for this model.
    // Uses a dynamic router so it can managed orders for any emp
    this.orderRouter =
this.custRouter.createChildRouter('order').configure(function(stateId)
    {
```

```
      if (stateId)
      {
        return new oj.RouterState(stateId, { value: stateId === 'orders' ?
'listOrder' : stateId });
      }
    });

    // This is the main logic to switch the module based on both router states.
    this.moduleConfig = ko.pureComputed(function()
    {
      var moduleConfig;
      var orderStateId = this.orderRouter.stateId();

      if (orderStateId)
      {
        if (orderStateId === 'orders')
        {
          // Pass a reference of the orders array to the child module to render
the list
          moduleConfig = $.extend(true, {}, this.orderRouter.moduleConfig,
          {
            'params': { 'data': this.custRouter.currentValue.peek().orders }
          });
        }
        else
        {
          // Change the module name and pass the orderId down to the module
          moduleConfig = $.extend(true, {}, this.orderRouter.moduleConfig,
          {
            'name': 'viewOrder',
            'params': { 'data': orderStateId }
          });
        }
      }
      else if (this.custRouter.stateId() === 'list')
      {
        // Pass a reference of empArray to the child module to render the list
        moduleConfig = $.extend(true, {}, this.custRouter.moduleConfig,
        {
          'params': { 'data': custArray }
        });
      }
      else
      {
        // Change the module name and pass the emp down to the module
        moduleConfig = $.extend(true, {}, this.custRouter.moduleConfig,
        {
          'name': 'view',
          'params': { 'data': this.custRouter.currentValue.peek() }
        });
      }

      return moduleConfig;
    }, this);

    this.handleActivated = function()
    {
      // Now that the router for this view exist, synchronise it with the URL
      return oj.Router.sync().
      then(
         null,
```

```
            function(error) {
               oj.Logger.error('Error during refresh: ' + error.message);
            }
         );
      };

      this.dispose = function()
      {
        // Every router is destroyed on dispose.
        this.custRouter.dispose();
      };

    }

    // Return constructor function
    return viewModel;
  });
```

> **Note:**
>
> In this example, ojModule's `handleActivated` life cycle callback plays an important role in making the page work. `oj.Router.sync()` synchronizes the new router state with the current URL. The synchronization gives the `custRouter` child router a chance to transition to its default state, which is `list` in this example.

5. Add any remaining code needed to complete the content or display.

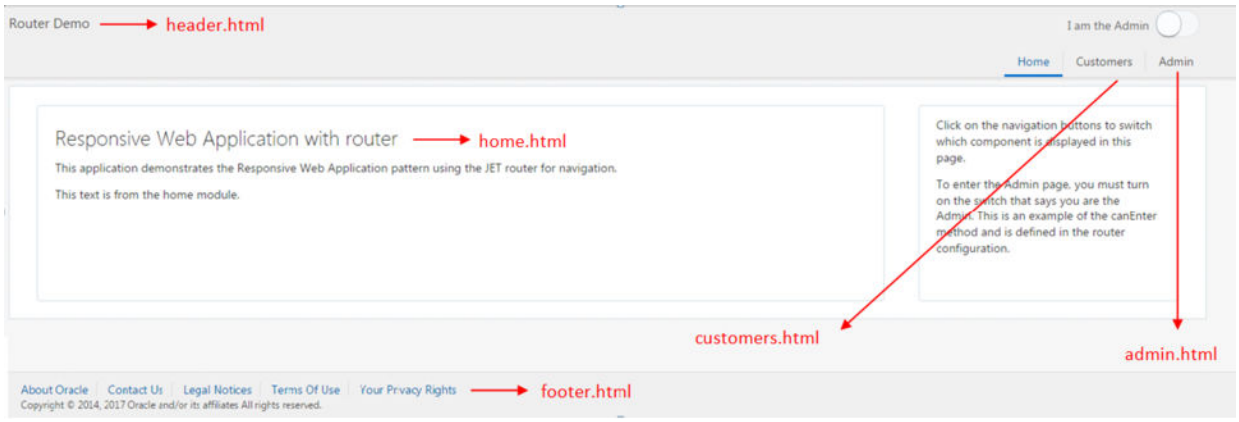
You can download the Oracle JET SPA-ojModule-ojRouter application at SPA-ojModule-ojRouter.zip. The sample application includes the simple `index.html` page displayed in the preceding example and the content for the Customers and Admin sections.

The Oracle JET Cookbook also includes additional Routing Demos that implement `oj.Router` and a link to the oj.Router API documentation.

Using ojModule

Many of the Oracle JET Cookbook and sample applications use `ojModule` to manage the Knockout binding. With `ojModule`, you can store your HTML content for a page section in an HTML fragment or template file and the JavaScript functions that contain your viewModel in a viewModel file.

For example, the SPA-ojModule-ojRouter single-page sample application uses various view templates to define the HTML markup for the application. The image below shows the main page for the application as it runs on the desktop, with arrows pointing to the names of five such templates defining view content for the application.



> **Note:**
>
> The **Customers** tab also uses templates to display details of a single customer, list of customers, orders for a customer, and details of a specific order.

The `home` view template is shown below. In the highlighted code, the first `data-bind` definition uses Knockout's `text` binding to set the `h2` element's content to a text node containing the value of `title`. The second `data-bind` definition uses Knockout's `foreach` binding to iterate through the content, and the list items display the text defined in `line`.

```
<div class="oj-flex oj-flex-items-pad">
  <div class="oj-xl-9 oj-lg-9 oj-md-9 oj-sm-12 oj-flex-item">
    <div role="main" class="oj-panel demo-page-content-area">
      <h2 data-bind="text: title"></h2>
      <!-- ko foreach: description -->
        <p data-bind="text: line"></p>
      <!-- /ko -->
    </div>
  </div>
  <div class="oj-xl-3 oj-lg-3 oj-md-3 oj-sm-12 oj-flex-item">
    <div class="oj-panel oj-panel-alt2 demo-page-content-area">
      <p>Click on the navigation buttons to switch which component is displayed in
this page.</p>
          <p>To enter the Admin page, you must turn on the switch that says you are
the Admin.
             This is an example of the canEnter method and is defined in the router
configuration.</p>
    </div>
  </div>
</div>
```

The code that defines the `title` and `line` is contained in the viewModel for the `home` section's content, stored in the `home.js` viewModel.

```
/* Home content module, example of singleton view model object. */
define(['knockout'], function(ko) {
  var viewModel = {
    title: 'Responsive Web Application with router',
    description:[{
                line: 'This application demonstrates the Responsive Web Application
```

```
pattern using the \
JET router for navigation.'
      },{
        line: 'This text is from the home module.'
      }]
  };
  return viewModel;
});
```

To use `ojModule` in your Oracle JET application:

If needed, create the application that will house your main HTML5 page and supporting JavaScript. For additional information, see Getting Started with Oracle JET Web Application Development.

1. To modify the default location of the application's view templates and viewModel scripts, add code to the application's main script that initializes the location of the viewModels and view templates and defines the suffix for the view templates.

   The `ojModule` binding namespace contains the default location and names of managed view templates and viewModels.

Property	Description
`viewPath`	Defines the path to the view templates. Defaults to `text!views/`
`viewSuffix`	Defines the suffix for view templates. Defaults to `.html`.
`modelPath`	Defines the location of the viewModel scripts. Defaults to `viewModels/`

   The Oracle JET SPA-ojModule-ojRouter sample application uses the default locations and paths for its view templates and viewModels. If you want to modify the paths, you can add the following code to your main application script, substituting the paths appropriate for your application.

   ```
   oj.ModuleBinding.defaults.modelPath = 'viewModels/';
   oj.ModuleBinding.defaults.viewPath = 'text!views/';
   oj.ModuleBinding.defaults.viewSuffix = '.html';
   ```

2. In your RequireJS bootstrap file (typically `main.js`) add `ojs/ojmodule` to the list of RequireJS modules, along with any other modules that your application uses.

   ```
   require(['ojs/ojcore', 'knockout', 'jquery',
   'ojs/ojknockout', 'ojs/ojbutton', 'ojs/ojmenu', 'ojs/ojtoolbar', 'ojs/
   ojnavigationlist',
   'ojs/ojoffcanvas', 'ojs/ojarraytabledatasource', 'ojs/ojmodule', 'ojs/ojrouter',
   'text', 'ojs/ojcheckboxset', 'ojs/ojswitch'],
   ```

3. If needed, create a `views` folder in the default location or in the `viewPath` if you defined one in a previous step.

4. Create your view templates and add them to the `viewPath` folder.

   Use the default `viewSuffix` or the value you set for the `viewSuffix` in Step 1.

5. If needed, create a folder for your viewModels in the default location or in the `modelPath` you specified in Step 1.

6. As needed, create your viewModel scripts and add them to the `modelPath` folder.

Use the same name for the module that you used for the template in Step 4. For example, if your template is named `customers.html`, use `customers.js` for the viewModel's name.

7. Add code to the application's HTML page to reference the view template or viewModel in the `ojModule` binding.

   If the page section contains only a view template, use `ojModule` with the `viewName` option:

   ```
   <header id="headerWrapper" role="banner" data-bind="ojModule: { viewName:
   'sampleView'}"></div>
   ```

   To reference a viewModel with a view template, use `ojModule` with the `name` option:

   ```
   <header id="headerWrapper" role="banner" data-bind="ojModule: { name:
   'sampleViewModel'}"></div>
   ```

   If you're using `oj.Router`, use `ojModule` with the `router.moduleConfig` option.

   ```
   <div id="content" role="main" class="oj-panel" style="padding-bottom:30px" data-
   bind="ojModule: router.moduleConfig"></div>
   ```

For more information about `oj.Router` and `ojModule`, see the Oracle JET oj.Router and ojModule API documentation.

> **Tip:**
>
> `ojModule` is not specific to single-page applications, and you can also use it to reuse content in multi-page applications. However, if you plan to reuse or share your content across multiple applications, consider creating Oracle JET composite components instead of using `ojModule`. Oracle JET composite components also contain reusable pieces of user interface code, and they offer the following additional features to facilitate sharing and reuse.
>
> - Composite components, also known as CCAs, have a contract. The API for a CCA is well defined by its `component.json`, which describes its supported properties, methods, and events in a standard, universal, and self-documenting way. Providing a standardized contract makes it easier for external tools or other applications to consume composite components.
> - CCAs include version and dependency metadata, making it clear which versions of Oracle JET they support and what other components they may require for operation.
> - CCAs are self-contained. A CCA definition can contain all the libraries, styles, images, and translations that it needs to work.
>
> To learn more about composite component features, see Working with Oracle JET Composite Components.

Working with ojModule's ViewModel Life Cycle

The `ojModule` binding provides life cycle listeners that allow you to specify actions to take place at defined places in the ViewModel's life cycle.

For example, you can specify actions to take place when the ViewModel is about to be used for the View transition, after its associated View is inserted into the document DOM, and after its View and ViewModel are inactive.

The following table lists the available methods with a description of their usage.

Method Name	Description
`handleActivated()`	Use to perform tasks when the View is about to use the ViewModel. This method is typically used to fetch data before the View transitions. For an example, see the Oracle JET Cookbook at: Dynamic Data Fetch.
`handleAttached()`	Use to perform tasks after the View is inserted into the DOM.
`handleBindingsApplied()`	Use to perform tasks after the bindings are applied on the View. If the current View is retrieved from cache, the bindings will not be re-applied, and this callback will not be invoked.
`handleDeactivated()`	Use to perform tasks after the View and ViewModel become inactive.
`handleDetached()`	Use to perform tasks after the View is removed from the DOM.
`handleTransitionCompleted()`	Use to perform tasks after a new View is complete, including any possible animation between the old and new View.
`dispose()`	Use to perform cleanup tasks before all binding references to the ViewModel are released. Note that this method will be invoked only once.
`initialize()`	Use to perform tasks when the ViewModel is created. This method will only be called if the ViewModel factory or the AMD module returns a ViewModel instance instead of a constructor function. This method will be invoked only once when the ViewModel is created and will not be called when the View is retrieved from cache.

You can also find stub methods for using the `ojModule` life cycle methods in some of the Oracle JET templates. For example, the navbar template, available as a template when you Scaffold a Hybrid Mobile Application with the Oracle JET CLI, defines stub methods for `handleActivated()`, `handleAttached()`, `handleBindingsApplied()`, and `handleDetached()`. Comments describe the expected parameters and use cases.

```
function DashboardViewModel() {
  var self = this;
  // Below are a subset of the ViewModel methods invoked by the ojModule binding
  // Please reference the ojModule jsDoc for additional available methods.

  /**
   * Optional ViewModel method invoked when this ViewModel is about to be
   * used for the View transition.  The application can put data fetch logic
   * here that can return a Promise which will delay the handleAttached function
   * call below until the Promise is resolved.
   * @param {Object} info - An object with the following key-value pairs:
   * @param {Node} info.element - DOM element or where the binding is attached. This
may be a 'virtual' element (comment node).
   * @param {Function} info.valueAccessor - The binding's value accessor.
   * @return {Promise|undefined} - If the callback returns a Promise, the next phase
(attaching DOM) will be delayed until
   * the promise is resolved
   */
  self.handleActivated = function(info) {
    // Implement if needed
  };
```

```
  /**
   * Optional ViewModel method invoked after the View is inserted into the
   * document DOM.  The application can put logic that requires the DOM being
   * attached here.
   * @param {Object} info - An object with the following key-value pairs:
   * @param {Node} info.element - DOM element or where the binding is attached. This
may be a 'virtual' element (comment node).
   * @param {Function} info.valueAccessor - The binding's value accessor.
   * @param {boolean} info.fromCache - A boolean indicating whether the module was
retrieved from cache.
   */
  self.handleAttached = function(info) {
    // Implement if needed
  };

  /**
   * Optional ViewModel method invoked after the bindings are applied on this View.
   * If the current View is retrieved from cache, the bindings will not be re-applied
   * and this callback will not be invoked.
   * @param {Object} info - An object with the following key-value pairs:
   * @param {Node} info.element - DOM element or where the binding is attached. This
may be a 'virtual' element (comment node).
   * @param {Function} info.valueAccessor - The binding's value accessor.
   */
  self.handleBindingsApplied = function(info) {
    // Implement if needed
  };

  /*
   * Optional ViewModel method invoked after the View is removed from the
   * document DOM.
   * @param {Object} info - An object with the following key-value pairs:
   * @param {Node} info.element - DOM element or where the binding is attached. This
may be a 'virtual' element (comment node).
   * @param {Function} info.valueAccessor - The binding's value accessor.
   * @param {Array} info.cachedNodes - An Array containing cached nodes for the View
if the cache is enabled.
   */
  self.handleDetached = function(info) {
    // Implement if needed
  };
}
```

For more information about `ojModule`, see ojModule.

6

Understanding Oracle JET User Interface Basics

Oracle JET User Interface (UI) components extend the `htmlElement` prototype to implement the World Wide Web Consortium (W3C) web component specification for custom elements. Custom elements provide a more declarative way of working with Oracle JET components and allow you to access properties and methods directly on the DOM layer.

Topics:

- Typical Workflow for Working with the Oracle JET User Interface
- About the Oracle JET User Interface
- Adding an Oracle JET Custom Element to Your Page

Typical Workflow for Working with the Oracle JET User Interface

Identify Oracle JET user interface components, understand their common functionality, and identify reserved words. You should also understand the steps required to add a component to your HTML5 page.

To work with the Oracle JET user interface, refer to the typical workflow described in the following table.

Task	Description	More Information
Understand the Oracle JET UI toolkit	Identify Oracle JET UI components, understand their common functionality, and identify Oracle JET reserved words.	About the Oracle JET User Interface
Add an Oracle JET component to your page	Identify the steps you must take to add an Oracle JET component to your HTML5 page. Optionally, add animation effects to your application.	Adding an Oracle JET Custom Element to Your Page

About the Oracle JET User Interface

Oracle JET includes components, patterns, and utilities to use in your application. The toolkit also includes an API specification (if applicable) and one or more code examples in the Oracle JET Cookbook.

Topics:

- Identifying Oracle JET UI Components, Patterns, and Utilities
- Common Functionality in Oracle JET Components

- Oracle JET Reserved Namespaces and Prefixes

Identifying Oracle JET UI Components, Patterns, and Utilities

The Oracle JET Cookbook lists all the components, design patterns, and utilities available for your use. By default, the cookbook is organized into functional sections, but you can also click **Sort** to arrange the contents alphabetically.

The Cookbook contains samples that you can edit online and see the effects of your changes immediately. You'll also find links to the API documentation, if applicable.

Common Functionality in Oracle JET Components

All Oracle JET components are implemented as custom HTML elements, and programmatic access to these components is similar to interacting with any HTML element.

Custom Element Structure

Oracle JET custom element names start with `oj-`, and you can add them to your page the same way you would add any other HTML element. In the following example, the `oj-label` and `oj-input-date-time` custom elements are added as child elements to a standard HTML `div` element.

```
<div id="div1">
  <oj-label for="dateTime">Default</oj-label>
  <oj-input-date-time id="dateTime" value='{{value}}'>
  </oj-input-date-time>
</div>
```

Each custom element can contain one or more of the following:

- Attributes: Modifiers that affect the functionality of the element.

  String literals will be parsed and coerced to the property type. Oracle JET supports the following string literal type coercions: boolean, number, string, Object, Array, and any. The any type, if used by an element, is marked with an asterisk (*) in the element's API documentation and coerced to Objects, Arrays, or strings.

  In the `oj-input-date-time` element defined above, `value` is an attribute that contains a Date object. It is defined using a binding expression that indicates whether the element's ViewModel should be updated if the attribute's value is updated.

  ```
  <oj-input-date-time id="dateTime" value='{{value}}'>
  ```

  The `{{...}}` syntax indicates that the element's `value` property will be updated in the element's ViewModel if it's changed. To prevent the attribute's value from updating the corresponding ViewModel, use the `[[...]]` syntax.

- Methods: Supported methods

  Each custom element's supported method is documented in its API.

- Events: Supported events

  Events specific to the custom element are documented in its API. Define the listener's method in the element's ViewModel.

```
var listener = function( event )
{
  // Check if this is the end of "inline-open" animation for inline message
  if (event.detail.action == "inline-open") {
    // Add any processing here
  }
};
```

Reference the listener using the custom element's DOM attribute, JavaScript property, or the `addEventListener()`.

- Use the DOM attribute.

  Declare a listener on the custom element using `on-event-name` syntax.

  ```
  <oj-input-date-time on-oj-animate-start='[[listener]]'</oj-input-date-time>
  ```

  Note that in this example the listener is declared using the `[[...]]` syntax since its value is not expected to change.

- Use the JavaScript property.

  Specify a listener in your ViewModel for the `.onEventName` property.

  ```
  myInputDateTime.onOjAnimateEnd = listener
  ```

  Note that the JavaScript property uses camelCase for the `onOjAnimateEnd` property. CamelCased properties are mapped to attribute names by inserting a dash before the uppercase letter and converting that letter to lower case, for example, `on-oj-animate-end`.

- Use the `addEventListener()` API.

  ```
  myInputDateTime.addEventListener('ojAnimateEnd', listener);
  ```

By default, JET components will also fire `propertyChanged` custom events whenever a property is updated, for example, `valueChanged`. You can define and add a listener using any of the three methods above. When referencing a `propertyChanged` event declaratively, use `on-property-changed` syntax.

```
<oj-input-date-time value="{{currentValue}}" on-value-
changed="{{valueChangedListener}}" </oj-input-date-time>
```

- Slots

  Oracle JET elements can have two types of child content that determine the content's placement within the element.

  - Any child element with a supported slot attribute will be moved into that named slot. All supported named slots are documented in the element's API. Child elements with unsupported named slots will be removed from the DOM.

    ```
    <oj-table>
      <div slot='bottom'<oj-paging-control></oj-paging-control></div>
    </oj-table>
    ```

  - Any child element lacking a slot attribute will be moved to the default slot, also known as a regular child.

A custom element will be recognized only after its module is loaded by the application. Once the element is recognized, Oracle JET will register a busy state for the element and will begin the process of upgrading the element from a normal element to a custom element. The element will not be ready for interaction until the upgrade process is complete. The application should listen to either the page-level or element-

scoped `oj.BusyContext` before attempting to interact with any JET custom elements. See the oj.BusyContext API documentation on how BusyContexts can be scoped.

The upgrade of custom elements relies on a binding provider which manages the data binding. The binding provider is responsible for setting and updating attribute expressions. Any custom elements within its managed subtree will not finish upgrading until the provider applies bindings on that subtree. By default, there is a single binding provider for a page, but subtree specific binding providers can be added by using the `data-oj-binding-provider` attribute with values of `none` and `knockout`. The default binding provider is `knockout`, but if a page or DOM subtree does not use any expression syntax or knockout, the application can set `data-oj-binding-provider="none"` on that element so that its dependent JET custom elements do not wait for bindings to be applied to finish upgrading.

Other Common Functionality

Oracle JET custom elements also have the following functionality in common:

- Context menus

  Custom elements support the `slot` attribute to add context menus to Oracle JET custom elements, described in each element's API documentation.

  ```
  <oj-some-element>
    <-- use the contextMenu slot to designate this as the context menu for this
  component -->
    <oj-menu slot="contextMenu" style="display:none" aria-label="Some element's
  context menu"
      ...
    </oj-menu>
  </oj-some-element>
  ```

- Keyboard navigation and other accessibility features

  Oracle JET components that support keyboard navigation list the end user information in their API documentation. For additional information about Oracle JET components and accessibility, see Developing Accessible Applications.

- Drag and drop

  Oracle JET includes support for standard HTML5 drag and drop and provides the `dnd-polyfill` library to extend HTML5 drag and drop behavior to supported mobile and desktop browsers. In addition, some Oracle JET custom elements such as `oj-table` support drag and drop behavior through the `dnd` attribute. For specific details, see the component's API documentation and cookbook examples. To learn more about HTML5 drag and drop, see http://www.w3schools.com/html/html5_draganddrop.asp.

- Deferred rendering

  Many Oracle JET custom elements support the ability to defer rendering until the content shown using `oj-defer`. To use `oj-defer`, wrap it around the custom element.

  ```
  <oj-collapsible id="defer">
    <h4 id="hd" slot="header">Deferred Content</h4>
      <oj-defer>
        <div data-bind="ojModule: 'deferredRendering/content'"></div>
      </oj-defer>
  </oj-collapsible>
  ```

Add the deferred content to the application's view and ViewModel, `content.html` and `content.js`, as specified in the data-bind definition. For the complete code example, see Collapsibles - Deferred Rendering.

For a list of custom elements that support `oj-defer`, see oj-defer.

Custom Element Examples and References

The Oracle JET Cookbook and JavaScript API Reference for Oracle® JavaScript Extension Toolkit (JET) provide examples that illustrate how to work with custom elements. In addition, the Cookbook provides demos with editing capability that allow you to modify the sample code directly and view the results without having to download the sample.

To learn more about the World Wide Web Consortium (W3C) web component specification for custom elements, see Custom Elements.

Oracle JET Reserved Namespaces and Prefixes

Oracle JET reserves the `oj` namespace and prefixes. This includes, but is not limited to component names, namespaces, pseudo-selectors, public event prefixes, CSS styles, Knockout binding keys, and so on.

Adding an Oracle JET Custom Element to Your Page

Use the Oracle JET Cookbook recipes and API documentation to locate examples that illustrate the specific element and functionality you want to add to your page.

To add an Oracle JET custom element to your page:

If needed, create the application that will house your main HTML5 page and supporting JavaScript. For additional information, see Getting Started with Oracle JET Application Development.

1. Using the Oracle JET Cookbook, select the Oracle JET element that you want to add.

   Each cookbook demo will usually contain several examples illustrating common use cases. Look for an example that comes closest to your use case.

2. Follow the example's recipe and add the markup, modifying as needed, to your HTML page.

   All Oracle JET recipes include the basic format for initializing the component using custom element syntax. The code below shows a portion of the markup for the `oj-input-date-time` example shown in the Date and Time Pickers demos.

```
<div id="div1">
  <oj-label for="dateTime">Default</oj-label>
  <oj-input-date-time id="dateTime" value='{{value}}'>
  </oj-input-date-time>

  <br/><br/>

  <span class="oj-label">Current component value is:</span>
  <span data-bind="text: value"></span>
</div>
```

In this example, the `oj-input-date-time` element is declared with its `value` attribute using `{{...}}` expression syntax, which indicates that changes to the value will also update the corresponding value in the ViewModel. Each Oracle JET custom element includes additional attributes that are defined in the custom element's API documentation.

3. Use the Oracle JET Cookbook for example scripts and the syntax to use for adding the custom element's Require module and ViewModel to your RequireJS bootstrap file or module.

   For example, the basic demo for `oj-input-date-time` includes the following script that you can use in your application.

```
require(['ojs/ojcore', 'knockout', 'jquery', 'ojs/ojknockout', 'ojs/
ojdatetimepicker'],
function (oj, ko, $)
{

  function SimpleModel()
  {
    this.value = ko.observable(oj.IntlConverterUtils.dateToLocalIso(new
Date(2013, 0, 1)));
  }

  $(document).ready(function ()
  {
    ko.applyBindings(new SimpleModel(), document.getElementById('div1'));
  });

});
```

> **Important:**
>
> If you're working with a Starter Template or define modules for different sections of your page, you can't simply copy and paste the cookbook example. The Cookbook uses a `require()` call to load and use the needed libraries in a single bootstrap file. The starter file that you are pasting into is a RequireJS module which uses `define()` instead to create a module that can be used by other parts of your application. For additional details, see Modifying the Web Nav Drawer Starter Template Content.

If you already have a RequireJS bootstrap file or module, compare your file with the Cookbook sample and merge in the differences. For details about working with RequireJS, see Using RequireJS for Modular Development.

Adding Animation Effects

You can use the `ojModule` binding's `animation` property in conjunction with the `ModuleAnimations` namespace to configure animation effects when the user transitions between or drills into views. If you're not using `ojModule`, you can use the

`oj.AnimationUtils` namespace instead to add animation to Oracle JET components or HTML elements.

Adding Animation Effects Using the ojModule Binding

The oj.ModuleAnimations namespace includes pre-configured implementations that you can use to configure the following animation effects:

- `coverStart`: The new view slides in to cover the old view.
- `coverUp`: The new view slides up to cover the old view.
- `drillIn`: Animation effect is platform-dependent.
  - Web and iOS: `coverStart`
  - Android: `coverUp`
  - Windows: `zoomIn`
- `drillOut`: Animation effect is platform-dependent.
  - Web and iOS: `revealEnd`
  - Android: `revealDown`
  - Windows: `zoomOut`
- `fade`: The new view fades in and the old view fades out.
- `goLeft`: Navigate to sibling view on the left. Default effect is platform-dependent.
  - Web and iOS: none
  - Android and Windows: `pushRight`
- `goRight`: Navigate to sibling view on the right. Default effect is platform-dependent.
  - Web and iOS: none
  - Android and Windows: `pushLeft`
- `pushLeft`: The new view pushes the old view out to the left.
- `pushRight`: The new view pushes the old view out to the right.
- `revealDown`: The old view slides down to reveal the new view.
- `revealEnd`: The new view slides left or right to reveal the new view, depending on the locale.
- `zoomIn`: The new view zooms in.
- `zoomOut`: The old view zooms out.

For examples that illustrate how to add animation with the `ojModule` binding, see Animation Effects with ojModule Binding.

Adding Animation Effects Using oj.AnimationUtils

The oj.AnimationUtils namespace includes methods that you can use to configure the following animation effects on HTML elements and Oracle JET components:

- `collapse`: Use for collapsing the element
- `expand`: Use for expanding the element
- `fadeIn` and `fadeOut`: Use for fading the element into and out of view.

- `flipIn` and `flipOut`: Use for rotating the element in and out of view.
- `ripple`: Use for rippling the element.
- `slideIn` and `slideOut`: Use for sliding the element into and out of view.
- `zoomIn` and `zoomOut`: Using for zooming the element into and out of view.

Depending on the method's options, you can configure properties like `delay`, `duration`, and `direction`. For examples that illustrate how to configure animation using the `oj.AnimationUtils` namespace, see Animation Effects.

7

Working with Oracle JET User Interface Components

Oracle JET provides a variety of user interface (UI) components that you can configure for use in your application. The Oracle JET Cookbook includes examples for working with collections, controls, forms, visualizations, and other features. You may also find this collection of tips and tricks helpful.

Topics

- Typical Workflow for Working with Oracle JET UI Components
- Working with Collections
- Working with Controls
- Working with Forms
- Working with Layout and Navigation
- Working with Visualizations

Typical Workflow for Working with Oracle JET UI Components

Identify tips and tricks for working with Oracle JET user interface components.

To work with the Oracle JET user interface components, refer to the typical workflow described in the following table. Use the task descriptions to locate the component that you want to add to your application from the Description column.

Task	Description	More Information
Work with collections	Identify tips and tricks for working with data grids, list views, pagination, row expanders, tables, and trees.	Working with Collections
Work with controls	Identify tips and tricks for working with buttons, button sets, conveyor belts, film strips, menus, progress indicators, HTML tags, toolbars, and trains.	Working with Controls
Work with forms	Identify tips and tricks for working with form elements, including checkbox and radio sets, comboboxes, form controls, form layout, input, labels, selection, validation and user assistance components.	Working with Forms
Work with layout and navigation components, elements, and patterns	Identify tips and tricks for working with accordions, collapsibles, dialogs, drawer utilities, panels, popups, and tabs.	Working with Layout and Navigation

Task	Description	More Information
Work with visualizations	Identify tips and tricks for working with data visualizations, including charts, diagrams, gauges, NBoxes, sunbursts, tag clouds, thematic maps, timelines, and treemaps.	Working with Visualizations

Working with Collections

Use Oracle JET data collection components to display data in tables, data grids, list views, or trees.

The Oracle JET data collection components include `oj-table`, `oj-data-grid`, `oj-tree`, and `oj-list-view`, and you can use them to display records of data. `oj-table` and `oj-data-grid` both display data in rows and columns, and your choice of component depends upon your use case. Use `oj-tree-view` to display hierarchical data such as a directory structure and `oj-list-view` to display a list of data. The toolkit also includes pagination and row expanders that display hierarchical data in a data grid or table row.

The Oracle JET Cookbook and JavaScript API Reference for Oracle® JavaScript Extension Toolkit (JET) include complete demos and examples for using the collection components, and you may also find the following tips and tricks helpful.

Topics:

- Choosing a Table or Data Grid
- Working with Data Grids
- Working with List Views
- Working with Pagination
- Working with Row Expanders
- Working with Tables
- Working with Tree Views

Choosing a Table or Data Grid

Oracle JET provides both the `oj-table` and `oj-data-grid` components for displaying records of data in rows and columns, and this section can help you decide which component to use in your application.

The `oj-table` component displays records of data on a row basis. It's best used when you have simple data that can be presented as rows of fields, and it should be your first choice unless you require advanced features. A selection in the table provides you with the row of data and all of the fields in that row or record. The sizing of the table is based on the content itself. The height and width of the cells is adjusted for the content included. `oj-table` is based on the HTML `oj-table` element. You can write templates using `oj-table` elements such as `tr`, `td`, `th`, and so on.

The `oj-data-grid` is designed to provide grid functionality. It provides the ability to select individual or ranges of cells of data. It's best used when you need to display totals or tallies of data across columns or rows of data. The `oj-data-grid` is designed to be much more flexible in its layout and functionality than the `oj-table` component.

It's a low-level component that you can shape in your application to how you want the data to be displayed. The overall size of the data grid is not determined by its content, and the developer specifies the exact height and width of the container. The data grid acts as a viewport to the contents, and its size doesn't affect the size of the columns and rows like a table does. `oj-data-grid` is based on the HTML `oj-data-grid` element, and you can host the template content inside it.

The table below provides a feature comparison of the `oj-table` and `oj-data-grid` components.

Feature	oj-table	oj-data-grid
Column/Row sizing	Controlled by content or CSS styles. Percent values supported for width and height.	Controlled by cell dimensions. Does not support percent values for width and height.
User-resizable column/row	No	Yes
Row reordering	No	Yes
Column sorting	Yes	Yes
Column/Row selection	Yes	Yes
Cell selection	No	Yes
Marquee selection	No	Yes
Row header support	No	Yes
Pagination	Page, high water mark	Page, high water mark, virtual scrolling (see note)
Custom cell templates	Yes	Yes
Custom row templates	Yes	No
Custom cell renderers	Yes	Yes
Custom row renderers	Yes	No
Row expander support	Yes	Yes
Cell stamping	Yes	Yes
Cell merging	Yes	Yes
Render aggregated cubic data	No	Yes
Base HTML element	`oj-table`	`oj-data-grid`
Custom footer template	Yes (provides access to column data for passing to a JavaScript function)	No (cell level renderers used for column and row data manipulations)
Cell content editing	Yes	Yes
Content filtering	Yes	Yes

> **Note:**
>
> True virtual scrolling is available as a feature of `oj-data-grid`. Modern design principles should be considered and implemented before implementing virtual scrolling. It is much more desirable to present the end user with a filtered list of data that will be more useful to them, than to display thousands of rows of data and expect them to scroll through it all. True virtual scrolling means that you can perform scrolling in both horizontal and vertical directions with data being added and removed from the underlying DOM. High water mark scrolling (with lazy loading) is the preferred method of scrolling, and you should use it as a first approach.

Working with Data Grids

The `oj-data-grid` component displays data in cells inside a grid consisting of rows and columns. `oj-data-grid` is themable and supports WAI-ARIA authoring practices for accessibility. You can configure the `oj-data-grid` for cell selection with row and column headers or for row selection with column headers.

Cell Based

	0	1	2
0	1	2	3
1	4	5	6
2	7	8	9

Row Based

	First Name	Last Name	Email Address	Phone #	Date Hired	Salary
100	Steven	King	SKING	515.123.4567	Jun 17, 1987	$24,000.00
101	Neena	Kochhar	NKOCHHAR	515.123.4568	Sep 21, 1989	$17,000.00
102	Lex	De Haan	LDEHAAN	515.123.4569	Jan 13, 1993	$17,000.00
103	Alexander	Hunold	AHUNOLD	590.423.4567	Jan 3, 1990	$9,000.00
104	Bruce	Ernst	BERNST	590.423.4568	May 21, 1991	$6,000.00
105	David	Austin	DAUSTIN	590.423.4569	Jun 25, 1997	$4,800.00
106	Valli	Pataballa	VPATABAL	590.423.4560	Feb 5, 1998	$4,800.00
107	Diana	Lorentz	DLORENTZ	590.423.5567	Feb 7, 1999	$4,200.00
108	Nancy	Greenberg	NGREENBE	515.124.4569	Aug 17, 1994	$12,000.00
109	Daniel	Faviet	DFAVIET	515.124.4169	Aug 16, 1994	$9,000.00
110	John	Chen	JCHEN	515.124.4269	Sep 28, 1997	$8,200.00
111	Ismael	Sciarra	ISCIARRA	515.124.4369	Sep 30, 1997	$7,700.00
112	Jose Manuel	Urman	JMURMAN	515.124.4469	Mar 7, 1998	$7,800.00

> **Note:**
>
> When you configure the data grid for row selection, the grid has a look and feel that is similar to the `oj-table` component. However, there are distinct differences in functionality between the `oj-data-grid` and `oj-table` components. For additional information, see Choosing a Table or Data Grid.

To create the data grid, define an `oj-data-grid` element in your HTML file and assign a meaningful id, width, and height. Also, specify the data property.

```
<oj-data-grid id="datagrid"
    style="width:100%;max-width:234px;height:130px"
```

```
    aria-label="Data Grid Cell Based Grid Demo"
    data="[[dataSource]]">
</oj-data-grid>
```

In this example, the `aria-label` is added to the `oj-data-grid` element for accessibility. `oj-data-grid` implicitly defines an Aria role as `application`, and you must add the `aria-label` to distinguish it from other elements defined with the Aria `application` role. For additional information about Oracle JET and accessibility, see Developing Accessible Applications.

The data is defined in the `dataSource` object and can be one of the following:

- `oj.ArrayDataGridDataSource`: Use to define data in a static array.

  The array can be a single array where each item in the array represents a row in the data grid, or a two dimensional array where each item represents a cell in the grid. For the data grid shown in this section, the data is defined as a two dimensional array.

  ```
  require(['ojs/ojcore', 'knockout', 'jquery', 'ojs/ojknockout', 'promise',
  'ojs/ojdatagrid', 'ojs/ojarraydatagriddatasource'],
  function(oj, ko, $)
  {
      function viewModel()
      {
          var self = this;
          var dataArray = [
              ['1', '2', '3'],
              ['4', '5', '6'],
              ['7', '8', '9']
          ];
          self.dataSource = new oj.ArrayDataGridDataSource(dataArray);
      }

      $(
          function()
          {
              ko.applyBindings(new viewModel(),
  document.getElementById('datagrid'));
          }
      );
  });
  ```

  `oj.ArrayDataGridDataSource` also supports custom sort behavior through its `comparator` property. For details, consult the oj.ArrayDataGridDataSource API documentation.

- `oj.CollectionDataGridDataSource`: Use to define data using the Oracle JET Common Model. The data grid will respond to events generated by the underlying `oj.Collection`.

  For more details about `oj.CollectionDataGridDataSource`, see the oj.CollectionDataGridDataSource API documentation.

- `oj.PagingDataGridDataSource`: Use to include pagination. For additional information, see Working with Pagination.

- `oj.FlattenedTreeDataGridDataSource`: Use to display hierarchical data in the data grid. The user can expand and collapse rows in the data grid. For additional information, see Working with Row Expanders.

- `oj.CubeDataGridDataSource`: Use to include aggregate values on column headers, row headers, or both. For additional information, see Working with oj. CubeDataGridDataSource.
- Custom data source: Use to provide your own data source for the data grid. The Oracle JET Cookbook contains examples for creating custom data sources, including an example with nested headers.

The Oracle JET Cookbook includes the complete example for the data grid used in this section at Data Grids. The cookbook also includes examples that show row-based data grids, editable data grids, and data grids with visualizations, end headers, custom cell renderers, merged cells, and custom data sources.

Working with oj. CubeDataGridDataSource

Use `oj.CubeDataGridDataSource` to render aggregated cubic data in your data grid. You can aggregate values on column headers, row headers, or both column and row headers. You can also define a page axis to filter or otherwise restrict the display.

For example, you may have a collection that contains data for sales, number of units sold, and sales tax data for vehicles sold by car dealerships, and you'd like to aggregate the data to show sales, unit, and tax data by year and city. Your data also contains the type of vehicle sold, its color, and type of drive train, and you'd also like to aggregate the data to show the sales, unit, and tax data grouped by product, color, and drive train.

You can configure `oj-data-grid` with the `oj.CubeDataGridDataSource` to achieve the desired grouping. The following image shows the runtime display.

			New York						Redwood Shores						Boston				
			2014			2015			2014			2015			2014			2015	
			units	sales	tax	units	sales	tax	units	sales	tax	units	sales	tax	units	sales	tax	units	sales
Coupe	White	FWD	107	889	6.11%	181	1453	5.56%	121	1326	6.39%	137	177	5.01%	81	349	7.03%	131	1726
		AWD	23	1116	2.38%	54	1099	7.37%	31	897	4.87%	120	149	4.35%	69	1712	5.86%	49	640
	Black	FWD	93	647	5.36%	62	933	5.58%	55	833	6.54%	107	862	7.37%	169	869	5.58%	112	1062
		AWD	59	1735	5.76%	124	1337	3.40%	83	1266	2.36%	29	1103	3.45%	148	1499	3.97%	59	808
	Red	FWD	65	363	6.58%	120	1157	1.73%	61	1269	4.34%	74	1326	1.38%	127	826	4.26%	84	1010
		AWD	160	596	4.88%	74	544	4.58%	108	660	7.07%	38	577	4.01%	187	1144	6.17%	127	547
Sedan	White	FWD	99	456	5.97%	3	955	4.87%	135	1775	4.89%	94	1094	3.99%	55	734	4.27%	103	1490
		AWD	45	754	8.55%	81	675	2.02%	83	796	7.18%	184	1012	6.00%	166	779	4.61%	31	821
	Black	FWD	74	676	8.26%	12	64	6.39%	135	1010	0.97%	115	104	6.46%	105	1797	6.19%	25	930
		AWD	98	980	8.44%	54	1258	2.15%	165	471	3.93%	134	497	8.37%	87	677	8.30%	95	179
	Red	FWD	69	884	6.19%	62	1599	7.74%	61	1828	1.15%	45	1527	8.03%	67	376	5.01%	190	1505
		AWD	45	1749	5.28%	121	268	6.34%	111	914	4.73%	67	540	2.71%	105	578	6.55%	123	255

The data grid uses JSON data for its data source. The code sample below shows a portion of the JSON array.

```
[
  {
    "index": 0,
    "units": 80,
```

```
    "sales": 535,
    "tax": 0.0234,
    "year": "2014",
    "gender": "Male",
    "product": "Coupe",
    "city": "New York",
    "drivetrain": "FWD",
    "color": "White"
  },
  {
    "index": 1,
    "units": 95,
    "sales": 610,
    "tax": 0.0721,
    "year": "2015",
    "gender": "Male",
    "product": "Coupe",
    "city": "New York",
    "drivetrain": "FWD",
    "color": "White"
  },
  {
    "index": 2,
    "units": 27,
    "sales": 354,
    "tax": 0.0988,
    "year": "2014",
    "gender": "Female",
    "product": "Coupe",
    "city": "New York",
    "drivetrain": "FWD",
    "color": "White"
  },

]
```

The data also contains a column for the gender of the buyer which isn't included in the display. The totals displayed in the grid come from applying the aggregation across any JSON rows that match up on type, color, drivetrain, year, and city. For the data in this example, this has the effect of grouping the Male and Female values and applying the aggregation. For example, the units shown in the grid for the New York sales in 2014 of white FWD coupes comes from totaling the highlighted values:

```
80 + 27 = 107 units
```

The following code sample shows the markup for the data grid.

```
<oj-data-grid id="datagrid"
    style="width:100%;height:400px;max-width:851px;"
    aria-label="Cubic Data Source Grid Demo"
    data="[[dataSource]]"
    cell.renderer="[[cellRenderer]]"
></oj-data-grid>
```

In this example, the `datasource` is defined in the application's main script.

```
require(['ojs/ojcore', 'knockout', 'jquery', 'ojs/ojknockout', 'ojs/ojmodel',
     'promise', 'ojs/ojcube', 'ojs/ojdatagrid'],
function(oj, ko, $)
{
  function dataGridModel() {
```

```
    var vm = this;
    var cube = null;
    var topLevelItems = [];

    var collection = new oj.Collection(null, {
        url: 'cookbook/dataCollections/dataGrid/cubeGrid/cubedata.json'
    });
    var dataArr = null;

    function generateCube(dataArr, axes) {
      return new oj.DataValueAttributeCube(dataArr, axes,
              [{attribute:'units',aggregation:oj.CubeAggType['SUM']},
                  {attribute:'sales'},
                  {attribute:'tax',aggregation:oj.CubeAggType['AVERAGE']}]});
    };

    collection.fetch({success:function() {
      dataArr = collection.map(function(model) {
        return model.attributes;
      });
      cube = generateCube(dataArr, axes);

      var topLevelItems = getItemsForLevel(2, 0);
      vm.colors(topLevelItems);

      vm.currentColor(topLevelItems[0].value);

      vm.dataSource(new oj.CubeDataGridDataSource(cube));
    }});

    var axes = [
        {axis: 0, levels: [
                {attribute: 'city'},
                {attribute: 'year'},
                {dataValue: true}]},
        {axis: 1, levels:[
                {attribute: 'product'}
                {attribute: 'color'},
                {attribute: 'drivetrain'}]}];

    this.dataSource = ko.observable();

     $(function() {
     ko.applyBindings(new dataGridModel(),
         document.getElementById('wrapper'));
  });
});
```

The `oj.CubeDataGridDataSource` parameter is instantiated with the return value of the `generateCube()` function, which is an `oj.Cube` object. The `oj.Cube` class provides functions for the `oj.DataValueAttributeCube` class which creates the object to convert the row set data into grouped, cubic data. The `oj.DataValueAttributeCube` constructor takes the following parameters:

- `rowset`: An array of objects containing name-value pairs

  In this example, the JSON data is mapped to the `dataArr` object which is defined as an `oj.Collection` object.

```
var collection = new oj.Collection(null, {
    url: 'cookbook/dataCollections/dataGrid/cubeGrid/cubedata.json'
});
```

- `layout`: An array of objects that contains the axis number and levels to use for the aggregation

  The axis number indicates where you want the aggregated data displayed: 0 for column headers, and 1 for row headers. The levels tell the cube which values to aggregate and the order to display them. The code sample below shows the layout used in this example. The `dataValue` property indicates which level to display for the units, sales, and tax aggregated values.

```
var axes = [
            {axis: 0, levels: [
                    {attribute: 'city'},
                    {attribute: 'year'},
                    {dataValue: true}]},
            {axis: 1, levels: [
                    {attribute: 'product'},
                    {attribute: 'color'},
                    {attribute: 'drivetrain'}]}];
```

  You can configure an additional axis that you can use to filter or otherwise restrict the display of data. For example, you can add a third axis which includes the color and drivetrain attributes, as shown in the following code sample:

```
var axes = [
            {axis: 0, levels: [
                    {attribute: 'city'},
                    {attribute: 'year'},
                    {dataValue: true}]},
            {axis: 1, levels: [
                    {attribute: 'product'}]},
            {axis: 2, levels:[
                    {attribute: 'color'},
                    {attribute: 'drivetrain'}]}];
```

  When you add a third axis to the page, the axis will not be visible. However, you can use the `setPage()` method to use the axis as a page axis to filter the display. For example, setting `color` and `drivetrain` on the third axis has the effect of restricting the display to aggregate only those values that match both the `color` and `drivetrain` attributes.

  In this example, you have six page combinations: White FWD, White AWD, Black FWD, Black AWD, Red FWD, and Red AWD products. If you set the `color` attribute to White and the `drivetrain` attribute to 4WD as shown in the following image, when you render the page only the values for White, 4WD products (Coupe, Sedan, Wagon, SUV, Van, and Truck) display on the screen.

color
White
drivetrain
FWD

	New York						Redwood Shores						Boston			
	2014			2015			2014			2015			2014			
	units	sales	tax	units	sales	tax	units	sales	tax	units	sales	tax	units	sales	tax	units
Coupe	107	889	6.11%	181	1453	5.56%	121	1326	6.39%	137	177	5.01%	81	349	7.03%	131
Sedan	99	456	5.97%	3	955	4.87%	135	1775	4.89%	94	1094	3.99%	55	734	4.27%	103
Wagon	82	593	6.82%	49	534	6.76%	77	1334	5.85%	155	1060	0.43%	33	791	7.27%	106
SUV	54	548	4.59%	69	1182	6.86%	107	1457	6.70%	102	1080	9.07%	117	622	5.38%	74
Van	110	614	5.35%	80	885	8.38%	13	1578	5.35%	148	713	2.12%	99	1513	8.47%	158
Truck	83	1420	6.74%	135	918	0.88%	99	1435	5.24%	136	782	3.72%	146	768	5.20%	78

- `dataValues`: An array of objects that contains the name of the attribute in the row set that represents the data, an optional label, and the aggregation type. The aggregation type is also optional and defaults to `SUM`. You can also set it to one of the aggregation types shown in the following table.

Aggregation Type	Description
`AVERAGE`	Average the values.
`COUNT`	Count the number of values.
`CUSTOM`	Specify a custom callback function to do the aggregation.
`FIRST`	Substitute the first value encountered in the collection.
`MAX`	Calculate the maximum of the values.
`MIN`	Calculate the minimum of the values.
`NONE`	Substitute a null for the value.
`STDDEV`	Calculate the standard deviation of the values.
`SUM`	Total the values.
`VARIANCE`	Calculate the variance of the values.

The code sample below shows the function that defines the `dataValues` for the cube used in this section. The cube will sum the units and sales data and will average the sales tax data.

```
function generateCube(dataArr, axes) {
        return new oj.DataValueAttributeCube(dataArr, axes,
                [{attribute:'units',aggregation:oj.CubeAggType['SUM']},
                    {attribute:'sales'},
                    {attribute:'tax',aggregation:oj.CubeAggType['AVERAGE']}]);
    };
```

The Oracle JET Cookbook demos at Data Grids (Cubic Data Source) contain the complete code for the example used in this section, including the complete code for setting the page axis and rendering the cell content. The oj.CubeDataGridDataSource API documentation also contains additional detail and methods that you can use for working with the cube.

Working with List Views

The Oracle JET `oj-list-view` component enhances the HTML list (`ul`) element to provide a themable, WAI-ARIA compliant component that displays a list of data. `oj-list-view` supports single and multiple selection, high water mark scrolling when working with table data, and hierarchical content.

Topics:

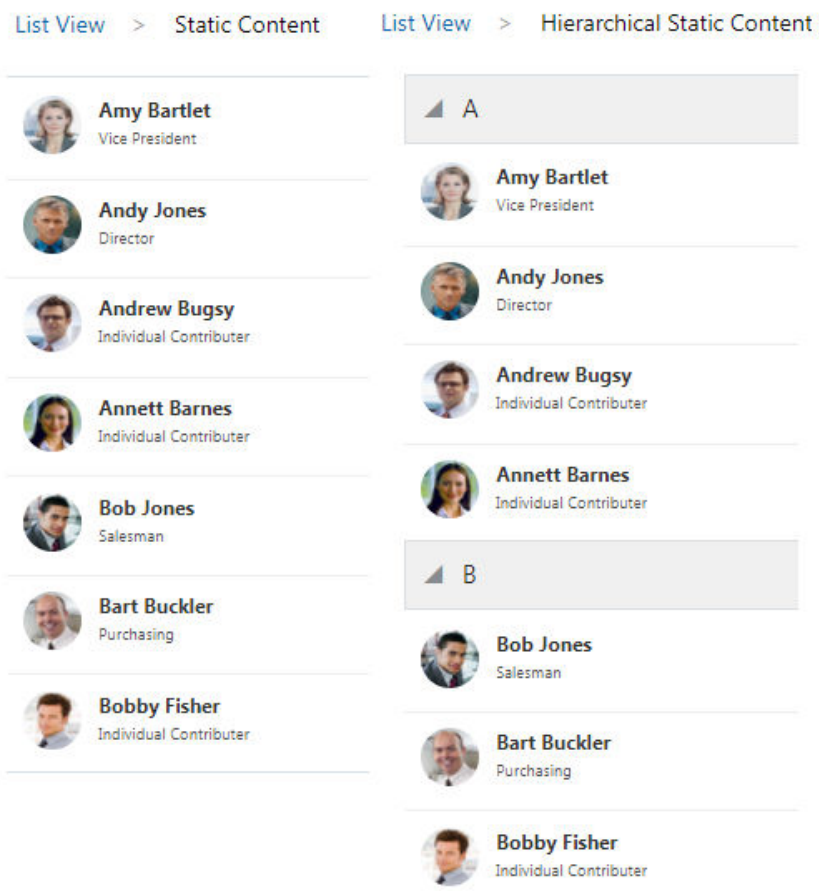
- Understanding Data Requirements for List Views
- Working with List Views and Knockout Templates

Understanding Data Requirements for List Views

The data source for the `oj-list-view` component can be one of the following:

- Flat or hierarchical static HTML

  The following image shows examples of list views that display static content, one with flat content and one using a nested list for hierarchical content.



The following code sample shows a portion of the markup used to create the list view with hierarchical static content using the `oj-list-view` element. The element allows multiple selection and expands both list item groups A and B upon initial display.

```
<oj-list-view id="listview" aria-label="list with hierarchical data" selection-
mode="multiple" item.selectable="[[itemOnly]]">
  <ul>
    <li id="a">
      <span>A</span>
        <ul>
          <li id="amybartlet">
            <span class="avatar" style="background-image: url('images/nBox/
1.png')"></span>
              <div class="container">
                <div>
                  <span class="name">Amy Bartlet</span>
                </div>
                <div>
                  <span class="oj-text-xs oj-text-secondary-color">Vice
President</span>
                </div>
              </div>
          </li>
        ... contents omitted
        </ul>
```

```
    </li>
    <li id="b">
      <span>B</span>
        <ul>
     ... contents omitted
        </ul>
    </li>
  </ul>
</oj-list-view>
```

The code to apply the binding is shown below.

```
require(['ojs/ojcore', 'knockout', 'jquery', 'ojs/ojknockout', 'promise', 'ojs/
ojlistview'],
function(oj, ko, $)
{
    function viewModel()
    {
        this.itemOnly = function(context)
        {
            return context['leaf'];
        };
    }

    $(
        function()
        {
            ko.applyBindings(new viewModel(),
document.getElementById('listview'));
        }
    );
});
```

See List View Using Static Data for the complete example to create `oj-list-view` components using static data and static hierarchical data.

- Oracle JET `TreeDataSource`, including `oj.JsonTreeDataSource` and `oj.CollectionTreeDataSource`.

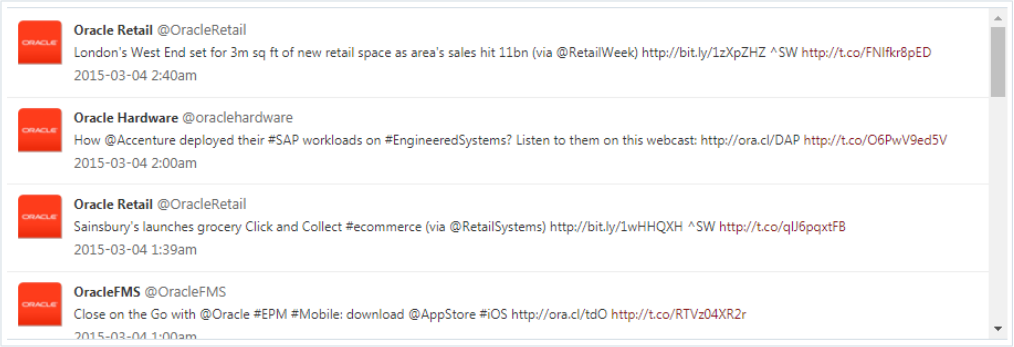
  To use a `TreeDataSource`, you specify the method that returns the tree data in the `data` attribute for the component.

```
<oj-list-view id="listview" aria-label="list using json data"
               data="[[dataSource]]" selection-mode="single"
item.renderer="[[renderer]]" item.focusable="[[itemOnly]]"
               item.selectable="[[itemOnly]]" drill-mode="none">
</oj-list-view>
```

  The Oracle JET Cookbook contains the complete example for creating a list view with an `oj.JsonTreeDataSource` at List View Using Hierarchical JSON Data. You can also find an example for creating a list view with an `oj.JsonTreeDataSource` that also contains an index at Indexer.

- Oracle JET `TableDataSource`, including `oj.ArrayTableDataSource`, `oj.CollectionTableDataSource`, and `oj.PagingTableDataSource`

  Use the `TableDataSource data` attribute when you have flat data or want to add scrolling or pagination to your `oj-list-view`. The following image shows a list view using an `oj.CollectionTableDataSource` object for its data.



In this example, high water mark scrolling is enabled through the component attribute's `scroll-policy`.

```
<oj-list-view id="listview" aria-label="list using collection" style="width:
100%;height:300px;overflow-x:hidden"
       data="[[dataSource]]"
       item.renderer="[[oj.KnockoutTemplateUtils.getRenderer('tweet_template')]]"
       selection-mode="single"
       scroll-policy="loadMoreOnScroll" scroll-policy-options.fetch-size="15">
</oj-list-view>
```

For the complete example, including the script that creates the `oj.CollectionTableDataSource` object, see List View Using oj.Collection.

You can also find cookbook examples that add Pull to Refresh and Swipe to Reveal touch capability to an `oj-list-view` created with the `oj.CollectionTableDataSource` object.

> **Note:**
>
> If you do not specify a data source in the list view component's `data` attribute, Oracle JET will examine the child elements inside the root element and use it for static content. If the root element has no children, Oracle JET will render an empty list.

Working with List Views and Knockout Templates

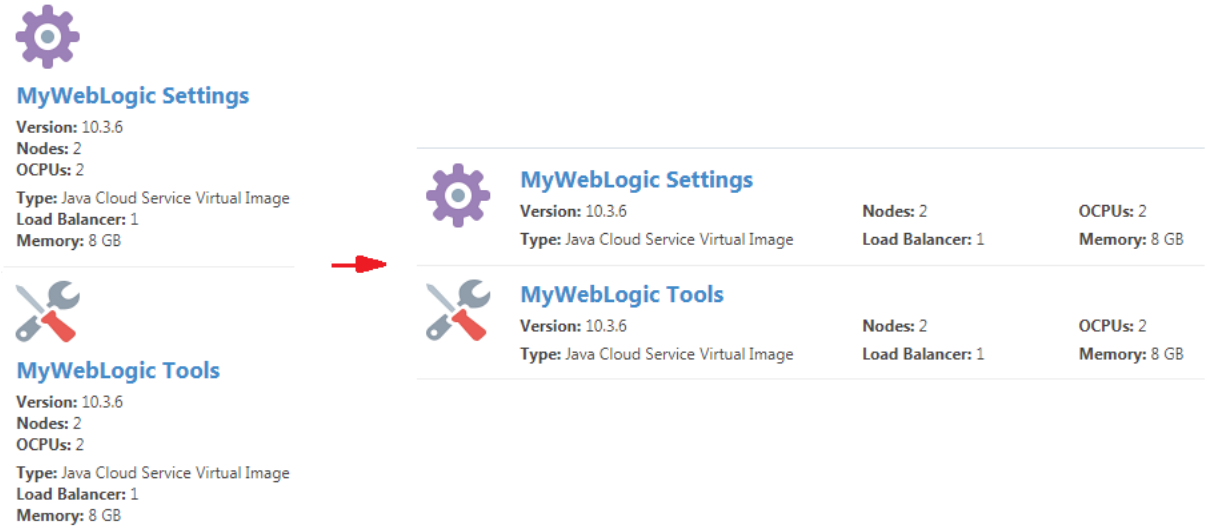
You can use a Knockout template to contain the markup for your list item content and reference the name of the template in the `oj-list-view` component's `item` attribute.

The code sample below shows a portion of the markup and template for a list view using an `oj.ArrayTableDataSource` object for its data. In this example, the template is named `server_template`.

```
<oj-list-view id="listview" aria-label="list using array"
                data="[[dataSource]]"
item.renderer="[[oj.KnockoutTemplateUtils.getRenderer('server_template', true)]]"
selection-mode="single">
</oj-list-view>
<script type="text/html" id="server_template">
    <li data-bind="attr: {id: $data['name']}">
```

```
        <div class="oj-flex">
            <div class="oj-lg-2 oj-md-2 oj-sm-12" style="max-width:70px;padding-
bottom:5px">
                <img data-bind="attr: {src: 'images/listView/'+ name +'.png', alt:
name}"></img>
            </div>
            <div class="oj-lg-10 oj-md-10 oj-sm-12">
                <div class="oj-flex row">
                    <div class="oj-lg-12">
                        <span class="header" data-bind="text: 'MyWebLogic '+name"></
span>
                    </div>
                </div>
                  ... contents omitted
            </div>
        </div>
    </li>
</script>
```

The template uses the Oracle JET responsive grid classes to change the display from one column for smaller displays to multiple columns for larger displays.



You can find the complete code sample for this list view at: List Views.

For additional information about the Oracle JET responsive grid, see Designing Responsive Applications. For additional information about working with Knockout templates, see Using ojModule.

Working with Pagination

Use the `oj-paging-control` component to add pagination to the `oj-table` and `oj-data-grid` components or the HTML list element. Pagination displays the number of pages and rows in the table or grid, and the user can use pagination to move between pages, jump to a specific page, or navigate to the first or last page of data.

In the following image, `oj-paging-control` element is added to the `oj-table` element and initialized with the default display.

Department Id	Department Name	Location Id	Manager Id
10015	ADFPM 1001 neverending	200	300
556	BB	200	300
10	Administration	200	300
20	Marketing	200	300
30	Purchasing	200	300
40	Human Resources1	200	300
50	Administration2	200	300
60	Marketing3	200	300
70	Purchasing4	200	300
80	Human Resources5	200	300

Page 1 of 3 (1-15 of 45 items) 1 2 3 Last

To add pagination to a table, define the table's data as a `PagingTableDataSource` object, and add the `oj-paging-control` using the same `PagingTableDataSource` object for its data attribute. Specify the number of rows to display in the `oj-paging-control` element's `page-size` attribute. The code sample below shows the markup defining the table and pagination components. In this example, `page-size` is set to 10.

```
<div id="pagingControlDemo">
    <oj-table id="table" summary="Department List" aria-label="Departments Table"
              data='[[pagingDatasource]]'
              columns='[{"headerText": "Department Id", "field": "DepartmentId"},
                        {"headerText": "Department Name", "field": "DepartmentName"},
                        {"headerText": "Location Id", "field": "LocationId"},
                        {"headerText": "Manager Id", "field": "ManagerId"}]'
              style='width: 100%;'>
        <oj-paging-control id="paging" data='[[pagingDatasource]]' page-size='10'
slot='bottom'>
        </oj-paging-control>
    </oj-table>
</div>
```

The script that populates the `pagingDatasource` with data and completes the Knockout binding is shown below. In this example, the table's data is defined in an `ArrayTableDataSource` object, and the `pagingDatasource` defines the `PagingTableDataSource` as a new `ArrayTableDataSource` object.

```
require(['ojs/ojcore', 'knockout', 'jquery', 'ojs/ojknockout', 'ojs/ojcheckboxset',
'ojs/ojtable', 'ojs/ojpagingcontrol',
 'ojs/ojpagingtabledatasource', 'ojs/ojarraytabledatasource'],
function(oj, ko, $)
{
  function viewModel()
  {
    var self = this;

    var deptArray = [{DepartmentId: 10015, DepartmentName: 'ADFPM 1001 neverending',
LocationId: 200, ManagerId: 300},
        {DepartmentId: 556, DepartmentName: 'BB', LocationId: 200, ManagerId: 300},
        {DepartmentId: 10, DepartmentName: 'Administration', LocationId: 200,
```

```
ManagerId: 300},
        {DepartmentId: 20, DepartmentName: 'Marketing', LocationId: 200, ManagerId:
300},
        ...contents omitted
        {DepartmentId: 13022, DepartmentName: 'Human Resources15', LocationId: 200,
ManagerId: 300}];
    self.pagingDatasource = new oj.PagingTableDataSource(new
oj.ArrayTableDataSource(deptArray, {idAttribute: 'DepartmentId'}));
  }

  var vm = new viewModel;

  $(document).ready
  (
    function()
    {
      ko.applyBindings(vm, document.getElementById('pagingControlDemo'));
    }
  );
});
```

To add a paging control to `oj-data-grid`, define the data grid's data as a `oj.PagingDataGridDataSource` object, and add the `oj-paging-control` element using the same `oj.PagingDataGridDataSource` object for its data attribute. Set the `page-size` attribute equal to the fetch size for the data collection, if creating the data grid from an `oj.CollectionDataGridDataSource` object.

The Oracle JET Cookbook contains complete examples for adding pagination to `oj-table`, `oj-data-grid`, and HTML lists at Pagination. You can also find the link to the oj-paging-control API documentation as well as examples that show different options for customizing the paging display.

For additional information about working with the `oj-table` component, see Working with Tables. For more information about working with the `oj-data-grid` component, see Working with Data Grids.

Working with Row Expanders

Use the Oracle JET `oj-row-expander` component to expand or collapse rows in a data grid or table to display hierarchical data. `oj-row-expander` renders the expand/collapse icon with the appropriate indentation and works directly with the flattened data source. In the following image, the row expander is used with an `oj-data-grid` component, and the user can expand the tasks to display subtasks and dates.

	Resource	Start Date	End Date
Task 1	Chadwick	1/1/2014	10/1/2014
Task 1-1	Chris	1/1/2014	3/1/2014
Task 1-2	Jim	3/1/2014	6/1/2014
Task 1-2-1	Jay	3/1/2014	5/1/2014
Task 1-2-2	Karin	5/1/2014	6/1/2014
Task 1-3	Chadwick	6/1/2014	8/1/2014
Task 1-4	Chris	8/1/2014	10/1/2014
Task 2	Henry	4/1/2014	12/1/2014
Task 3	Jay	5/1/2014	11/1/2014
Task 4	Victor	11/1/2014	12/1/2014

To use `oj-row-expander` with `oj-data-grid`, create an `oj-data-grid` element and assign a meaningful ID to it and specify properties on the `oj-data-grid`. In the HTML file, specify a row header template that adds the `oj-row-expander` element to the row header. Specify the grid's data in an `oj.FlattenedTreeDataGridDataSource` object. In the code sample below, the `oj-row-expander` is defined in the data grid's row template. The `context` option references the object obtained from the data grid's column renderer.

```
<oj-data-grid
    id="datagrid"
    style="width:100%;max-width:502px;height:400px"
    aria-label="Data Grid with Row Expander"
    data="[[dataSource]]"
    selection-mode.cell="single"

header.column.renderer="[[oj.KnockoutTemplateUtils.getRenderer('column_header_templat
e')]]"
    header.column.style="width:100px;"
    header.column.resizable.width="enable"

header.row.renderer="[[oj.KnockoutTemplateUtils.getRenderer('row_header_template')]]"
    header.row.style="width:200px;"
    cell.class-name="oj-helper-justify-content-flex-start"
></oj-data-grid>

<script type="text/html" id="column_header_template">
    <!-- ko if: $context.key=='resource' -->
        <span data-bind="text: 'Resource'"></span>
    <!-- /ko -->
        <!-- ko if: $context.key=='start' -->
        <span data-bind="text: 'Start Date'"></span>
```

```
    <!-- /ko -->
        <!-- ko if: $context.key=='end' -->
        <span data-bind="text: 'End Date'"></span>
    <!-- /ko -->
</script>

<script type="text/html" id="row_header_template">
    <oj-row-expander context="[[$context]]"></oj-row-expander>
    <span data-bind="text: $context.data"></span>
</script>
```

The data for the `oj.FlattenedTreeDataGridDataSource` object can come from local or fetched JSON, or an `oj.Collection` object. In the example in this section, the data is read from a JSON file. The code sample below shows a portion of the JSON.

```
[
    {"attr": {"id": "t1",
              "name": "Task 1",
              "resource": "Chadwick",
              "start": "1/1/2014",
              "end": "10/1/2014"
             },
     "children": [
                     {"attr": {"id": "t1:1",
                               "name": "Task 1-1",
                               "resource": "Chris",
                               "start": "1/1/2014",
                               "end": "3/1/2014"
                              },
                      "children": [
                                       {"attr": {"id": "t1:1:1",
                                                 "name": "Task 1-1-1",
                                                 "resource": "Henry",
                                                 "start": "1/1/2014",
                                                 "end": "2/1/2014"
                                                }
                                       },
                                       {"attr": {"id": "t1:1:2",
                                                 "name": "Task 1-1-2",
                                                 "resource": "Victor",
                                                 "start": "2/1/2014",
                                                 "end": "3/1/2014"
                                                }
                                       }
                                   ]
                     },

                     ...contents omitted

                  ]
    },
    ... contents omitted
    {"attr": {"id": "t4",
              "name": "Task 4",
              "resource": "Victor",
              "start": "11/1/2014",
              "end": "12/1/2014"
             }
    }
]
```

The script that reads the JSON file and defines the `datasource` object as an `oj.FlattenedTreeDataGridDataSource` is shown below.

```
require(['ojs/ojcore', 'knockout', 'jquery', 'ojs/ojknockout', 'promise',
    'ojs/ojdatagrid', 'ojs/ojrowexpander', 'ojs/ojflattenedtreedatagriddatasource',
'ojs/ojjsontreedatasource'],
function(oj, ko, $)
{
    function viewModel()
    {
        var self = this;

        var options = {
            'rowHeader': 'name',
            'columns': ['resource', 'start', 'end']
        };
        self.dataSource = ko.observable();

        $.getJSON( "cookbook/dataCollections/rowExpander/dataGridRowExpander/
projectData.json",
            function(data)
            {
                self.dataSource(new oj.FlattenedTreeDataGridDataSource(new
oj.JsonTreeDataSource(data), options));
            }
        );
    }

    $(
        function()
        {
            ko.applyBindings(new viewModel(), document.getElementById('datagrid'));
        }
    );
});
```

To use the row expander with `oj-table`, add the `oj-row-expander` element to the HTML markup, and specify the table's data in an `oj.FlattenedTreeTableDataSource` object. The Oracle JET Cookbook at Row Expanders contains an example that uses the row expander with `oj-table`. The cookbook also contains the complete code for the example in this section and a link to the API documentation for `oj-row-expander`. In addition, you can find examples that use an `oj.Collection` object for the table's data and initialize the row expander with one or more rows expanded.

For additional information about working with the `oj-data-grid` component, see Working with Data Grids. For more information about working with the `oj-table` component, see Working with Tables.

Working with Tables

The Oracle JET `oj-table` component enhances the HTML table element to provide support for accessibility, custom cell and row templates and renderers, theming, row expansion, pagination, and editable array and collection tables.

Department Id	Department Name	Location Id	Manager Id
1001	ADFPM 1001 neverending	200	300
556	BB	200	300
10	Administration	200	300
20	Marketing	200	300
30	Purchasing	200	300
40	Human Resources1	200	300
50	Administration2	200	300
60	Marketing3	200	300
70	Purchasing4	200	300
80	Human Resources5	200	300
90	Human Resources11	200	300
100	Administration12	200	300
110	Marketing13	200	300
120	Purchasing14	200	300
130	Human Resources15	200	300

You can define the table's data in an array using the `oj.ArrayTableDataSource` object or in a Collection using the `oj.CollectionTableDataSource` object. The code sample below shows the markup for the `oj-table` component shown in this example.

```
<oj-table id='table'
          aria-label='Departments Table'
          data='[[datasource]]'
          columns-default.sortable='disabled'
          columns='[{"headerText": "Department Id",
                     "field": "DepartmentId"},
                    {"headerText": "Department Name",
                     "field": "DepartmentName"},
                    {"headerText": "Location Id",
                     "field": "LocationId"},
                    {"headerText": "Manager Id",
                     "field": "ManagerId"}]'>
</oj-table>
```

> **Note:**
>
> The `aria-label` attribute on the `oj-table` element describes the purpose of the table and is an accessibility requirement. See the oj-table API documentation for details about accessibility and supported keystrokes and gestures.

The script that defines the `datasource` object and completes the component binding is shown below. For the sake of brevity, some table data is omitted.

```
require(['ojs/ojcore', 'knockout', 'jquery', 'ojs/ojknockout', 'promise', 'ojs/
ojtable', 'ojs/ojarraytabledatasource'],
function(oj, ko, $)
```

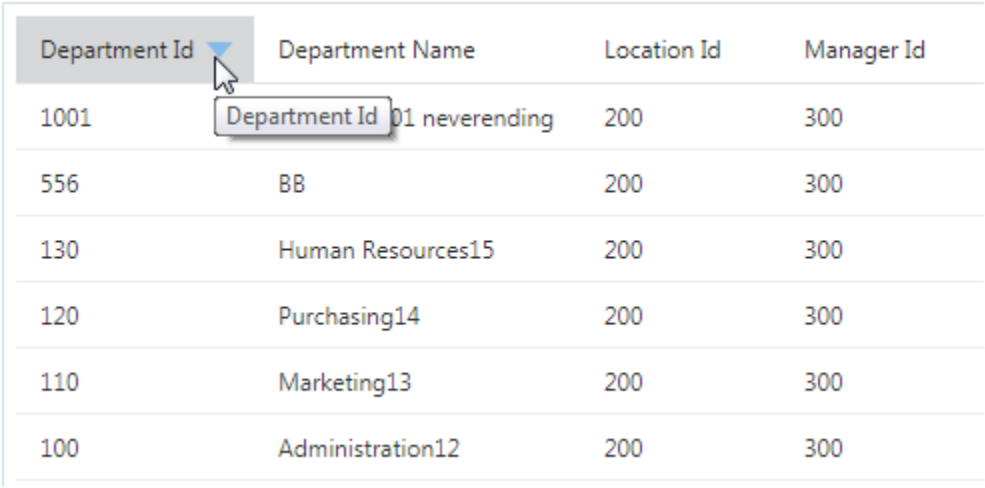
```
{
  function viewModel()
  {
    var self = this;

    var deptArray = [{DepartmentId: 1001, DepartmentName: 'ADFPM 1001 neverending',
LocationId: 200, ManagerId: 300},
        {DepartmentId: 556, DepartmentName: 'BB', LocationId: 200, ManagerId: 300},
        {DepartmentId: 10, DepartmentName: 'Administration', LocationId: 200,
ManagerId: 300},
        {DepartmentId: 20, DepartmentName: 'Marketing', LocationId: 200, ManagerId:
300},
        ... contents omitted
        {DepartmentId: 130, DepartmentName: 'Human Resources15', LocationId: 200,
ManagerId: 300}];
    self.datasource = new oj.ArrayTableDataSource(deptArray, {idAttribute:
'DepartmentId'});
  }
  var vm = new viewModel;

  $(document).ready
  (
    function()
    {
      ko.applyBindings(vm, document.getElementById('table'));
    }
  );
});
```

The Tables demos in the Oracle JET Cookbook include the complete example for this table and a link to the oj-table API documentation. The cookbook also includes an example that creates the `oj-table` component using data defined in an `oj.CollectionTableDataSource` object and examples that show tables with custom row and cell templates, selection, sorting, reordering, scrolling, custom cell renderers, and drag and drop support.

Understanding oj-table and Sorting

`oj-table` enables single column sorting by default if the underlying data supports sorting. Using the `sortable` property of the `oj-table` component's `columnDefaults` attribute, you can control sorting for the entire table, or you can use the `sortable` property of the `columns` attribute to control sorting on specific columns.

When you configure a column for sorting, the column header displays arrow indicators to indicate that the column is sortable when the user hovers over the column header. In the following image, the Department ID is sortable, and the sort indicator is showing a down arrow to indicate that the sort is currently descending. The user can select the down arrow to change the sort back to ascending.



The `sortable` attribute supports the following options:

- `auto`: Sort the table or indicated column if the underlying data supports sorting.
- `disabled`: Disable sorting on the table or indicated column.
- `enabled`: Enable sorting on the entire table or indicated column.

To enable sorting on specific columns:

- Set the `sortable` property to `none` on the table's `columnDefaults` attribute to remove the `auto` default behavior.
- Set the `sortable` property to `enabled` on the columns that you want to sort.

The following code sample shows the markup to create the department table shown in this section. In this example, the table is configured to enable sorting on only the Department Id column.

```
<oj-table id="table"
          aria-label="Departments Table"
          data='[[datasource]]'
          columns-default='{"sortable": "disabled"}'
          columns='[{"headerText": "Department Id",
                     "field": "DepartmentId",
                     "sortable": "enabled"},
                    {"headerText": "Department Name",
                     "field": "DepartmentName"},
                    {"headerText": "Location Id",
                     "field": "LocationId"},
                    {"headerText": "Manager Id",
                     "field": "ManagerId"}]'>
</oj-table>
```

`oj-table` sorting uses standard JavaScript array sorting. If your application requires custom sorting, you can use oj.ArrayTableDataSource and its `comparator` property. For an example, see Tables - Custom Sort.

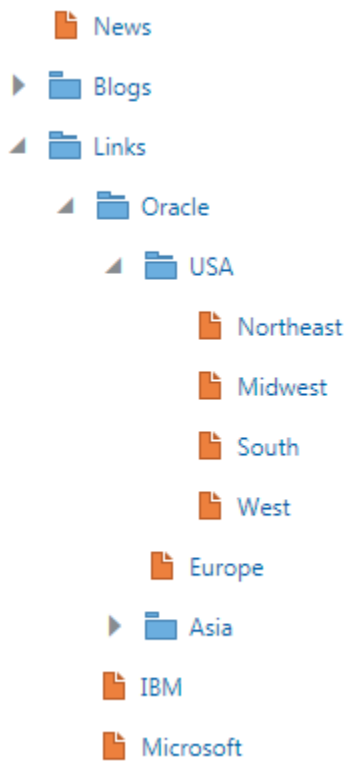
For additional information about `oj-table` and sorting options, see the oj-table API documentation.

For examples in the Oracle JET Cookbook that implement tables and table sorting, see Tables.

Working with Tree Views

The `oj-tree-view` component displays the hierarchical relationship between the items of a tree.

Each element in the tree is called a node, and the top levels of the hierarchy are referred to as the root nodes. The descendents of the root nodes are its children, and each child node can also contain children. Users select a node to display its children.



In this example, the expanded Links node is a root node, and the Oracle, IBM, and Microsoft nodes are its children. The Oracle child node contains the USA, Europe, and Asia nodes which also contain child nodes. The USA node is expanded to show its Northeast, Midwest, South, and West child nodes.

To create the `oj-tree-view` component, add the `oj-tree-view` element to the HTML markup. Assign it a meaningful ID and specify attributes on the `oj-tree-view` element. Construct the tree using a predefined HTML unordered list (`ul`) element in the `oj-tree-view` element.

Understanding Data Requirements for Tree Views

You can supply data to the tree view using static HTML content or a `TreeDataSource` object. The method you choose will depend upon the type of data you provide.

- static HTML content: Use when you want to define your data within HTML views. The code sample below shows the static HTML content used to render the tree view above.

```
<oj-tree-view id="treeview" data-oj-binding-provider="none" aria-label="Tree
View with Static HTML">
  <ul>
    <li id="news">
      <span class="oj-treeview-item-icon"></span
      ><span class="oj-treeview-item-text">News</span>
    </li>
    <li id="blogs">
      <span class="oj-treeview-item-icon"></span
      ><span class="oj-treeview-item-text">Blogs</span>
      <ul>
       ... contents omitted
      </ul>
    </li>
    <li id="links">
      <span class="oj-treeview-item-icon"></span
      ><span class="oj-treeview-item-text">Links</span>
      <ul>
        <li id="oracle">
          <span class="oj-treeview-item-icon"></span
          ><span class="oj-treeview-item-text">Oracle</span>
          <ul>
            <li id="usa">
              <span class="oj-treeview-item-icon"></span
              ><span class="oj-treeview-item-text">USA</span>
              <ul>
                <li id="northeast">
                  <span class="oj-treeview-item-icon"></span
                  ><span class="oj-treeview-item-text">Northeast</span>
                </li>
                <li id="midwest">
                  <span class="oj-treeview-item-icon"></span
                  ><span class="oj-treeview-item-text">Midwest</span>
                </li>
                <li id="south">
                  <span class="oj-treeview-item-icon"></span
                  ><span class="oj-treeview-item-text">South</span>
                </li>
                <li id="west">
                  <span class="oj-treeview-item-icon"></span
                  ><span class="oj-treeview-item-text">West</span>
                </li>
              </ul>
            </li>
            <li id="europe">
              <span class="oj-treeview-item-icon"></span
              ><span class="oj-treeview-item-text">Europe</span>
            </li>
            <li id="asia">
              <span class="oj-treeview-item-icon"></span
              ><span class="oj-treeview-item-text">Asia</span>
              <ul>
              ... contents omitted
              </ul>
            </li>
          </ul>
        </li>
```

```
        <li id="ibm">
          <span class="oj-treeview-item-icon"></span
          ><span class="oj-treeview-item-text">IBM</span>
        </li>
        <li id="microsoft">
          <span class="oj-treeview-item-icon"></span
          ><span class="oj-treeview-item-text">Microsoft</span>
        </li>
      </ul>
    </li>
  </ul>
</oj-tree-view>
```

> **Tip:**
>
> To ensure that there is no extra white space between the icon and the text in the display, place the icon `span` and the text `span` elements on a single line or use the broken closing bracket notation.
>
> ```
> <li id="news">
> </span
> >News
> </li>
> ```

- `oj.JsonTreeDataSource`: Use when the underlying data is a JSON object.

  In this example below, the tree view uses JSON data. The sample code also shows the Knockout `applyBindings()` call to complete the component binding.

```
require(['ojs/ojcore', 'knockout', 'jquery', 'ojs/ojknockout',
      'ojs/ojtreeview', 'ojs/ojjsontreedatasource'
    ],
    function(oj, ko, $) {
      function TreeViewModel() {
        var jsonData = [{
          "attr": {
            "title": "News",
            "id": "news"
          }
        }, {
          "attr": {
            "title": "Blogs",
            "id": "blogs"
          },
          "children": [{
            "attr": {
              "title": "Today",
              "id": "today"
            }
          }, {
            "attr": {
              "title": "Yesterday",
              "id": "yesterday"
            }
          }, {
            "attr": {
              "title": "Archive",
              "id": "archive"
            }
```

```
    }]
  }, {
    "attr": {
      "title": "Links",
      "id": "links"
    },
    "children": [{
      "attr": {
        "title": "Oracle",
        "id": "oracle"
      },
      "children": [{
        "attr": {
          "title": "USA",
          "id": "usa"
        },
        "children": [{
          "attr": {
            "title": "Northeast",
            "id": "northeast"
          }
        }, {
          "attr": {
            "title": "Midwest",
            "id": "midwest"
          }
        }, {
          "attr": {
            "title": "South",
            "id": "south"
          }
        }, {
          "attr": {
            "title": "West",
            "id": "west"
          }
        }]
      }, {
        "attr": {
          "title": "Europe",
          "id": "europe"
        }
      }, {
        "attr": {
          "title": "Asia",
          "id": "asia"
        },
        "children": [{
          "attr": {
            "title": "Japan",
            "id": "japan"
          }
        }, {
          "attr": {
            "title": "China",
            "id": "china"
          }
        }, {
          "attr": {
            "title": "India",
            "id": "india"
```

```
                    }
                  }]
                }]
              }, {
                "attr": {
                  "title": "IBM",
                  "id": "ibm"
                }
              }, {
                "attr": {
                  "title": "Microsoft",
                  "id": "microsoft"
                },
              }]
            }];
            this.data = new oj.JsonTreeDataSource(jsonData);
          }

          $(document).ready(function() {
            ko.applyBindings(new TreeViewModel(),
    document.getElementById('treeview'));
          });
        }
      );
```

- `oj.CollectionTreeDataSource` : Use when your data is coming from an `oj.Collection` object, typically representing an external data source.

Specifying Initial Expansion

By default, each node's children are hidden upon initial display. To specify that one or more nodes are expanded when the tree view first loads, use the `expanded` attribute and specify the key set containing the keys of the items that should be expanded.

```
<oj-tree-view id="treeview"
              data="[[data]]"
              item.renderer="[[renderer]]"
              expanded="{{expanded}}"
              aria-label="Tree View Expansion Demo">
</oj-tree-view>
```

In your application's viewModel, define the key set. The example below sets the tree view to expand the Links node on initial display. Note that you must also add the `ojknockout-keyset` module and `keySet` function to the require definition.

```
require(['ojs/ojcore', 'ojs/ojknockout-keyset', 'knockout', 'jquery',
    'ojs/ojknockout', 'ojs/ojbutton', 'ojs/ojtreeview',
    'ojs/ojjsontreedatasource'
    ],
    function(oj, keySet, ko, $, jsonData) {
      function TreeViewModel() {
        var self = this;

        self.data = new oj.JsonTreeDataSource(JSON.parse(jsonData));
        self.expanded = new keySet.ObservableExpandedKeySet().add(['links']);
        ... contents omitted
);
```

Examples

The Oracle JET Cookbook contains the complete recipe and code for the samples used in this section at Tree Views. The cookbook also includes a link to the oj-tree-view API documentation which includes additional examples for creating tree views.

Working with Controls

Oracle JET includes buttons, menus, and container elements to control user actions or display progress against a task. For HTML elements such as simple lists, you can use the standard HTML tags directly on your page, and Oracle JET will apply styling based on the application's chosen theme.

For example, you can use the `oj-button` element as a standalone element or include in `oj-buttonset` , `oj-menu`, and `oj-toolbar` container elements.

Navigation components such as `oj-conveyor-belt`, `oj-film-strip`, and `oj-train` use visual arrows or dots that the user can select to move backward or forward through data.

To show progress against a task in a horizontal meter, you can use the `oj-progress` element.

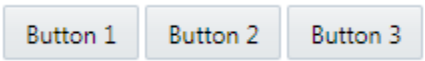
The Oracle JET Cookbook and JavaScript API Reference for Oracle® JavaScript Extension Toolkit (JET) include complete demos and examples for using Oracle JET controls, and you may also find the following tips and tricks helpful.

Topics:

- Working with Buttons
- Working with Button Sets
- Working with Conveyor Belts
- Working with File Picker
- Working with Film Strips
- Working with Menus
- Working with Progress Indicators
- Working with Tags
- Working with Toolbars
- Working with Trains

Working with Buttons

Use the Oracle JET `oj-button` element to display a push button. A push button is an ordinary button that does not stay pressed when clicked. Oracle JET buttons are WAI-ARIA compliant and themable, with appropriate styles for `hover`, `active`, `checked`, and `disabled`.



Create a push button by adding the `oj-button` element to the HTML markup and adding a listener in your view model to respond to click events. The code sample below shows the markup for **Button 1**, **Button 2**, and **Button 3**.

```
<div id='buttons-container'>
    <oj-button id='button1' on-click='[[buttonClick]]'>Button 1</oj-button>
    <oj-button id='button2' on-click='[[buttonClick]]'>
        <span data-bind="text: button2Text"></span>
    </oj-button>
    <oj-button id='button3' on-click='[[buttonClick]]'>
         <!--ko text: button3Text--><!--/ko-->
    </oj-button>
</div>
```

Each button's `on-click` attribute specifies the `buttonClick` listener. In this example, the listener simply sets the value of `clickedButton` to the id of the button that the user clicked: `button1`, `button2`, or `button3`.

```
function buttonModel(){
    var self = this;

    self.button2Text = "Button 2";
    self.button3Text = "Button 3";

    self.clickedButton = ko.observable("(None clicked yet)");
    self.buttonClick = function(event){
        self.clickedButton(event.currentTarget.id);
        return true;
    }
}
```

The Oracle JET Cookbook at Buttons includes complete examples for all buttons, including a section on button styling that shows you how to add icons, control chroming, and configure colors, width, and responsive behavior. oj-button API documentation describes support for keyboard interaction, accessibility, and event handling.

Working with Button Sets

Use `oj-buttonset-many` or `oj-buttonset-one` to group related toggle buttons, such as a group of checkboxes or radio buttons.

The `oj-buttonset-many` and `oj-buttonset-one` components are themable, visual, and semantic grouping containers for Oracle JET buttons that are also compliant with WAI-ARIA. The `oj-buttonset-many` element allows the user to toggle multiple buttons, as shown in the figure below, and `oj-buttonset-one` allows the user to toggle only one.

Create the button set by adding the `oj-buttonset-many` element or `oj-buttonset-one` element to the application's markup. Set a `value` attribute on the `oj-buttonset-many` or `oj-buttonset-one` element. Inside the button set, create each button from an `oj-option` element.

```
<div id='buttons-container'>
  <oj-buttonset-many id="formatset" value="{{formats}}">
```

```
    <oj-option value="bold">Bold</oj-option>
    <oj-option value="italic">Italic</oj-option>
    <oj-option value="underline">Underline</oj-option>
  </oj-buttonset-many>
</div>
```

In your view model, set an initial value for the button set and apply the bindings.

```
require(['ojs/ojcore', 'knockout', 'jquery', 'ojs/ojknockout',  'ojs/ojbutton'],
function(oj, ko, $)
{
  function ButtonModel() {
  var self = this;
  // observable array bound to the Buttonset:
  self.formats = ko.observableArray(["bold", "underline"]);
  }

  $(function() {
    ko.applyBindings(new ButtonModel(), document.getElementById('buttons-
container'));
  });
});
```

The Oracle JET cookbook at Button Sets contains the complete recipe to create the button set shown above.

Here are some tips for working with button sets:

- A button set that contains radios should contain all radios in the radio group.
- Checkboxes and radio buttons created by `oj-option` in the button set should specify the value attribute, since the `value` attribute on the `oj-buttonset` refers to that attribute.
- The application should not do anything to interfere with the focus management. For example, it should not set the `tabindex` of the buttons.
- Enabled buttons should remain user visible. Otherwise, the arrow-key navigation to the button will cause the focus to seemingly disappear.
- The button set's `focusManagement` attribute should be set to `none` when placing the button set in an `oj-toolbar` element.
- The application is responsible for applying WAI-ARIA aria-label and aria-controls attributes to the button set, as appropriate.

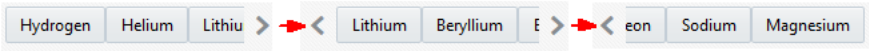
  ```
  aria-label="Choose only one.  Use left and right arrow keys to navigate."
  aria-controls="myTextEditor"
  ```

- If the value attribute and DOM get out of sync (for example, if a set of buttons contained in a button set changes, possibly due to a Knockout binding), then the application is responsible for updating the `value` attribute.
- The application doesn't need to listen for this event, since the `value` binding will update the bound observable whenever the value state changes.

For additional information about the `oj-buttonset-many` and `oj-buttonset-one` components' attributes, events, and methods, see the oj-buttonset API documentation.

Working with Conveyor Belts

The Oracle JET `oj-conveyor-belt` custom element manages overflow for a group of sibling child elements to control the number of child elements displayed and provides horizontal or vertical scrolling to cycle through the other child elements.



Define the `oj-conveyor-belt` element in the HTML file. Specify a group of child sibling elements to be managed by the `oj-conveyor-belt`. You can add the sibling child elements as either direct children of the conveyor belt or as the nested children of a container element that is itself a direct or nested child of the conveyor belt. In this example, the `oj-conveyor-belt` is configured for horizontal scrolling with a maximum width that varies on the screen width, and the sibling child elements are defined as Oracle JET `oj-button` elements.

```
<div id="conveyorbelt-horizontal-example">
 <div class="oj-flex">
   <oj-conveyor-belt class="oj-lg-6 oj-md-9 oj-sm-12">
     <oj-button class="demo-button">Hydrogen</oj-button>
     <oj-button class="demo-button">Helium</oj-button>
     <oj-button class="demo-button">Lithium</oj-button>
     <oj-button class="demo-button">Beryllium</oj-button>
     <oj-button class="demo-button">Boron</oj-button>
     <oj-button class="demo-button">Carbon</oj-button>
     <oj-button class="demo-button">Nitrogen</oj-button>
     <oj-button class="demo-button">Oxygen</oj-button>
     <oj-button class="demo-button">Fluorine</oj-button>
     <oj-button class="demo-button">Neon</oj-button>
     <oj-button class="demo-button">Sodium</oj-button>
     <oj-button class="demo-button">Magnesium</oj-button>
   </oj-conveyor-belt>
 </div>
</div>
```

> **Note:**
>
> The `oj-conveyor-belt` component does not provide accessibility features such as keyboard navigation. It is the responsibility of the application developer to make the items in the conveyor belt accessible. For tips and additional detail, see the oj-conveyor-belt API documentation.

When you configure the child elements as direct children of the `oj-conveyor-belt`, the element will ensure that they are laid out according to the specified orientation. If, however, you configure the child elements as the children of a container element, you must take additional steps to ensure the correct display. For details, see the oj-conveyor-belt API documentation.

The Oracle JET Cookbook Conveyor Belts demos contain the complete code for this example. In addition, you will find examples for a vertical conveyor belt and conveyor belts with nested content, tab-based scrolling, and programmatic scrolling.

> **Note:**
>
> The Oracle JET `oj-film-strip` component also manages a group of sibling child elements to provide horizontal or vertical scrolling to cycle through the other child elements. However, it also provides the ability to:
>
> - lay out a set of items across discrete logical pages.
> - control which and how many items are shown.
> - hide items outside the current viewport from tab order and screen readers.
>
> For additional information, see Working with Film Strips.

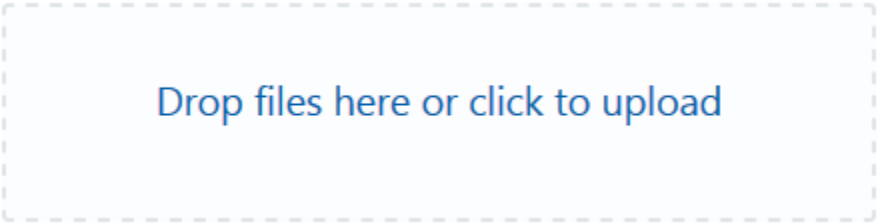
Working with File Picker

Use `oj-file-picker` element to provide a clickable drop zone for users to select a single or multiple files.

By default, the Oracle JET file picker displays a clickable drop zone to select file(s) or drag and drop file(s) from the file system to the drop zone. You can customize the default drop zone text or the entire drop zone by using the named `slot` set to `trigger`.

The following example provides the default clickable drop zone:

```
<oj-file-picker on-select='[[selectListener]]'></oj-file-picker>
```



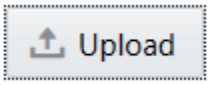
The following example provides the clickable drop zone with customized text:

```
<oj-file-picker on-select='[[selectListener]]'>
  <div tabindex='0' slot='trigger' class='oj-filepicker-dropzone'>
    <p class='oj-filepicker-text'>I'm a clickable dropzone</p>
  </div>
</oj-file-picker>
```

The following example provides the customized file picker and the slot attribute is set to trigger the custom content of the file picker:

```
<oj-file-picker class='oj-filepicker-custom' selectOn='click' on-
select='[[selectListener]]'>
  <oj-button slot='trigger'>
```

```
    <span slot='startIcon' class='oj-fwk-icon oj-fwk-icon-arrowbox-n'></span>
    Upload
    </oj-button>
</oj-file-picker>
```

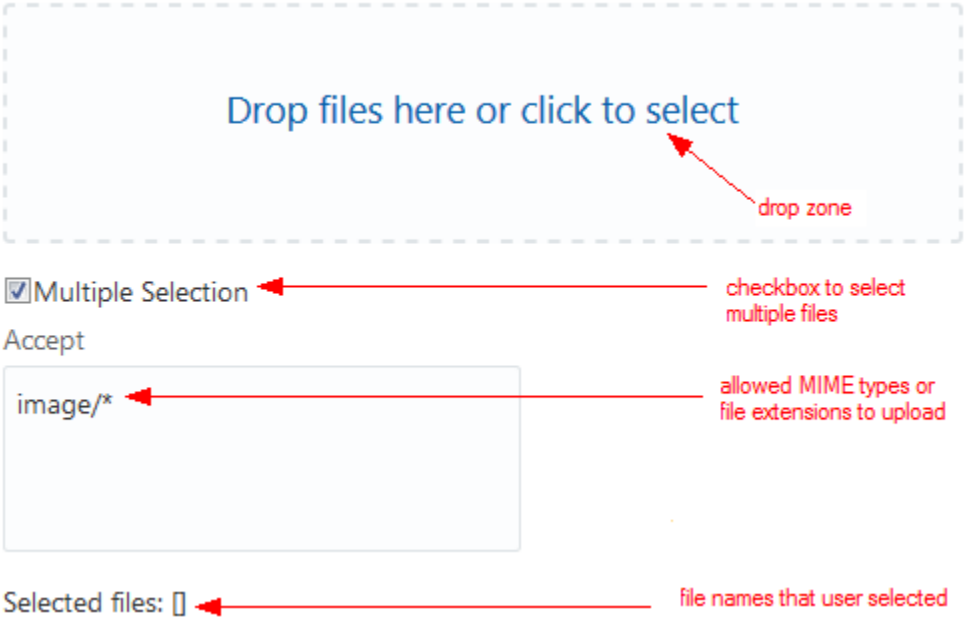


By specifying the `accept` attribute, you can allow the file picker to accept an array of strings of allowed MIME types. Default value is `undefined`. If `accept` attribute is not specified, all file types are accepted.

> **Note:**
>
> If the `accept` attribute is specified, files with empty string type will be rejected if no match is found in the `accept` attribute value.

To allow multiple selection of files, set the `selection-mode` attribute to `multiple`. Also, to get the selected files, add an `on-select` listener.

The following image illustrates the default file picker with `selection mode` set to multiple file selection and `accept` attribute set to any image type.




```
<div id="parentContainer" style="padding:10px">
  <oj-file-picker accept="[[acceptArr]]"
                  selection-mode="[[multipleStr]]"
                  on-select="[[selectListener]]">
  </oj-file-picker>

  <div style="padding-top:8px"></div>
  <div class="oj-choice-item">
    <input id="selection" type="checkbox" class="oj-checkbox-nocomp"
```

```
          data-bind="checked: multiple" />
    <label for="selection" class="oj-checkbox-label-nocomp">Multiple Selection</
label>
  </div>

  <label for="acceptFld"
         class="oj-label-nocomp">Accept</label>
  <textarea id="acceptFld"
            rows="3"
            class="oj-form-control oj-textarea-nocomp"
            data-bind="value: acceptStr">
  </textarea>

  <div style="padding-top:8px">
    <label for="selected">Selected files: </label>
    <span id="selected" data-bind="text: ko.toJSON(fileNames)"></span>
  </div>
</div>
```

In this example, the file picker uses Knockout `array` to accept the selected files, identify the file name of the selected file, and to display the file name.

```
require(['ojs/ojcore', 'knockout', 'jquery', 'ojs/ojknockout', 'ojs/ojfilepicker'],
  function(oj, ko, $) {
    var self = this;

    function basicModel() {
      self.multiple = ko.observable(true);
      self.multipleStr = ko.pureComputed(function() {
        return multiple() ? "multiple" : "single";
      }, self);

      self.acceptStr = ko.observable("image/*");
      self.acceptArr = ko.pureComputed(function() {
        var accept = acceptStr();
        return accept ? accept.split(",") : [];
      }, self);

      self.fileNames = ko.observableArray([]);

      self.selectListener = function(event) {
        var files = event.detail.files;
        for (var i = 0; i < files.length; i++) {
          self.fileNames.push(files[i].name);
        }
      }
    }
    $(function() {
      ko.applyBindings(new basicModel(), document.getElementById('parentContainer'));
    });

  });
```

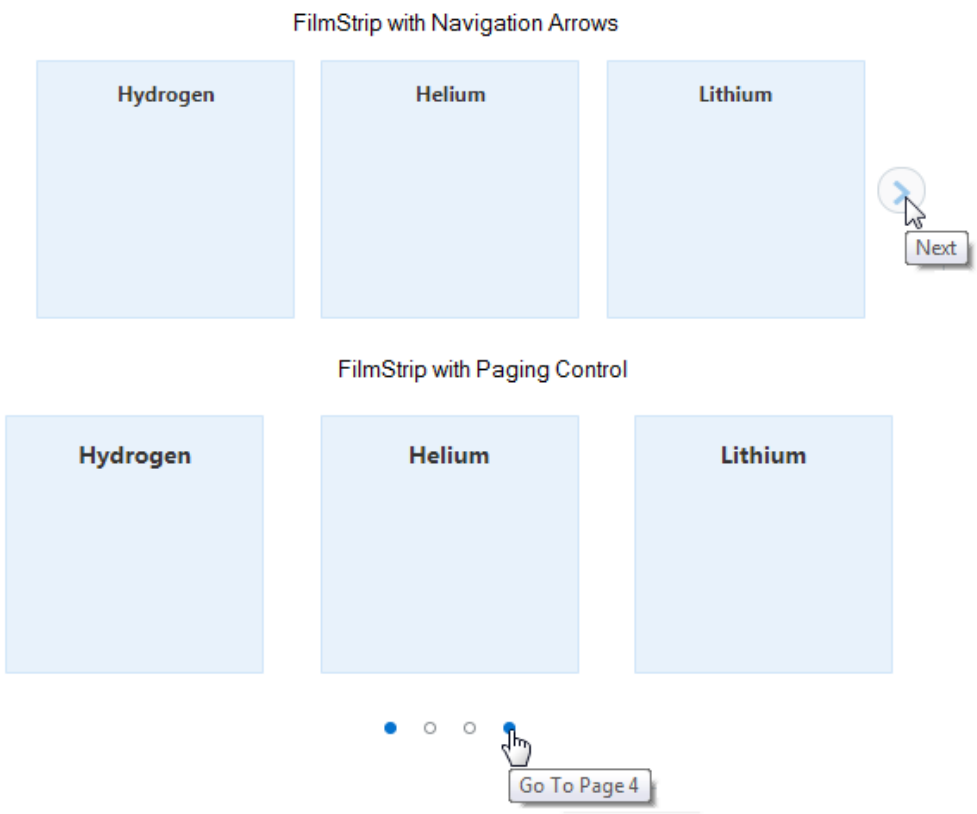
The Oracle JET Cookbook File Picker demos contain the complete code for this example.

Working with Film Strips

The Oracle JET `oj-film-strip` component manages a group of sibling child elements to lay out its children in a single row or column across logical pages and provides

horizontal or vertical scrolling to cycle through the other child elements. You can configure the film strip to use arrows or add the `oj-paging-control` element that uses dots for scrolling through the child elements.

The following image shows two `oj-film-strip` elements configured for horizontal scrolling. The sibling child elements are panels, using the Oracle JET panel design pattern. In the top film strip, the user selects arrows for navigating through the content, and in the bottom film strip, the user selects dots for navigating through the content.



Configuring Film Strips

Create an `oj-film-strip` element. In the HTML file, specify a group of sibling child elements to be laid out by the `oj-film-strip`. The code sample below shows the markup for the film strip example using arrows for navigation.

```
<div id="filmStripDiv" class="oj-panel" style="margin: 20px;">
  <oj-film-strip id="filmStrip"
       aria-label="Set of chemicals"
       arrow-placement="[[currentNavArrowPlacement]]"
       arrow-visibility="[[currentNavArrowVisibility]]">
    <!-- ko foreach: chemicals -->
    <div class="oj-panel oj-panel-alt2 demo-filmstrip-item"
         data-bind="style: {
           display: getItemInitialDisplay($index())}">
      <span data-bind="text: name"></span>
    </div>
    <!-- /ko -->
  </oj-film-strip>
</div> <!-- end filmStripDiv -->
```

You can use the `arrow-placement` attribute to control the location of the arrows. By default, it is set to `adjacent` which displays arrows outside the content, but you can set it to `overlay` to overlay the arrows on the content.

In this example, the film strip uses the Knockout `foreach` binding to iterate through the list of chemicals defined in the application's main script, shown below.

```
require(['ojs/ojcore', 'knockout', 'jquery',
         'ojs/ojknockout', 'ojs/ojfilmstrip'],
function(oj, ko, $)
{
  $(
    function()
    {
      function MyModel() {
        var self = this;

        self.chemicals = [
            { name: 'Hydrogen' },
            { name: 'Helium' },
            { name: 'Lithium' },
            { name: 'Beryllium' },
            { name: 'Boron' },
            { name: 'Carbon' },
            { name: 'Nitrogen' },
            { name: 'Oxygen' },
            { name: 'Fluorine' },
            { name: 'Neon' },
            { name: 'Sodium' },
            { name: 'Magnesium' }
        ];

        self.currentNavArrowPlacement = ko.observable("adjacent");
        self.currentNavArrowVisibility = ko.observable("auto");

        getItemInitialDisplay = function(index)
        {
          return index <3 ? '' : 'none';
        };
      };

      var model = new MyModel();

      ko.applyBindings(model,
        document.getElementById('filmstrip-navarrows-example'));
    }
  );
});
```

`oj-film-strip` will lay out the child items across multiple logical pages and allow for changing between logical pages. When the element is resized, the layout will adjust automatically, and the number of pages and items shown per page may change.

The Oracle JET Cookbook at Film Strips includes the complete code for the example used in this section. You can also find examples for film strips with pagination, vertical film strips, lazy loading a film strip, film strips that contain master-detail data, and film strips that use the `getPagingModel()` to display paging information.

> **Note:**
>
> `oj-film-strip` is a layout element, and it is the responsibility of the application developer to make the items in the film strip accessible. For tips and additional detail, see the Accessibility section in the oj-film-strip API documentation.

Working with Menus

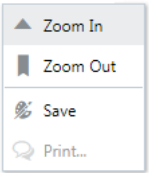
The `oj-menu` component is a themable, WAI-ARIA compliant menu with mouse and keyboard interactions for navigation. Use the `oj-menu` element on the `oj-button` element to create a menu button, or add it as a context menu to an Oracle JET element.

Topics:

- Working with oj-menu
- Working with Menu Buttons
- Working with Context Menus

Working with oj-menu

To create a menu, use the `oj-menu` element with an `oj-option` element representing each menu item. Ideally, menus should have a manageable number of items. When your menu is displayed on a small screen, such as on a mobile device, the menu will behave like a sheet menu, and its contents will slide up from the bottom of the screen.



The following code sample shows the markup used to create the basic menu. To handle menu selection, add an `action` listener.

```
<div id='menubutton-container'>
  <oj-menu-button id="menuButton">
    Actions
    <!-- To handle menu item selection, use an action listener as shown, not a click
listener. -->
    <oj-menu id="myMenu" slot="menu" style="display:none" on-oj-
action="[[menuItemAction]]">
      <oj-option id="zoomin" value="Zoom In">
          <span class="oj-menu-item-icon oj-fwk-icon oj-fwk-icon-arrow-n"
slot="startIcon"></span>Zoom In
      </oj-option>
      <oj-option id="zoomout" value="Zoom Out">
          <span class="oj-menu-item-icon demo-icon-font demo-bookmark-icon-16"
slot="startIcon"></span>Zoom Out
      </oj-option>
      <oj-option id="divider"></oj-option>
```

```
      <oj-option id="save" value="Save">
          <span class="oj-menu-item-icon demo-icon-font demo-palette-icon-24"
slot="startIcon"></span>Save
      </oj-option>
      <oj-option id="print" value="Print..." disabled="true">
          <span class="oj-menu-item-icon demo-icon-font demo-chat-icon-24"
slot="startIcon"></span>Print...
      </oj-option>
    </oj-menu>
  </oj-menu-button>
</div>
```

The following script defines the action listener.

```
require(['ojs/ojcore', 'knockout', 'jquery', 'ojs/ojknockout', 'ojs/ojbutton', 'ojs/
ojmenu', 'ojs/ojoption'],
function(oj, ko, $)
{

function MenuModel() {
    var self = this;
    self.selectedMenuItem = ko.observable("(None selected yet)");

    self.menuItemAction = function( event ) {
        self.selectedMenuItem(event.detail.item.value);
    };
}

$(function() {
    ko.applyBindings(new MenuModel(), document.getElementById('menubutton-
container'));
});

});
```

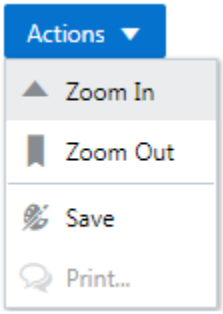
For additional information about the `oj-menu` component's attributes, events, and methods, see the oj-menu API documentation.

The Oracle JET cookbook includes advanced examples for working with `oj-menu`, including demos for working with menus programmatically and using templates. For details, see Menu (Advanced).

Working with Menu Buttons

Menu buttons are `oj-button` components that display an `oj-menu` component when the user does one of the following:

- Clicks on the button.
- Sets focus on the button and presses the Enter, Spacebar, or Arrow Down key.



To create the menu button, add the basic `oj-menu` element as a child of an `oj-menu-button` element.

The following code sample shows the markup for the menu button, with the details for the basic menu omitted.

```
<div id='menubutton-container'>
  <oj-menu-button id="menuButton">
    Actions
    <!-- To handle menu item selection, use an action listener as shown, not a click
listener. -->
    <oj-menu id="myMenu" slot="menu" style="display:none" on-oj-
action="[[menuItemAction]]">
      <oj-option id="zoomin" value="Zoom In">
        <span class="oj-menu-item-icon oj-fwk-icon oj-fwk-icon-arrow-n"
slot="startIcon"></span>Zoom In
      </oj-option>
      ... contents omitted
    </oj-menu>
  </oj-menu-button>
  <p>
  <p>
  <p class="bold">Last selected menu item:
    <span id="results" data-bind='text: selectedMenuItem'></span>
  </p>
</div>
```

The following code sample shows the code that initializes the menu button.

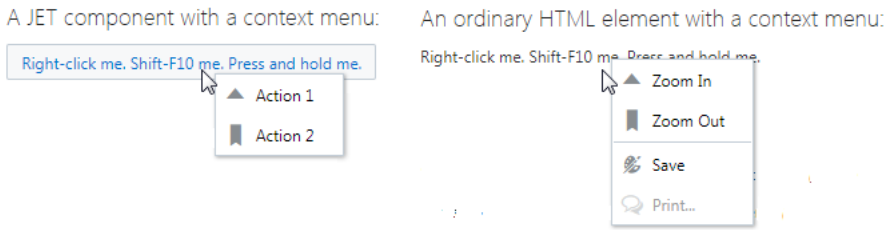
```
require(['ojs/ojcore', 'knockout', 'jquery', 'ojs/ojknockout', 'ojs/ojbutton', 'ojs/
ojmenu', 'ojs/ojoption'],
function(oj, ko, $)
{

function MenuModel() {
    var self = this;
    self.selectedMenuItem = ko.observable("(None selected yet)");

    self.menuItemAction = function( event ) {
        self.selectedMenuItem(event.detail.item.value);
    };
}

$(function() {
    ko.applyBindings(new MenuModel(), document.getElementById('menubutton-
container'));
});
```

```
});
```

The Oracle JET Cookbook includes the complete code sample for the menu button shown in this section. For details, see Menu Buttons.

Working with Context Menus

Context menus are `oj-menu` elements placed on the Oracle JET component using the `slot` attribute or an HTML5 element using the `contextMenu` attribute. The element on which the context menu binding is placed should be user-focusable, for keyboard accessibility through the Shift+F10 key, since focus is returned to the element when the menu is dismissed.



In this example, the `oj-menu` element is placed on an `oj-button` element and the HTML `anchor` element. The code sample below shows the markup for the `oj-button` element and the HTML `anchor` element, with the details for the basic menu omitted.

```
<div id="button-container">
  <h3>A JET component with a context menu:</h3>

  <!-- Use the contextMenu slot for JET components -->
  <oj-button id="myButton">
    Right-click me. Shift-F10 me. Press and hold me.
    <oj-menu slot="contextMenu" style="display:none" aria-label="Order Actions" on-
oj-action="[[menuItemAction]]" >
      <oj-option id="action1" value="Action 1">
        <span class="oj-menu-item-icon oj-fwk-icon oj-fwk-icon-arrow-n"
slot="startIcon"></span>Action 1
      </oj-option>
      <oj-option id="action2" value="Action 2">
        <span class="oj-menu-item-icon demo-icon-font demo-bookmark-icon-16"
slot="startIcon"></span>Action 2
      </oj-option>
    </oj-menu>
  </oj-button>

  <br><br>

  <h3>An ordinary HTML element with a context menu:</h3>

  <!-- Use the ojContextMenu binding for elements that aren't JET components
       The tabindex makes this div user-focusable per the requirements.  -->
  <div id="myDiv" contextmenu="myMenu" tabindex="0"
       data-bind="ojContextMenu: {}">Right-click me. Shift-F10 me. Press and hold
me.</div>
```

```
  <!-- To handle menu item selection, use a select listener as shown, not a click
listener. -->
  <oj-menu id="myMenu" style="display:none" aria-label="Order Actions" on-oj-
action="[[menuItemAction]]" >
    <oj-option id="zoomin" value="Zoom In">
      <span class="oj-menu-item-icon oj-fwk-icon oj-fwk-icon-arrow-n"
slot="startIcon"></span>Zoom In
    </oj-option>
    <oj-option id="zoomout" value="Zoom Out">
      <span class="oj-menu-item-icon demo-icon-font demo-bookmark-icon-16"
slot="startIcon"></span>Zoom Out
    </oj-option>
    <oj-option id="divider"></oj-option>
    <oj-option id="save" value="Save">
      <span class="oj-menu-item-icon demo-icon-font demo-palette-icon-24"
slot="startIcon"></span>Save
    </oj-option>
    <oj-option id="print" value="Print..." disabled="true">
      <span class="oj-menu-item-icon demo-icon-font demo-chat-icon-24"
slot="startIcon"></span>Print...
    </oj-option>
 </oj-menu>

 <br><br>

 <h3>Results:</h3>

 <p>Last selected menu item in the button's context menu:
   <span id="buttonResults" class="italic" data-bind="text: selectedItem.myButton"></
span>
 </p>
 <p>Last selected menu item in the div's context menu:
   <span id="divResults" class="italic" data-bind="text: selectedItem.myDiv"></span>
 </p>
</div>
```

The following code sample shows the code to initialize the context menu.

```
require(['ojs/ojcore', 'knockout', 'jquery', 'ojs/ojknockout', 'ojs/ojbutton', 'ojs/
ojmenu', 'ojs/ojoption'],
function(oj, ko, $)
{

function MenuModel() {
    var self = this;
    self.selectedItem = {
        myButton: ko.observable("(None selected yet)"),
        myDiv: ko.observable("(None selected yet)")
    };

    self.menuItemAction = function( event ) {
        var launcherId = event.target.id == "myMenu" ? "myDiv" : "myButton";

        self.selectedItem[launcherId](event.detail.item.textContent);

    };
}

$(function() {
    ko.applyBindings(new MenuModel(), document.getElementById('button-container'));
});
```

```
});
```

The Oracle JET cookbook contains the complete example, including the CSS for the menu icons, at Context Menus.

Working with Progress Indicators

You can use the `oj-progress` component to indicate progress against a task in a horizontal meter. Set the value for the progress indicator in the `oj-progress` element's `value` attribute. In the image below, the progress indicator's `value` option is set to 70 to indicate that the task is 70% complete.

To indicate that the value is indeterminate, set the `value` option to -1, and the progress indicator will change to reflect the indeterminate status.

To create a progress indicator, use the `oj-progress` element with a defined value attribute.

```
<div id="progressbarWrapper">
  <oj-progress style="width:50%" value="{{progressValue}}"></oj-progress>
</div>
```

The script that sets the value defines a Knockout observable and sets the initial value. In this example, the progress indicator's initial value is set to -1, to indicate that the value is indeterminate.

```
require(['ojs/ojcore', 'knockout', 'jquery', 'ojs/ojknockout', 'ojs/ojprogress'],
  function(oj, ko, $)
  {
    function ViewModel()
    {
      var self = this;
      self.progressValue = ko.observable(-1);
    }

    ko.applyBindings(new ViewModel(), document.getElementById('progressbarWrapper'));
  });
```

The Oracle JET Cookbook at Progress contains a complete example for an `oj-progress` that shows the effect of adjusting the progress indicator's value. You can also find examples that show a progress indicator with labeling and a progress indicator that is linked to changing data.

Working with Tags

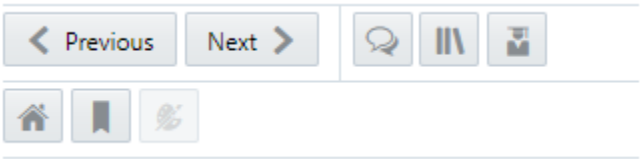
You can add HTML tags to your Oracle JET page as you would for any other HTML application. However, you should be aware that Oracle JET provides styling directly on the following tags:

- Header (`h1`, `h2`, `h3`, `h4`)
- Horizontal rule (`hr`)
- Link (`a`)
- List (`ul`, `ol`, `li`)
- Paragraph (`p`)

There may be use cases where you do not want to use aspects of the default theming because it causes compatibility issues. For example, you may be embedding JET components or regions in a page controlled by another technology, such as Oracle ADF. In these cases, Oracle JET provides options for theming for compatibility. For additional information, see Understanding Oracle JET Theming For Compatibility.

Working with Toolbars

The `oj-toolbar` component is a WAI-ARIA compliant toolbar, with arrow key navigation and a single tab stop. The tab stop updates on navigation, so that tabbing back into the toolbar returns to the most recently focused button.



The `oj-toolbar` can contain buttons, menu buttons, button sets, and non-focusable content such as separator icons.

Here are some tips for working with toolbars:

- A toolbar that contains radio buttons should contain all radio buttons in the radio group.
- The application should not do anything to interfere with the focus management.
- Enabled buttons should remain user visible. Otherwise, the arrow-key navigation to the button would cause the focus to seemingly disappear.
- The button set's `focusManagement` attribute should be set to `none` when placed in a toolbar.

The Oracle JET Cookbook includes toolbar examples at Toolbars. For additional information about the `oj-toolbar` component's attributes, events, and methods, see the oj-toolbar API documentation.

Working with Trains

The `oj-train` component displays a navigation visual that enables the user to move back and forth between different points. Typically, the train displays steps in a task or process.

Each step can display information about its `visited` state (`visited`, `unvisited`, or `disabled`) and a message icon of type `confirmation`, `error`, `fatal`, `warning`, or `info`.

To create a train, you can directly add the `oj-train` element in the HTML file. Then you can define the selected-step and steps attributes. The code sample below shows a portion of the markup for the first `oj-train` shown in this section.

```
<div id="train-container">
  <oj-train id="train"
            class="oj-train-stretch"
            style="max-width:700px;margin-left:auto;margin-right:auto;"
            on-selected-step-changed="[[updateLabelText]]"
            selected-step="{{selectedStepValue}}"
            steps="[[stepArray]]">
  </oj-train>
  ... contents omitted
</div>
```

The code sample below shows the code that applies the binding, defines the steps and captures the step selection.

```
require(['ojs/ojcore', 'knockout', 'jquery', 'ojs/ojknockout', 'ojs/ojtrain', 'ojs/
ojbutton'],
function(oj, ko, $)
{
    function TrainData() {
          this.selectedStepValue = ko.observable('stp1');
          this.selectedStepLabel = ko.observable('Step One');
          this.stepArray =
            ko.observableArray(
                    [{label:'Step One', id:'stp1'},
                     {label:'Step Two', id:'stp2'},
                     {label:'Step Three', id:'stp3'},
                     {label:'Step Four', id:'stp4'},
                     {label:'Step Five', id:'stp5'}]);
    };

    this.updateLabelText = function(event) {
          var train = document.getElementById("train");
          trainModel.selectedStepLabel(train.getStep(event.detail.value).label);
```

```
    };

    var trainModel = new TrainData();

    $(function() {
          ko.applyBindings(trainModel, document.getElementById('train-container'));
    });
});
```

The Oracle JET Cookbook includes the complete code for this example at Trains. You can also find additional examples that show a stretched train, a train with messages, and a train with button navigation.

Working with Forms

Oracle JET includes classes to create responsive form layouts and components that you can add to your form to manage labels, form validation and messaging, input, and selection. The input components also include attributes to mark an input as disabled or read only when appropriate.

The Oracle JET Cookbook and JavaScript API Reference for Oracle® JavaScript Extension Toolkit (JET) include complete demos and examples for using forms, and you may also find the following tips and tricks helpful.

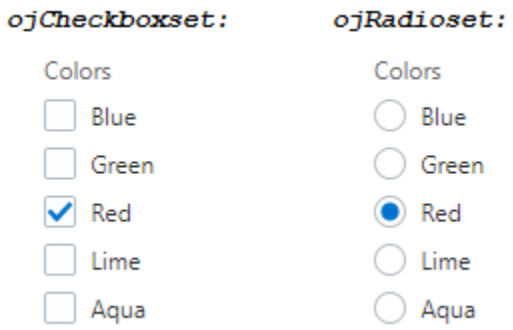
Topics:

- Working with Checkbox and Radio Sets
- Working with Color Pickers
- Working with Comboboxes
- Working with Form Controls
- Working with Form Layout Features
- Working with Input Components
- Working with Labels
- Working with Select
- Working with Sliders
- Working with Switches
- Working with Validation and User Assistance

> **Important:**
>
> When working with forms, use the HTML `div` element to surround any Oracle JET input component. Do not use the HTML `form` element because its postback behavior can cause unwanted page refreshes when the user submits or saves the form.

Working with Checkbox and Radio Sets

The Oracle JET `oj-checkboxset` and `oj-radioset` components enhance a group of HTML `input` elements.



The `oj-checkboxset` and `oj-radioset` components manage the selected value of their group and add required validation. In addition, the components manage the styles of the input elements, adding and removing the Oracle JET styles depending upon state.

To create the `oj-checkboxset` or `oj-radioset`, add an `oj-checkboxset` or `oj-radioset` node that wraps a set of `oj-option`s and creates the necessary `input` and `label` elements. The initial value of the checkbox or radio button is defined in the component's `value` option. Provide data with `oj-option`. The following code example shows the markup that defines the `oj-checkboxset` shown in the image above.

```
<div id="formId">
    <oj-label id="mainlabelid">Colors</oj-label>
    <!-- You need to set the aria-labelledby attribute
         to make this accessible.
         role="group" is set for you by oj-checkboxset. -->
    <oj-checkboxset id="checkboxSetId" labelled-by="mainlabelid"
          value="{{currentColor}}">
      <oj-option id="blueopt" name="color" value="blue">Blue</oj-option>
      <oj-option id="greenopt" name="color" value="green">Green</oj-option>
      <oj-option id="redopt" name="color" value="red">Red</oj-option>
      <oj-option id="limeopt" name="color" value="lime">Lime</oj-option>
      <oj-option id="aquaopt" name="color" value="aqua">Aqua</oj-option>
    </oj-checkboxset>
</div>
```

> **Note:**
>
> For accessibility, the `oj-checkboxset` and `oj-radioset` components require that you set the `labelled-by` attribute on the component element. For additional information about creating accessible Oracle JET components, see Using the Accessibility Features of Oracle JET Components.

The code that defines the `currentColor` value is defined in the `checkboxsetModel()` function, shown below.

```
require(['ojs/ojcore', 'knockout', 'jquery', 'ojs/ojknockout', 'ojs/ojcheckboxset',
'ojs/ojlabel'],
  function(oj, ko, $)
  {
    function checkboxsetModel()
    {
      var self = this;
      self.currentColor = ko.observable(["red"]);
    }

    var vm = new checkboxsetModel();

    $(document).ready(
      function()
      {
        ko.applyBindings(vm, document.getElementById('formId'));
      }
    );
  });
```

The Oracle JET Cookbook contains complete examples for configuring the `oj-checkboxset` and `oj-radioset` components. You can also find examples that show how to disable the component or one of its input elements, display the component inline, and test validation. For details, see Checkbox Sets and Radio Sets.

Working with Color Pickers

Use Oracle JET color picker components to select specific color values.

You can use Oracle JET `oj-color-palette` and `oj-color-spectrum` components to display a color palette with a predefined set of colors or to define a custom color value from a display that contains a saturation spectrum.

The `value` option of the color pickers is an object of the `oj.Color` type. You can create an `oj.Color` object instance from a CSS-like color string and then pass that instance.

The Oracle JET Cookbook contains the complete examples that you can use to create color pickers and define their behavior at Color Palette and Color Spectrum.

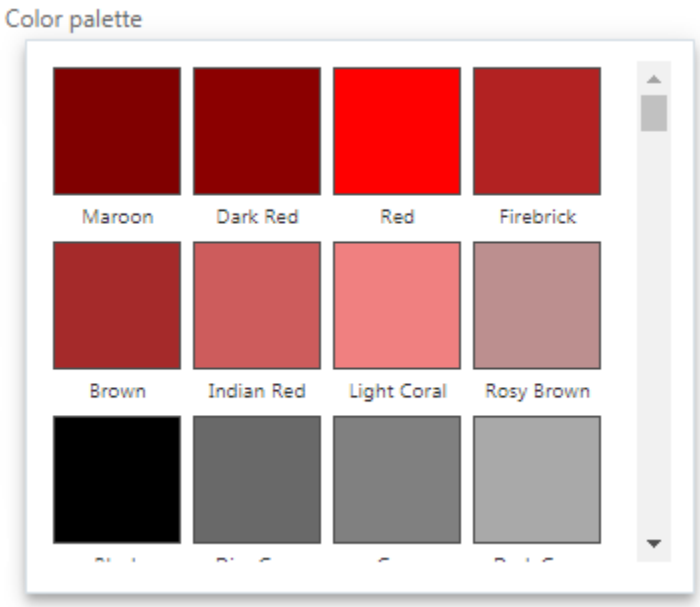
Topics:

- Working with oj-color-palette
- Working with oj-color-spectrum

Working with oj-color-palette

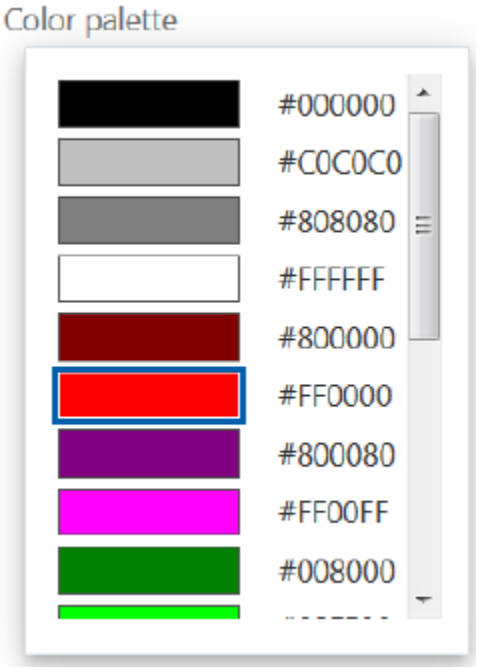
Use `oj-color-palette` to display a predefined set of colors from which a specific color can be selected. The `oj-color-palette` component supports a grid layout with or without labels and a list layout with or without labels.

The following image shows the `oj-color-palette` component using a grid layout and large color swatches with labels.



The swatch size of the `oj-color-palette` component using a grid layout can be large, small or extra small.

The following image shows the `oj-color-palette` component using a list layout with small color swatches with labels.



The swatch size of the `oj-color-palette` component using a list layout can be large or small.

To create the color palette, add the `oj-color-palette` element directly in the HTML file. The code sample below shows the markup for the color palette component that uses large color swatches with labels.

```
<div id="colorPaletteDemo">
```

```
  ... contents omitted

  <oj-label id="mainlabelid">Color palette</oj-label>
  <div class="demo-palette-panel oj-panel oj-panel-shadow-lg">
      <oj-color-palette class="demo-palette-picker" labelled-by="mainlabelid"
          palette="[[mypalette]]"
          swatch-size="[[swatchSize]]"
          label-display="[[labelDisplay]]"
          layout="grid"
          value="{{colorValue}}">
      </oj-color-palette>
  </div>
</div>
```

The colorValue is defined in the view model as follows:

```
require(['ojs/ojcore', 'knockout', 'jquery', 'ojs/ojknockout', 'ojs/ojcolorpalette',
'ojs/ojcolor', 'ojs/ojvalidation-base', 'ojs/ojbutton', 'ojs/ojlabel'],
 function(oj, ko, $)
 {
   function Model()
   {
      var self = this;
      self.colorValue = ko.observable(oj.Color.BISQUE);
      self.swatchSizes = ["lg", "sm", "xs"];
      self.labelDisplays = ["auto", "off"];
      self.swatchSize = ko.observable(self.swatchSizes[0]);
      self.labelDisplay = ko.observable(self.labelDisplays[0]);", "off"];
      self.swatchSize = ko.observable(self.swatchSizes[0]);
      self.labelDisplay = ko.observable(self.labelDisplays[0]);

      ... remaining contents omitted
```

To set a component to disabled, you set the `disabled` option in the markup.

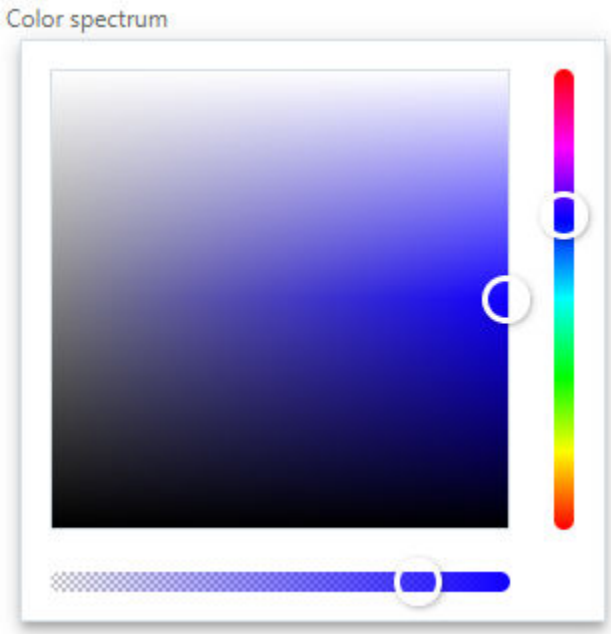
```
<div id="colorPaletteDemo">

   ... contents omitted

  <oj-label id="mainlabelid">Color palette</oj-label>
  <div class="demo-palette-panel oj-panel oj-panel-shadow-lg">
    <oj-color-palette class="demo-palette-picker" labelled-by="mainlabelid"
        palette="[[mypalette]]"
        swatch-size="lg"
        label-display="auto"
        layout="grid"
        value="{{colorValue}}"
        disabled="[[paletteDisabled]]">
    </oj-color-palette>
  </div>
</div>
```

For additional information about adding an Oracle JET component to your page, see Color Palette.

Working with oj-color-spectrum

Use `oj-color-spectrum` to display a saturation spectrum with hue and opacity sliders from which you can retrieve a custom color value.

The following image shows the `oj-color-spectrum` component.



To create the color spectrum, you can add the `oj-color-spectrum` element directly in the HTML file.

```
<div id="colorSpectrumDemo">

       ... contents omitted

  <oj-label id="mainlabelid">Color spectrum</oj-label>
  <div  class="demo-color-panel oj-panel oj-panel-shadow-lg">
    <oj-color-spectrum class="demo-color-spectrum" labelled-by="mainlabelid"
      value="{{colorValue}}"
      on-value-changed="[[updatePreviewColor]]"
    </oj-color-spectrum>
  </div>
</div>
```

You can disable the component. To set a component to disabled, you set the `disabled` option in the markup.

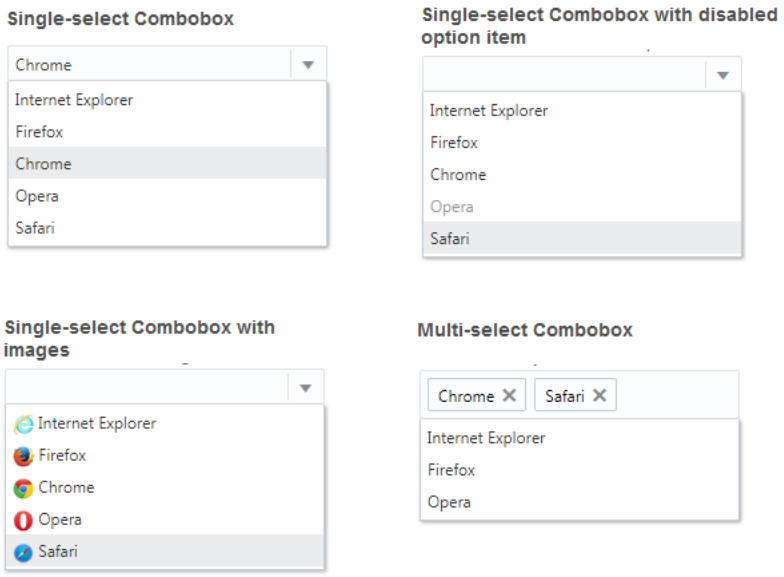
```
<div id="colorSpectrumDemo">

 ... contents omitted

 <oj-label id="mainlabelid">Color spectrum</oj-label>
  <div  class="demo-color-panel oj-panel oj-panel-shadow-lg">
    <oj-color-spectrum class="demo-color-spectrum" labelled-by="mainlabelid"
      value="{{colorValue}}"
      disabled="[[spectrumDisabled]]">
    </oj-color-spectrum>
  </div>

</div>
```

For additional information about adding an Oracle JET component to your page, see Color Spectrum.

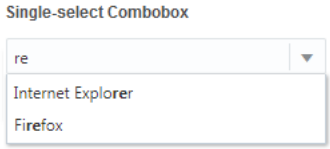
Working with Comboboxes

You can use the Oracle JET `oj-combobox-one` component or the `oj-combobox-many` component to display dropdown lists that are themable and WAI-ARIA compliant. The combobox component enhances the HTML input element and supports single-select, multi-select, free text input, and search filtering.

The image below shows three single-select comboboxes and one multi-select combobox. In this example, the first combobox displays the default dropdown list. The other single-select combo boxes illustrate a disabled item option and items that include images with the item text.



The user can select one of the items from the dropdown list or erase the current value and enter text in the input field to search the list of available options. The user can also enter non-matching text to add a new item. In the example below, the user typed `re` which matches the Internet Explorer and Firefox list items.



> **Tip:**
>
> You can use the Oracle JET `oj-select-one` component or the `oj-select-many` component to create read-only dropdown lists. For additional information, see Working with Select.

To create the combobox, you can use the `oj-combobox-one` element or the `oj-combobox-many` element. Use the `oj-option` component to display the options in the form of a data list in the combobox. The code sample below shows the markup for the single-select combobox with all option items enabled.

```
<div id="form1">
    <oj-label for="combobox">Single-select Combobox</oj-label>
  <oj-combobox-one id="combobox" value="{{val}}"
    style="max-width:20em">
    <oj-option value="Internet Explorer">Internet Explorer</oj-option>
        <oj-option value="Firefox">Firefox</oj-option>
        <oj-option value="Chrome">Chrome</oj-option>
        <oj-option value="Opera">Opera</oj-option>
        <oj-option value="Safari">Safari</oj-option>
  </oj-combobox-one>
  <div>
    <br/>
    <oj-label for="curr-value" class="oj-label">Curr value is: </oj-label>
    <span id="curr-value" data-bind="text: val"></span>
  </div>
</div>
```

The code that defines the `val` value is defined in the `ValueModel()` constructor function, shown below.

```
require(['ojs/ojcore', 'knockout', 'jquery', 'ojs/ojknockout', 'ojs/
ojselectcombobox'],
function(oj, ko, $)
{
  $(
    function()
    {
      function ValueModel() {
          this.val = ko.observableArray(["Chrome"]);
      }
      ko.applyBindings(new ValueModel(), document.getElementById('form1'));
    }
  );
});
```

You can initialize the combobox with the `options` array. Set the element's `options` attribute to a knockout `observableArray`. The array contains objects with the `value` and `label` fields in string format. Group data is also supported by specifying the `label` and `children`, which is an array of options inside the group. You can also redefine keys used in the array by specifying them in the `options-keys`. The code below defines the options binding for a multi-select combobox.

```
require(['ojs/ojcore', 'knockout', 'jquery', 'ojs/ojknockout', 'ojs/
ojselectcombobox'],
function(oj, ko, $)
{
  function comboboxModel () {
    this.optionsKeys1 = {label: 'regions', children: 'states', childKeys: {value:
'state_abbr', label: 'state_name'}};
    this.optionsKeys2 = {label: 'regions', children: 'states', childKeys: {value:
'state_abbr', label: 'state_name',
       children: 'cities', childKeys: {value: 'city_abbr', label: 'city_name'}}};
    this.browsers = ko.observableArray([
        {value: 'Internet Explorer', label: 'Internet Explorer'},
        {value: 'Firefox',    label: 'Firefox'},
        {value: 'Chrome',   label: 'Chrome'},
```

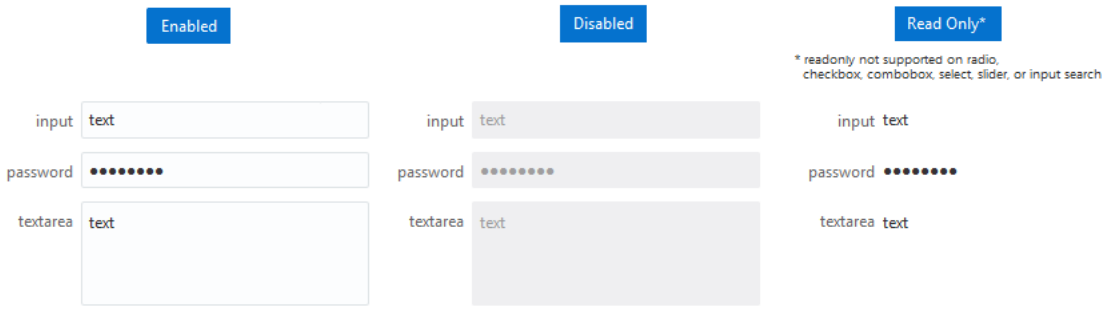
```
    {value: 'Opera',    label: 'Opera', disabled: true},
    {value: 'Safari',   label: 'Safari'}
]);

this.groupData = ko.observableArray([
  {label: "Alaskan/Hawaiian Time Zone",
    children: [
      {value: "AK", label: "Alaska"},
      {value: "HI", label: "Hawaii"}
  ]},
  {label: "Pacific Time Zone",
    children: [
      {value: "CA", label: "California"},
      {value: "NV", label: "Nevada"},
      {value: "OR", label: "Oregon"},
      {value: "WA", label: "Washington"}
  ]}
]);

this.groupDataWithKeys = ko.observableArray([
  {regions: "Alaskan/Hawaiian Time Zone",
    states: [
      {state_abbr: "AK", state_name: "Alaska"},
      {state_abbr: "HI", state_name: "Hawaii"}
  ]},
  {regions: "Pacific Time Zone",
    states: [
      {state_abbr: "CA", state_name: "California"},
      {state_abbr: "NV", state_name: "Nevada"},
      {state_abbr: "OR", state_name: "Oregon"},
      {state_abbr: "WA", state_name: "Washington"}
  ]}
]);

this.triLevelGroupData = ko.observableArray([
  {regions: "Alaskan/Hawaiian Time Zone",
    states: [
      {state_abbr: "AK", state_name: "Alaska",
        cities: [
          {city_abbr: "AN", city_name: "Anchorage"}
      ]},
      {state_abbr: "HI", state_name: "Hawaii",
        cities: [
          {city_abbr: "HO", city_name: "Honolulu"},
          {city_abbr: "HL", city_name: "Hilo"}
      ]}
  ]},
  {regions: "Pacific Time Zone",
    states: [
      {state_abbr: "CA", state_name: "California",
        cities: [
          {city_abbr: "SF", city_name: "San Francisco"},
          {city_abbr: "LA", city_name: "Los Angeles"}
      ]},
      {state_abbr: "NV", state_name: "Nevada",
        cities: [
          {city_abbr: "LV", city_name: "Las Vegas"}
      ]},
      {state_abbr: "OR", state_name: "Oregon",
        cities: [
          {city_abbr: "PL", city_name: "Portland"},
```

```
                {city_abbr: "BD", city_name: "Bend"}
            ]},
            {state_abbr: "WA", state_name: "Washington",
              cities: [
                {city_abbr: "ST", city_name: "Seattle"},
                {city_abbr: "SK", city_name: "Spokane"}
            ]}
        ]}
      ]);
    }
    $(
      function() {
        ko.applyBindings(new comboboxModel(),
document.getElementById('form1'));
      }
    );
});
```

The Oracle JET Cookbook contains complete examples for configuring the single-select and multi-select comboboxes at . You can also find examples for setting the width, handling events, adding new entries, and including images with the list items.

Working with Form Controls

Oracle JET components that support input, such as the `oj-input-text` component, provide form controls that you can use to indicate that a component is disabled or, in some cases, read only. When the component is disabled, the input fields appear grayed out, and keyboard navigation is also disabled. When the component is set to read only, the input field is not displayed, and keyboard navigation is disabled as well.



To set a component to disabled or read only, you set the `disabled` or `readOnly` option in the markup and specify the method that will be called when the component is marked `disabled` or `readOnly`. The following code sample shows the markup for the `oj-input-text` component.

```
<div class="oj-flex">
  <div class="oj-flex-item">
    <oj-label for="inputcontrol1">input</oj-label>
  </div>
  <div class="oj-flex-item">
    <oj-input-text id="inputcontrol1" placeholder="placeholder text"
value='{{placeholder()? null : "text"}}'
          disabled="[[disableFormControls()]]" readonly="[[readonlyFormControls]]"
          messages-custom="{{messages}}"></oj-input-text>
```

```
  </div>
</div>
```

The `disableFormControls()` and `readonlyFormControls()` methods set the Knockout observable to `false`.

```
require(['ojs/ojcore', 'knockout', 'jquery', 'ojs/ojknockout', 'ojs/ojinputtext'],
  function(oj, ko, $)
  {
    function StateModel() {
      this.disableFormControls = ko.observable(false);
      this.readonlyFormControls = ko.observable(false);
    }
    $(document).ready(
      function()
      {
        ko.applyBindings(new StateModel(),
        document.getElementById('form-container'));
      }
    );
  });
```

> **Note:**
>
> You can also set `disabled` as an attribute on the input element. If you use this method, then `disabled` will only be picked up at component creation. Changing the native input element's disabled state after creation will have no effect.

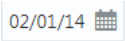
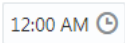
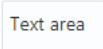
The Oracle JET Cookbook at Form Controls includes the complete example for the `oj-input-text` component as well as the other Oracle JET components that support `disabled` and `readOnly` options. In addition, you can find examples for using placeholder text and controlling the form's width and height.

Working with Form Layout Features

Oracle JET provides style classes that you can use to create form layouts that adjust to the size of the user's screen. For additional information, see Responsive Form Layouts. For examples that illustrate best practices for form layout in Oracle JET applications, see Form Layouts.

Working with Input Components

The Oracle JET input components enhance browser input elements. Enhancements include support for custom validation and conversion, accessibility, internationalization, and more. The Forms page in the Oracle JET Cookbook includes examples for working with the following Oracle JET input components.

Oracle JET Component	Image	HTML5 Element
`oj-input-date`		`<oj-input-date id="date" value={{value}}> </oj-input-date>`
`oj-input-date-time`		`<oj-input-date-time id="dateTime" value={{value}}> </oj-input-date-time>`
`oj-input-number`		`<oj-input-number id="inputnumber-id" max= "[[max]]" min= "[[min]]" step= "[[step]] value= "{{currentValue}}"> </oj-input-number>`
`oj-input-password`		`<oj-input-password id="password" value= "{{value}}"> </oj-input-password>`
`oj-input-text`		`<oj-input-text id="input-text" value="{{value}}"></oj-input-text>`
`oj-input-time`		`<oj-input-time id="time" value={{value}}> </oj-input-time>`
`oj-text-area`		`<oj-text-area id="text-area" value="{{value}}"> </oj-text-area>`

The editable input components include converters and validators that you can customize as needed. For additional information, see Validating and Converting Input.

The components also support help messages that you can customize to provide user assistance in your application. For additional information, see Working with User Assistance.

Working with Labels

The `oj-label` component decorates the label text with a required icon and help icon. The user can interact with the help icon (on hover, on focus, etc) to display help description text or to navigate to an URL for more information.

To create a label, add the `oj-label` element directly in the HTML file. Use `for` on the `oj-label` element to point to the id of the JET Form component.

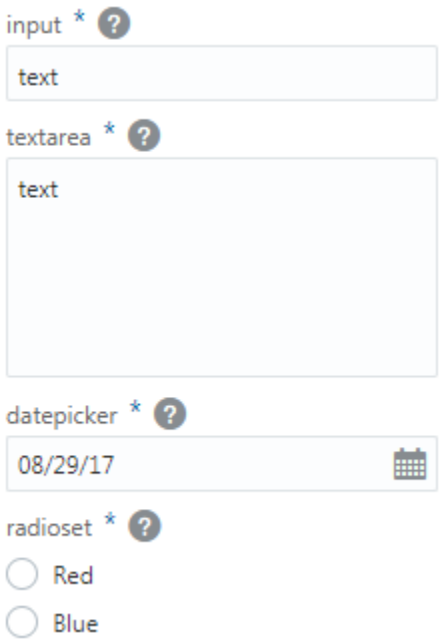
```
<oj-label for='input-text' show-required="[[isRequired]]"
help.definition='[[helpDef]]' help.source='[[helpSource]]'>input</oj-label>
    <oj-input-text id="input-text" required="[[isRequired]]" value="text"></oj-input-
text>
```

For accessibility, you must associate the `oj-label` component to its JET form component. For most JET form components you can do this using the `oj-label`'s `for` attribute and the JET form component's `id` attribute.

For a few JET form components (`oj-radioset`, `oj-checkboxset`, `oj-color-palette`, and `oj-color-spectrum`), you must associate the `oj-label` component to its JET form component using the `oj-label`'s `id` attribute and the JET form component's `labelled-by` attribute. For information, see the example below.

```
<oj-label id="radiosetlabel" show-required="[[isRequired]]"
help.definition='[[helpDef]]' help.source='[[helpSource]]'>radioset</oj-label>
<oj-radioset id="radioSetId"  required="[[isRequired]]" labelled-by="radiosetlabel">
  <oj-option name="color" value="red">Red</oj-option>
  <oj-option name="color" value="blue">Blue</oj-option>
</oj-radioset>
```

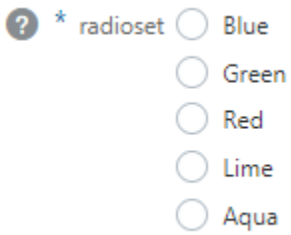
Labels are top aligned by default, following best practices for mobile devices.



You can modify the labels to display inline by adding the `oj-label-inline` class to the `oj-label` element when you don't want to use responsive design classes.

```
<div class='oj-form'>
  <div class='oj-flex-bar'>
    <div class='oj-flex-bar-start'>
      <oj-label id="radiosetlabel" show-required="[[isRequired]]" class="oj-label-
inline"
       help.definition='[[helpDef]]' help.source='[[helpSource]]'>radioset</oj-label>
    </div>
    <div class='oj-flex-bar-middle'>
      <oj-radioset id="radioSetId" labelled-by="radiosetlabel"
required='[[isRequired]]'>
        <oj-option id="blueopt" value="blue" name="rbb">Blue</oj-option>
        <oj-option id="greenopt" value="green" name="rbb">Green</oj-option>
        <oj-option id="redopt" value="red" name="rbb">Red</oj-option>
        <oj-option id="limeopt" value="lime" name="rbb">Lime</oj-option>
        <oj-option id="aquaopt" value="aqua" name="rbb">Aqua</oj-option>
      </oj-radioset>
    </div>
  </div>
</div>
```

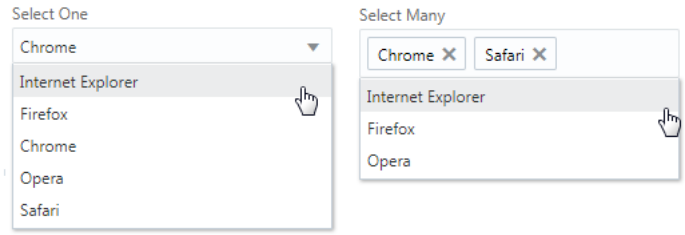
When the user runs the page, the label for the `oj-radioset` component displays inline.



The Oracle JET Cookbook includes additional examples for using help and required modifiers. In addition, the cookbook contains examples for displaying access keys, making multiple labels on one field accessible, and making multiple fields on one label accessible. For details, see Labels.

Working with Select

You can use the Oracle JET `oj-select-one` component or the `oj-select-many` component to display read-only, dropdown lists that are themable and WAI-ARIA compliant. The `oj-select` component enhances the HTML select element and supports single and multiple selection with search filtering.

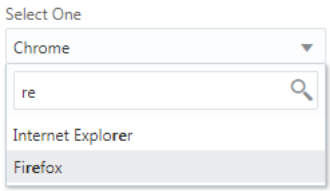


The dropdown list displays when the user does one of the following:

- Clicks on the select box.
- Sets focus on the select box and starts typing or presses the Enter, Arrow Up, or Arrow Down key.

If the number of options is less than the`minimumResultsForSearch` value, then by default the search box is not displayed when the dropdown is open. However, if the user starts typing when the select box is in focus, the dropdown will be open along with the search box displayed.

For example, there are five items in the dropdown list shown above so a search box is not displayed by default.



> **Tip:**
>
> You can also use the Oracle JET `oj-select-one` element and the`oj-select-many` element to create dropdown lists that support single and multiple selection respectively. The `oj-select` component also supports user input to add items to the dropdown list. For information, see Working with Comboboxes.

The code sample below shows the markup used to create the multi-select component shown in this section.

```
<div id="form1">
  <oj-label for="multiSelect">Select Many</oj-label>
  <oj-select-many id="multiSelect" value={{val}} style="max-width:20em">
    <oj-option value="IE">Internet Explorer</oj-option>
    <oj-option value="FF">Firefox</oj-option>
    <oj-option value="CH">Chrome</oj-option>
    <oj-option value="OP">Opera</oj-option>
    <oj-option value="SA">Safari</oj-option>
  </oj-select-many>
  <div>
    <br/>
    <oj-label for="curr-value">Current selected value is  </oj-label>
    <span id="curr-value" data-bind="text: ko.toJSON(val)"></span>
  </div>
</div>
```

> **Note:**
>
> For accessibility, the select component requires that you set the `for` attribute on the label element to point to the id of the `oj-select-one` element or the `oj-select-many` element. For information about creating accessible Oracle JET components, see Using the Accessibility Features of Oracle JET Components.

The code sample also defines `max-width` style attribute. Use `max-width` instead of `width` to ensure that the component adjusts its width automatically when the display width changes.

The code that defines the `val` value is defined in the `ValueModel()` function, shown below. `val` is defined as a Knockout observable, with its initial values set to `Chrome` and `Safari`.

```
require(['ojs/ojcore', 'knockout', 'jquery', 'ojs/ojknockout', 'ojs/
ojselectcombobox'],
function(oj, ko, $)
{
  $(
    function()
    {
      function ValueModel() {
        this.val = ko.observableArray(["CH","SA"]);
      }
      ko.applyBindings(new ValueModel(), document.getElementById('form1'));
    }
  );
});
```

You can populate an `oj-select-one` component or an`oj-select-many` component with the `options` array. Set the element's `options` attribute to a knockout `observableArray`. The array contains objects with `value` and `label` fields in string format. Group data is also supported by specifying the `label` and `children` , which is an array of options inside the group. You can also redefine keys used in the array by specifying them in the `options-keys`. The code below defines the options binding for the multi-select component.

```
require(['ojs/ojcore', 'knockout', 'jquery', 'ojs/ojknockout', 'ojs/
ojselectcombobox'],

function(oj, ko, $)
{
  function selectModel () {
    //simple select
    this.selectVal = ko.observableArray(['Chrome']);
    this.browsers = ko.observableArray([
      {value: 'Internet Explorer', label: 'Internet Explorer'},
      {value: 'Firefox',  label: 'Firefox'},
      {value: 'Chrome',   label: 'Chrome'},
      {value: 'Opera',    label: 'Opera'},
      {value: 'Safari',   label: 'Safari'}
    ]);

    //group
```

```
this.groupVal = ko.observableArray(['CA']);
this.groupData = ko.observableArray([
  {label: "Alaskan/Hawaiian Time Zone",
   children: [
      {value: "AK", label: "Alaska"},
      {value: "HI", label: "Hawaii"}
   ]},
  {label: "Pacific Time Zone",
   children: [
      {value: "CA", label: "California"},
      {value: "NV", label: "Nevada"},
      {value: "OR", label: "Oregon"},
      {value: "WA", label: "Washington"}
   ]}
]);

//tri level group
this.triLevelVal = ko.observableArray(['SF']);
this.triLevelGroupData = ko.observableArray([
  {label: "Alaskan/Hawaiian Time Zone",
   children: [
      {value: "AK", label: "Alaska",
       children: [
            {value: "AN", label: "Anchorage"}
            ]},
      {value: "HI", label: "Hawaii",
       children: [
            {value: "HO", label: "Honolulu"},
            {value: "HL", label: "Hilo"}
            ]}
   ]},
  {label: "Pacific Time Zone",
   children: [
      {value: "CA",
       label: "California",
       children: [
            {value: "SF", label: "San Francisco"},
            {value: "LA", label: "Los Angeles"}
            ]},
      {value: "NV", label: "Nevada",
       children: [
            {value: "LV", label: "Las Vegas"}
            ]},
      {value: "OR", label: "Oregon",
       children: [
            {value: "PL", label: "Portland"},
            {value: "BD", label: "Bend"}
            ]},
      {value: "WA", label: "Washington",
       children: [
            {value: "ST", label: "Seattle"},
            {value: "SK", label: "Spokane"}
            ]}
   ]}
]);

//option keys
this.optionsKeys = {label: 'regions', children: 'states',
                    childKeys: {value: 'state_abbr', label: 'state_name'}};
this.groupKeysVal = ko.observableArray(['CA']);
this.groupDataWithKeys = ko.observableArray([
```

```
      {regions: "Alaskan/Hawaiian Time Zone",
       states: [
          {state_abbr: "AK", state_name: "Alaska"},
          {state_abbr: "HI", state_name: "Hawaii"}
       ]},
      {regions: "Pacific Time Zone",
        states: [
          {state_abbr: "CA", state_name: "California"},
          {state_abbr: "NV", state_name: "Nevada"},
          {state_abbr: "OR", state_name: "Oregon"},
          {state_abbr: "WA", state_name: "Washington"}
      ]}
    ]);
  }

  $(
    function() {
      ko.applyBindings(new selectModel(), document.getElementById("containerDiv"));
    }
  );
});
```

The Oracle JET Cookbook contains the complete example for configuring the `oj-select-one` and `oj-select-many` components. You can also find examples that show how to disable the component, group related components in the dropdown list, bind an event handler to the component's `ValueChanged` event, and more. For details, see Select.

Working with Sliders

A slider component displays a horizontal or vertical bar representing a numeric value. The Oracle JET `oj-slider` component enhances the HTML `input` element to provide a slider component that is themable and WAI-ARIA compliant.

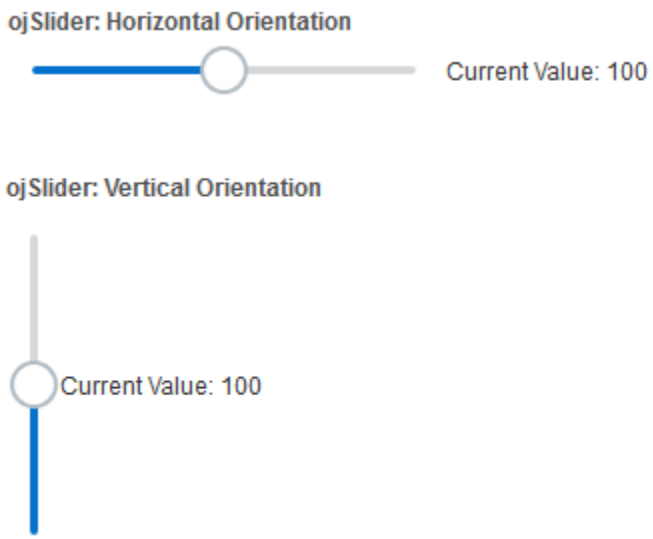
Topics

- About the oj-slider Component
- Creating Sliders
- Formatting Tips for oj-slider

About the oj-slider Component

The `oj-slider` component supports horizontal or vertical sliders with one thumb. The user can use gestures, mouse, or keyboard on the thumb to adjust the value within the slider's range.

The following image shows two `oj-slider` components. The two components illustrate sliders with horizontal and vertical orientation and their current value set to 100.



Creating Sliders

To create the `oj-slider` component, create an `oj-slider` element and assign an `id` to it. Create a HTML `oj-label` element to add the `for` attribute points to the `id` of the `oj-slider` component. Use the component's `min` and `max` attributes to set the slider range and the component's `value` attribute to set the thumb's initial value.

To add the `oj-slider` component to your page:

1. Create an `oj-slider` component using the `oj-slider` element. Set values for the slider's minimum, maximum, and step values.

```
<div id="slider-container">
     <oj-label for="slider"> slider component </oj-label>
     <oj-slider id="slider" value="{{value}}" min="[[min]]" max="[[max]]"
step="[[step]]"> </oj-slider>
</div>
```

> **Note:**
>
> For accessibility, the `div` container includes a label element that associates the `oj-slider` component with the label. See the oj-slider API documentation for accessibility details and associated keyboard and touch end user support.

2. Add code to your application script that sets the values for the attributes you specified in the previous step. The view model for the basic slider is shown below.

```
require(['ojs/ojcore', 'knockout', 'jquery', 'ojs/ojknockout', 'ojs/ojslider'],
function(oj, ko, $) {
  function SliderModel() {
    var self = this;
    self.max = ko.observable(200);
    self.min = ko.observable(0);
    self.value = ko.observable(100);
    self.step = ko.observable(10);
```

```
  }
  var sliderModel = new SliderModel();
  $(
    function() {
      ko.applyBindings(sliderModel, document.getElementById('slider-container'));
    }
  );
});
```

The `step` attribute indicates the size of the interval the slider takes between the `min` and `max` values.

> **Note:**
>
> The full specified value of the range (`max - min`) should be evenly divisible by `step`.

Formatting Tips for oj-slider

The `oj-slider` component provides options that allow you to customize a horizontal slider's width, change the slider's orientation to vertical, adjust a vertical slider's height, or disable it.

You may find the following tips helpful when working with sliders.

- To change the horizontal slider's width, enter a `style` value directly on the `oj-slider` element. The example below shows how you could specify an absolute width of 25 em on the `oj-slider` used in this section.

  ```
  <oj-label for="slider-id"> ojSlider component </oj-label>
    <br>
    <oj-slider id="slider-id"  value="{{value}}" min="[[min]]" max="[[max]]"
  step="[[step]]" style="max-width: 25em"> </oj-slider>
  ```

  To specify a width as a percentage of the maximum width available, set the max-width style to a percentage.

  ```
  style:'max-width:100%'
  ```

- To create a vertical slider, set the `oj-slider` component's orientation option to vertical.

  ```
  <oj-label for="slider-id"> ojSlider component </oj-label>
    <br>
    <oj-slider id="slider-id" orientation="vertical"  value="{{value}}"
  min="[[min]]" max="[[max]]" step="[[step]]"
  style="height: 150px"> </oj-slider>
  ```

- To change the vertical slider's height, set the `style` attribute directly on the `oj-slider` element.

  ```
  <oj-label for="slider-id"> ojSlider component </oj-label>
    <br>
    <oj-slider id="slider-id" orientation="vertical"  value="{{value}}"
  min="[[min]]" max="[[max]]" step="[[step]]"
  style="height: 150px"> </oj-slider>
  ```

- To display a slider that displays a value but does not allow interaction, set the component's `disabled` option to `true`.

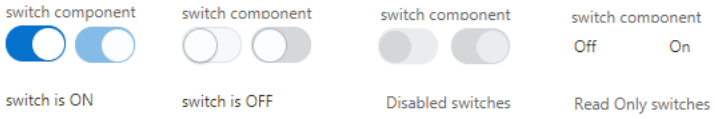
Cookbook Examples

The Oracle JET cookbook includes the complete examples shown in this section at Sliders. You can also find examples that show disabled sliders and sliders with icons on the bar to manipulate the thumb.

Working with Switches

A switch component displays two mutually exclusive choices to a user, typically ON or OFF. You can also disable a switch or make it read only using component attributes and display a switch inline using built-in style classes.

The following image shows an `oj-switch` component in on, off, disabled, and read only states. The display changes when the switch has focus to provide a visual clue to the user that the switch is selectable.



To create the `oj-switch` component, add the `oj-switch` element directly in the HTML file and assign it an id. Use the component's `value` attribute to set the initial state to `true` or `false`.

The following code sample shows the markup for the `oj-switch` shown in this section. For accessibility, the form container includes a label element where the `for` attribute points to the id of the `oj-switch` component. See the oj-switch API documentation for accessibility details and associated keyboard and touch end user support.

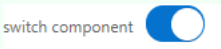
```
<div id="componentDemoContent" style="width: 1px; min-width: 100%;">
  <oj-label class="oj-label" for="switch">switch component</oj-label>
  <oj-switch id="switch" value="{{isChecked}}"></oj-switch><br/><br/>
  <span> switch is </span><span data-bind="text: ((isChecked()) ? 'ON' : 'OFF')"></
span>
</div>
```

The `isChecked` variable specified for the `oj-switch` component's `value` attribute is defined in the application's main script, shown below. In this example, the `isChecked` variable is a boolean set to `true` by a call to the Knockout `observable()` function.

```
define(['ojs/ojcore', 'knockout', 'ojs/ojswitch'],
  function(oj, ko) {
    function SwitchModel() {
      var self = this;
      self.isChecked = ko.observable(false);
    }
  return SwitchModel;
  }
);
```

Tip:

You can configure `oj-switch` to display inline with its label.



To configure an inline switch, add the `oj-label-inline` class to the switch's label.

```
<oj-label for="switch" class="oj-label-inline">switch component</oj-label>
```

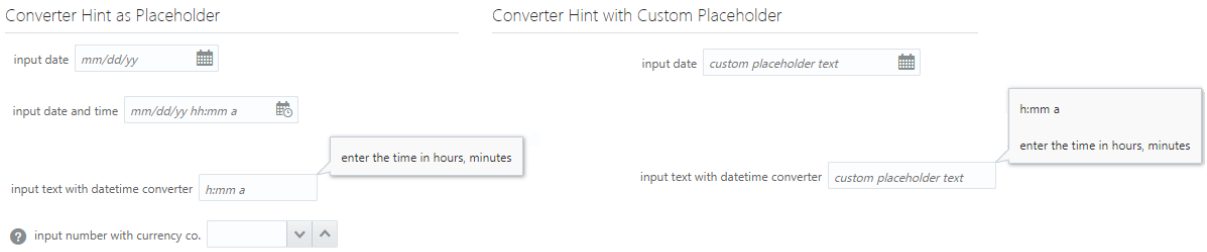
To disable the switch, set the component's `disabled` attribute to `true`. To make the switch read only, set the component's `readOnly` attribute to `true`.

The Oracle JET cookbook includes the complete example shown in this section at Switches. You can also find examples that implement disabled, read only, and inline switches.

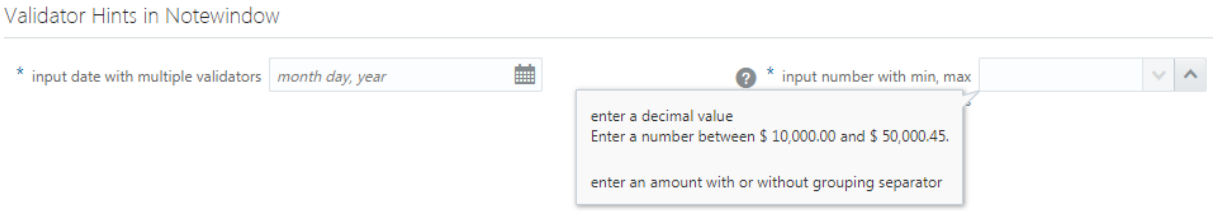
Working with Validation and User Assistance

The Oracle JET editable components display help, converter and validator hints, title, and messaging content by default. If needed, you can customize the defaults for displaying content.

The image below shows examples of converter hints and title. The example on the left uses default converter hints, and the example on the right uses custom placeholder text. The title text by default appears in the note window.



You can create multiple validators on a component. The image below shows an example that uses multiple validators on an `oj-input-number` component.



The Oracle JET Cookbook contains complete examples that you can use to customize help, converter and validator hints, and messaging content. See User Assistance.

For information about validating and converting input on the Oracle JET input components, see Validating and Converting Input. For information about using and customizing Oracle JET user assistance, see Working with User Assistance.

Working with Layout and Navigation

Use the Oracle JET `oj-accordian`, `oj-collapsible`, `oj-dialog`, `oj-flex*`, `oj-masonry-layout`, `oj-navigation-list`, `offCanvasUtils`, `oj-panel`, `oj-popup`, `oj-size*`, and `oj-tab-bar` components and patterns to control the initial data display and allow the user to access additional content by expanding sections, selecting tabs, or displaying dialogs and popups.

The Oracle JET Cookbook and JavaScript API Reference for Oracle® JavaScript Extension Toolkit (JET) include complete demos and examples for using the layout and navigation components, and you may also find the following tips and tricks helpful.

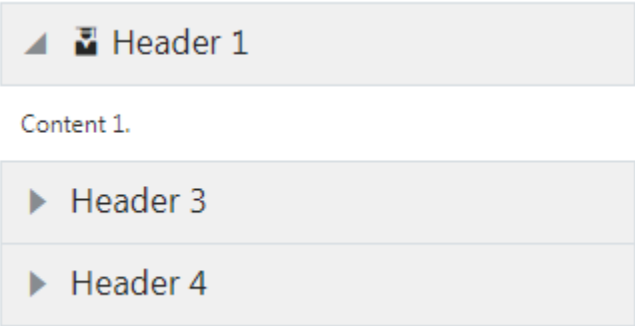
Topics:

- Working with Accordions
- Working with Collapsibles
- Working with Dialogs
- Working with Masonry Layouts
- Working with Nav Lists
- Working with offCanvasUtils
- Working with Panels
- Working with Popups
- Working with Tab Bars

For information about the flex layout (`oj-flex*`) and responsive grid (`oj-size`) classes, see Designing Responsive Applications

Working with Accordions

The Oracle JET `oj-accordion` component contains a list of `oj-collapsible` components.



You can define the JET Accordion by adding the `oj-accordion` element in the HTML file. Each child of the accordion must be an `oj-collapsible` element.

The following code shows a portion of the markup used to create the accordion shown in this section. In this example, the first collapsible (Header 1) is expanded.

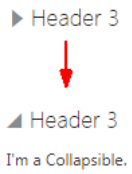
```
<oj-accordion id="accordionPage">
  <oj-collapsible id="c1">
    <span><span><span class="demo-icon-font demo-education-icon-24"></span> Header
1</span></span>
    <p class="oj-p">Content 1.</p>
  </oj-collapsible>
  <oj-collapsible id="c3">
    <span>Header 3</span>
    <p class="oj-p">Content 2.</p>
  </oj-collapsible>
  <oj-collapsible id="c4">
    <span>Header 4</span>
    <p class="oj-p">Content 3.</p>
  </oj-collapsible>
</oj-accordion>
```

The Oracle JET Cookbook at Accordions contains the sample code to create the accordion pictured in this section. You can also find examples for expanding multiple child elements and responding to events.

For additional information about working with the `oj-collapsible` component, see Working with Collapsibles.

Working with Collapsibles

The Oracle JET `oj-collapsible` component contains a header and a block of content that expands or collapses when clicked.



You can define the collapsible by directly adding the `oj-collapsible` element in the HTML file. Add the header and content elements as children of the `oj-collapsible` element. You can use any valid markup to represent the header and block of content. In this example, the collapsible is using the `h3` element to contain the header and a `p` element to contain the content.

```
<oj-collapsible id="collapsiblePage">
  <h3 id="h">Header 3</h3>
  <p id="c">I'm a Collapsible.</p>
</oj-collapsible>
```

To apply the binding, you can use the Knockout `applyBindings()` method and reference the id of the element that contains the `oj-collapsible` element.

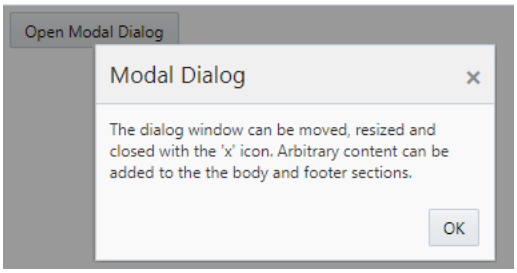
```
require(['knockout', 'jquery', 'ojs/ojcore', 'ojs/ojknockout', 'ojs/ojcollapsible'],
  function(ko, $) {
```

```
    $(
      function() {
        ko.applyBindings(null, document.getElementById('collapsiblePage'));
      }
    );
  }
);
```

The Oracle JET Cookbook Collapsibles demos contain the complete example for creating this collapsible as well as examples that show nested collapsibles, collapsibles with different header elements or borders, and event handling.

Working with Dialogs

You can use the Oracle JET `oj-dialog` component to display dialogs that are themable and follow WAI-ARIA authoring practices for accessibility. By default, the `oj-dialog` component renders modal dialogs which require user interaction before control returns to the calling window.



Typically, you create the dialog by directly adding the `oj-dialog` element in the HTML file. Define the dialog title in the `oj-dialog` element's `title` attribute. Add content to the dialog using the`slot="body"` and `slot="footer"` sections.

The `oj-dialog` element includes support for the header close icon and close handler, but you must add your own markup for creating the OK button in the `slot="footer"` section, as shown in the code sample below.

```
<div id="dialogWrapper">
  <oj-dialog style="display:none" id="modalDialog1" title="Modal Dialog">
    <div slot="body">
       The dialog window can be moved, resized and closed with the 'x' icon.
       Arbitrary content can be added to the the body and footer sections.
    </div>
    <div slot="footer">
      <oj-button id="okButton" data-bind="click: function() {
          document.querySelector('#modalDialog1').close();
        }">  OK
      </oj-button>
    </div>
  </oj-dialog>
  <oj-button id="buttonOpener"
    data-bind="click: function() {
      document.querySelector('#modalDialog1').open();
    }">
    Open Modal Dialog
  </oj-button>
</div>
```

You must also add code to handle the events on the OK and button opener buttons, in addition to the Knockout `applyBindings()` call that completes the component binding.

```
require(['ojs/ojcore', 'knockout', 'jquery', 'ojs/ojknockout', 'ojs/ojbutton', 'ojs/
ojdialog'],
function(oj, ko, $)
    {
      ko.applyBindings(null, document.getElementById('dialogWrapper'));
    });
```

> **Note:**
>
> Although the `oj–dialog` component is wrapped as a jQuery UI widget, its structure differs from the jQuery UI dialog widget. For additional detail, see the oj-dialog API documentation.

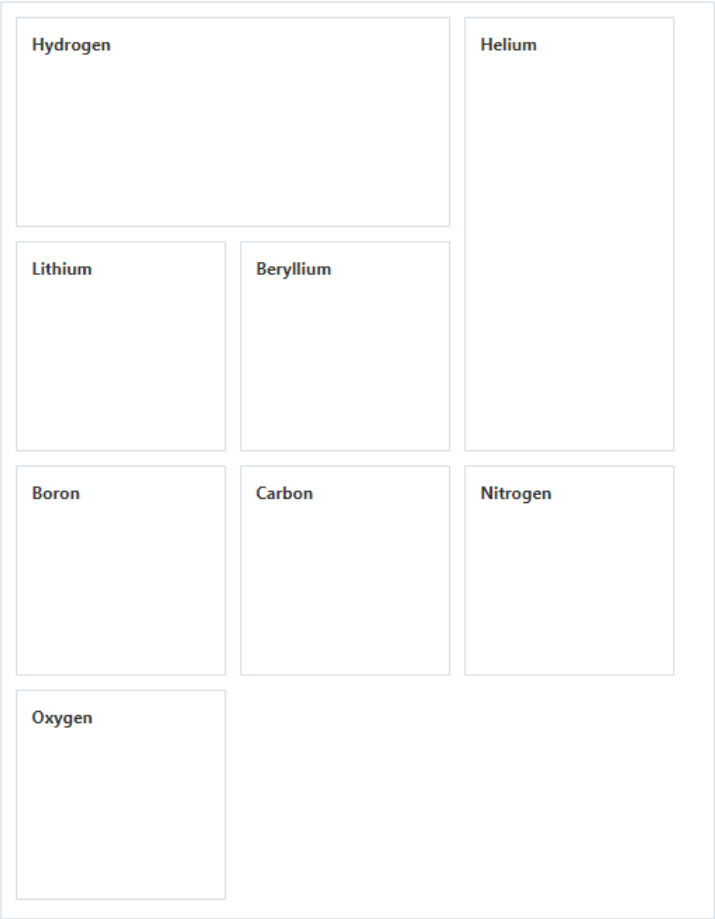
The Oracle JET Cookbook includes the complete code for this example at Dialogs. You can also find samples for creating a modeless `oj-dialog` and dialogs rendered with custom headers and other display options.

Working with Masonry Layouts

Use the Oracle JET `oj-masonry-layout` component to create a responsive grid of tiles containing arbitrary content. You can specify the size of each tile by defining the number of rows and columns that the tile will span relative to the other tiles in the layout.

Masonry layouts are advanced dashboard components that are useful when tiles are of different sizes because the component will shift tiles to fill in gaps, if possible. Masonry layouts are not intended for page layout, and you typically use them when you want to compact tiles in both horizontal and vertical directions at the same time. You can also use masonry layout when you want to allow users to operate on tiles, such as resizing, inserting, deleting, and reordering of tiles. Additionally, masonry layout animates the resulting layout changes to help preserve the context of the user.

The image below shows a basic masonry layout with eight tiles. In this example, the first tile in the layout occupies two columns and one row, the second tile occupies one column and two rows, and the remaining tiles occupy one column and one row.



Topics:

- Configuring Masonry Layouts
- Understanding the oj-masonry-layout Layout Process
- oj-masonry-layout Size Style Classes

Configuring Masonry Layouts

Define the masonry layout by directly adding `oj-masonry-layout` element in the HTML file. In the `oj-masonry-layout` element, add the group of sibling child elements that the `oj-masonry-layout` will manage. To specify the relative size of the tiles, add one of the `oj-masonry-layout` size style classes to each child element.

The code sample below shows the markup for the basic `oj-masonry-layout` shown in this section. The example uses the Knockout `foreach` binding to iterate through the child elements.

```
<div id="masonrylayout-basic-example">
  <div class="demo-scroll-container">
    <oj-masonry-layout id="masonryLayout">
      <!-- ko foreach: chemicals -->
      <div class="oj-panel demo-tile"
                  data-bind="text: name, css: sizeClass"></div>
      <!-- /ko -->
```

```
    </oj-masonry-layout>
  </div>
</div>
```

The size of each tile is determined by the value in the `sizeClass` property which is defined in the application's main script shown below.

```
require(['ojs/ojcore', 'knockout', 'jquery',
         'ojs/ojknockout', 'ojs/ojmasonrylayout'],
function(oj, ko, $)
{
  $(
    function()
    {
      function MyModel() {
        var self = this;

        self.chemicals = [
            { name: 'Hydrogen',
              sizeClass: 'oj-masonrylayout-tile-2x1' },
            { name: 'Helium',
              sizeClass: 'oj-masonrylayout-tile-1x2' },
            { name: 'Lithium',
              sizeClass: 'oj-masonrylayout-tile-1x1' },
            { name: 'Beryllium',
              sizeClass: 'oj-masonrylayout-tile-1x1' },
            { name: 'Boron',
              sizeClass: 'oj-masonrylayout-tile-1x1' },
            { name: 'Carbon',
              sizeClass: 'oj-masonrylayout-tile-1x1' },
            { name: 'Nitrogen',
              sizeClass: 'oj-masonrylayout-tile-1x1' },
            { name: 'Oxygen',
              sizeClass: 'oj-masonrylayout-tile-1x1' }
        ];
      }

      ko.applyBindings(new MyModel(),
        document.getElementById('masonrylayout-basic-example'));
    }
  );
});
```

The Oracle JET Cookbook at Masonry Layouts contains the complete example for the basic masonry layout shown in this section. You can also find demos that illustrate the different tile sizes, and masonry layouts that let you resize, reorder, and flip tiles.

> **Note:**
>
> The `oj-masonry-layout` component does not provide accessibility features such as keyboard navigation. It is the responsibility of the application developer to make the items in the layout accessible. For tips and additional detail, see the oj-masonry-layout API documentation.

Understanding the oj-masonry-layout Layout Process

`oj-masonry-layout` lays out its child tiles based on the size of the screen, the size of the tiles, and the order in which you define them.

To determine the layout, `oj-masonry-layout`:

- Processes the tiles in the order in which they originally appear in the DOM.
- Determines the number of columns to display based on the width of the `oj-masonry-layout` element and the width of a 1x1 tile.
- Determines the number of rows to display based on the number of columns and the number and sizes of tiles to lay out.
- Lays out the grid cells in a left-to-right, top-to-bottom order (or right-to-left, top-to-bottom order when the reading direction is right-to-left).

  Tiles will be positioned in the first empty cell in which they fit. This can result in empty cells in the layout. Subsequent tiles may fill those earlier gaps if they fit.

If the element is resized, `oj-masonry-layout` will redo the layout, and the number of columns and rows may change.

oj-masonry-layout Size Style Classes

`oj-masonry-layout` supports tile sizes ranging from one to three columns and one to three rows, as shown in the following table.

Style Class	Description
`oj-masonrylayout-tile-1x1`	A tile that spans one column and one row
`oj-masonrylayout-tile-1x2`	A tile that spans one column and two rows
`oj-masonrylayout-tile-1x3`	A tile that spans one column and three rows
`oj-masonrylayout-tile-2x1`	A tile that spans two columns and one row
`oj-masonrylayout-tile-2x2`	A tile that spans two columns and two rows
`oj-masonrylayout-tile-2x3`	A tile that spans two columns and three rows
`oj-masonrylayout-tile-3x1`	A tile that spans three columns and one row
`oj-masonrylayout-tile-3x2`	A tile that spans three columns and two rows

Working with Nav Lists

The Oracle JET `oj-navigation-list` component enhances the HTML list (`ul`) element to provide a themable, WAI-ARIA compliant component that displays a list of vertical or horizontal navigation links. You can configure the nav list to slide in and out of view, expand and collapse, and respond to changes in screen size.

Topics:

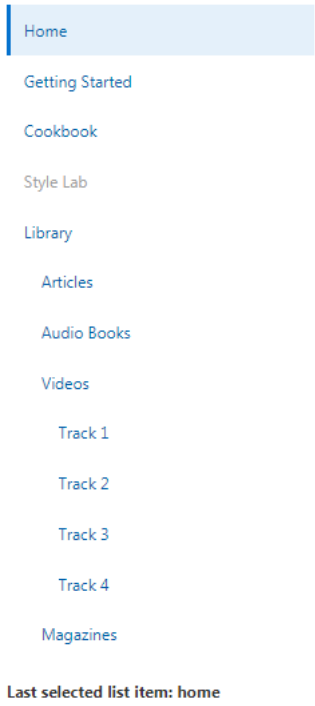
- Understanding Data Requirements for Nav Lists
- Working with Nav Lists and Knockout Templates

Understanding Data Requirements for Nav Lists

The data source for the `oj-navigation-list` component can be one of the following:

- Flat or hierarchical static HTML

  The following image shows a vertical nav list that displays static, hierarchical HTML.



The following code sample shows a portion of the markup used to create the `oj-navigation-list` element with hierarchical static content and the Oracle JET component binding. The markup uses `ul` and `li` HTML elements to define the nav list, with nested lists for the hierarchical data. The markup also specifies `selectedItem` as a Knockout observable in the component's `selection` attribute that will update when the user selects a list item.

```
<div id="navlistdemo">
  <p>
  <div id="navlistcontainer"  style="max-width:300px">
    <oj-navigation-list aria-label="Choose a navigation item"
selection="{{selectedItem}}">
      <ul >
        <li id="home" >
          <a href="#" >Home</a>
        </li>
        <li id="gettingStarted" >
          <a href="#" >Getting Started</a>
        </li>
        <li id="cookbook">
          <a href="#" >Cookbook</a>
        </li>
```

```
        <li id="stylelab" class="oj-disabled" >
          <a href="#" >Style Lab</a>
        </li>
        <li id="library" >
          <a href="#" >Library</a>
          <ul>
            <li id="articles">
              <a href="#" >Articles</a>
            </li>
            <li id="audios">
              <a href="#" >Audio Books</a>
            </li>
            <li id="videos">
              <a href="#"  >Videos</a>
              <ul>
                <li id="track1">
                  <a href="#" >Track 1</a>
                </li>
                <li id="track2">
                  <a href="#" >Track 2</a>
                </li>
                <li id="track3">
                  <a href="#" >Track 3</a>
                </li>
                <li id="track4">
                  <a href="#" >Track 4</a>
                </li>
              </ul>
            </li>
            <li id="magazines">
              <a href="#">Magazines</a>
            </li>
          </ul>
        </li>
      </ul>
    </oj-navigation-list>
  </div>
  <br>
  <div>
    <p class="bold">Last selected list item:
      <span data-bind='text: selectedItem'></span>
    </p>
  </div>
</div>
```

The code to apply the binding and define `selectedItem` is shown below. In this example, the initial value of the Knockout observable is set to `home`, and the nav list will initially display with the **Home** item selected.

```
require(['ojs/ojcore','knockout','jquery','ojs/ojknockout','ojs/
ojnavigationlist'],
  function(oj, ko, $)
  // this callback gets executed when all required modules are loaded
  {
    function ViewModel(){
      this.selectedItem = ko.observable("home");
    }
    var vm = new ViewModel();
    ko.applyBindings(vm, document.getElementById('navlistdemo'));
  }
);
```

- Oracle JET `TreeDataSource`, including `oj.JsonTreeDataSource` and `oj.CollectionTreeDataSource`.

  Typically, you use one of the Oracle JET `TreeDataSource` class when your list data contains groups. To use a `TreeDataSource`, specify the method that returns the tree data in the `data` attribute for the `oj-navigation-list` element.

  ```
  <div id="navlistdemo">
    <div style="max-width:300px">
      <oj-navigation-list
        drill-mode="sliding"
        selection="{{selectedListItem}}"
        data="[[dataSource]]"
        item.renderer="[[oj.KnockoutTemplateUtils.getRenderer('folder_template',
  true)]]">
      </oj-navigation-list>
    </div>
  </div>
  ```

- Oracle JET `TableDataSource`, including `oj.ArrayTableDataSource` and `oj.CollectionTableDataSource`

  Use `oj.ArrayTableDataSource` when the underlying data is a static array or Knockout `observableArray`. If you use an `observableArray`, the nav list will automatically react when items are added or removed from the array.

  Use `oj.CollectionTableDataSource` when `oj.Collection` is the model for the underlying data. The nav list will automatically react to model events from the underlying `oj.Collection`.

  To use a `TableDataSource`, specify the method that returns the table data in the `data` attribute for the `oj-navigation-list` element.

  ```
  <div id="navlistdemo">
    <div style="max-width:300px">
      <oj-navigation-list
        selection="{{selectedItem}}"
        data="[[dataSource]]"
        item.renderer="[[oj.KnockoutTemplateUtils.getRenderer('server_template',
  true)]]">
      </oj-navigation-list>
   </div>
  </div>
  ```

The Oracle JET Cookbook contains a number of `oj-navigation-list` examples, including ones that illustrate the use of each supported data source. For details, see Nav Lists.

Working with Nav Lists and Knockout Templates

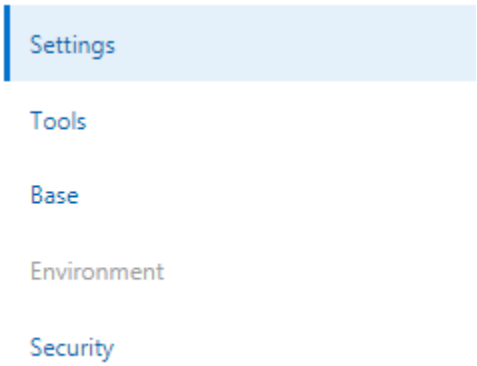
You can use a Knockout template to contain the markup for your list item content. Reference the name of the template in the `oj-navigation-list` element's `template` option.

The code sample below shows a portion of the markup and template for a nav list using an `oj.ArrayTableDataSource` object for its data. In this example, the template is named `server_template`.

```
<div id="navlistdemo">
  <div style="max-width:300px">
    <oj-navigation-list
      selection="{{selectedItem}}"
```

```
      data="[[dataSource]]"
      item.renderer="[[oj.KnockoutTemplateUtils.getRenderer('server_template',
true)]]">
    </oj-navigation-list>
</div>
</div>
<script type="text/html" id="server_template">
    <li data-bind="attr:{ id : $data['id']}, css:{'oj-
disabled' : $data['disabled']=='true'}">
        <a href="#" data-bind="text: $data['name']">
        </a>
    </li>
</script>
```

At runtime, the nav list iterates through the array and displays the content contained in `name` and applies styling for disabled items.

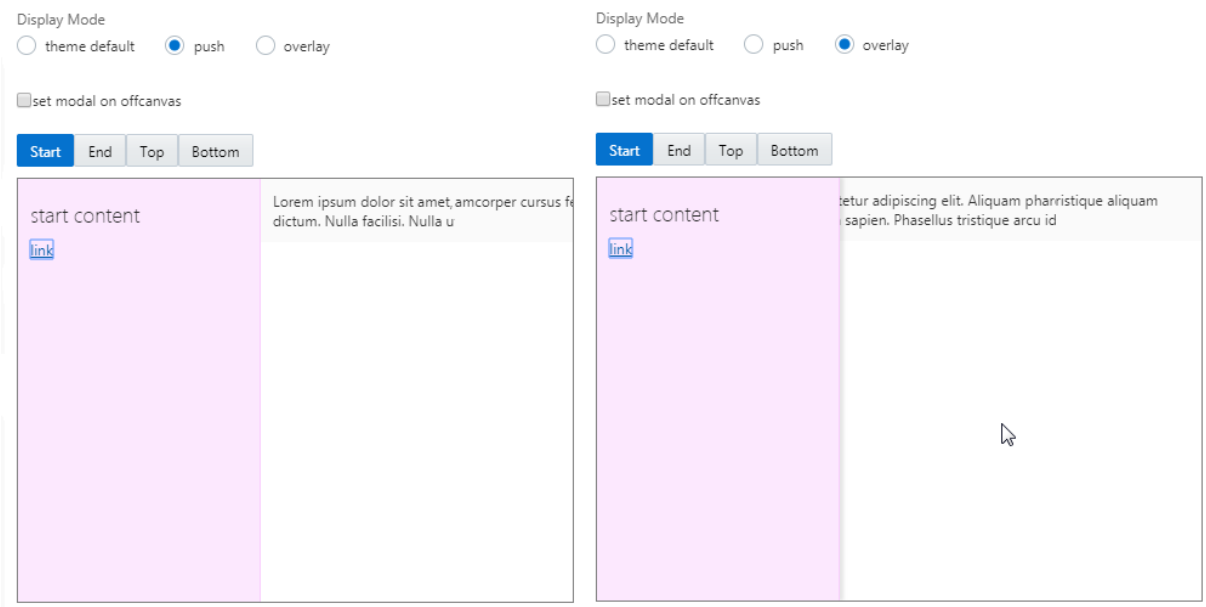


You can find the complete code sample for this nav list at Nav List (ArrayTableDataSource).

Working with offCanvasUtils

Oracle JET `offCanvasUtils` are useful if you want to store content in a region that is not part of the viewport. With the `offCanvasUtils`, you can configure the region to slide in and out of view at the start, end, top, or bottom of the viewport, either by pushing existing content out of view or by overlaying the region.

The following image shows two examples of regions configured to slide in at the start of the viewport. In the region configured for push, `offCanvasUtils` pushes aside as much of the viewport content as needed to display the region. In the region configured for overlay, the content is displayed over the main content, obscuring the content.



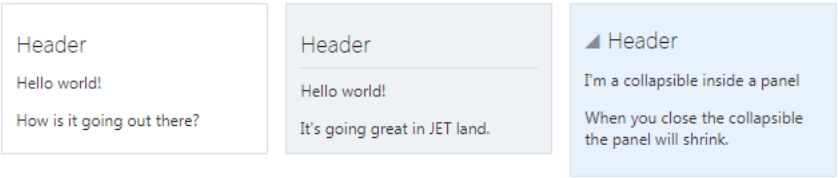
Configuring an Off-Canvas Partition

To configure an off-canvas partition, create a HTML `div` wrapper to contain both the content and the off-canvas partition and use `offCanvasUtils` to show, hide, or toggle the partition view.

The Oracle JET Cookbook Drawer Utilities demos contain the sample code that implements the off-canvas partitions shown in this section and links to the oj.OffcanvasUtils API documentation. You can also find examples that implement the off-canvas partitions as responsive design patterns that change their position from off screen to fixed inside the viewport based on a media query. For more information about responsive design and media queries, see Designing Responsive Applications.

Working with Panels

Panels use Oracle JET styles included with the Alta theme to display a bounded section on the page. You can customize the size, color, and content of the panel.



In this example, three panels are displayed with three different colors. The content for the first two panels is defined using the HTML header and paragraph elements. The content for the third panel is defined as an `oj-collapsible` element. The `oj-panel-*` classes are defined on the `div` elements that represent the panels.

```
<div id="panelPage">
  <div class="oj-flex">
```

```
    <div class="oj-panel oj-sm-margin-2x demo-mypanel">
      <h3>Header</h3>
      <p>Hello world!</p>
      How is it going out there?
    </div>

    <div class="oj-panel oj-panel-alt1 oj-sm-margin-2x demo-mypanel">
      <h3 class="oj-header-border">Header</h3>
      <p>Hello world!</p>
      It's going great in JET land.
    </div>

    <div class="oj-panel oj-panel-alt2 oj-sm-margin-2x demo-mypanel">
    <oj-collapsible id="collapsiblePage">
        <h3>Header</h3>
        <div>
          <p>I'm a collapsible inside a panel</p>
           When you close the collapsible the panel will shrink.
        </div>
    </oj-collapsible>
    </div>
  </div>
</div>
```

When panels are on the same row they will have equal height because they are in a `div` element defined with the `oj-flex` style class. You can change the default behavior by adding responsive classes to set alignment. For an example, see Flex Layouts - Align.

You can customize the size or display using styles, or you can create a custom style class. In this example, the panel is customized using the `demo-mypanel` class, shown below.

```
.demo-mypanel {
  width:  200px;
}
```

The Oracle JET Cookbook contains the complete code for this example at Panel Content. You can also find an example that shows the available panel colors at Panel Colors.

For additional information about using and customizing the Oracle JET themes included with Oracle JET, see Theming Applications. For more information about the `oj-collapsible` component, see Working with Collapsibles.

Working with Popups

The Oracle JET popup framework includes the `oj-popup` element that you can use to create popups on your page. To manage stacking order of `oj-popup` and all Oracle JET elements that use popups internally, such as `oj-dialog` and `oj-menu`, the popup framework uses the CSS `z-index` property and re-parents the popup DOM into a `zorder` container.

Topics:

- Working with oj-popup
- Working with the Oracle JET Popup Framework

Working with oj-popup

You can use the Oracle JET `oj-popup` component to create themable popups that follow WAI-ARIA authoring practices for accessibility. The image below shows a popup displayed with default options, but you can customize the popup's modality, position, tail, and chrome.

You can create the popup by directly adding the `oj-popup` element in the HTML file. Add the popup's content as the child of that element. For the image shown above, the popup is defined on the `oj-popup` element, with the content defined in the `span` element.

```
<oj-popup style="display:none" id="popup1">
  <span class="blink-rainbow">Hello World!!!</span>
</oj-popup>
```

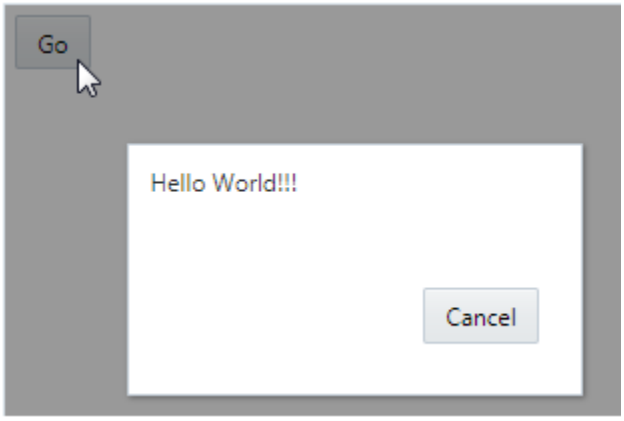
In this example, the popup displays when the user clicks **Go**. The **Go** button is defined as an `oj-button` element that references the popup's `id` attribute to display the popup when clicked.

```
<div id="popupWrapper">
  <oj-popup style="display:none" id="popup1">
    <span class="blink-rainbow">Hello World!!!</span>
  </oj-popup>
  <oj-button id="btnGo"
    data-bind="click: function(data)
    {
      document.querySelector('#popup1').open('#btnGo > button');
    }">
    Go
  </oj-button>
</div>
```

In this example, the popup's modality defaults to modeless which means that the popup will not block user input on the page behind the popup. You can change this to modal by setting the element's `modality` attribute as described in the oj-popup API documentation.

```
<oj-popup class="demo-popup" id="popup1"
  tail="none" position.my.horizontal="center" position.my.vertical="bottom"
  position.at.horizontal="center" position.at.vertical="bottom"
  position.of="window" position.offset.y="-10"
  modality="modal"
  data-bind="event:{'ojAnimateStart': startAnimationListener}">
```

When the user clicks the **Go** button now, the popup blocks user input of the page behind the popup with a blocking overlay pane.



The popup's default modality is defined in the `$popupModalityOptionDefault` SCSS variable and is specific to the chosen theme. For the Alta desktop theme, the SCSS variable is set to `null` which defaults to `modeless`, and the code sample below shows the generated CSS:

```
.oj-popup-option-defaults {
  font-family: "{}"; }
```

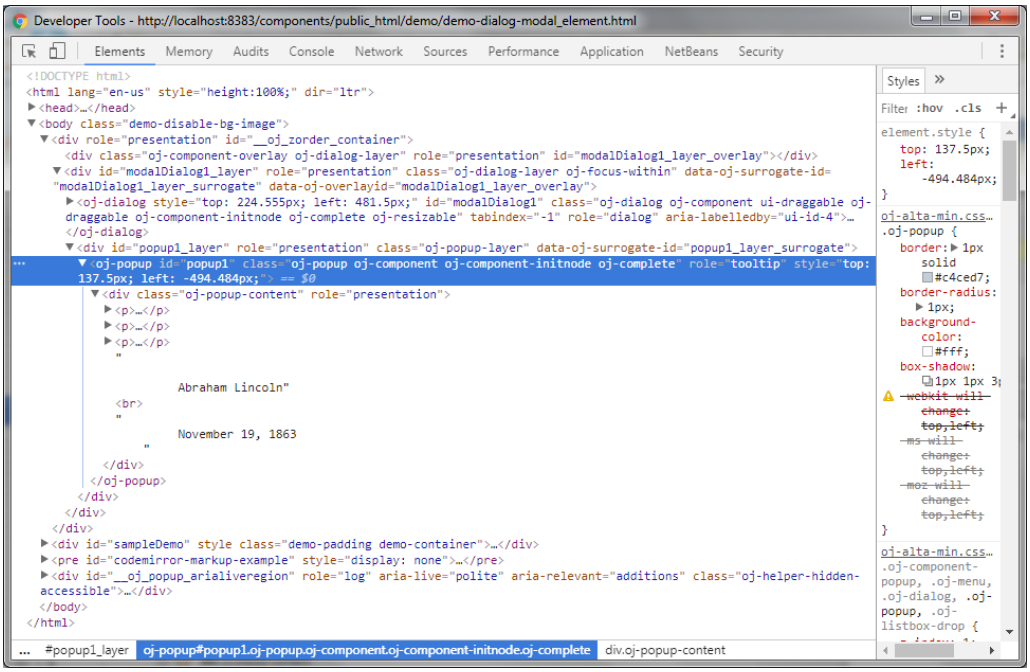
The Oracle JET Cookbook contains the complete example for this popup at Popups. You can also find examples that customize the popup and that use the popup as a tooltip for accessibility.

> **Note:**
>
> The `oj-popup` component defines default roles for accessibility and keyboard navigation. You can find details about the `oj-popup` component's accessibility support in the oj-popup API documentation. In addition, you can find some general recommendations about best practices for making the gestures that launch the popup accessible.

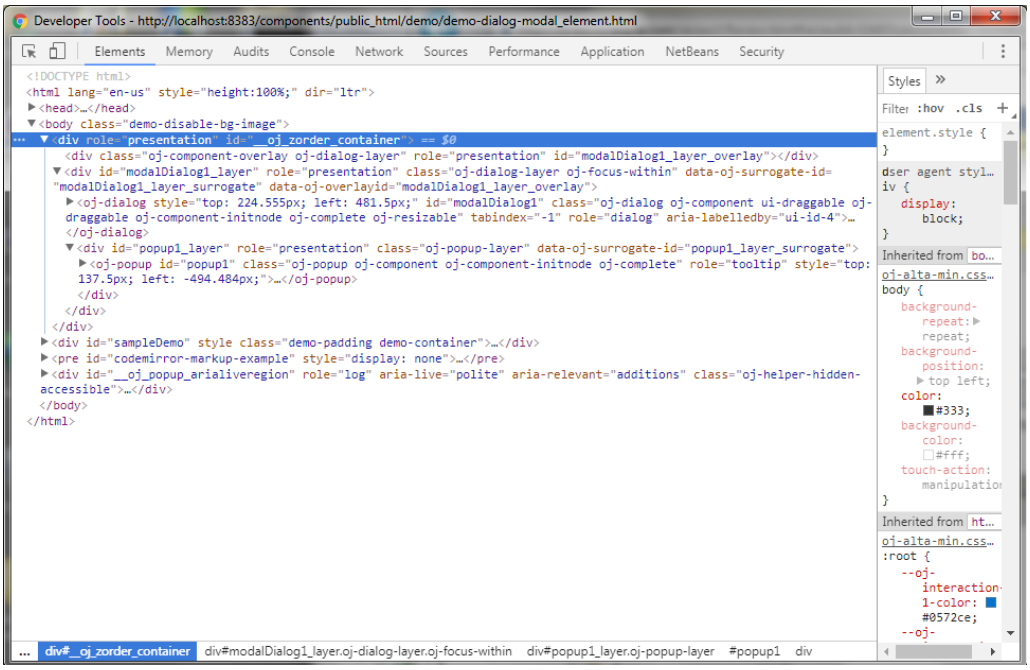
Working with the Oracle JET Popup Framework

The popup framework uses the CSS `z-index` property and re-parents the popup DOM to manage the stacking order of `oj-popup` and all Oracle JET components that use popups internally, such as `oj-dialog` and `oj-menu`. `z-index` defaults should be sufficient for most purposes, but you can modify them using SCSS variables to generate new CSS or modify the CSS styles directly.

When a popup of any type opens, Oracle JET re-parents it into a `zorder` container in the document `body` and re-parents it back to its original location when closed. For example, if you add the `oj-popup` element shown in Working with oj-popup to the main content in an Oracle JET Web Nav Drawer Starter Template, the browser DOM reflects its initial placement within the document body.



When the user clicks the **Go** button to display the popup, Oracle JET re-parents it into the `oj_zorder_container` as a direct child of the `body` element.



The purpose of re-parenting is to better manage the stacking context as it relates to how popups are used versus where they are defined in the document. The `zorder` container has a default root container that holds popups open. When the popup is re-parented, it is wrapped in a `div` containing the `oj-popup-layer` style and assigned a z-index weight. If there are multiple open popups on the page, each popup is re-

parented to the `zorder` container, and the active popup will be the popup with the highest z-index weight.

For example, at initial display, the `oj-popup's` layer is marked with the `oj-popup-layer` style which has a z-index value of 1000. Popups of the same type are assigned the same z-index values. If there are multiple popups open, the popup that has active focus will be assigned a greater z-index value of 1001, and the `oj-popup-layer.oj-focus-within` style is applied.

In most cases, you should never need to modify the CSS z-index defaults since the resulting behavior may be unpredictable. If needed, however, you can update the SCSS variables used to generate the application's CSS or modify the CSS styles directly. The following table shows the default CSS style selectors, SCSS variables, and z-index values.

CSS Selector	SCSS Variable	Z-Index Value
`oj-popup-layer`	`$popupZindex`	1000
`oj-popup-layer.oj-focus-within`	`$popupZindex` + 1	1001
`oj-popup-layer.oj-popup-tail-simple`	`$noteWindowZindex`	1030
`oj-popup-layer.oj-popup-tail-simple.oj-focus-within`	`$noteWindowZindex` + 1	1031
`oj-listbox-drop-layer`	`$popupZindex`	1000
`oj-menu-layer`	`$popupZindex`	1000
`oj-dialog-layer`	`$dialogZindex`	1050
`oj-dialog-layer.oj-focus-within`	`$dialogZindex` + 1	1051
`oj-component-overlay`	Uses z-index value of associated `oj-`***`popup_type`***`-layer`. For example, the z-index of `oj-dialog-layer` is 1050, and `oj-component-overlay` will use 1050 when the associated popup is a dialog.	Varies according to `oj-`***`popup_type`***`-layer`.

When popup elements are initially defined on the page, they are defined with a default z-index value of 1 using the CSS selectors shown in the following table. The root popup is absolutely positioned on the page in relation to its parent container. Setting the value to 1 ensures that the root popup's children will display above the popup.

Oracle JET Components	CSS Selector	SCSS Variable	Z-Index Value
`oj-menu`	`oj-menu`	`$defaultZindex`	1
`oj-popup`	`oj-popup`	`$defaultZindex`	1
`oj-dialog`	`oj-dialog`	`$defaultZindex`	1
`oj-combobox, oj-select`	`oj-listbox-drop`	`$defaultZindex`	1
`oj-input-date-time`, `oj-input-date`, `oj-input-time`	`oj-popup`	`$defaultZindex`	1
Editable elements using note windows for messaging	`oj-popup.oj-popup-tail-simple`	`$defaultZindex`	1

For additional information about Oracle JET's popup strategy, see the oj-popup API documentation.

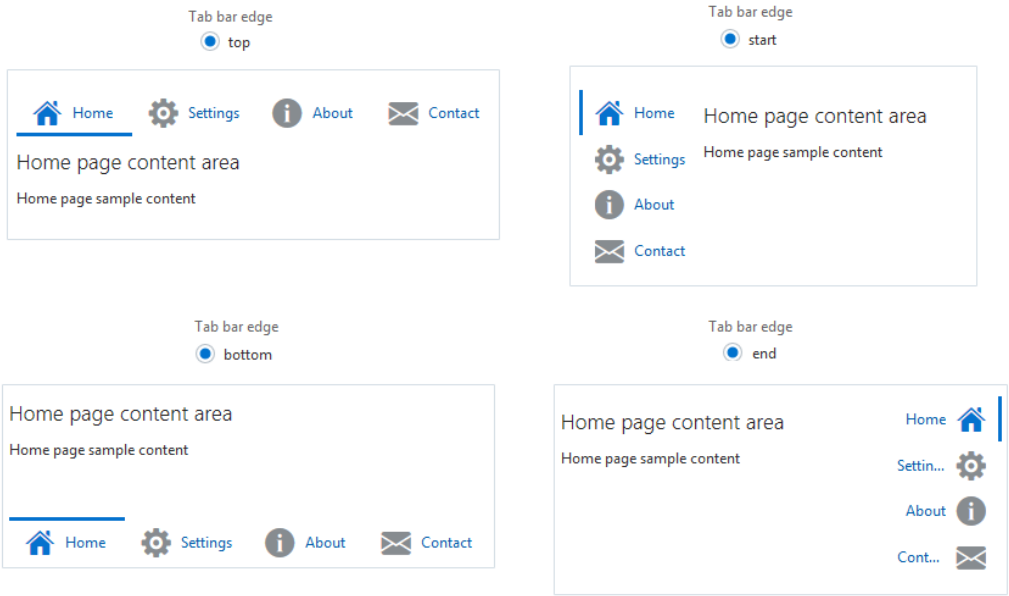
For additional information about Oracle JET's use of CSS and SCSS, see Theming Applications. For more information about the CSS `z-index` property, see http://www.w3.org/wiki/CSS/Properties/z-index.

Working with Tab Bars

Use the Oracle JET `oj-tab-bar` element to create themable, sortable, and removable tabs that toggle between related content sections.

> **Tip:**
>
> Tab bars are designed for toggling between content only. If you want to perform operations on the content within a tab, use `oj-toolbar` instead. See Working with Toolbars.

The Oracle JET tab bar enhances a HTML list element into a WAI-ARIA compliant, mobile friendly component with advanced interactive features. The tab bar is customizable, and you can configure the tabs to show icons, text, or both. The tab bar's position is also customizable, and you can display the tab bar on the top, bottom, or side (start or end) of the tab's content.



To create a tab bar, add the `oj-tab-bar` element to a HTML container element, typically a `div` element, on your page. Add the HTML unordered list element (`ul`) as a child to the tab bar, and add `li` child elements to the list for each tab. Each list item must include an anchor which contains the list item's text and optional icon.

The code sample below shows the markup for an `oj-tab-bar` that contains the four tabs shown in the image. The `edge` attribute specifies the position of the `oj-tab-bar`. In this example, it's set to `top` , and you can also set it to `bottom`, `start`, or `end`.

```
<div id="demo-container" class="oj-flex demo-edge-top">
  <oj-tab-bar id="hnavlist"
              edge="top"
              selection="{{selectedItem}}">
    <ul>
      <li id="home">
        <a href="#" aria-controls="home-tab-panel" id="home-tab">
          <span class="oj-tabbar-item-icon demo-home-icon-24 demo-icon-font-24">
          </span>
          Home
        </a>
      </li>
      <li id="settings">
        <a href="#" aria-controls="settings-tab-panel" id="settings-tab">
          <span class="oj-tabbar-item-icon demo-gear-icon-16 demo-icon-font-24">
          </span>
          Settings
        </a>
      </li>
      <li id="about">
        <a href="#"  aria-controls="about-tab-panel" id="about-tab">
          <span class="oj-tabbar-item-icon demo-info-icon-24 demo-icon-font-24">
          </span>
          About
        </a>
      </li>
      <li id="contact">
        <a href="#" aria-controls="contact-tab-panel" id="contact-tab">
          <span class="oj-tabbar-item-icon demo-email-icon-24 demo-icon-font-24">
          </span>
          Contact
        </a>
      </li>
    </ul>
  </oj-tab-bar>
</div>
```

> **Tip:**
>
> You can also use the tab bar's `data` attribute to populate the tab bar with data from a table data source or array data provider. See the Tab Bars cookbook demos for examples.

Adding Content

To provide content to the tabs, add the `oj-switcher` element to the tab bar's container. `oj-switcher` dynamically decides which child element to display based on the value in its `value` attribute, which must match the value in the tab bar's `selection` attribute. The switcher's `slot` attribute also determines which tab's content to display and must match an id in the tab bar's list.

```
<div id="demo-container" class="oj-flex demo-edge-top">
  <oj-tab-bar id="hnavlist"
              edge="top"
```

```
            selection="{{selectedItem}}">
    <ul>
      <li id="home">
        <a href="#" aria-controls="home-tab-panel" id="home-tab">
          <span class="oj-tabbar-item-icon demo-home-icon-24 demo-icon-font-24">
          </span>
          Home
        </a>
      </li>
      ... contents omitted
   </oj-tab-bar>
   <oj-switcher value="[[selectedItem]]">
     <div slot="home"
          id="home-tab-panel"
          role="tabpanel"
          aria-labelledby="home-tab">
       <div class="demo-tab-content">
         <h2>Home page content area</h2>
         <p>Home page sample content</p>
       </div>
     </div>
     <div slot="settings"
          id="settings-tab-panel"
          role="tabpanel"
          aria-labelledby="settings-tab">
       <div class="demo-tab-content">
         <h2>Settings content area</h2>
         <p>Settings sample content</p>
       </div>
     </div>
     <div slot="about"
          id="about-tab-panel"
          role="tabpanel"
          aria-labelledby="about-tab">
       <div class="demo-tab-content">
         <h2>About content area</h2>
         <p>About sample content</p>
       </div>
     </div>
     <div slot="contact"
          id="contact-tab-panel"
          role="tabpanel"
          aria-labelledby="contact-tab">
       <div class="demo-tab-content">
         <h2>Contact page content area</h2>
         <p>Contact sample content</p>
       </div>
     </div>
   </oj-switcher>
</div>
```

In this example, `selectedItem` is an observable, which is initially set to `home` in the application's view model.

```
require(['ojs/ojcore',
'knockout',
'jquery',
'ojs/ojknockout',
'ojs/ojnavigationlist',
'ojs/ojswitcher'
],
```

```
function(oj, ko, $) // this callback gets executed when all required modules are
loaded
{

  function ViewModel(){
    this.selectedItem = ko.observable("home");
  }
  var vm = new ViewModel();
  $(function() {
      ko.applyBindings(vm, document.getElementById('tabbardemo'));
  });
}
);
```

For a complete example that illustrates the use of `oj-switcher` with `oj-tab-bar`, see Switcher.

Adding Interactivity

You can add the ability to add, remove, or reorder a tab within the tab bar. To enable this functionality, first create the tab bar using a supported data source.

- Removing a tab

  Add the `oj-removable` class to each removable tab. Add the `on-oj-remove` attribute to the `oj-tab-bar` to specify an event handler to remove the tab from the data source.

> **Important:**
>
> For accessibility, add the `oj-menu` child element to the tab bar as a contextmenu slot with an option marked as `data-oj-command="oj-tabbar-remove"`.
>
> ```
> <oj-tab-bar contextmenu="tabmenu" id="hnavlist"
>  selection="{{selectedItem}}"
>  current-item="{{currentItem}}"
>  edge="top"
>  data="[[dataSource]]"
>
> item.renderer="[[oj.KnockoutTemplateUtils.getRenderer('server_template',
> true)]]"
>  on-oj-remove="[[onRemove]]">
>  <oj-menu slot="contextMenu"
>  style="display:none"
>  aria-label="Actions">
>  <oj-option data-oj-command="oj-tabbar-remove">
>  Removable
>  </oj-option>
>  </oj-menu>
>  </oj-tab-bar>
> ```

- Adding a tab

  To add a tab, add code to your view model to push the new tab to the data source. Provide the user with a dialog or other prompt to add the tab and reference the event handler that will add the tab to the data source.

```
<oj-button id="addTab" on-click="[[openDialog]]">Add Tab</oj-button>
```

- Reordering tabs

  Add the `reorderable` attribute to the `oj-tab-bar` definition and set it to `enabled`. You must also add the `on-oj-reorder` attribute to specify an event handler to move the tab location in the data source.

> **Important:**
>
> For accessibility, define a context menu on the page that includes entries for cutting and pasting items by specifying the `data-oj-command` attribute on the menu item with one of the values: `oj-tabbar-cut`, `oj-tabbar-paste-before`, and `oj-tabbar-paste-after`.
>
> ```
> <oj-tab-bar id="tabbar"
>  aria-label="Tabs using json data"
>  selection="settings"
>  data="[[dataSource]]"
>  item.renderer="[[renderer]]"
>  reorderable="enabled"
>  edge="[[edge]]"
>  on-oj-reorder="[[handleReorder]]">
>  <oj-menu id="itemMenu" slot="contextMenu" style="display:none" aria-
> label="Item Reorder">
>  <oj-option id="cutItem" data-oj-command="oj-tabbar-cut"></oj-option>
>  <oj-option id="pasteBeforeItem" data-oj-command="oj-tabbar-paste-
> before"></oj-option>
>  <oj-option id="pasteAfterItem" data-oj-command="oj-tabbar-paste-
> after"></oj-option>
>  </oj-menu>
> </oj-tab-bar>
> ```

For the code that implements the event handlers for adding, removing, and reordering tabs, see the Tab Bar - Add and Remove and Tab Bar - Reorder demos in the Oracle JET Cookbook.

Working with Visualizations

The Oracle JET visualization components include charts, gauges, and other components that you can use and customize to present flat or hierarchical data in a graphical display for data analysis.

Topics:

- Choosing a Data Visualization Component for Your Application
- Using Attribute Groups With Data Visualization Components

Choosing a Data Visualization Component for Your Application

The visualization components include charts, gauges, and a variety of other visualizations including diagrams, timelines, thematic maps, and so on that you can

use for displaying data. You may find the following usage suggestions helpful for determining which visualization to use in your application.

Charts

Charts show relationships among data and display values as lines, bars, and points within areas defined by one or more axes.

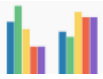
Chart Type	Image	Description	Usage Suggestions
Area		Displays series of data whose values are represented by filled-in areas. Areas can be stacked or unstacked. The axis is often labeled with time periods such as months.	Use to show cumulative trends over time, such as sales for the last 12 months. Area charts require at least two groups of data along an axis. If you are working with multiple series and want to display unstacked data, use line or line with area charts to prevent values from being obscured.
Bar		Displays data as a series of rectangular bars whose lengths are proportional to the data values. Bars display vertically or horizontally and can be stacked or unstacked.	Use to compare values across products or categories, or to view aggregated data broken out by a time period.
Box Plot		Displays the minimum, quartiles, median, and maximum values of groups of numerical data. Groups display vertically or horizontally. You can also vary the box width to make the width of the box proportional to the size of the group.	Use to analyze the distribution of data. Box plots are also called box and whisker diagrams.
Bubble		Displays three measures using data markers plotted on a two-dimensional plane. The location of the markers represents the first and second measures, and the size of the data markers represents the proportional values of the third measure.	Use to show correlations among three types of values, especially when you have a number of data items and you want to see the general relationships. For example, use a bubble chart to plot salaries (x-axis), years of experience (y-axis), and productivity (size of bubble) for your work force. Such a chart enables you to examine productivity relative to salary and experience.
Combination		Displays series of data whose values are represented by a combination of bars, lines, or filled-in areas.	Combination charts are commonly configured as lines with bars for lines with stacked bars. For example, you can use a line to display team average rating with bars to represent individual team member ratings on a yearly basis.

Chart Type	Image	Description	Usage Suggestions
Funnel		Visually represents data related to steps in a process as a three-dimensional chart that represents target and actual values, and levels by color. The steps appear as vertical slices across a horizontal cone-shaped section. As the actual value for a given step or slice approaches the quota for that slice, the slice fills.	Use to watch a process where the different sections of the funnel represent different stages in the process, such as a sales cycle. The funnel chart requires actual values and target values against a stage value, which might be time.
Line		Displays series of data whose values are represented by lines.	Use to compare items over the same time. Charts require data for at least two points for each member in a group. For example, a line chart over months requires at least two months. Typically a line of a specific color is associated with each group of data such as the Americas, Europe, and Asia. Lines should not be stacked which can obscure data. To display stacked data, use area or line with area charts.
Line with Area		Displays series of data whose values are represented as lines with filled-in areas.	Use for visualizing trends in a set of values over time and comparing those values across series.
Pie		Represents a set of data items as proportions of a total. The data items are displayed as sections of a circle causing the circle to look like a sliced pie.	Use to show relationship of parts to a whole such as how much revenue comes from each product line. Consider treemaps or sunbursts if you are working with hierarchical data or you want your visual to display two dimensions of data.
Polar		Displays series of data on a polar coordinate system. The polar coordinate system can be used for bar, line, area, combination, scatter, and bubble charts. Polygonal grid shape (commonly known as radar) is supported for polar line and area charts.	Use to display data with a cyclical x-axis, such as weather patterns over months of the year, or for data where the categories in the x-axis have no natural ordering, such as performance appraisal categories.
Pyramid		Displays values as slices in a pyramid. The area of each slice represents its value as a percentage of the total value of all slices.	Use to display hierarchical, proportional and foundation-based relationships, process steps, organizational layers, or topics interconnections.
Range		Displays a series of data whose values are represented either as an area or bar proportional to the data values.	Use to display a range of temperatures for each day of a month for a city.

Chart Type	Image	Description	Usage Suggestions
Scatter		Displays two measures using data markers plotted on a two-dimensional plane.	Use to show correlation between two different kinds of data values, such as sales and costs for top products. Scatter charts are especially useful when you want to see general relationships among a number of items.
Spark		Display trends or variations as a line, bar, floating bar, or area. Spark charts are simple and condensed.	Use to provide additional context to a data-dense display. Sparkcharts are often displayed in a table, dashboard, or inline with text.
Stock		Display stock prices and, optionally, the volume of trading for one or more stocks. When any stock or candlestick chart includes the volume of trading, the volume appears as bars in the lower part of the chart.	

Gauges

Gauges focus on a single value, displayed in relation to minimum, maximum, or threshold values.

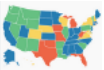
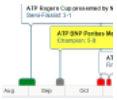
Gauge Type	Image	Description	Usage Suggestions
Dial		Displays a metric value plotted on a circular axis in relation to the minimum and maximum possible values for the metric. An indicator points to the dial gauge's metric value on the axis	The circular status meter is usually a better choice because of its more modern look and feel and efficient use of space.
LED		Graphically depicts a measurement, such as a key performance indicator (KPI). Several styles of shapes are available, including round or rectangular shapes that use color to indicate status, and triangles or arrows that point up, left, right, or down in addition to the color indicator.	Use to highlight a specific metric value in relation to its threshold.
Rating		Displays and optionally accepts input for a metric value.	Use to show ratings for products or services, such as the star rating for a movie.

Gauge Type	Image	Description	Usage Suggestions
Status Meter		Displays a metric value on a horizontal, vertical, or circular axis. An inner rectangle shows the current level of a measurement against the ranges marked on an outer rectangle. Optionally, status meters can display colors to indicate where the metric value falls within predefined thresholds.	

Other Data Visualizations

Other data visualizations include maps, timelines, Gantt charts and various other components that don't fit into the chart or gauge category.

Data Visualization Component	Image	Description	Usage Suggestions
Diagram		Models, represents, and visualizes information using a shape called a node to represent data, and links to represent relationships between nodes.	Use to highlight both the data objects and the relationships between them.
Gantt		Displays bars that indicate the start and end date of tasks.	Use to display project schedules.
Legend		Displays a panel which provides an explanation of the display data in symbol and label pairs.	Consider using the legend component when multiple visualizations on the same page are sharing a coloring scheme. For an example using `ojLegend` with a bubble chart, see Using Attribute Groups With Data Visualization Components.
NBox		Displays data items across two dimensions. Each dimension can be split into multiple ranges, whose intersections result in distinct cells representing data items.	Use to visualize and compare data across a two-dimensional grid, represented visually by rows and columns.

Data Visualization Component	Image	Description	Usage Suggestions
PictoChart		Uses stamped images to display discrete data as a visualization of an absolute number or the relative size of different parts of a population.	Common in infographics. Use when you want to use icons to: • visualize a discrete value, such as the number of people in a sample that meets a specified criteria. • highlight the relative sizes of the data, such as the number of people belonging to each age group in a population sample.
Sunburst		Displays quantitative hierarchical data across two dimensions, represented visually by size and color. Uses nodes to reference the data in the hierarchy. Nodes in a radial layout, with the top of the hierarchy at the center and deeper levels farther away from the center.	Use for identifying trends for large hierarchical data sets, where the proportional size of the nodes represents their importance compared to the whole. Color can also be used to represent an additional dimension of information. Use sunbursts to display the metrics for all levels in the hierarchy.
Tag Cloud		Displays textual data where font style and size emphasizes the importance of each data item.	Use for quickly identifying the most prominent terms to determine their relative importance.
Thematic Map		Displays data that is associated with a geographic location.	Use to show trends or patterns in data with a spatial element to it.
Timeline		Displays a set of events in chronological order and offers rich support for graphical data rendering, scale manipulation, zooming, resizing, and objects grouping.	Use to display time specific events in chronological order.
Treemap		Displays quantitative hierarchical data across two dimensions, represented visually by size and color. Uses nodes to reference the data in the hierarchy. Nodes are displayed as a set of nested rectangles.	Use for identifying trends for large hierarchical data sets, where the proportional size of the nodes represents their importance compared to the whole. Color can also be used to represent an additional dimension of information Use treemaps if you are primarily interested in displaying two metrics of data using size and color at a single layer of the hierarchy.

For examples that implement visualization components, see the Oracle JET Cookbook at Data Visualizations.

> **Note:**
>
> To use an Oracle JET data visualization component, you must configure your application to use RequireJS. For details about adding RequireJS to your application, see Use RequireJS to Manage Library, Link, and Script References.

Using Attribute Groups With Data Visualization Components

Attribute groups allow you to provide stylistic values for color and shape that can be used as input for supported data visualization components, including bubble and scatter charts, sunbursts, thematic maps, and treemaps. In addition, you can share the attribute values across components, such as a thematic map and a legend, using an attribute group handler.

Using attribute groups is also one way that you can easily provide visual styling for data markers for a given data set. Instead of manually choosing a color for each unique property and setting a field in your data model, you can use an attribute group handler to get back a color or shape value given a data value. Once an attribute group handler retrieves a color or shape value given a data value, all subsequent calls that pass in the same data value will always return that color or shape.

Oracle JET provides the following classes that you can use for adding attribute groups to your data visualization components:

- `oj.ColorAttributeGroupHandler`: Creates a color attribute group handler that will generate color attribute values.

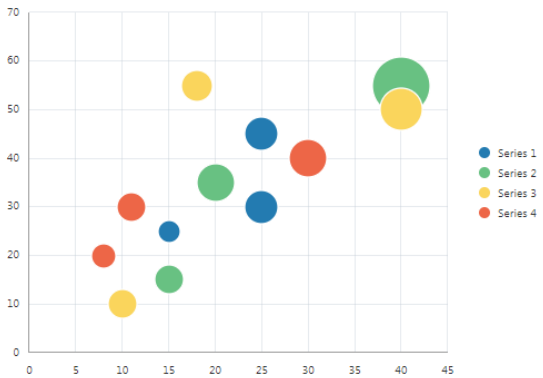
  Colors are generated using the values in the `.oj-dvt-category-index*` tag selectors.

- `oj.ShapeAttributeGroupHandler`: Creates a shape attribute group handler that will generate shape attribute values.

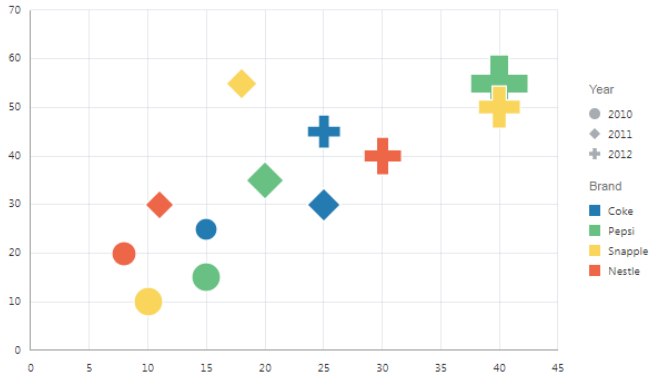
  Supported shapes include `square`, `circle`, `human`, `triangleUp`, `triangleDown`, `diamond`, and `plus`.

You can see the effect of applying attribute groups to a bubble chart in the following figure. In this example, the shape of the markers (`round`, `diamond`, and `plus`) indicates the year for which the data applies. The color differentiates the brand. The example also uses the `oj.Legend` data visualization component to provide a legend for the bubble chart.

Bubble Chart with Default Colors and Shapes



Bubble Chart with Attribute Groups for Color and Shape



The code excerpt below shows the JavaScript to create the bubble chart with color and shape attribute groups. The code relating to the attribute groups is highlighted in bold font.

```
require(['ojs/ojcore', 'knockout', 'jquery', 'ojs/ojknockout', 'ojs/ojchart','ojs/
ojlegend'],
function(oj, ko, $)
{
var colorHandler = new oj.ColorAttributeGroupHandler();
    var shapeHandler = new oj.ShapeAttributeGroupHandler();
    shapeHandler.getValue();

    function ChartModel() {
        var data = [
            {x: 15, y: 25, z: 5, company: "Coke", year: "2010"},
            {x: 25, y: 30, z: 12, company: "Coke", year: "2011"},
            {x: 25, y: 45, z: 12, company: "Coke", year: "2012"},
            {x: 15, y: 15, z: 8, company: "Pepsi", year: "2010"},
            {x: 20, y: 35, z: 14, company: "Pepsi", year: "2011"},
            {x: 40, y: 55, z: 35, company: "Pepsi", year: "2012"},
            {x: 10, y: 10, z: 8, company: "Snapple", year: "2010"},
            {x: 18, y: 55, z: 10, company: "Snapple", year: "2011"},
            {x: 40, y: 50, z: 18, company: "Snapple", year: "2012"},
            {x: 8, y: 20, z: 6, company: "Nestle", year: "2010"},
            {x: 11, y: 30, z: 8, company: "Nestle", year: "2011"},
            {x: 30, y: 40, z: 15, company: "Nestle", year: "2012"}
        ];

        this.seriesItems = [
            {name: "Coke", displayInLegend: 'off', items: []}, {name: "Pepsi",
displayInLegend: 'off', items:[]},
            {name: "Snapple", displayInLegend: 'off', items: []}, {name: "Nestle",
displayInLegend: 'off', items:[]}
        ];
        this.bubbleGroups = [];

        for(var i = 0; i < data.length; i++){
            var seriesIndex = Math.floor(i/3);
            this.seriesItems[seriesIndex].items.push({
                x: data[i].x, y: data[i].y, z: data[i].z,
                color: colorHandler.getValue(data[i].company),
                markerShape: shapeHandler.getValue(data[i].year),
                categories: [ data[i].company, data[i].year],
```

```
                shortDesc: data[i].company + " " + data[i].year + "<br/>X: " +
                    data[i].x + "<br/>Y: " + data[i].y + "<br/>Z: " + data[i].z
            });
        }

        this.bubbleGroups= ["2010", "2011", "2012"];

        var legendSections = {sections: [
            {title: "Year", items: [
                {markerShape:shapeHandler.getValue("2010"), text: "2010", id:
"2010"},
                {markerShape:shapeHandler.getValue("2011"), text: "2011", id:
"2011"},
                {markerShape:shapeHandler.getValue("2012"), text: "2012", id: "2012"}
            ]},
            {title: "Brand", items:[
                {color: colorHandler.getValue("Coke"), text: "Coke", id: "Coke"},
                {color: colorHandler.getValue("Pepsi"), text: "Pepsi", id: "Pepsi"},
                {color: colorHandler.getValue("Snapple"), text: "Snapple", id:
"Snapple"},
                {color: colorHandler.getValue("Nestle"), text: "Nestle", id:
"Nestle"}
            ]}
        ]};
        self.legendSections = ko.observable(legendSections);
    }

    var chartModel = new ChartModel();

    $(
        function(){
            ko.applyBindings(chartModel, document.getElementById('chart-container'));
        }
    );
});
```

The bubble chart's legend uses the same attribute group handlers for color and shape. Since the color and shape values were initially set in the `colorHandler.getValue()` and `shapeHandler.getValue()` calls above, the calls to `getValue()` below will return the same values for color and shape.

```
    ...content omitted
        var legendSections = {sections: [
            {title: "Year", items: [
                {markerShape:shapeHandler.getValue("2010"), text: "2010", id:
"2010"},
                {markerShape:shapeHandler.getValue("2011"), text: "2011", id:
"2011"},
                {markerShape:shapeHandler.getValue("2012"), text: "2012", id: "2012"}
            ]},
            {title: "Brand", items:[
                {color: colorHandler.getValue("Coke"), text: "Coke", id: "Coke"},
                {color: colorHandler.getValue("Pepsi"), text: "Pepsi", id: "Pepsi"},
                {color: colorHandler.getValue("Snapple"), text: "Snapple", id:
"Snapple"},
                {color: colorHandler.getValue("Nestle"), text: "Nestle", id:
"Nestle"}
            ]}
        ]};
        self.legendSections = ko.observable(legendSections);
    }
```

```
var chartModel = new ChartModel();

$(
    function(){
        ko.applyBindings(chartModel, document.getElementById('chart-container'));
```

The Oracle JET Cookbook provides the complete code for implementing both bubble charts at Bubble Charts.

You can also initialize an attribute group with match rules which consist of a map of key value pairs for categories and the matching attribute values. For example, if you wanted to specify colors for specific categories instead of using the default colors, you could define the color attribute group with match rules.

```
var colorHandler = new oj.ColorAttributeGroupHandler({"soda":"#336699",
                                                      "water":"#CC3300",
                                                      "iced tea":"#F7C808"});
```

For detailed information about `oj.ColorAttributeGroupHandler`, see the oj.ColorAttributeGroupHandler API documentation. For more information about `oj.ShapeAttributeGroupHandler`, see the oj.ShapeAttributeGroupHandler API documentation.

8

Working with Oracle JET Composite Components

Oracle JET composite components are reusable pieces of user interface code that you can embed as custom HTML elements. Composite components can contain Oracle JET components, other composite components, HTML, JavaScript, and CSS.

Topics

- Typical Workflow for Working with Oracle JET Composite Components
- About Composite Components
- Creating Composite Components
- Testing Composite Components
- Adding Composite Components to Your Page

Typical Workflow for Working with Oracle JET Composite Components

Understand Oracle JET composite components and the steps required to create and add them to your page. Optionally, you can learn how to customize composite component styling and access the component's internal parts for testing.

To work with Oracle JET composite components, refer to the typical workflow described in the following table. Most tasks are optional, but you may find the first two helpful for understanding the Oracle JET composite component architecture and adding Oracle composite components to your page.

Task	Description	More Information
Understand the Oracle JET composite component architecture	Understand the structure and functionality of composite components.	About Composite Components
Create a composite component	Identify the steps you must take to create an Oracle JET composite component.	Creating Composite Components
Test your composite component.	Understand the testing guidelines for composite components.	Testing Composite Components
Add a composite component to your page.	Identify the steps you must take to add a composite component to your page.	Adding Composite Components to Your Page

About Composite Components

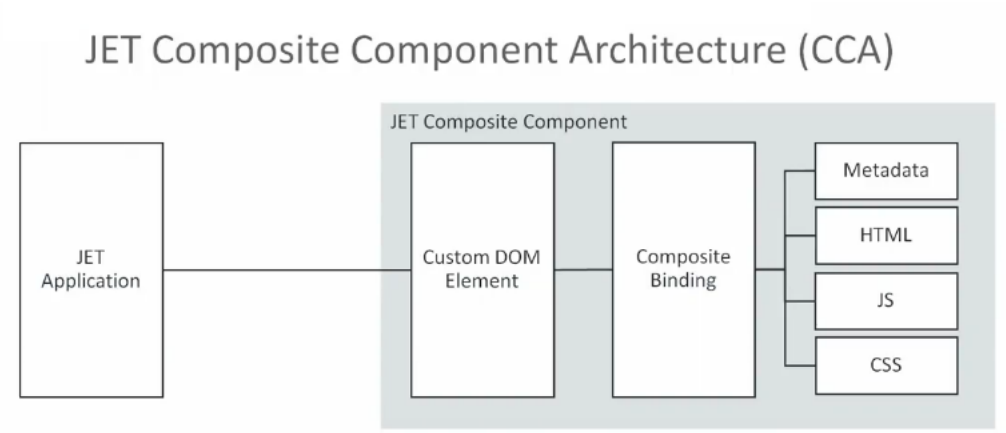
Oracle JET composite components are packaged as standalone modules that your application can load using RequireJS. The framework supplies APIs that you can use

to register composite components. Knockout currently provides one and two way data binding, template markup, and composite component activation.

Composite components follow much of the W3C Web Components - Custom Elements specification which describes a mechanism for enabling authors to define and use custom DOM elements. For detail discussion on Composite Components visit the Groundside blog: https://blogs.oracle.com/groundside/cca.

Composite Component Architecture

The following image shows a high-level view of the JET Composite Component architecture. In this example, an Oracle JET application is consuming the composite component, but you can add composite components to other JavaScript or Oracle applications where supported.



Composite components contain:

- A custom DOM element: Named element that functions as a HTML element.

  ```
  <my-composite-component attribute1="value1" attribute2="value2" ...>
  </my-composite>
  ```

- Composite binding definition: Knockout expression for setting composite component attributes.

  ```
  <my-composite-component attribute1="value1" attribute2="[[value2]]"
  attribute3="{{value3}}">
  </my-composite>
  ```

  attribute1's value can be a primitive JavaScript type (boolean, number, string) or a JSON object, *attribute2*'s value uses one way data binding, and *attribute3* 's value uses a two way binding. One way bindings on composite components specify that the expression will not update the application's ViewModel if the value changes. In two way bindings, the expression will update and the value written back to the application's ViewModel.

  In the following code sample, the composite component is declared with three attributes: `type`, `data`, and `axis-labels`.

  ```
  <my-chart type=bubble data="[[salesData]]" axis-labels={{showAxisLabels}} ...
  </my-chart>
  ```

Because the `salesData` expression is declared using one way binding (`[[salesData]]`), it will not be written back to if the `data` property is updated by the composite component's ViewModel. The `data` property will contain the current value, but the `salesData` expression will not be updated. Alternatively, if the `axisLabels` property is updated by the ViewModel, both the `axisLabel` property and the `{{showAxisLabels}}` expression will contain the updated value.

- Metadata: Data provided in JSON format which defines the composite component's required properties: `name`, `version`, and `jetVersion`. Metadata may also define optional properties, including `description`, `displayName`, `compositeDependencies`, `icon`, `methods`, `events`, and `slots`.

  Composite components support both runtime and design time metadata. Design time metadata isn't required at runtime and is useful for design time tools and property editors. Design time tools can define tools-specific metadata extensions to the composite component's metadata. For any tool-specific metadata extensions, refer to the documentation for that specific tool. For additional information about metadata properties, see oj.Composite in the API documentation.

> **Note:**
>
> The following sample shows some of the available metadata fields with descriptions of their content and whether they are not used at run time. Required metadata are highlighted in bold.

```
{
  "name": "The component tag name",
  "version": "The component version. Note that changes to the metadata even for
minor updates like updating the jetVersion should result in at least a minor
composite version change, e.g. 1.0.0 -> 1.0.1.",
  "jetVersion": "The semantic version of the supported JET version(s). Composite
authors should not specify a semantic version range that includes unreleased JET
major versions as major releases may contain non backwards compatible changes.
Authors should instead recertify composites with each major release and update
the composite metadata or release a new version that is compatible with the new
release changes.",
  "description": "A high-level description for the component. Not used at run
time.",
  "displayName": "A user friendly, translatable name of the component. Not used
at run time.",

  "properties": {
    "property1": {
      "description": "A description for the property. Not used at run time.",
      "displayName": "A user friendly, translatable name of the property. Not
used at run time.",
      "readOnly": "Boolean that determines whether a property can be updated
outside of the ViewModel. False by default.",
      "type": "The type of the property, following Google's Closure Compiler syntax.",
      "value": "Object containing an optional default value for a property.",
      "writeback": "Boolean that determines whether an expression bound to this
property should be written back to. False by default.",
      "enumValues": "An optional array of valid enum values for a string
property. An error is thrown if a property value does not match one of the
provided enumValues.",
      "properties": "A nested properties object for complex properties.
```

```
Subproperties exposed using nested properties objects in the metadata can be set
using dot notation in the attribute. See the Subproperties section for more
details on working with subproperties."
      },

    "property2": {
         ... contents omitted
      }
  },

  "methods": {
    "method1": {
      "description": "A description for the method. Not used at run time.",
      "displayName": "A user friendly, translatable name of the method. Not used
at run time.",
      "internalName": "An optional ViewModel method name that is different from,
but maps to this method.",
      "params": "An array of objects describing the method parameter . Not used
at run time.",
      "return": "The return type of the method, following Closure Compiler
syntax. Not used at run time."
     },
    "method2": {
     ... contents omitted
     }
  },

  "events": {
    "event1": {
      "bubbles": "Boolean that indicates whether the event bubbles up through
the DOM or not. Defaults to false. Not used at run time.",
      "cancelable": "Boolean that Indicates whether the event is cancelable or
not. Defaults to false. Not used at run time.",
      "description": "A description for the event. Not used at run time.",
      "displayName": "A user friendly, translatable name of the method. Not used
at run time.",
      "detail": {
        "field name": "Describes the properties available on the event's detail
property which contains data passed when initializing the event. Not used at run
time."
       }
     },
    "event2": {
     ... contents omitted
     }
   },

  "slots": {
    "slot1": {
      "description": "A description for the slot. Not used at run time.",
      "displayName": "A user friendly, translatable name of the method. Not used
at run time."
     }
   }
}
```

HTML markup: (Required) Contains the View definition which describes how to render the composite component.

- JavaScript: Optional script for defining the ViewModel and custom events.

The ViewModel is also where you define callbacks for various stages of the composite component's lifecycle. Composite components support the following optional lifecyle methods: `initialize (context)`, `activated (context)`, `attached (context)`, `bindingsApplied (context)`, and `detached (element)`.

- CSS: Optional styling for the composite component.

  CSS is not scoped to composite components, and you must define styles appropriately.

- SCSS: Optional files containing Sass variables to generate the composite component's CSS.

  If you're defining only a few styles for your component, then adding the CSS manually may be sufficient. However, there may be use cases where you want to use Sass variables to generate the CSS. In those cases, create and place the SCSS files in the composite component's folder and use the tooling to add node-sass to your application. See Step 8 - Creating Composite Components.

> **Important:**
>
> You must add the Sass files manually to the composite component's folder. The tooling will compile any Sass files if they exist, but it will not create them for you.

Composite Component Files

Place composite component files in a folder with the same name as the composite component tag. Typically, you place the folder within your application in a `jet-composites` folder: *`application-path`*`/jet-composites/my-composite-component/`.

You can also place your composite component in a different file location or reference a composite component on a different server using RequireJS path mapping. For examples, see oj.Composite - Packaging and Registration.

Each composite component file should use the following naming convention to match the purpose:

- *`my-composite-component`*`.`**`html`**: view template
- *`my-composite-component`*`.`**`js`**: ViewModel
- `component`**`.json`**: metadata
- `styles`**`.css`**: CSS styling
- `styles`**`.scss`**: Sass variables to generate CSS for composite components
- `loader`**`.js`**: RequireJS module defining the dependencies for its metadata, View, ViewModel, and CSS This file should also include the composite component registration.

Composite Component Slotting

Use slotting to add child components (which can also be composites) that get slotted into specified locations within the composite's View markup. The following example contains a portion of the View markup for a composite component named demo-columns.

```
<div class="oj-flex oj-flex-item-pad">
  <div role="group"
       data-bind="attr: {'aria-label': $props.headers[0]}"
       class="oj-flex-item colA oj-flex oj-sm-flex-direction-column oj-sm-align-
items-center">
    <h3 data-bind="text: $props.headers[0]"></h3>
    <oj-bind-slot name="columnA">
    </oj-bind-slot>
  </div>
  ... content omitted
</div>
```

In this example, the demo-columns composite component defines an `oj-bind-slot` named `columnA`. As shown below, a developer can specify a child component with a slot named `columnA` when adding the demo-columns composite to the page.

```
<demo-columns id="composite-container" headers='["Sales", "Human Resources",
"Support"]'>
  <!-- ko foreach: sales -->
    <demo-card slot="columnA" name="[[name]]" work-title="[[title]]"></demo-card>
  <!-- /ko -->
  ... contents omitted
</demo-columns>
```

Composite Component Examples

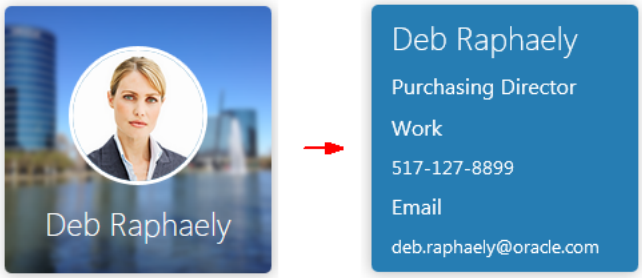
The Oracle JET Cookbook contains complete examples for creating basic and advanced composite components. You can also find examples that use slotting and data binding. For details, see Composite Component - Basic.

For additional information about composite component fields and methods, see oj.Composite in the API documentation.

Creating Composite Components

Use the Oracle JET command-line interface (CLI) to create a composite component template that you can populate with content. If you're not using the tooling, you can add the composite component files and folders manually to your Oracle JET application.

The following image shows a simple Oracle JET composite component configured to display contact cards with the contact's name and image if available. When the user selects the card, the content flips to show additional detail about the contact. You can find the complete example at Composite Component - Basic. You can download the demo files by clicking the download button ( ) on the cookbook.



To create this composite component or one of your own:

1. Determine a name for your composite component.

   The Web Component specification restricts custom element names as follows:

   - Names must contain a hyphen.
   - Names must start with a lowercase ASCII letter.
   - Names must not contain any uppercase ASCII letters.
   - Names should use a unique prefix to reduce the risk of a naming collision with other components.

     A good pattern is to use your organization's name as the first segment of the component name, for example, *org-component-name*.
   - Names must not be any of the reserved names. For the complete list, see valid custom element name.

   > **Note:**
   >
   > Oracle JET also reserves the `oj` namespace and prefixes.

   For example, use `demo-card` to duplicate the contact card example.

2. Determine where to place your composite component, using one of the following options.

   - Add the composite component to an existing Oracle JET application that you created with the Oracle JET CLI as described in Create a Web Application Using the Oracle JET Command-Line Interface or Create a Hybrid Mobile Application.

     If you use this method, you'll use the CLI to create a composite component template that contains the folders and files you'll need to store the composite component's content.
   - Use the Oracle JET CLI to scaffold a new composite component, based on a template, as well as a new Oracle JET application that you can use to view and test the composite component.

     If you use this method, you don't need to create a new Oracle JET application. However, you'll still need to install the Oracle JET CLI as described in Install the Prerequisite Packages. Also, the application contains an `index.html` file without UI, navigation, or layout, and you'll have to add those features manually to your application.
   - Manually add the composite component to an existing Oracle JET application that doesn't use the Oracle JET CLI.

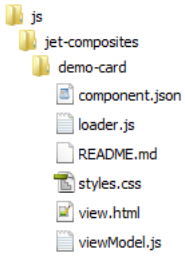
     If you use this method, you'll create the folders and files manually to store the composite component's content.

3. Depending upon the choice you made in the previous step, perform one of the following tasks to create the composite component.

   - If you used the Oracle JET CLI to create your application, in the application's top level directory, enter the following command at a terminal prompt to generate the composite component template:

```
ojet create component component-name
```

For example, enter `ojet create component demo-card` to create a composite component named `demo-card`. The command will add `jet-composites/demo-card` to the application's `js` folder and files containing stub content for the composite component.

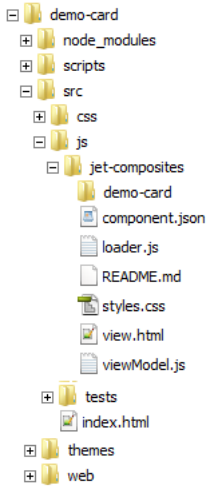


- If you installed the Oracle JET CLI and want to create a composite component and an Oracle JET application that you can use for testing, enter the following command at a terminal prompt in a folder of your choosing:

  ```
  ojet create component component-name
  ```

  For example, enter `ojet create component demo-card` to create an application and a composite component named `demo-card`.

  The command will add a `demo-card` folder containing the Oracle JET application files, `jet-composites/demo-card` to the new application's `js` folder, and files containing stub content for the composite component.



- If you're not using the Oracle JET CLI, create a `jet-composites` folder in your application's `js` folder, and add folders containing the name of each composite component you will create.

  For the demo-card example, create the `jet-composites` folder and add a `demo-card` folder to it. You'll create the individual composite component files in the remaining steps.

4. Determine the properties, methods, and events that your composite component will support and add them to the `component.json` file in the composite component's root folder, creating the file if needed.

> **Important:**
>
> The name of the composite component properties, event listeners, and methods should avoid collision with the existing HTMLElement properties, event listeners, and methods. Additionally, the property name `slot` should not be used. Also, you must not re-define the global attributes and events listed in the following link: Global attributes.

The demo-card example defines properties for the contact's full name, employee image, title, work number, email address, and background image. The required metadata are highlighted in bold.

```
 {
  "name": "demo-card",
  "description": "A card element that can display an avatar or initials on one
side and employee information on the other.",
  "version": "1.0.2",
  "displayName": "Demo Card",
  "jetVersion": ">=4.0.0 <5.0.0",
  "properties": {
     "name": {
       "description": "The employee's full name.",
       "type": "string"
       },
     "avatar": {
       "description": "The url of the employee's image.",
       "type": "string"
       },
     "workTitle": {
       "description": "The employee's job title.",
       "type": "string"
       },
     "workNumber": {
       "description": "The employee's work number.",
       "type": "number"
       },
    "email": {
      "description": "The employee's email.",
      "type": "string"
      },
    "backgroundImage": {
      "description": "The url of the background to use for the employee's card.",
      "type": "string"
      },
    }
 }
```

This example only defines properties for the composite component. You can also add metadata that defines methods and events. The metadata lists the name of the method or event and supported parameters.

```
{
  "properties": {
    ... contents omitted
```

```
  },
  "methods": {
     "flipCard" {
       "description": "Method to toggle flipping a card"
     },
     "enableFlip" {
       "description": "Enables or disables the ability to flip a card.",
       "params": [
         {
           "name": "bEnable",
           "description": "True to enable card flipping and false otherwise.",
           "type": "boolean"
          }
        ]
      },
    },
   "events": {
     "cardClick": {
       "description": "Triggered when a card is clicked and contains the value
of the clicked card..",
       "bubbles": true,
       "detail": {
         "value": {
           "description": "The value of the card.",
           "type": "string"
                }
            }
        }
    }
}
```

5. If your composite component contains a ViewModel, add its definition to `viewModel.js` in the composite component's root folder, creating the file if needed.

   The code sample below shows the ViewModel for the demo-card composite component. Comments describe the purpose, parameters, and return value of each function.

```
define(
    ['ojs/ojcore', 'knockout', 'jquery'], function (oj, ko, $) {
    'use strict';

    function model (context) {
    var self = this;
    self.initials = null;
    self.workFormatted = null;
    var element = context.element;

    /**
      * Formats a 10 digit number as a phone number.
      * @param  {number} number The number to format
      * @return {number}        The formatted phone number
      */
    var formatPhoneNumber = function(number) {
      return number.replace(/(\d{3})(\d{3})(\d{4})/, '$1-$2-$3');
    }

    // The props field on context is a Promise. Once that resolves,
    // we can access the properties that were defined in the composite metadata
    // and were initially set on the composite DOM element.
    context.props.then(function(properties) {
      if (properties.name) {
```

```
        var initials = properties.name.match(/\b\w/g);
        self.initials = (initials.shift() + initials.pop()).toUpperCase();
      }
      if (properties.workNumber)
        self.workFormatted = formatPhoneNumber(properties.workNumber);
    });

    /**
      * Flips a card
      * @param  {MouseEvent} event The click event
      */
    self.flipCard = function(model, event) {
      if (event.type === 'click' || (event.type === 'keypress' && event.keyCode
=== 13)) {
        $(element.childNodes[0]).toggleClass('flipped');
      }
    };
  }
  return model;
});
```

6. In the composite component's root folder, add the View definition to `view.html`, creating the file if needed.

   The View for the demo-card composite component is shown below. Any property defined in the component's metadata is accessed using the `$props` property of the View binding context.

```
<div tabindex="0" role="group" class="flip-container"
  data-bind="event: {click: flipCard, keypress: flipCard}, attr: {'aria-
label': $props.name + ' Press Enter for more info.'}">

  <div class="front-side" data-bind="attr: {style: $props.backgroundImage ?
'background-image:url(' + $props.backgroundImage + ')' : ''}">
    <!-- ko if: $props.avatar == null -->
      <div class="avatar" data-bind="text: initials"></div>
    <!-- /ko -->

    <!-- ko if: $props.avatar != null -->
      <img class="avatar" data-bind="attr: {src: $props.avatar,
alt: $props.name}">
    <!-- /ko -->
    <h2 data-bind="text: $props.name"></h2>
  </div>

  <div class="back-side">
    <div class="inner-back-side">
      <h2 data-bind="text: $props.name"></h2>
      <h5 data-bind="text: $props.workTitle"></h5>

      <!-- ko if: $props.workNumber != null -->
        <h5>Work</h5>
        <a href="#" data-bind="text: workFormatted"></a>
      <!-- /ko -->

      <!-- ko if: $props.email != null -->
        <h5>Email</h5>
        <a href="#" data-bind="text: $props.email"></a>
      <!-- /ko -->
    </div>
  </div>
</div>
```

For accessibility, the View's role is defined as `group`, with `aria-label` specified for the contact's name. In general, follow the same accessibility guidelines for the composite View markup that you would anywhere else within the application.

7. If you're not using the Oracle JET CLI, create the `loader.js` RequireJS module and place it in the composite component's root folder.

   The `loader.js` module defines the composite dependencies and registers the composite's `tagName`, `demo-card` in this example.

```
define(['ojs/ojcore', 'text!./view.html', './viewModel', 'text!./
component.json', 'css!./styles', 'ojs/ojcomposite'],
  function(oj, view, viewModel, metadata) {
    oj.Composite.register('demo-card', {
      view: {inline: view},
      viewModel: {inline: viewModel},
      metadata: {inline: JSON.parse(metadata)}
    });
  }
);
```

   In this example, the CSS is loaded through a RequireJS plugin (`css!./styles`), and you do not need to pass it explicitly in `oj.Composite.register()`.

8. Configure any custom styling that your composite component will use.

   - If you only have a few styles, add them to `styles.css` file in the composite component's root folder, creating the file if needed.

     For example, the demo-card composite component defines styles for the demo card's display, width, height, margin, padding, and more.

```
/* This is to prevent the flash of unstyled content before the composite
properties have been setup. */
demo-card:not(.oj-complete) {
  visibility: hidden;
}

demo-card {
  display: block;
  width: 200px;
  height: 200px;
  perspective: 800px;
  margin: 10px;
  box-sizing: border-box;
  cursor: pointer;
}

demo-card h2,
demo-card h5,
demo-card a,
demo-card .demo-card-avatar  {
  color: #fff;
  padding: 0;
}

... remaining contents omitted
```

   - If you used the Oracle JET tooling to create your application and want to use Sass to generate your CSS:

a. If needed, at a terminal prompt in your application's top level directory, type the following command to add node-sass to your application: `ojet add sass`.

b. Create `styles.scss` and place it in the composite component's top level folder.

c. Edit `styles.scss` with any valid SCSS syntax and save the file.

   In this example, a variable defines the demo card size:

```
$demo-card-size: 200px;

/* This is to prevent the flash of unstyled content before the composite
properties have been setup. */
demo-card:not(.oj-complete) {
  visibility: hidden;
}

demo-card {
  display: block;
  width: $demo-card-size;
  height: $demo-card-size;
  perspective: 800px;
  margin: 10px;
  box-sizing: border-box;
  cursor: pointer;
}

demo-card h2,
demo-card h5,
demo-card a,
demo-card .demo-card-avatar  {
  color: #fff;
  padding: 0;
}
... remaining contents omitted
```

d. To compile Sass, at a terminal prompt type `ojet build` or `ojet serve` with the `--sass` flag and application-specific options.

```
ojet build|serve [options] --sass
```

   `ojet build --sass` will compile your application and generate `styles.css` and `styles.css.map` files in the default platform's folder. For a web application, the command will place the CSS in **`web`**`/js/js-composites/`*`composite-name`*.

   `ojet serve --sass` will also compile your application but will display the web application in a running browser with `livereload` enabled. If you save a change to `styles.scss` in the application's **`src`**`/js/jet-composites/composite-name` folder, Oracle JET will compile Sass again and refresh the display.

> **Tip:**
>
> For help with `ojet` command syntax, type `ojet help` at a terminal prompt.

9. If you want to add documentation for your composite component, add content to `README.md` in your composite component's root folder, creating the file if needed.

Your `README.md` file should include an overview of your component with well-formatted examples. Include any additional information that you want to provide to your component's consumers. The recommended standard for `README` file format is markdown. For help with markdown, refer to https://guides.github.com/features/mastering-markdown/.

Recommended Standard Patterns and Coding Practices

Composite components should comply with all recommended patterns and coding practices to ensure interoperability both with other composite components and consuming frameworks.

Component Versioning

Your composite component must be assigned a version number in semantic version format.

> **Important:**
>
> When assigning and incrementing the version number associated with your composite components, be sure to follow semantic version rules and update Major, Minor and Patch version numbers appropriately. By doing so, component consumers will have a clear understanding about the compatibility and costs of migrating between different versions of your component.

JET Version Compatibility

You must use the semantic version rules to specify the `jetVersion` of the supported JET version(s). Composite authors should not specify a semantic version range that includes unreleased JET major versions as major releases may contain non backwards compatible changes. Authors should instead recertify composites with each major release and update the composite metadata or release a new version that is compatible with the new release changes.

Translatable Resources

Components should use the standard Oracle JET mechanism using the `ojL10n requireJS` plugin. You must store the translation bundles in a `resources/nls` subdirectory within your composite component's root folder. You can declare the languages and locals that you support in the composite metadata.

Peer-to-Peer Communication

Components must prefer a shared observable provided by the consumer over any kind of secret signaling mechanism when you are dealing with a complex integration. For example, a filter component and a data display component. By using a shared observable you can pre-seed and programmatically interact with the components through the filter.

Alternatively, you can use events and public methods based on one of the following approaches being used:

- A hierarchical relationship between the source and receiver of the event
- The identity of the source being passed to the receiver

> **Note:**
>
> In some runtime platforms, the developer doing the wiring may not have access to component IDs to pass the relevant identity.

- Listeners attached by components at the document level.

  In this case, you have to be responsible for the cleanup of those listeners, management of duplicates, and so on. Also, such listeners should preferably be based on custom composite events, not common events such as click, which might be overridden by intermediate nodes.

> **Note:**
>
> Under the web-component standards (shadow DOM), events will be re-targeted as they transition the boundary between the component and the consuming view. That is, the apparent identity of the raising element might be changed, particularly in the case of nested composite component architecture where the event would get tagged with the element representing the outer composite rather than the inner composite. Therefore, you should not rely on the `event.target` attribute to identify the composite source when listening at the document level. Instead, the `event.deepPath` attribute can be used to understand the actual origin of the event.

Access to External Data

You should not allow components to use the knockout binding hierarchy to obtain data from outside the composite context, for example, `$root`, `$parent[1]`, and so on. If you do so, an invisible API for the component will be created which cannot be captured in the metadata. All data transfer in and out of the component must be through the formal properties, methods, and events. Knockout binding expressions used in the context of, for example, a `ko foreach:` binding within the CCA view may use such expressions, providing that they do not attempt to reach outside of the logical root binding context of the CCA.

Object Scope

All properties and functions of composite components should be confined to the scope of the view model. No window or global scope objects should be created. Similarly, the existence of window scope objects should not be assumed by the composite author. If a consumer composite defined externally at window or global level is required for read or write then that composite must be passed in by the consuming view model through a formal property. Even if a well known global reference is needed from outside of the component, for example jQuery, it should be formally injected using the require `define()` function and declared as a dependency in the composite metadata.

External References

If a composite component must reference an external component, it should be part of the formal API of the component. The formal API passes the component reference through a property. For example, to allow the registration of a listener, the composite code requires a component reference defined externally. You must not allow composites to obtain IDs from hard-coded values, global storage, or walking the DOM.

Subcomponent IDs

Within the framework if any component needs a specific ID, use `context.unique` or `context.uniqueId` value to generate the ID. This ID is unique within the context of the page.

ID Storage

Any generated IDs should not be stored across invocation, such as in local storage or in cookies. The `context.unique` value for a particular composite may change each time a particular view is executed.

LocalStorage

It is difficult to consistently identify a unique instance of a composite within an application. So, it is advised not to allow a composite to utilize the local storage of a browser for persisting information that is specific to an instance of that composite. However, if the application provides a unique key through the public properties of the component you can then identify the unique instance of the component.

Additionally, do not use local storage as a secret signaling mechanism between composites. You cannot assure the availability of the capability and so it is recommended to exchange information through a shared JavaScript object or events as part of the public API for the component(s).

String Overrides

Composites will often contain string resources internally to service their default needs for UI and messages. However, sometimes you may want to allow the consumer to override these strings. To do this, expose a property for this purpose on the composite. By convention such a property would be called `translations`, and within it you can have sub-properties for each translatable string that relates to a required property on the composite, for example `requiredHint`, `requiredMessageSummary`, and so on. These properties can then be set on the component tag using sub-property references. For example:

```
"properties" : {
  "translations": {
    "description": "Property to allow override of default messages",
    "writeback" : false,
    "properties" : {
      "requiredHint": {
        "description": "Change the text of the required hint",
        "type": "string"
         },
      "requiredMessageSummary": {
        "description": "...",
        "type": "string"
        },
      "requiredMessageDetail": {
          "description": "...",
          "type": "string"
        }
      }
     }
   }
}
```

Logging

Use `oj.Logger` to write log messages from your code in preference to `console.log()`. The composites should respect the logging level of the consuming application and not change it. You should ideally prefix all log messages emitted from the composite with an identification of the source composite. As a preference, you can use the composite name. The composites should not override the writer defined by the consuming application.

Expensive Initialization

Composites should carry out minimum work inside the constructor function. Expensive initialization should be deferred to the `activated` lifecycle method or later. The constructor of a composite is invoked even if the component is not actually added to the visible DOM. For example, if a constructor is invoked within a Knockout `if` block. The further lifecycle phases will only occur when the component is actually needed.

Service Classes

The use of global service classes, that is functionality shared across multiple composites, can constitute an invisible contract that the consumer of your composite has to know about. To avoid this, we recommend:

- Create the service as a module that every composite can explicitly set it as `require()` block, thus removing the need for the consumer to do this elsewhere.
- Consider the timing issues that might occur if your service class needs some time to initialize, for example fetching data from a remote service. In such cases, you should be returning promises to the service object so that the components can safely avoid trying to use the information before it is actually available.

Using ojModule

If you use `ojModule` in a composite component and plan to distribute the composite outside of your application, you must take additional steps to ensure that the contained `ojModule` could be loaded from the location relative to the location of the composite. Unless the View and ViewModel instances are being passed to `ojModule` directly, you will need to provide the require function instance and the relative paths for views and view models. The require function instance should be obtained by the composite loader module by specifying `require` as a dependency.

```
<div data-bind="ojModule: {require: {instance: require_instance, viewPath:
"path_to_composite's_Views", modelPath: "path_to_composite's_ViewModels"}}"></div>
```

`require` Option	Type	Description
`instance`	Function	Function defining the require instance
`viewPath`	String	String containing the path to the composite component's Views
`modelPath`	String	String containing the path to the composite component's ViewModels

For additional information about working with `ojModule`, see ojModule.

Archiving Composite Components for Distribution

If you want to create a zip file for packaging, create an archive with the same name as the component itself. You may add version-identifying suffixes to the zip file name for operational reasons. The composite component artifacts must be placed in the root of the zip file, and there should be no intermediate directory structure before reaching the files.

Testing Composite Components

Test Oracle JET composite components using your favorite testing tools for client-side JavaScript applications.

Regardless of the test method you choose, be sure that your tests fully exercise the composite component's:

- ViewModel (if it exists)

  Ideally, your test results should be verifiable via code coverage numbers.
- HTML view

  Be sure to include any DOM branches that might be conditionally rendered, and test all slots with and without default content.
- Properties and property values
- Events
- Methods
- Accessibility
- Security

For additional information about testing Oracle JET applications, see Testing Oracle JET Applications.

Adding Composite Components to Your Page

To use a composite component, require its registration file and include the composite element in the application's HTML.

1. Open `component.json` and verify that your version of Oracle JET is compatible with the version specified in `jetVersion`.

   For example, the demo-card example specifies the following `jetVersion`:

   ```
   "jetVersion": ">=3.0.0 <5.0.0"
   ```

   According to semantic version syntax, this indicates that the composite is compatible with JET versions greater than or equal to 3.0.0 and less than 5.0.0.

   If your version of Oracle JET is lower than the `jetVersion`, you must update your version of Oracle JET before using the composite. If your version of Oracle JET is greater than the `jetVersion`, contact the developer to get an updated version of the composite.
2. In your application's `index.html` or main application HTML, add the composite component and any associated property declarations.

For example, to use the `demo-card` composite component, add it to your `index.html` file and add declarations for `name`, `avatar`, `work-title`, `work-number`, `email`, and `background-image`.

```
<div id="composite-container" class="oj-flex oj-sm-flex-items-initial">
  <!-- ko foreach: employees -->
    <demo-card class="oj-flex-item" name="[[name]]" avatar="[[avatar]]" work-
title="[[title]]"
      work-number="{{work}}" email="[[email]]" background-
image="[[backgroundImage]]">
    </demo-card>
  <!-- /ko -->
</div>
```

> **Note:**
>
> The framework maps the attribute names in the markup to the composite component's properties.
>
> - Attribute names are converted to lowercase. For example, a `workTitle` attribute will map to a `worktype` property.
> - Attribute names with dashes are converted to camelCase by capitalizing the first character after a dash and then removing the dashes. For example, the `work-title` attribute will map to a `workTitle` property.
>
> You can access the mapped properties programmatically as shown in the following markup:
>
> ```
> <h5 data-bind="text: $props.workTitle"></h5>
> ```

3. In your application's ViewModel, set values for the properties you declared in the previous step and add the composite component's loader file and the `ojs/ojcomposite` module to the list of application dependencies.

   For example, the following code adds the view Module to the application's RequireJS bootstrap file. The code also defines the `ojs/ojcomposite` and `jet-composites/demo-card-loader` dependencies.

```
require(['ojs/ojcore', 'knockout', 'jquery', 'ojs/ojknockout', 'ojs/
ojcomposite', 'jet-composites/demo-card/loader'],
function(oj, ko, $) {
  function model() {
    var self = this;
    self.employees = [
      {
        name: 'Deb Raphaely',
        avatar: 'images/composites/debraphaely.png',
        title: 'Purchasing Director',
        work: '5171278899',
        email: 'deb.raphaely@oracle.com',
        backgroundImage: 'images/composites/card-background_1.png'
      },
      {
        name: 'Adam Fripp',
        avatar: null,
        title: 'IT Manager',
        work: '6501232234',
        email: 'adam.fripp@oracle.com',
```

```
        backgroundImage: null
      }
    ];
  }

  $(function() {
      ko.applyBindings(new model(), document.getElementById('composite-
container'));
  });

});
```

9

Using the Common Model and Collection API

The Oracle JET Common Model and Collection API provides a two-way data binding model using Knockout. Use the Model and Collection classes to build Create, Read, Update, Delete (CRUD) applications that integrate REST services or data from any web service that returns data in the form of JSON objects.

Topics:

- Typical Workflow for Binding Data in Oracle JET
- About Oracle JET Data Binding
- Using the Oracle JET Common Model and Collection Framework
- Integrating REST Services
- Creating a CRUD Application Using Oracle JET

Typical Workflow for Binding Data in Oracle JET

Understand Oracle JET's support for the Model-View-ViewModel design. Learn how to use the Oracle JET Common Model and Collection API and how to integrate REST services into your Oracle JET application.

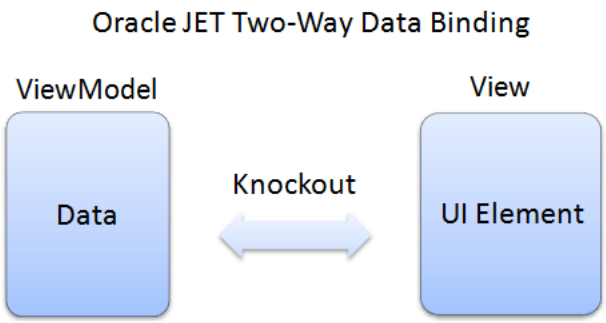
To use the Oracle JET Common Model for building CRUD applications, refer to the typical workflow described in the following table:

Task	Description	More Information
Understand Oracle JET support for data binding	Identify Oracle JET's support for the Model-View-ViewModel design.	About Oracle JET Data Binding
Use Oracle JET's Common Model and Collection	Identify the Oracle JET Common Model and Collection API, its relationship with Knockout, and how to use it in your Oracle JET application.	Using the Oracle JET Common Model and Collection Framework
Integrate REST services	Understand how to integrate REST services into your Oracle JET application.	Integrating REST Services
Create a CRUD application	Create a CRUD application using the Oracle JET Common Model and Collection API.	Creating a CRUD Application Using Oracle JET

About Oracle JET Data Binding

Oracle JET supports two-way data binding between the View and Model layers in the Model-View-ViewModel (MVVM) design. Data changes in the ViewModel are sent to

the UI components, and user input from the UI components is written back into the ViewModel.



Oracle JET uses Knockout to perform the data binding between the UI elements and the ViewModel. The ViewModel normally contains data fields for the UI state as well as references to external data. One of the ways to provide external data is to use the Common Model and Collection API.

Data for an Oracle JET application can come from any web data source that generates JSON data, such as a REST service, Server Sent Event (SSE), or WebSocket. In addition, Oracle JET also provides specific support for integrating web service data based on the Oracle REST standard.

Oracle JET also provides UI components for the View layer that include properties for data binding with Knockout. For additional information about Oracle JET's UI components and data binding options, see Understanding Oracle JET User Interface Basics.

Using the Oracle JET Common Model and Collection Framework

The Oracle JET Common Model and Collection API provides a collection-of-records object model that includes classes for bringing external data into an Oracle JET application and mapping the data to the application's view model.

Topics:

- About the Oracle JET Common Model and Collection API
- About Oracle JET Data Binding and Knockout
- Using the Oracle JET Common Model and Collection API

About the Oracle JET Common Model and Collection API

The Oracle JET Common Model and Collection API provides a collection-of-records object model that includes the following classes:

- `oj.Model`: Represents a single record from a data service such as a REST service
- `oj.Collection`: Represents a set of data records and is a list of `oj.Model` objects of the same type

- `oj.Events`: Provides methods for event handling
- `oj.KnockoutUtils`: Provides methods for mapping the attributes in a `oj.Model` or `oj.Collection` object to Knockout observables for use with component ViewModels

oj.Model and oj.Collection include client-side API that provides one way to bring external data into an Oracle JET application. oj.KnockoutUtils provides the `map()` method to map the attributes in a model object or the attributes of all models in a collection object to the application's view data model.

About Oracle JET Data Binding and Knockout

Knockout provides bindings between components as well as binding data from a ViewModel to components or HTML elements in an Oracle JET application. Knockout is an integral part of Oracle JET's toolkit and is included in the Oracle JET distribution.

The Oracle JET Model and Collection API includes the `KnockoutUtils` class which contains the `map()` method to convert the attributes in a model object (or the attributes of all models in a collection object) into Knockout observables for use with components' ViewModels.

The example below maps the `data` collection to the `tasks` ViewModel.

```
renderTaskViews = function(tasksData) {
    this.tasks = oj.KnockoutUtils.map(data);
}
```

To utilize the data model in Oracle JET applications, Oracle JET provides custom elements which support Knockout variables in the `data` attribute. For additional information about working with Oracle JET UI components, see Understanding Oracle JET User Interface Basics.

```
<oj-table  data="[[dataSource]]">
</oj-table>
```

For additional information about Knockout, see http://www.knockoutjs.com.

Using the Oracle JET Common Model and Collection API

To use a `Model` or `Collection` class, an application must extend `oj.Model` or `oj.Collection` to create a foundation object to represent a data record or list of records from its data service. The application provides the data service's URL used for fetching and updating task records (when a task record's ID is appended), along with various options giving users the ability to map data service records to their client-side ViewModel and vice versa.

To use the Oracle JET Common Model and Collection API:

1. Add JavaScript code to your application that extends `oj.Model`.

   The following script shows a simple example of extending `oj.Model` to create a model object which defines a `Department`. In this example, the data is returned from the REST service at the indicated URL. The `parse` callback parses the data and maps the attributes received from the REST service to desired ViewModel attribute names.

   ```
   var self = this;
   self.serviceURL = 'http://RESTServerIP:port/stable/rest/Departments';
   self.Department = oj.Model.extend({
   ```

```
        urlRoot: self.serviceURL,
        parse: self.parseDept,
        idAttribute: 'DepartmentId'
    });

/**
 * Callback to map attributes returned from RESTful data service to desired view
model attribute names
 */
self.parseDept = function(response) {
        return {DepartmentId: response['DepartmentId'],
            DepartmentName: response['DepartmentName'],
            LocationId: response['LocationId'],
            ManagerId: response['ManagerId']};
};
```

2. Add JavaScript code to your application that extends `oj.Collection`.

   The following code example creates a collection object for its entire data set (or list) of task records and ties that to a specific instance of a task record model. The `fetch()` method tells the collection to go to the data service and asynchronously retrieve the data services' data set using the given URL, through jQuery AJAX calls. The application's `success` callback method is invoked when the call successfully returns and the collection has been populated with model records.

```
self.DeptCol = ko.observable();
self.myDept = new self.Department();

// Create a base object "class" for the entire dataset
self.DeptCollection = oj.Collection.extend({
        url: self.serviceURL + "?limit=50",
        model: self.myDept
    });

// Create a specific instance for the departments.  This will be filled with
instances of the
// model "department" for each record when the data is retrieved from the data
service
self.DeptCol(new self.DeptCollection());
self.datasource(new oj.CollectionTableDataSource(self.DeptCol()));
```

3. Bind the returned data collection to a Knockout ViewModel to make it ready for consumption by one or more components on the application page.

   The following code sample maps the `vm Collection` object to the element in the view with the ID `'table'` using the `ko.applyBindings()` function.

```
var vm = new viewModel;

$(document).ready(function() {
        ko.applyBindings(vm, document.getElementById('table'));
    });
```

4. Add code to the application's index.html or main page that consumes the Knockout ViewModel.

   The following code examples shows a simple table that is defined with four columns: `Department Id`, `Department Name`, `Location Id`, and `Manager Id`. The `data` attribute binds the `datasource` collection to the ViewModel using Knockout.

```
<oj-table id="table" summary="Department List" aria-label="Departments Table"
                                        data='[[datasource]]'
                                        columns='[{"headerText": "Department
```

```
Id",
                                                        "field": "DepartmentId"},
                                                       {"headerText": "Department
Name",
                                                        "field": "DepartmentName"},
                                                       {"headerText": "Location Id",
                                                        "field": "LocationId"},
                                                       {"headerText": "Manager Id",
                                                        "field": "ManagerId"}]'>
</oj-table>
```

For a complete list of Oracle JET Common Model and Collection API properties and functions, see the oj.Model and oj.Collection API documentation.

Integrating REST Services

Oracle JET is designed to work with any web service that returns data in the form of JSON objects. Because the content of JSON objects can vary widely from service type, the application developer is responsible for examining the JSON object's content and defining the ViewModel appropriately.

Topics:

- About Oracle JET Support for Integrating REST Services
- Passing Custom AJAX Options in Common Model CRUD API calls
- Supplying a customURL Callback Function
- Replacing oj.sync or oj.ajax Functions

About Oracle JET Support for Integrating REST Services

The Model and Collection classes provide native support for mapping JSON objects obtained from REST web services to Knockout observables. The `oj.Collection.url` property specifies the path to the web service data.

```
oj.Collection.extend({url:"http://myserver/departments",...})
```

Model objects also include the `url` property. By default, this is set to the model ID appended to the `oj.Collection.url` property: http://*myserver*/departments/*modelID*.

To override the default path set by the Collection object, set the `oj.model.urlRoot` property, and the application will use the *`url.Root/modelID`* as the path to the data.

Oracle JET also provides three ways to customize the AJAX requests that Oracle JET makes when accessing REST services through the Common Model.

- Pass custom AJAX options in Common Model CRUD API calls.
- Supply a `customURL` callback function.
- Replace the `oj.sync` or `oj.ajax` functions.

Passing Custom AJAX Options in Common Model CRUD API calls

Use this method if the default URL behavior is acceptable, but your application needs to pass custom properties such as headers to the REST server.

To customize the AJAX options, add property/value pairs to the `options` argument of the `oj.Collection.create(create)`, `oj.Collection.fetch(read)`,`oj.Model.save(update)`, and `oj.Model.destroy(delete)` functions.

The following code example shows how an application could pass in a custom header when doing a read from the REST service.

```
myOjCollection.fetch(
                     {headers:{my-custom-header:"header-value"},
                      beforeSend: myBeforeSendCallbackFunc,
                      success:function(collection){
                     }});
```

Supplying a customURL Callback Function

If you need to customize the URL or AJAX behavior, set the `oj.Collection.customURL` property to a user-defined callback function when defining the collection.

```
function myCustomFunction(operation, collection, options) {
...
}
var myCollection = oj.Collection.extend({customURL:myCustomFunction,...})
```

The callback function should include the following parameters which are passed to it by default:

- `operation`: `create`, `read`, `update`, `patch`, or `delete`
- `collection`: The `oj.Model` or `oj.Collection` object requesting the `operation`
- `options`: Information relevant to the operation

Typical options include:

- `recordID`: Id of the record involved in the operation
- `fetchSize`: The number of records to return; returns all if not set
- `startIndex`: Starting record number of the data set
- `fromID`: Retrieve records starting with or after the record with the given unique ID. Based on your REST server's interpretation, the record with the given ID may or may not be included.
- `since`: Retrieve records with timestamps after the given timestamp
- `until`: Retrieve records with timestamps up to the given timestamp
- `sort`: Field or fields by which to sort
- `sortDir`: Specific ascending or descending order

The following example shows a `customURL` callback. By setting this user-created callback function on the Collection's `customURL` property, the Collection will call out to this function when it needs information to make a REST call.

```
function myCustomFunction(operation, collection, options) {
     // Use the default if operation is create
     if (operation === "create") {
         return null;
    }
     if (operation === "delete") {
        // Just set a URL if it's delete
        return "http://destroy/model='+options['recordID'];
```

```
    }
    var retObj = {};
    if (operation === "read") {
        retObj['url'] = "http://fetch";
        if (options['sort']) {
          retObj['url'] += "/order="+options['sort'] + ";" + options['sortDir'];
        }
        retObj['headers'] = {my-custom-header:"header-value"};
        retObj['mimeType'] = "text/plain";
        return retObj;
    }
    // Update or patch
    retObj['url'] = "http://update/model="+options['recordID'];
    retObj['type'] = "POST";
    retObj['beforeSend'] = myBeforeSendCallback;
    return retObj;
}
```

Replacing oj.sync or oj.ajax Functions

You can replace `oj.sync()` or `oj.ajax()` when you need to change the entire client-server transport mechanism. For example, this option is best if you want to use WebSockets instead of AJAX calls to the server.

The `oj.sync` method is the master server access method for all models and collections and accepts the following parameters:

```
oj.sync = function(method, model, options)
```

Valid values for the `method` parameter are the CRUD operations: `create`, `read`, `update`, `patch`, and `delete`. The `model` parameter accepts either the `oj.Model` being created, read, updated, patched, or deleted, or the `oj.Collection` being read. Options are passed down from the higher-level common model API call and vary with the type of operation.

The replacement `oj.sync()` method is completely responsible for implementing all of these operations using whatever transport mechanism is being used, whether it be the application's own AJAX routines, WebSockets, a JavaScript server, or something else. The method must return a Promise object similar to the AJAX `XMLHttpRequest` (XHR) in order to maintain compatibility with potential virtual API calls being made by the `oj.Model` and `oj.Collection` object.

> **Note:**
>
> Replacing `oj.sync()` replaces the transport mechanism for the JET common model used by all calls within the application. This is a very advanced use of the Oracle JET common model.

The `oj.ajax()` method is the master AJAX entry point for all `oj.Model` and `oj.Collection` server interactions, when they are using the default sync implementations. `oj.ajax()` passes its parameters and return value through to the `jQuery.ajax()` method by default. For additional information about the jQuery.ajax() method's expected parameters and return value, see http://api.jquery.com/jquery.ajax.

For additional information about the `oj.sync()` or `oj.ajax()` methods, see the Oracle JET oj.sync() and oj.ajax() API documentation.

Creating a CRUD Application Using Oracle JET

Use Knockout and the Oracle JET Common Model API to create applications that perform CRUD (Create, Read, Update, Delete) operations on data returned from a REST Service API.

Topics:

- Defining the ViewModel
- Reading Records
- Creating Records
- Updating Records
- Deleting Records

> **Note:**
>
> The application shown in this section also includes code for defining and displaying header and footer detail. You can download the complete sample application here: OracleJET-CommonModel-CRUD.zip.

Defining the ViewModel

Identify the data source for your application and create the ViewModel.

1. Identify your data source and examine the data. For data originating from a REST service, identify the service URL and navigate to it in a browser.

   The following example shows a portion of the output of a REST service that returns department data for a fictitious organization from a REST server named `MockRESTServer`.

```
{
  "Departments" : [ {
    "DepartmentId" : 10,
    "DepartmentName" : "Administration",
    "ManagerId" : null,
    "LocationId" : null,
    "version" : "ACED00057...contents truncated",
    "links" : {
      "self" : {
        "rel" : "self",
        "href" : "http://mockrest/stable/rest/Departments/10"
      },
      "canonical" : {
        "rel" : "canonical",
        "href" : "http://mockrest/stable/rest/Departments/10"
      },
      "Employees" : {
        "rel" : "child",
        "href" : "http://mockrest/stable/rest/Departments/10/Employees"
```

```
      }
    }
  }, {
    "DepartmentId" : 20,
    "DepartmentName" : "Retail Marketing",
    "ManagerId" : null,
    "LocationId" : null,
    "version" : "ACED00057...contents truncated",
    "links" : {
      "self" : {
        "rel" : "self",
        "href" : "http://mockrest/stable/rest/Departments/20"
      },
      "canonical" : {
        "rel" : "canonical",
        "href" : "http://mockrest/stable/rest/Departments/20"
      },
      "Employees" : {
        "rel" : "child",
        "href" : "http://mockrest/stable/rest/Departments/20/Employees"
      }
    }
  }, {
  ... contents omitted
  } ],
  "links" : {
    "self" : {
      "rel" : "self",
      "href" : "http://mockrest/stable/rest/Departments"
    }
  },
  "_contextInfo" : {
    "limit" : 25,
    "offset" : 0
  }
}
```

In this example, each department is identified by its `DepartmentId` and contains information about its name (`DepartmentName`), manager (`ManagerId`), and location (`LocationId`). Each department also contains employees (`Employees`) which are children of each department.

> **Tip:**
>
> The Oracle JET Common Model CRUD sample application uses JSON data returned from a mock rest service. You can find the mock rest service scripts in the `public_html/js/rest` folder. You can find the sample JSON data in the `departments.json` file located in the `public_html/js` folder.

2. Determine the data you will need for your collection.

   For example, the following figure shows a simple Oracle JET table that uses the `DepartmentId`, `DepartmentName`, `LocationId`, and `ManagerId` returned from the REST service identified in the previous step to display a table of department IDs, names, location IDs, and manager IDs. The table element is defined as an `oj-table` element, which is included in the Oracle JET UI component library.

Department Id	Department Name	Location Id	Manager Id
10	Administration		
20	Marketing		
30	Transportation		
40	Shipping		
50	Human Resources		
60	Operations		
70	Inventory		
80	Sales	2500	2500
100	Finance		
110	Documentation		
130	Billing	1700	1700
140	Control And Credit	1700	1700

3. Add a JavaScript function to your application that will contain your ViewModel.

   The following code shows a skeleton function that defines the ViewModel in the Oracle JET Common Model CRUD application. In this example, the function is stored in a file named `app.js`. The code to complete the ViewModel will be defined in upcoming steps.

```
define(['ojs/ojcore', 'knockout', 'ojs/ojmodel', 'MockRESTServer'],
        function(oj, ko)
        {
            function viewModel() {
            // To be defined
            };
            return {'deptVM': viewModel};
        }
)
```

> **Note:**
>
> This example uses RequireJS for modular development. The RequireJS bootstrap file will be shown in a later step. For additional information about using RequireJS, see Using RequireJS for Modular Development.

4. Add a JavaScript function to the function you defined in the previous step that defines the data model using `oj.Model.extend()`.

   The highlighted code in the example below defines the data model for the application shown in the preceding figure. The `Department` variable represents a single record in the database and is displayed as a row in the table. `Department` is declared using the `oj.Model.extend` function call and instantiated in the declaration for `myDept`. The `urlRoot` property defines the data source, which in this case is the REST service URL.

```
define(['ojs/ojcore', 'knockout', 'ojs/ojmodel', 'MockRESTServer'],
       function(oj, ko)
       {
           function viewModel() {
               var self = this;
               self.serviceURL = 'http://mockrest/stable/rest/Departments';;

           function parseDept(response) {
               if (response['Departments']) {
                   var innerResponse = response['Departments'][0];
                   if (innerResponse.links.Employees == undefined) {
                       var empHref = '';
                   } else {
                       empHref = innerResponse.links.Employees.href;
                   }
                   return {DepartmentId: innerResponse['DepartmentId'],
                           DepartmentName: innerResponse['DepartmentName'],
                           links: {Employees: {rel: 'child', href: empHref}}};
               }
               return {DepartmentId: response['DepartmentId'],
                       DepartmentName: response['DepartmentName'],
                       LocationId:response['LocationId'],
                       ManagerId:response['ManagerId'],
                       links: {Employees: {rel: 'child', href: response['links']
['Employees'].href}}};
           }

           // Think of this as a single database record or a single table row.
               var Department = oj.Model.extend({
                   urlRoot: self.serviceURL,
                   parse: parseDept,
                   idAttribute: 'DepartmentId'
               });

               var myDept = new Department();
           };

           return {'deptVM': viewModel};
       };
}
```

The `parse` property is an optional user callback function to allow parsing of JSON record objects as they are returned from the data service. In this example, `parseDept` is the callback function and simply maps the `DepartmentId` and `DepartmentName` returned from the REST service to `DepartmentId` and `DepartmentName`. If the `LocationId` or `ManagerId` records contain data, `parseDept` maps the attributes to `LocationId` and `ManagerId`.

The parse callback can be useful for mapping database attribute names to names that may make more sense. For example, if your database uses `Id` and `Name` as the attributes that represent the department ID and department name, you could replace the return call in the `parseDept` function with:

```
return {DepartmentId: response['Id'], DepartmentName: response['Name']};
```

For a complete list of `oj.Model` properties and functions, see the oj.Model API documentation.

5. Define the collection that will hold the data model object you defined in the previous step using `oj.Collection.extend()`.

The highlighted code in the example below defines the collection object for the `Department` model object. The `DeptCollection` variable is declared in the `viewModel()` function using the `oj.Collection.extend` function and instantiated in the declaration for `self.DeptCol`. The `url` property defines the data source, which in this case is the REST service URL, and limits the collection to 50 records.

```
define(['ojs/ojcore', 'knockout', 'ojs/ojmodel', 'MockRESTServer'],
    function(oj, ko){
        function viewModel() {
            var self = this;
            self.serviceURL = 'http://mockrest/stable/rest/Departments';
            self.Departments = ko.observableArray([]);
            self.DeptCol = ko.observable();
            self.datasource = ko.observable();

            var parseDept = function(response) {
            ... contents omitted
            };

            var Department = oj.Model.extend({
                urlRoot: self.serviceURL,
                parse: parseDept,
                idAttribute: 'DepartmentId'
            });

            var myDept = new Department();

            // this defines our collection and what models it will hold
                var DeptCollection = oj.Collection.extend({
                    url: self.serviceURL + "?limit=50",
                    model: myDept
                });

                self.DeptCol(new DeptCollection());
                self.datasource(new
oj.CollectionTableDataSource(self.DeptCol()));

            }
            return {'deptVM': viewModel};
        }
);
```

Both the `DeptCol` and `datasource` objects are defined as Knockout observables so that changes to the data collection can be handled. The `datasource` object will contain the column data needed by the `oj-table` element and is passed the `DeptCol` observable as a parameter to `oj.CollectionTableDataSource()`.

The `Departments` object is defined as a Knockout observable array and will be populated in a later step.

For a complete list of `oj.Collection` properties and functions, see the oj.Collection API documentation.

6. Populate the collection with data by calling `oj.Collection.fetch()` to read the data from the data service URL.

   The highlighted code in the code sample below calls `oj.collection.fetch()` to add data to the `DeptCol` data collection and complete the ViewModel.

```
define(['ojs/ojcore', 'knockout', 'ojs/ojmodel', 'MockRESTServer'],
         function(oj, ko){
            function viewModel() {
```

```
                var self = this;
                self.serviceURL = 'http://mockrest/stable/rest/Departments';
                self.Departments = ko.observableArray([]);
                self.DeptCol = ko.observable();
                self.datasource = ko.observable();
                self.fetch = function(successCallBack) {
                    // populate the collection by calling fetch()
                    self.DeptCol().fetch({
                        success: successCallBack,
                        error: function(jqXHR, textStatus, errorThrown){
                            console.log('Error in fetch: ' + textStatus);
                        }
                    });
                };

                var parseDept = function(response) {
                   ... contents omitted
                };

                var Department = oj.Model.extend({
                    urlRoot: self.serviceURL,
                    parse: parseDept,
                    idAttribute: 'DepartmentId'
                });

                var myDept = new Department();

                var DeptCollection = oj.Collection.extend({
                    url: self.serviceURL + "?limit=50",
                    model: myDept
                });

                self.DeptCol(new DeptCollection());
                self.datasource(new
oj.CollectionTableDataSource(self.DeptCol()));

            }
            return {'deptVM': viewModel};
        }
);
```

The `fetch()` function also defines an error callback that will log a message to the console if the `fetch()` call fails.

7. Add the ViewModel or the file containing the name of your ViewModel to your RequireJS bootstrap file, typically `main.js`.

   If you created your Oracle JET application using an Oracle JET Starter template or modified your Oracle JET download as described in Use RequireJS to Manage Library, Link, and Script References, you should already have a `main.js` file. Locate the line that defines the require modules and add your file to the list.

   For example, the code below lists the modules defined for the Common Model Sample. The application stores its ViewModel in the `app.js` file. The reference to the `app.js` file is highlighted in bold.

```
require(['ojs/ojcore',
        'knockout',
        'jquery',
        'app',
        'footer',
        'MockRESTServer'
```

```
        'ojs/ojmodel',
        'ojs/ojknockout',
        'ojs/ojknockout-model',
        'ojs/ojdialog',
        'ojs/ojinputtext',
        'ojs/ojinputnumber',
        'ojs/ojbutton',
        'ojs/ojtable',
        'ojs/ojcollectiontabledatasource'],
```

You must also add the app reference in the callback definition as shown in the following example.

```
// this callback gets executed when all required modules are loaded
function(ko, $, oj, app, footer, MockRESTServer)
{
    ...
}
```

8. Update your RequireJS bootstrap file to instantiate the ViewModel, create the Knockout bindings, and display the content on the page.

   The highlighted code in the code sample below creates a Knockout observable for each element in the `deptData` collection and assigns the resulting array to the `Departments` Knockout observable array you defined in a previous step.

```
require(['ojs/ojcore',
        'knockout',
        'jquery',
        'app',
        'footer',
        'MockRESTServer'
        'ojs/ojmodel',
        'ojs/ojknockout',
        'ojs/ojknockout-model',
        'ojs/ojdialog',
        'ojs/ojinputtext',
        'ojs/ojinputnumber',
        'ojs/ojbutton',
        'ojs/ojtable',
        'ojs/ojcollectiontabledatasource'],
        function(ko, $, oj, app, footer, MockRESTServer) // this callback gets
executed when all required modules are loaded
        {
          var fvm = new footer.footerVM();
          $(document).ready(function(){
                $.getJSON("js/departments.json",
                        function (data) {
                    new MockRESTServer(data, {id:"DepartmentId",
                        url:/^http:\/\/mockrest\/stable\/rest\/Departments(\?
limit=([\d]*))?$/i,
                        idUrl:/^http:\/\/mockrest\/stable\/rest\/Departments\/
([\d]+)$/i});

            var vm = new app.deptVM();
            ko.applyBindings(fvm, document.getElementById('footerContent'));
            vm.fetch(
                function(collection, response, options){
                  var deptData = collection;
                  // This will create a ko.observable() for each element
                  // in the deptData response and assign the resulting array
                  // to the Departments ko observeableArray.
```

```
                        vm.Departments = oj.KnockoutUtils.map(deptData, null, true);
                        //perform a Knockout applyBindings() call binding this
                        // viewModel with the current DOM
                        ko.applyBindings(vm, document.getElementById('mainContent'));
                        //Show the content div after the REST call is completed.
                        $('#mainContent').show();
                    });
                      });
                            });
                }
        );
```

Reading Records

To read the records, define the Oracle JET elements that will read the records in your main HTML5 page.

The following sample code shows a portion of the `index.html` file that displays a table of records using the ViewModel defined in the previous steps and the `oj-table` element. In this example, the `mainContent div` includes the table definition that creates `Department Id`, `Department Name`, `Location Id`, and `Manager Id` as the table header text and defines the content for each row.

```
<div id="mainContent" class="oj-md-12 oj-flex-item page-padding" style="display:
none;">
    <div class="page-padding">
        <oj-table id="table" data="[[datasource]]"
              columns='[{"headerText": "Department Id",
                         "field": "DepartmentId", "sortable": "enabled"},
                        {"headerText": "Department Name",
                         "field": "DepartmentName", "sortable": "enabled"},
                        {"headerText": "Location Id",
                         "field": "LocationId"},
                        {"headerText": "Manager Id",
                         "field": "ManagerId"}]'>
        </oj-table>
    </div>
</div>
```

The `data` attribute reads the variable `datasource` from the REST service and binds it to the `oj-table` element.

Creating Records

To add the ability to create new records, add elements to your HTML5 page that accept input from the user and create a function that sends the new record to the REST server.

The figure below shows the result of adding a form using `oj-input-*` elements to the Oracle JET Common Model sample application. The user can enter a new department number in the provided field or use the side arrows to increment or decrement the value. The user then enters the name and clicks **Add**.

Department Id	Department Name	Location Id	Manager Id
10	Administration		
20	Marketing		
30	Transportation		
40	Shipping		
50	Human Resources		
60	Operations		
70	Inventory		
80	Sales	2500	2500
100	Finance		
110	Documentation		
130	Billing	1700	1700
140	Control And Credit	1700	1700

New Department

Department Id 555

Department Name *enter new name*

Add Department

To add the ability to create new records to the application's ViewModel and HTML5 page:

1. Add elements to the application's main page that accept input from the user.

   The highlighted code in the example below adds `oj-input-*` elements to the `index.html` page shown in the previous task.

```
<div id="mainContent" class="oj-md-12 oj-flex-item page-padding" style="display:
none;">
    <div class="page-padding">
        <div id="deptList" class="oj-md-9 oj-flex-item">
            <oj-table id="table" summary="Demo Table"
                ... contents omitted
            </oj-table>
            <br/>
            <div id="addDept" class="oj-flex-item oj-md-3 oj-sm-12 right">
              <div id="quickUpdate" class="frame">
                <div id="newDeptForm">
                  <div class="oj-form oj-sm-odd-cols-12 oj-md-odd-cols-6 oj-md-
labels-inline">
                    <h3>New Department</h3><hr/>
                    <div class="oj-flex">
                      <div class="oj-flex-item">
                        <oj-label for="newDepartId">Department Id</oj-label>
                      </div>
```

```
                      <div class="oj-flex-item">
                        <oj-input-number id="newDepartId" value= "555"></oj-
input-number>
                      </div>
                    </div>
                    <div class="oj-flex">
                      <div class="oj-flex-item">
                        <oj-label for="newDepartName">Department Name</oj-
label>
                      </div>
                      <div class="oj-flex-item">
                        <oj-input-text id="newDepartName" maxlength='30'
placeholder="enter new name"></oj-input-text>
                        <oj-button id="saveBtn" on-click="[[addDepartment]]"
label='Add Department'></oj-button>
                      </div>
                    </div>
                  </div>
                </div>
              </div>
            </div>
          </div>
    </div>
</div>
```

The form is defined to contain two input fields: an `oj-input-number` for the department ID, and an `oj-input-text` for the department name. The `oj-button` component is used for the form's button.

The `oj-button`'s `on-click` attribute is bound to the `addDepartment` function which is defined in the next step.

2. Add code to the ViewModel to add the user's input as a new record (model) in the data collection.

   The highlighted code in the example below shows the `addDepartment()` function that adds the new department number and department name to the `DeptCol` data collection. In this example, the function calls the `oj.Collection.create()` method which creates the new model instance, adds it to the data collection, and saves the new `DepartmentId` and `DepartmentName` to the data service.

```
define(['ojs/ojcore', 'knockout', 'ojs/ojmodel'],
         function(oj, ko)        {
            function viewModel() {
                var self = this;
                self.serviceURL = 'http://RESTServerIP:Port/stable/rest/
Departments';
                self.Departments = ko.observableArray([]);
                self.DeptCol = ko.observable();
                self.datasource = ko.observable();
                self.fetch = function(successCallBack) {
                    ... contents omitted
                };

                function parseDept(response){
                    ... contents omitted
                };

                var Department = oj.Model.extend({
                    urlRoot: self.serviceURL,
                    parse: parseDept,
```

```
                    parseSave:parseSaveDept,
                    idAttribute: 'DepartmentId'
                });

                var myDept = new Department();

                var DeptCollection = oj.Collection.extend({
                    url: self.serviceURL + "?limit=50",
                    model: myDept
                    comparator: 'DepartmentId'
                });

                self.DeptCol(new DeptCollection());
                self.datasource(new
oj.CollectionTableDataSource(self.DeptCol()));

            function parseSaveDept(response){
                return {DepartmentId: response['DepartmentId'],
                        DepartmentName: response['DepartmentName'],
                        LocationId: response['LocationId'],
                        ManagerId: response['ManagerId'],
                        links: {Employees: {rel: 'child', href:
response['links']['Employees'].href}}};
                };

            self.addDepartment = function (formElement, event) {
                var id = $("#newDepartId").val();
                var recordAttrs = {DepartmentId: id,
                        DepartmentName: $("#newDepartName").val(),
                        ManagerId: "", LocationId: "",
                        links: {Employees: {rel: 'child', href:
self.serviceURL + '/' + id + '/Employees'}}};
                this.DeptCol().create(recordAttrs,{
                    'contentType': 'application/vnd.oracle.adf.resource+json',
                    success: function (response) {
                        console.log('Success in Create');
                    },
                    error: function(jqXHR, textStatus, errorThrown){
                        console.log('Error in Create: ' + textStatus);
                        }
                    });
                }
            }
            return {'deptVM': viewModel};
        }
);
```

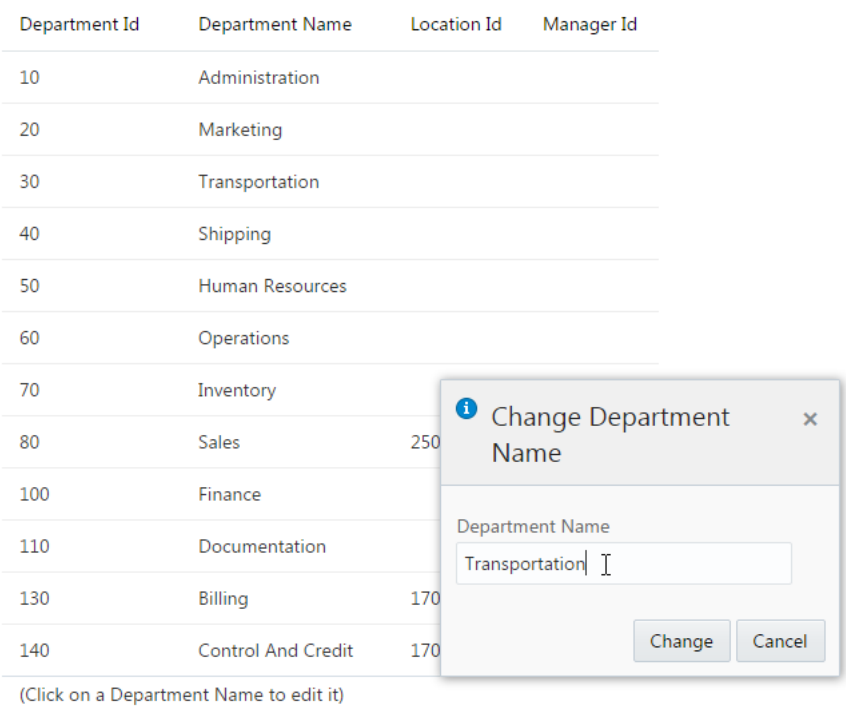
The `oj.Collection.create()` function accepts options to control the record save. In this example, simple error checking is added. In addition, the `create()` function sends a `contentType` to the REST server. Depending upon the REST service you are using, this option may not be required or may need modification.

Updating Records

To add the ability to update records, add elements to your HTML5 page that accept input from the user and create a function that sends the updated record to the REST server.

The figure below shows the Oracle JET Common Model sample application configured to allow updates to the department name. When the user moves the focus over a

department name, a tooltip appears that prompts the user to click to edit. department name, a tooltip prompts the user to click to edit. If the user clicks the department name, a dialog appears that enables the user to change the name. The user can type in a new department name and click **Change** to update the record or Cancel to keep the existing department name.



To add the ability to update records to the application's ViewModel and HTML5 page:

1. Add elements to the application's main page that identifies updatable elements and enables the user to perform an action to update them.

   The highlighted code in the example below adds the `oj-dialog` element to the page and provides the prompt to the user to click to edit the page.

```
<div id="mainContent" class="oj-flex-item oj-sm-12 oj-md-12 demo-page-content-
area page-padding" style="display: none;">
    <div class="page-padding">
          <div id="deptList" class="oj-flex-item oj-md-9 oj-sm-12">
            <oj-table id="table" data="[[datasource]]"
              columns='[{"headerText": "Remove", "id": "column1", "sortable":
"disabled"},
                        {"headerText": "Department Id",
                         "field": "DepartmentId", "sortable": "enabled"},
                        {"headerText": "Department Name",
                         "field": "DepartmentName", "sortable": "enabled"},
                        {"headerText": "Location Id",
                         "field": "LocationId"},
                        {"headerText": "Manager Id",
                         "field": "ManagerId"}]'
              selectionMode='{"row": "none", "column": "none"}'
              row-renderer='[[oj.KnockoutTemplateUtils.getRenderer("row_tmpl",
true)]]'>
```

```
            </oj-table>
    </div>
    <oj-dialog id="editDialog" style="display:none" title="Change Department
Name" dragAffordance="title-bar" modality="modeless" resize-behavior="none">
              <div class="oj-form oj-sm-odd-cols-12">
                <!-- Example of custom dialog header. Setting the class "oj-
dialog-header" will tell the dialog
                     to use the contents of that <div> as the header content.
Leaving this out will cause the
                     default header to be used, which uses the title attribute
as the header content value. -->
                <div class="oj-dialog-header oj-helper-clearfix" aria-
labelledby="dialog-title-id">
                  <div>
                    <span id="infoIcon" class="oj-message-status-icon oj-
message-info-icon" style="float:left; margin-right: 10px"></span>
                    <span id="dialog-title-id" class="oj-dialog-title">Change
Department Name</span>
                  </div>
                </div>
                <div class="oj-dialog-body">
                  <div class="oj-md-odd-cols-4">
                    <oj-label class="oj-label oj-label-inline"
for="newName">Department Name</oj-label>
                    <oj-input-text id="newName"
value="{{currentDeptName}}"></oj-input-text>
                  </div>
                </div>
                <div class="oj-dialog-footer">
                  <oj-button id="submitBtn" on-click="[[updateDeptName]]"
label="Change"></oj-button>
                  <oj-button id="resetBtn" on-click="[[cancelDialog]]"
label="Cancel"></oj-button>
                </div>
              </div>
            </oj-dialog>
          </div>
        </div>
      </div>
    </div>
<script type="text/html" id="row_tmpl">
      <tr>
        <td><div id='deptId' data-bind="text: DepartmentId"></div></td>
        <td><div id="deptName" data-bind="text: DepartmentName, click:
function(data, event){$root.showChangeNameDialog(DepartmentId,data,event)}"></
div></td>
        <td><div id="locId" data-bind="text: LocationId"></div></td>
        <td><div id="mgrId" data-bind="text: ManagerId"></div></td>
      </tr>
    </script>
```

The `oj-dialog` element includes the `oj-dialog-header`, `oj-dialog-body`, and `oj-dialog-footer` formatting classes. The `oj-dialog-header` class is optional and is used in this example to format the `Info` icon and title.

The `oj-dialog-body` class formats the dialog body which includes the `oj-label` for `Department Name` and an `oj-input-text` element to capture the user's input. The `oj-dialog-footer` defines two `oj-button` components which add the `Change` and `Cancel` buttons to the dialog. When the user clicks **Change**, the `updateDepartmentName()` function handles the updates to the record.

The original table is also modified to define the `row-renderer` attribute which specifies the `row_tmpl` template to use for the row display. The template script calls the `showChangeNameDialog()` function to display the dialog.

The `updateDepartmentName()` and `showChangeNameDialog()` functions are defined in the next step. For additional information about the `oj-dialog` component, see the oj-dialog API documentation.

2. Add code to the ViewModel to update the record.

   The highlighted code in the example below shows the `updateDepartment()` and `showChangeNameDialog()` functions.

```
define(['ojs/ojcore', 'knockout', 'ojs/ojmodel', 'MockRESTServer'],
         function(oj, ko)        {
            function viewModel() {
                var self = this;
                self.serviceURL = 'http://mockrest/stable/rest/Departments';
                self.Departments = ko.observableArray([]);
                self.currentDeptName = ko.observable('default');
                self.workingId = ko.observable('');
                self.DeptCol = ko.observable();
                self.datasource = ko.observable();
                self.fetch = function(successCallBack) {
                    ... contents omitted
                };

                function parseDept(response){
                    ... contents omitted
                }

                function parseSaveDept(response){
                    ... contents omitted
                }

                var Department = oj.Model.extend({
                    urlRoot: self.serviceURL,
                    parse: parseDept,
                    parseSave:parseSaveDept,
                    idAttribute: 'DepartmentId'
                });

                var myDept = new Department();

                var DeptCollection = oj.Collection.extend({
                    url: self.serviceURL + "?limit=50",
                    model: myDept
                    comparator: 'DepartmentId'
                });

                self.DeptCol(new DeptCollection());
                self.datasource(new
oj.CollectionTableDataSource(self.DeptCol()));

                self.showChangeNameDialog = function(deptId, data, event) {
                    var currName = data.DepartmentName;
                    self.workingId(deptId);
                    self.currentDeptName(currName);
                    document.getElementById("editDialog").open();
                }
```

```
                self.updateDeptName = function(formData, event) {
                    var currentId = self.workingId();
                    var myCollection = self.DeptCol();
                    var myModel = myCollection.get(currentId);
                    var newName = self.currentDeptName();
                    if (newName != myModel.get('DepartmentName') && newName !=
''') {
                        myModel.save({'DepartmentName': newName}, {
                            success: function(myModel, response, options) {
                              document.getElementById("editDialog").close();
                            },
                            error: function(jqXHR, textStatus, errorThrown) {
                                alert("Update failed with: " + textStatus);
                                document.getElementById("editDialog").close();
                            }
                        });
                    } else {
                        alert('Department Name is not different or the new
name is not valid');
                        document.getElementById("editDialog").close();
                    }
                };
            return {'deptVM': viewModel};
        }
);
```

The `showChangeNameDialog()` function stores the selected department detail and opens the dialog with the existing department name shown in the `oj-input-text` field.

The `updateDepartment()` function calls the `oj.Model.save()` method to save the current `Model` object to the data source. The function also defines success and error callbacks to close the dialog upon success or issue an error message if the record was not updated.

Deleting Records

To add the ability to delete records, add elements to your HTML5 page that accept input from the user and create a function that sends the new record for deletion to the REST server.

The figure below shows the Oracle JET Common Model CRUD application configured to allow record deletion. The user can check one or more departments in the list and click **Remove Department** to delete the record or records.

Remove	Department Id	Department Name	Location Id	Manager Id
☐	10	Administration		
☐	20	Marketing		
☑	30	Transportation		
☐	40	Shipping		
☐	50	Human Resources		
☐	60	Operations		
☐	70	Inventory		
☐	80	Sales	2500	2500
☐	100	Finance		
☐	110	Documentation		
☐	130	Billing	1700	1700
☐	140	Control And Credit	1700	1700

(Click on a Department Name to edit it)

Remove Department

To add the ability to delete records to the application's ViewModel and HTML5 page:

1. Add elements to the application's main page that identifies records marked for deletion and enables the user to perform an action to delete them.

   The highlighted code in the example below adds the `Remove` column with a check box to the department list and adds the `Remove Department` button below the list.

```
<div id="mainContent" class="oj-flex-item oj-sm-12 oj-md-12 demo-page-content-
area page-padding" style="display: none;">
        <div class="page-padding">
          <div id="deptList" class="oj-flex-item oj-md-9 oj-sm-12">
            <oj-table id="table" data="[[datasource]]"
              columns='[{"headerText": "Remove", "id": "column1", "sortable":
"disabled"},
                        {"headerText": "Department Id",
                         "field": "DepartmentId", "sortable": "enabled"},
                        {"headerText": "Department Name",
                         "field": "DepartmentName", "sortable": "enabled"},
                        {"headerText": "Location Id",
                         "field": "LocationId"},
                        {"headerture": "Manager Id",
                         "field": "ManagerId"}]'
              selectionMode='{"row": "none", "column": "none"}'
              row-renderer='[[oj.KnockoutTemplateUtils.getRenderer("row_tmpl",
true)]]'>
            </oj-table>
            <br/>
            <button id="deleteDept_btn" data-bind="enable: somethingChecked,
click: deleteDepartment">Remove Department</button>
          </div>
        </div>
```

```
      </div>

<script type="text/html" id="row_tmpl">
 <tr>
  <td><input type="checkbox" data-bind="attr: {id: DepartmentId},
click: $root.enableDelete"/></td>
  <td><div id='deptId' data-bind="text: DepartmentId"></div></td>
  <td><div id="deptName" data-bind="text: DepartmentName, click: function(data,
event){$root.showChangeNameDialog(DepartmentId,data,event)}"></div></td>
  <td><div id="locId" data-bind="text: LocationId"></div></td>
  <td><div id="mgrId" data-bind="text: ManagerId"></div></td>
 </tr>
</script>
```

The original table is modified to include the `Remove` column. The `row-renderer` attribute specifies the `row_tmpl` template to use for the row display. The template script adds the `checkbox` input element to the first column and the value of `DepartmentId`, `DepartmentName`, and `LocationId` to the remaining columns.

The button's `click` action is bound to the `deleteDepartment` function which is created in the next step.

2. Add code to the ViewModel to delete the record or records submitted by the user.

   The highlighted code in the example below shows the `deleteDepartment()` function. In this example, the function calls the `oj.Collection.remove()` method which removes the model or models from the data collection. To delete the record from the data source, the function calls the `oj.Model.destroy()` method.

```
define(['ojs/ojcore', 'knockout', 'ojs/ojmodel', 'MockRESTServer'],
         function(oj, ko)         {
             function viewModel() {
                var self = this;
                self.serviceURL = 'http://mockrest/stable/rest/Departments';
                self.Departments = ko.observableArray([]);
                self.DeptCol = ko.observable();
                self.datasource = ko.observable();
                self.somethingChecked = ko.observable(false);
                self.fetch = function(successCallBack) {
                    ... contents omitted
                }

                var Department = oj.Model.extend({
                    urlRoot: self.serviceURL,
                    idAttribute: 'DepartmentId'
                });

                var myDept = new Department();

                var DeptCollection = oj.Collection.extend({
                    url: self.serviceURL + "?limit=50",
                    model: myDept
                    comparator: 'DepartmentId'
                });

                self.DeptCol(new DeptCollection());
                self.datasource(new
oj.CollectionTableDataSource(self.DeptCol()));

                self.enableDelete = function() {
                    if (!$('input[type=checkbox]:checked').length) {
                        self.somethingChecked(false);
```

```
                    } else {
                        self.somethingChecked(true);
                    }
                    return true;
                }

                self.deleteDepartment = function(data, event) {
                    var deptIds = [];
                    deptIds = self.findDeptIds();
                    var collection = data.DeptCol();
                    deptIds.forEach(function(value, index, arr) {
                        var model = collection.get(parseInt(value));
                        if (model) {
                            collection.remove(model);
                            model.destroy();
                        }
                    });
                    self.enableDelete();
                    $('#table').ojTable('refresh');
                }

                self.findDeptIds = function() {
                    var selectedIdsArray = [];
                    $("input:checkbox").each(function() {
                        var cb = $(this);
                        if (cb.is(":checked")) {
                            selectedIdsArray.push(cb.attr("id"));
                        }
                    });
                    return selectedIdsArray;
                }
            }
            return {'deptVM': viewModel};
        }
);
```

The `deleteDepartment()` function calls the `findDeptIds()` function which returns the list of selected departments marked for deletion. The `enableDelete()` function resets the check box after the department list is deleted.

10
Validating and Converting Input

Oracle JET includes validators and converters on a number of Oracle JET editable elements, including `oj-combobox`, `oj-input*`, and `oj-text-area`. You can use them as is or customize them for validating and converting input in your Oracle JET application. Some editable elements such as `oj-checkboxset`, `oj-radioset`, and `oj-select` have a simple attribute for `required` values that implicitly creates a built-in validator.

> **Note:**
>
> The `oj-input*` mentioned above refers to the family of input components such as `oj-input-date-time`, `oj-input-text`, and `oj-input-password`, among others.

Topics:

- Typical Workflow for Validating and Converting Input
- About Oracle JET Validators and Converters
- Using Oracle JET Converters
- Using Oracle JET Validators

Typical Workflow for Validating and Converting Input

Understand Oracle JET's validation and conversion framework before working with it. Optionally, learn how to create custom converters and validators to extend the framework.

To validate and convert input in Oracle JET, refer to the typical workflow described in the following table:

Task	Description	More Information
Understand Oracle JET's validation and conversion framework	Understand the validators and converters included in Oracle JET and identify when you might need a custom validator or converter.	About Oracle JET Validators and Converters
Use Oracle JET converters in your application	Use `oj.ColorConverter`, `oj.IntlNumberConverter`, and `oj.IntlDateTimeConverter` to convert color, number, and date-time inputs to values expected by the view model.	Using Oracle JET Converters
Use a custom converter in your Oracle JET application	How to reference a custom converter in an Oracle JET application.	Using Custom Converters in Oracle JET

Task	Description	More Information
Use Oracle JET validators in your application	Use `oj.DateTimeRangeValidator`, `oj.DateRestrictionValidator`, `oj.LengthValidator`, `oj.NumberRangeValidator`, `oj.RegExpValidator`, and `oj.RequiredValidator` to validate user input.	Using Oracle JET Validators
Use a custom validator in your Oracle JET application	How to reference a custom validator in an Oracle JET application.	Using Custom Validators in Oracle JET

About Oracle JET Validators and Converters

Oracle JET provides converter classes that convert user input strings into the data type expected by the application and validator classes that enforce a validation rule on those input strings.

For example, you can use Oracle JET's `oj.IntlDateTimeConverter` to convert a user-entered date to a `Date` object for use by the application's ViewModel and then use `oj.DateTimeRangeValidator` to validate that input against a specified time range. You can also use converters to convert `Date` or `Number` objects to a string suitable for display or convert color object formats.

To retrieve the converter or validator factory for a registered type, Oracle JET provides the `oj.Validation` class which includes methods to register and retrieve converter and validator factories.

If the converters or validators included in Oracle JET are not sufficient for your application, you can create custom converters or validators. Optionally, you can provide a custom factory that implements the contract for a converter using `oj.ConverterFactory` or a validator using `oj.ValidatorFactory` and register the converter or validator with the `oj.Validation` class. The `oj.Validation` class enables you to access your custom converter or validator using the same mechanisms as you would use with the Oracle JET standard converters and validators.

Topics:

- About Validators
- About Converters

About Validators

All Oracle JET editable elements support a `value` attribute and provide UI elements that allow the user to enter or choose a value. These elements also support other attributes that page authors can set that instruct the element how it should validate its value.

An editable element may implicitly create a built-in converter and/or built-in validators for its normal functioning when certain attributes are set.

For example, editable elements that support a `required` property create the required validator implicitly when the property is set to `true`. Other elements like `oj-input-date`,

`oj-input-date-time`, and `oj-input-time` create a datetime converter to implement its basic functionality.

Topics

- Oracle JET Validators
- Oracle JET Component Validation Attributes
- Oracle JET Component Validation Methods

Oracle JET Validators

The following table describes the Oracle JET validators and provides links to the API documentation:

Validator	Description	Link to API
`oj.DateTimeRangeValidator`	Validates that the input date is between two dates, between two times, or within two date and time ranges	oj.DateTimeRangeValidator
`oj.DateRestrictionValidator`	Validates that the input date is not a restricted date	oj.DateRestrictionValidator
`oj.LengthValidator`	Validates that an input string is within a specified length	oj.LengthValidator
`oj.NumberRangeValidator`	Validates that an input number is within a specified range	oj.NumberRangeValidator
`oj.RegExpValidator`	Validates that the regular expression matches a specified pattern	oj.RegExpValidator
`oj.RequiredValidator`	Validates that a required entry exists	oj.RequiredValidator

Oracle JET Component Validation Attributes

The attributes that a component supports are part of its API, and the following validation specific attributes apply to most editable elements.

Element Attribute	Description
`converter`	When specified, the `converter` instance is used over any internal converter the element might create. On elements such as `oj-input-text`, you may need to specify this attribute if the value must be processed to and from a number or a date value.
`countBy`	When specified on `oj.LengthValidator`, `countBy` enables you to change the validator's default counting behavior. By default, this property is set to `codeUnit`, which uses JavaScript's String length property to count a UTF-16 surrogate pair as length === 2. Set this to `codePoint` to count surrogate pairs as length ===1.
`max`	When specified on an Oracle JET element like `oj-input-date` or `oj-input-number`, the element creates an implicit range validator.
`min`	When specified on an Oracle JET element like `oj-input-date` or `oj-input-number`, the component creates an implicit range validator.
`pattern`	When specified on an Oracle JET element like `oj-input-text`, `oj-input-password`, or `oj-text-area`, the component creates an implicit regExp validator using the pattern specified.

Element Attribute	Description
`placeholder`	When specified, it displays placeholder values in most elements.
`required`	When specified on an Oracle JET element, the element creates an implicit required validator.
`validators`	When specified, the element uses these validators along with the implicit validators to validate the UI value.

Some editable elements do not support specific validation attributes as they might be irrelevant to its intrinsic functioning. For example, `oj-radioset` and `oj-checkboxset` do not support a converter attribute since there is nothing for the converter to convert. For an exact list of attributes and how to use them, refer to the `Attributes` section in the element's API documentation. For Oracle JET API documentation, see JavaScript API Reference for Oracle® JavaScript Extension Toolkit (JET). Select the component you're interested in viewing from the API list.

Oracle JET Component Validation Methods

Oracle JET editable elements support the following methods for validation purposes. For details on how to call this method, its parameters and return values, refer to the component's API documentation.

Element Method	Description
`refresh()`	Use this method when the DOM the element relies on changes, such as the help attribute tooltip on an `oj-label` changing due to a change in locale.
`reset()`	Use this method to reset the element by clearing all messages and messages attributes - `messagesCustom` - and update the element's display value using the attribute value. User entered values will be erased when this method is called.
`validate()`	Use this method to validate the component using the current display value.

For details on calling a element's method, parameters, and return values, See the `Methods` section of the element's API documentation in JavaScript API Reference for Oracle® JavaScript Extension Toolkit (JET). You can also find detail on how to register a callback for or bind to the event and for information about what triggers the events. Select the component you're interested in viewing from the API list.

About Converters

The Oracle JET converters include date-time, number, and color converters and are described in the following table.

Converter	Description	Link to API
`oj.ColorConverter`	Converts oj.Color object formats	oj.ColorConverter
`oj.IntlDateTimeConverter`	Converts a string to a `Date` or a `Date` to a string	oj.IntlDateTimeConverter
`oj.IntlNumberConverter`	Converts a string to a number or formats a number or `Number` object value to a string	oj.IntlNumberConverter

Using Oracle JET Converters

The Oracle JET color, date-time, and number converters, `oj.ColorConverter`, `oj.IntlDateTimeConverter`, and `oj.IntlNumberConverter`, extend the `oj.Converter` object which defines a basic contract for converter implementations.

The converter API is based on the ECMAScript Internationalization API specification (ECMA-402 Edition 1.0) and uses the Unicode Common Locale Data Repository (CLDR) for its locale data. Both converters are initialized through their constructors, which accept options defined by the API specification. For additional information about the ECMA-402 API specification, see http://www.ecma-international.org/ecma-402/1.0. For information about the Unicode CLDR, see http://cldr.unicode.org.

The Oracle JET implementation extends the ECMA-402 specification by introducing additional options, including an option for user-defined patterns. For the list of additional options, see the oj.ColorConverter, oj.IntlDateTimeConverter, and oj.IntlNumberConverter API documentation.

For examples that illustrate the date-time and number converters in action, see the Converters section in the Oracle JET Cookbook. For examples using the color converter, see the Color Palette and Color Spectrum Cookbook samples.

> **Note:**
>
> The bundles that hold the locale symbols and data used by the Oracle JET converters are downloaded automatically based on the locale set on the page when using RequireJS and the `ojs/ojvalidation-base`, `ojs/ojvalidation-datetime`, or `ojs/ojvalidation-number` module. If your application does not use RequireJS, the locale data will not be downloaded automatically.

You can use the converters with an Oracle JET component or instantiate and use them directly on the page.

Topics:

- Using Oracle JET Converters with Oracle JET Components
- Understanding Oracle JET Converters Lenient Parsing
- Understanding Time Zone Support in Oracle JET
- Using Custom Converters in Oracle JET
- Using Oracle JET Converters Without Oracle JET Components

Using Oracle JET Converters with Oracle JET Components

Oracle JET elements that accept user input, such as `oj-input-date`, already include an implicit converter that is used when parsing user input. However, you can also specify an explicit converter on the element which will be used instead when converting data from the model for display on the page and vice versa. An explicit converter is required if you want to include time zone data.

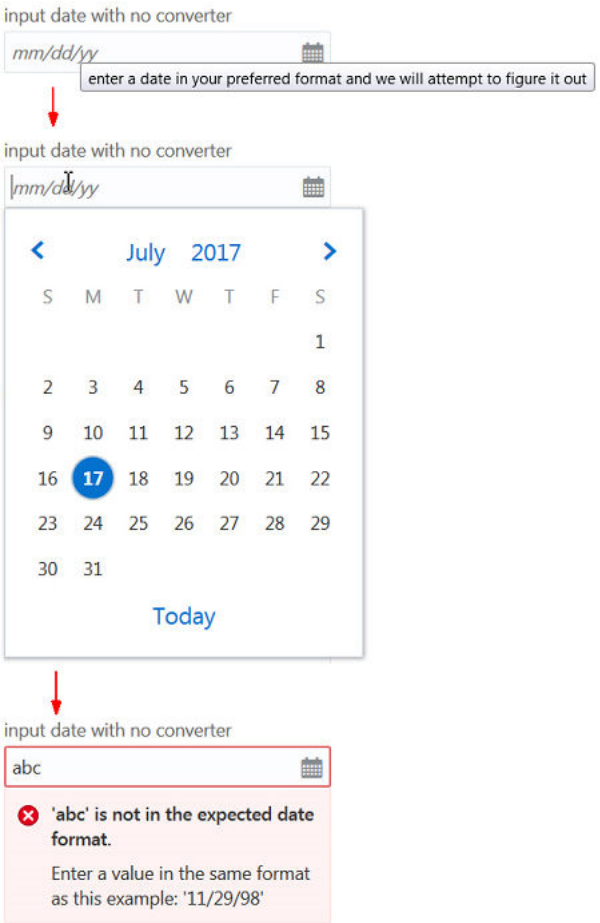
For example, the following code sample shows a portion of a form containing an `oj-input-date` component that uses the default converter supplied by the component implicitly. The highlighted code shows the `oj-input-date` component.

```
<div id="datetime-converter-example">
  ... contents omitted
  <div class="oj-flex">
    <div class="oj-flex-item">
      <oj-label for="date1">input date with no converter</oj-label>
    </div>
    <div class="oj-flex-item">
      <oj-input-date id="date1" value="{{date}}" name="date1"
                     help.instruction="enter a date in your preferred format and
we will attempt to figure it out">
      </oj-input-date>
    </div>
  </div>
</div>
```

The script to create the view model for this example is shown below.

```
require(['ojs/ojcore', 'knockout', 'jquery', 'ojs/ojknockout', 'ojs/
ojdatetimepicker', 'ojs/ojlabel'],
  function(oj, ko, $)
  {
    function MemberViewModel()
    {
      var self = this;
      self.date = ko.observable();
      self.datetime = ko.observable();
      self.time = ko.observable();
    };

    $(
      function()
      {
        ko.applyBindings(new MemberViewModel(),
          document.getElementById('datetime-converter-example'));
      }
    );
  });
```

When the user runs the page, the `oj-input-date` element displays an input field with the expected date format. In this example, the element also displays a hint when the user hovers over the input field, and displays a calendar when the user clicks in the input field. If the user inputs data that is not in the expected format, the built-in converter displays an error message with the expected format.



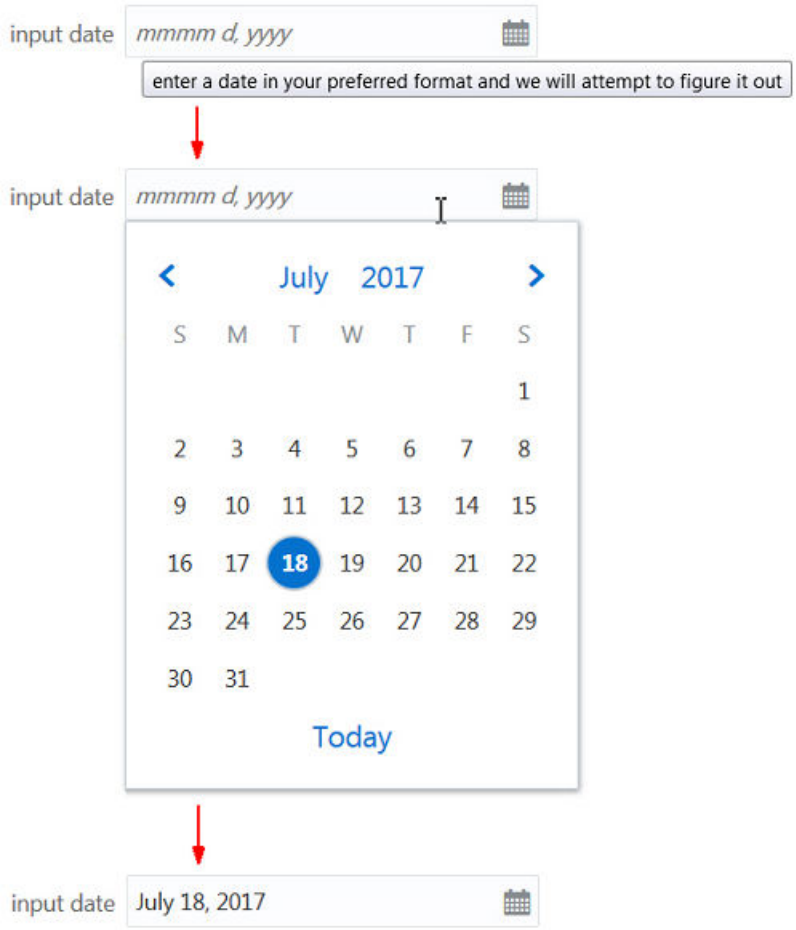
The error that the converter throws when there are errors during parsing or formatting operations is represented by the `oj.ConverterError` object, and the error message is represented by an object that duck-types `oj.Message`. The messages that Oracle JET converters use are resources that are defined in the translation bundle included with Oracle JET. For more information about messaging in Oracle JET, see Working with User Assistance.

You can also specify the converter directly on the element's `converter` attribute, if it exists. The code excerpt below defines another `oj-input-date` element on the sample form and specifies the `oj.IntlDateTimeConverter` converter with options that will convert the user's input to a `numeric` year, `long` month, and `numeric` day according to the conventions of the locale set on the page. The `options` parameter is an object literal that contains the ECMA-402 options as name-value pairs.

```
<div class="oj-flex">
  <div class="oj-flex-item">
    <oj-label for="date2">input date</oj-label>
  </div>
  <div class="oj-flex-item">
    <oj-input-date id="date2" value="{{date}}" name="date2"
                   help.instruction="enter a date in your preferred format and we
will attempt to figure it out"
                   converter= '{
                     "type":"datetime",
                     "options": {"year": "numeric", "month": "long", "day":
```

```
"numeric"}}'>
    </oj-input-date>
  </div>
</div>
```

When the user runs the page in the `en-us` locale, the `oj-input-date` element displays an input field that expects the user's input date to be in the `mmmm d, yyyy` format. The converter will accept alternate input if it makes sense, such as `18/07/17` (`MM/dd/yy`), and perform the conversion, but will throw an error if it cannot parse the input. For details about Oracle JET converters and lenient parsing support, see Understanding Oracle JET Converters Lenient Parsing.



Parsing of narrow era, weekday, or month name is not supported because of ambiguity in choosing the right value. For example, if you initialize the date time converter with options `{weekday: 'narrow', month: 'narrow', day: 'numeric', year: 'numeric'}`, then for the en-US locale, the converter will format the date representing `May 06, 2014` as `T, M 6, 2014`, where `T` represents Tuesday. If the user inputs `T, M 6, 2014`, the converter can't determine whether the user meant **`Thursday`**`, `**`March`** `6, 2014` or **`Tuesday`**`, `**`May`** `6, 2014`. Therefore, Oracle JET expects that user inputs be provided in either their short or long forms, such as `Sat, March 02, 2013`.

For additional details about the `oj.IntlDateTimeConverter` and `oj.IntlNumberConverter` component options, see oj.IntlDateTimeConverter and oj.IntlNumberConverter.

Understanding Oracle JET Converters Lenient Parsing

The Oracle JET converters support lenient number and date parsing when the user input does not exactly match the expected pattern. The parser does the lenient parsing based on the leniency rules for the specific converter.

`oj.IntlDateTimeConverter` provides parser leniency when converting user input to a date and enables the user to:

- Input any character as a separator irrespective of the separator specified in the associated pattern. For example, if the expected date pattern is set to `y-M-d`, the date converter will accept the following values as valid: `2013-11-16`, `2013/11-16`, and `2013aaa11xxx16`. Similarly, if the expected time pattern is set to `mm:ss:SS:`, the converter will accept the following values as valid: `11.24.376`.
- Specify a 4-digit year in any position relative to day and month. For example, both `11-2013-16` and `16-11-2013` are valid input values.
- Swap month and day positions, as long as the date value is greater than 12 when working with the Gregorian calendar. For example, if the user enters `2013-16-11` when `y-M-d` is expected, the converter will autocorrect the date to `2013-11-16`. However, if both date and month are less or equal to 12, no assumptions are made about the day or month, and the converter parses the value against the exact pattern.
- Enter weekday and month names or mix short and long names anywhere in the string. For example, if the expected pattern is `E, MMM, d, y`, the user can enter any of the following dates:

  ```
  Tue, Nov 26 2013
  Nov, Tue 2013 26
  2013 Tue 26 Nov
  ```

- Omit weekdays. For example, if the expected pattern is `E, MMM d, y`, then the user can enter `Nov 26, 2013`, and the converter autocorrects the date to `Tuesday, Nov 26, 2013`. Invalid weekdays are not supported. For instance, the converter will throw an exception if the user enters `Wednesday, Nov 26, 2013`.

`oj.IntlNumberConverter` supports parser leniency as follows:

- If the input does not match the expected pattern, Oracle JET attempts to locate a number pattern within the input string. For instance, if the pattern is `#,##0.0`, then the input string `abc-123.45de` will be parsed as `-123.45`.
- For the currency style, the currency symbol can be omitted. Also, the negative sign can be used instead of a negative prefix and suffix. As an example, if the `pattern` option is specified as `"\u00a4#,##0.00;(\u00a4#,##0.00)"`, then `($123)`, `(123)`, and `-123` will be parsed as `-123`.
- When the style is percent, the percent sign can be omitted. For example, `5%` and `5` will both be parsed as `0.05`.

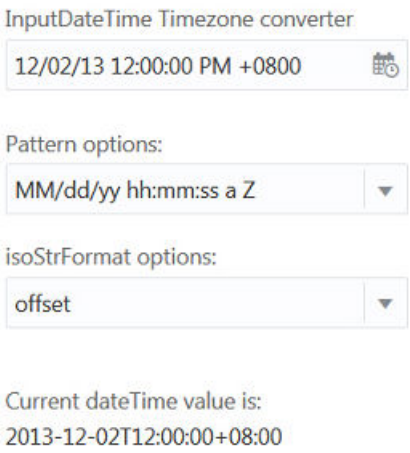
Understanding Time Zone Support in Oracle JET

By default, the `oj-input-date-time` and `oj-input-time` elements and `oj.IntlDateTimeConverter` support only local time zone input. You can add time zone support by including the `ojs/ojtimezonedata` RequireJS module and creating a converter with the desired pattern.

Oracle JET supports time zone conversion and formatting using the following patterns:

Token	Description	Example
z, zz, zzz	Abbreviated time zone name, format support only	PDT, PST
zzzz	Full time zone name, format support only	Pacific Standard Time, Pacific Daylight Time
Z, ZZ, ZZZ	Sign hour minutes	-0800
X	Sign hours	-08
XX	Sign hours minutes	-0800
XXX	Sign hours:minutes	-08:00
VV	Time Zone ID	America/Los Angeles

The image below shows the basic `oj-input-date-time` element configured for time zone support. In this example, the component is converted using the Z pattern.



The `oj-input-date-time` element is initialized with its `converter` attribute, in this case a method named `dateTimeConverter`.

```
<div id="div1">
  <oj-label for="timezone">InputDateTime Timezone converter</oj-label>
  <oj-input-date-time id="timezone" value={{dateTimeValue}}
converter=[[dateTimeConverter]]>
  </oj-input-date-time>
  <br/><br/>
  <p>
  <oj-label for="patternSelector">Pattern options:</oj-label>
  <oj-combobox-one id="patternSelector" value="{{patternValue}}">
    <oj-option value="MM/dd/yy hh:mm:ss a Z">MM/dd/yy hh:mm:ss a Z</oj-option>
    <oj-option value="MM-dd-yy hh:mm:ss a VV">MM-dd-yy hh:mm:ss a VV</oj-option>
    <oj-option value="MM-dd-yy hh:mm X">MM-dd-yy hh:mm X</oj-option>
  </oj-combobox-one>
  </p>
  <p>
  <oj-label for="isoStrFormatSelector">isoStrFormat options:</oj-label>
    <oj-combobox-one id="isoStrFormatSelector" value="{{isoStrFormatValue}}">
      <oj-option value="offset">offset</oj-option>
```

```
      <oj-option value="zulu">zulu</oj-option>
      <oj-option value="local">local</oj-option>
    </oj-combobox-one>
  </p>
  <br/>
  <span class="oj-label">Current dateTime value is: </span>
  <span data-bind="text: dateTimeValue"></span>
  //...contents omitted
</div>
```

The ViewModel contains the `dateTimeConverter()` definition. Note that you must also add the `ojs/timezonedata` to your RequireJS definition to access the time zone data files.

```
require(['ojs/ojcore', 'knockout', 'jquery', 'ojs/ojknockout', 'ojs/
ojdatetimepicker',
             'ojs/ojselectcombobox', 'ojs/ojtimezonedata', 'ojs/ojlabel'],

    function (oj, ko, $)
    {

      function FormatModel()
      {
        var self = this;

        this.dateTimeValue = ko.observable("2013-12-01T20:00:00-08:00");
        this.patternValue = ko.observable("MM/dd/yy hh:mm:ss a Z");
        this.isoStrFormatValue = ko.observable("offset");
        this.dateTimeConverter =
ko.observable(oj.Validation.converterFactory(oj.ConverterFactory.CONVERTER_TYPE_DATET
IME).
        createConverter(
        {
          pattern : self.patternValue(),
          isoStrFormat: self.isoStrFormatValue(),
          timeZone:'Etc/GMT-08:00'
        }));
        //Note that ojCombobox's value is always encapsulated in an array
        this.patternValue.subscribe(function (newValue)
        {

self.dateTimeConverter(oj.Validation.converterFactory(oj.ConverterFactory.CONVERTER_T
YPE_DATETIME).
          createConverter(
          {
            pattern : newValue,
            isoStrFormat: self.isoStrFormatValue(),
            timeZone:'Etc/GMT-08:00'
          }));
        });

        this.isoStrFormatValue.subscribe(function (newValue)
        {

self.dateTimeConverter(oj.Validation.converterFactory(oj.ConverterFactory.CONVERTER_T
YPE_DATETIME).
          createConverter(
          {
            pattern : self.patternValue(),
            isoStrFormat: newValue,
            timeZone:'Etc/GMT-08:00'
```

```
      }));
    });

//...contents omitted
  $(function ()
  {
    ko.applyBindings(new FormatModel(), document.getElementById('div1'));
  });

});
```

For an additional example illustrating how to add time zone support to `oj-input-date-time` and `oj-input-time` elements, see Input Date and Time - Time Zone.

Using Custom Converters in Oracle JET

You can create custom converters in Oracle JET by extending `oj.Converter` or by duck typing it. You can also create a custom converter factory to register the converter with Oracle JET and make it easier to instantiate the converter.

Custom converters can be used with Oracle JET components, provided they don't violate the integrity of the component. As with the built-in Oracle JET converters, you can also use them directly on the page.

The figure below shows an example of a custom converter used to convert the current date to a relative term. The `Schedule For` column uses a `RelativeDateTimeConverter` to convert the date that the page is run in the `en-US` locale to display `Today`, `Tomorrow`, and the date in two days.

Schedule For	Start Time	Class Name		Instructor	Duration
Today	2:30 PM	Early Morning Power Hour		Marcus Levi	1 hour
	4:00 PM	Beginners Yoga		Rachel Donald	1 hour & 15 minutes
	8:00 PM	Spinning	Sign Up	Chris Sharp	45 minutes
	9:00 PM	Pilates Mat		Sue Miller	1 hour
	2:30 AM	Salsa Dance		Paige Davis	1 hour
	3:30 AM	Zumba		Lisa Rogers	1 hour
Tomorrow	2:30 PM	Barre Fit	Sign Up	Paige Davis	1 hour
	4:00 PM	Hatha Yoga		Ramu	1 hour & 15 minutes
	8:00 PM	Body Conditioning		Birgette Peters	1 hour
	9:00 PM	Stretch and Roller		Sue Leon	1 hour
	2:30 AM	Core		Jose Marquis	1 hour
	3:30 AM	Spinning	Sign Up	Giselle Chan	1 hour
In 2 Days	2:30 PM	Body Conditioning		Chris Sharp	1 hour
	4:00 PM	Yoga Flow		Chris Carrington	1 hour & 15 minutes
	8:00 PM	Pilates Mat		Sue Miller	1 hour
	9:00 PM	Basic Step		Jose Marquis	1 hour
	2:30 AM	Zumba		Lisa Rogers	1 hour
	3:30 AM	Hip-Hop	Sign Up	Diane	1 hour

To create and use a custom converter in Oracle JET:

1. Define the custom converter.

   The code sample below shows a portion of the `RelativeDateTimeConverter` definition. The converter wraps the Oracle JET `IntlDateTimeConverter` by providing a specialized `format()` method that turns dates close to the current date into relative terms for display. For example, in the `en-US` locale, the relative terms will display `Today`, `Yesterday`, and `Tomorrow`. If a relative notation for the date value does not exist, then the date is formatted using the regular Oracle JET `format()` method supported by the Oracle JET `IntlDateTimeConverter`.

```
require(['ojs/ojcore', 'knockout', 'jquery', 'ojs/ojknockout',
      'ojs/ojtable', 'ojs/ojbutton', 'ojs/ojvalidation-datetime',
      'ojs/ojarraytabledatasource'],
    function (oj, ko, $)
    {
      ...contents omitted
      function RelativeDateTimeConverter(options) {
        this.Init(options);
      };

      RelativeDateTimeConverter._DEFAULT_RELATIVE_DATETIME_CONVERTER_OPTIONS =
      {
```

```
        'formatUsing' : "displayName",
        'dateField' : "week"
      };

      // Subclass from oj.Object
      oj.Object.createSubclass(RelativeDateTimeConverter, oj.Converter,
"RelativeDateTimeConverter");

      // Initializes converter instance with the set options
      RelativeDateTimeConverter.prototype.Init = function(options)
      {
        var cf;
        // Subclasses of oj.Object must call their superclass' Init
        RelativeDateTimeConverter.superclass.Init.call(this);
        this._options = options;
        //create datetime converter and then wrap it.
        cf =
oj.Validation.converterFactory(oj.ConverterFactory.CONVERTER_TYPE_DATETIME);
        this._wrappedConverter = cf.createConverter(options);
      };

      // Returns the options set on the converter
instanceRelativeDateTimeConverter.prototype.getOptions = function()
      {
      return this._options;
      };

      // Does not support parsing
      RelativeDateTimeConverter.prototype.parse = function(value)
      {
        return null;
      };

      // Formats a value using the relative format options. Returns the formatted
      // value and a title that is the actual date, if formatted value is a
relative date.
      RelativeDateTimeConverter.prototype.format = function(value)
      {
        var base;
        var formattedRelativeDate;

        // We get our wrapped converter and call its formatRelative function and
store the
        // return value ("Today", "Tomorrow" or null) in formatted variable
        // See oj.IntlDateTimeConverter#formatRelative(value, relativeOptions)
        // where relativeOptions has formatUsing and dateField options.
dateField is
        // 'day', 'week', 'month', or 'year'.
        formattedRelativeDate = this._getWrapped().formatRelative(value,
this._getRelativeOptions());

        // We get our wrapped converter and call its format function and store
the returned
        // string in base variable. This will be the actual date, not a relative
date.
        base = this._getWrapped().format(value);

        // Capitalize the first letter of the string
        if (formattedRelativedate &amp;amp;&amp;amp; typeof formattedRelativeDate ===
"string")
              {
```

```
            formattedRelativeDate = formattedRelativeDate.replace(/(\w)(\w*)/g,
            function (match, i, r) {
              return i.toUpperCase() + (r !== null ? r : "");
            });
          }
        return {value: formatted || base, title: formattedRelativeDate ? base :
""}
        };

        // Returns a hint
        RelativeDateTimeConverter.prototype.getHint = function ()
        {
          return "";
        };

        RelativeDateTimeConverter.prototype._getWrapped = function ()
        {
          return this._wrappedConverter;
        };

        RelativeDateTimeConverter.prototype._getRelativeOptions = function ()
        {
          var options = this._options;
          var relativeOptions = {};

          if (options &amp;&amp; options["relativeField"])
          {
            relativeOptions['formatUsing'] = "displayName";
            relativeOptions['dateField'] = options["relativeField"];
          }
          else
          {
            relativeOptions =
RelativeDateTimeConverter._DEFAULT_RELATIVE_DATETIME_CONVERTER_OPTIONS;
          }
          return relativeOptions;
        };
        ... contents omitted
});
```

The custom converter relies on the `IntlDateTimeConverter` converter's `formatRelative()` method. For additional details about the `IntlDateTimeConverter` converter's supported methods, see the oj.IntlDateTimeConverter API documentation.

2. Optionally, create a converter factory for the custom converter that supports a `createConverter()` method, meeting the contract defined by `oj.ConverterFactory`.

   The following code sample shows a simple converter factory for the `RelativeDateTimeConverter`. The code also registers the factory with Oracle JET as a new `relativeDate` type using the `oj.Validation` module.

```
/**
 * A converter factory for "relativeDate" that supports custom
 * formatting of normal and relative date times. A relative date example is
"Today",
 * "Tomorrow", or "This Week", etc.
 */
RelativeDateTimeConverterFactory = (function () {
  /**
  * Private function that takes regular and relative options.
```

```
  */
  function _createRelativeDateTimeConverter(options)
  {
    return new RelativeDateTimeConverter(options);
  }

  return {
    'createConverter': function (options) {
      return _createRelativeDateTimeConverter(options);
    }
  };
}());

/** Register the  factory with JET */
oj.Validation.converterFactory("relativeDate",
RelativeDateTimeConverterFactory);
```

3. Add code to your application that uses the custom converter.

   The code sample below shows how you could add code to your script to use the custom converter.

```
var rdtcf = oj.Validation.converterFactory("relativeDate");
var rdOptions = {relativeField: 'day', year: "numeric", month: "numeric", day:
"numeric"};
var rdConverter = rdtcf.createConverter(rdOptions);
```

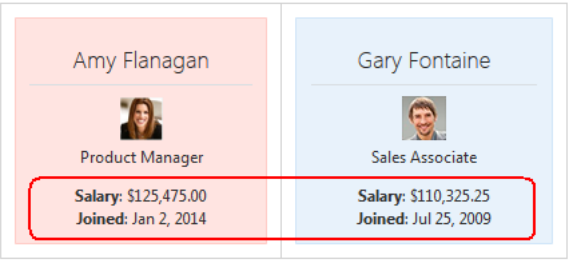
4. Add the Oracle JET element or elements that will use the custom converter to your page.

   For the code sample that creates the `oj-table` element and displays the `Schedule For` column in relative dates, see Converters (Custom).

Using Oracle JET Converters Without Oracle JET Components

If you want to use a converter without binding it to an Oracle JET component, create the converter using `oj.Validation.converterFactory.createConverter()`.

The Oracle JET Cookbook includes the Converters Factory demo that shows how to use the number and date time converters directly in your pages without binding them to an Oracle JET component. In the demo image, the salary is a number formatted as currency, and the start date is an ISO string formatted as a date.



The sample code below shows a portion of the viewModel that defines a `salaryConverter` to format a number as `currency` and a `dateConverter` that formats the start date using the `date` format style and `medium` date format.

```
require(['ojs/ojcore', 'knockout', 'jquery', 'ojs/ojknockout', 'ojs/ojvalidation-
```

```
datetime',
  'ojs/ojvalidation-number'],
    function(oj, ko, $)
    {
      function DemoViewModel()
      {
        var self = this;
        // for salary fields
        var salOptions = {style: 'currency', currency: 'USD'};
        var salaryConverter =
          oj.Validation.converterFactory("number").createConverter(salOptions);
        self.amySalary = ko.observable(salaryConverter.format(125475.00));
        self.garySalary = ko.observable(salaryConverter.format(110325.25));

        // for date fields
        var dateOptions = {formatStyle: 'date', dateFormat: 'medium'};
        var dateConverter =
          oj.Validation.converterFactory("datetime").createConverter(dateOptions);
        self.amyStartDate = ko.observable(dateConverter.format("2014-01-02"));
        self.garyStartDate = ko.observable(dateConverter.format("2009-07-25"));
        ... contents omitted
});
```

The code sample below shows the portion of the markup that sets the display output to the formatted values contained in `amySalary`, `amyStartDate`, `garySalary`, `garyStartDate`.

```
<td>
  <div class="oj-panel oj-panel-alt4 demo-panel-customizations">
    <h3 class="oj-header-border">Amy Flanagan</h3>
    <img src="images/Amy.png" alt="Amy">
    <p>Product Manager</p>
    <span style="white-space:nowrap;"><b>Salary</b>:
      <span data-bind="text: amySalary"></span></span>
    <br/>
    <span style="white-space:nowrap;"><b>Joined</b>:
      <span data-bind="text: amyStartDate"></span></span>
    <br/>
  </div>
</td>
<td>
  <div class="oj-panel oj-panel-alt2 demo-panel-customizations">
    <h3 class="oj-header-border">Gary Fontaine</h3>
    <img src="images/Gary.png" alt="Gary">
    <p>Sales Associate</p>
    <span style="white-space:nowrap;"><b>Salary</b>:
      <span data-bind="text: garySalary"></span></span>
    <br/>
    <span style="white-space:nowrap;"><b>Joined</b>:
        <span data-bind="text: garyStartDate"></span></span>
    <br/>
  </div>
</td>
```

Using Oracle JET Validators

Oracle JET validators provide properties that allow callers to customize the validator instance. The properties are documented as part of the validators' API. Unlike converters where only one instance of a converter can be set on an element, page authors can associate one or more validators with an element.

When a user interacts with the element to change its value, the validators associated with the element are run in order. When the value violates a validation rule, the `value` attribute is not populated, and the validator highlights the element with an error.

You can use the validators with an Oracle JET element or instantiate and use them directly on the page.

Topics:

- Using Oracle JET Validators with Oracle JET components
- Using Custom Validators in Oracle JET

Using Oracle JET Validators with Oracle JET components

Oracle JET editable elements, such as `oj-input-text` and `oj-input-date`, set up validators both implicitly, based on certain attributes they support such as `required`, `min`, `max`, and so on, and explicitly by providing a means to set up one or more validators using the component's `validators` attribute. As with the Oracle JET converters, the `validators` attribute can be specified either using JSON array notation or can be an array of actual validator instances.

For example, the following code sample shows a portion of a form containing an `oj-input-date` element that uses the default validator supplied by the component implicitly. The highlighted code shows the HTML5 attribute set on the `oj-input-date` element. When the `oj-input-date` reads the `min` attribute, it creates the implicit `oj.DateTimeRangeValidator`.

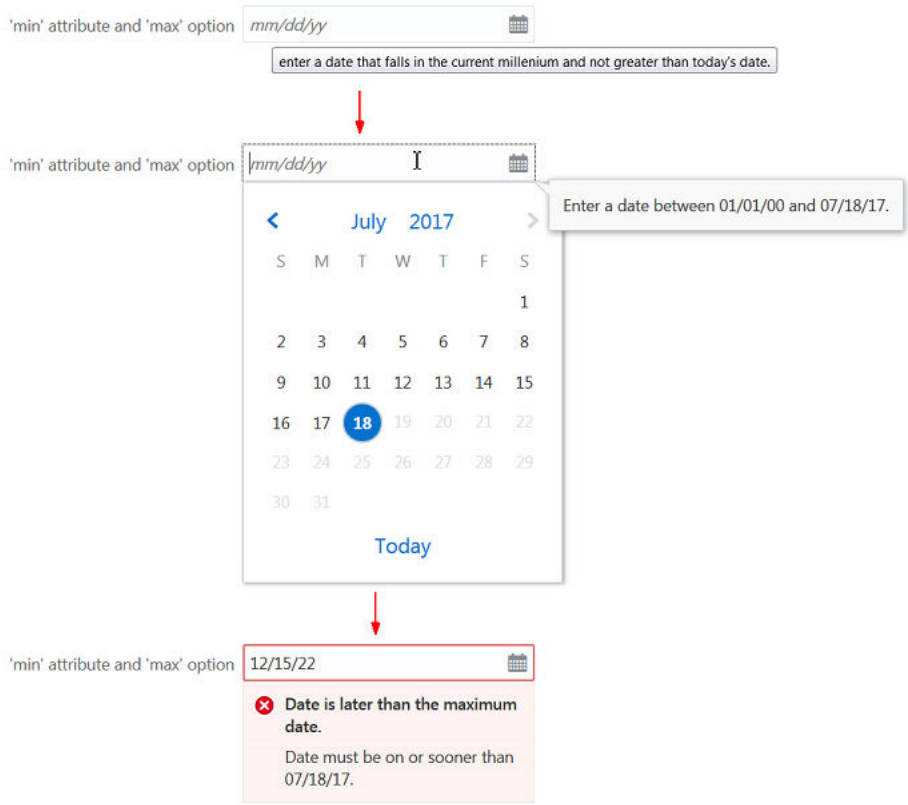
```
<div id="validator-example" class="oj-form oj-md-odd-cols-4 oj-md-labels-inline oj-
sm-odd-cols-12">
  <div class="oj-flex">
    <div class="oj-flex-item">
      <oj-label for="dateTimeRange1">'min' attribute and 'max' option</oj-label>
    </div>
    <div class="oj-flex-item">
      <oj-input-date id="dateTimeRange1" name="dateTimeRange1" value="{{dateValue1}}"
             min="2000-01-01T08:00:00.000"
             help.instruction="enter a date that falls in the current millenium and
not greater than today's date."
             max= '[[todayIsoDate]]'></oj-input-date>
    </div>
  </div>
</div>
```

The script to create the view model for this example is shown below.

```
require(['ojs/ojcore', 'knockout', 'jquery', 'ojs/ojknockout', 'ojs/ojinputnumber',
    'ojs/ojinputtext', 'ojs/ojdatetimepicker', 'ojs/ojvalidation-datetime', 'ojs/
ojlabel'],
    function(oj, ko, $)
    {
      function DemoViewModel()
      {
        var self = this;
        self.dateValue1 = ko.observable();
        self.dateValue2 = ko.observable();
        self.todayIsoDate = ko.observable(oj.IntlConverterUtils.dateToLocalIso(new
Date()));
        self.milleniumStartIsoDate =
ko.observable(oj.IntlConverterUtils.dateToLocalIso(new Date(2000, 00, 01)));
```

```
        self.validators = ko.computed(function()
        {
          return [{
            type: 'dateTimeRange',
            options: {
              max: self.todayIsoDate(),
              min: self.milleniumStartIsoDate(),
              hint: {
                'inRange': 'Enter a date that falls in the current millennium.'}}}];
        });
      };

      $(
        function()
        {
          ko.applyBindings(new DemoViewModel(), document.getElementById('validator-
example'));
        }
      );
    });
```

When the user runs the page, the `oj-input-date` element displays an input field with the expected date format. The `help.instruction` attribute set on the element displays as a tooltip upon hovering. When the user clicks on the field, the validator hint provided by the implicitly created `oj.DateTimeRangeValidator` is shown in a note window, along with a calendar popup. If the user inputs data that is not within the expected range, the built-in validator displays an error message with the expected range.



The error thrown by the Oracle JET validator when validation fails is represented by the `oj.ValidatorError` object, and the error message is represented by an object that duck-types `oj.Message`. The messages and hints that Oracle JET validators use when they throw an error are resources that are defined in the translation bundle included with Oracle JET. For more information about messaging in Oracle JET, see Working with User Assistance.

You can also specify the validator on the element's `validators` attribute, if it exists. The code sample below adds another `oj-input-date` element to the sample form and calls a function which specifies the `oj.DateTimeRangeValidator` validator (`dateTimeRange`) in the `validators` attribute.

```
<div class="oj-flex">
  <div class="oj-flex-item">
    <oj-label for="dateTimeRange2">'dateTimeRange' type in 'validators' option</oj-
label>
  </div>
  <div class="oj-flex-item">
    <oj-input-date id="dateTimeRange2" name="dateTimeRange2" value="{{dateValue2}}"
                   validators="[[validators]]"
                   help.instruction="enter a date that falls in the current millenium
and not greater than today's date.></oj-input-date>
  </div>
</div>
```

The highlighted code below shows the additions to the viewModel, including the defined function, with options that set the valid minimum and maximum dates and a hint that displays when the user sets the focus in the field.

```
require(['ojs/ojcore', 'knockout', 'jquery', 'ojs/ojknockout', 'ojs/ojinputnumber',
    'ojs/ojinputtext', 'ojs/ojdatetimepicker', 'ojs/ojvalidation-datetime', 'ojs/
ojlabel'],
    function(oj, ko, $)
    {
      function DemoViewModel()
      {
        var self = this;
        self.dateValue1 = ko.observable();
        self.dateValue2 = ko.observable();
        self.todayIsoDate = ko.observable(oj.IntlConverterUtils.dateToLocalIso(new
Date()));
        self.milleniumStartIsoDate =
ko.observable(oj.IntlConverterUtils.dateToLocalIso(new Date(2000, 00, 01)));
        self.validators = ko.computed(function()
        {
          return [{
            type: 'dateTimeRange',
            options: {
              max: self.todayIsoDate(),
              min: self.milleniumStartIsoDate(),
              hint: {
                'inRange': 'Enter a date that falls in the current millennium.'}}}];
        });
      };

      $(
        function()
        {
          ko.applyBindings(new DemoViewModel(), document.getElementById('validator-
example'));
        }
```

```
    );
});
```

When the user runs the page for the `en-US` locale, the `oj-input-date` element displays an input field that expects the user's input date to be between `01/01/00` and the current date. When entering a date value into the field, the date converter will accept alternate input as long as it can parse it unambiguously. This offers end users a great deal of leniency when entering date values.

For example, typing `1-2-3` will convert to a `Date` that falls on the 2nd day of January, 2003. If the `Date` value also happens to fall in the expected `Date` range set in the validator, then the value is accepted. If validation fails, the component will display an error.

The options that each validator accepts are specified in JavaScript API Reference for Oracle® JavaScript Extension Toolkit (JET).

The Oracle JET Cookbook contains the complete example used in this section as well as examples that show the built-in validators for date restrictions, length, number range, regular expression, and required fields. For details, see Validators.

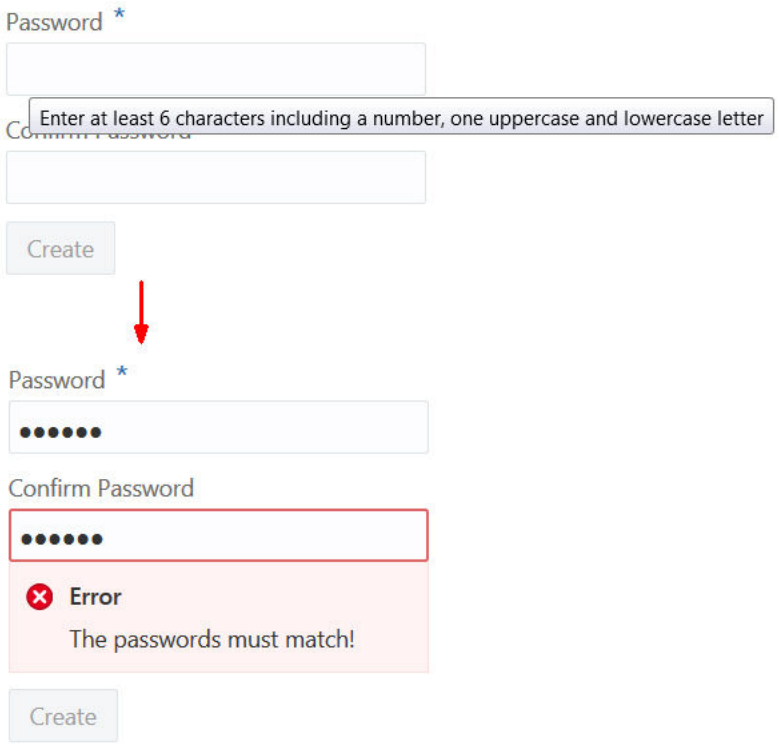
For more information about Oracle JET component validation, see Understanding How Validation and Messaging Works in Oracle JET Editable Components.

Using Custom Validators in Oracle JET

You can create custom validators in Oracle JET by extending `oj.Validator` or by duck typing it.

Custom validators can be used with Oracle JET components, provided they don't violate the integrity of the component. As with the built-in Oracle JET validators, you can also use them directly on the page. For information about messaging in Oracle JET, see Using the messages-custom Attribute.

The figure below shows an example of a custom validator used to validate password entries. If the user's password doesn't match, the validator displays an error message.



To create and use a custom validator in Oracle JET:

1. Define the custom validator.

   The highlighted code in the sample below shows the definition for the `equalToPassword` custom validator used in the figure above. The validator duck types the `oj.Validator` contract. Because the validator provides the methods expected of a validator, Oracle JET accepts it.

```
function DemoViewModel ()
{
  var self = this;
  self.password = ko.observable();
  self.passwordRepeat = ko.observable();

  // When password observable changes, validate the Confirm Password component
  // if it holds a non-empty value.
  self.password.subscribe (function (newValue)
  {
    var $cPassword = $("#cpassword"), cpUIVal = $cPassword.val();
    if (newValue && cpUIVal)
    {
      $cPassword.ojInputPassword("validate");
    }
  });

  // A validator associated to the Confirm Password field, that compares
  // the value in the password observable matches the value entered in the
  // Confirm Password field.

  self.equalToPassword = {
```

```
    validate: function(value)
    {
      var compareTo = self.password.peek();
      if (!value && !compareTo)
        return true;
      else if (value !== compareTo)
      {
        throw new Error(bundle['app']['validator-equalTo']['summary']);
      }
      return true;
    }
  };
}
$(
         function()
    {
        ko.applyBindings(new DemoViewModel(), document.getElementById('custom-
validator-example'));
    }
);
```

2. Add code to your application that uses the custom validator.

   The code sample below shows how you could add code to your page to use the custom validator. In this example, both input fields are defined as `oj-input-password` elements. The first `oj-input-password` element uses `oj.RegExpValidator` to validate that the user has input a password that meets the application's password requirements. The second `oj-input-password` element uses the `equalToPassword` validator to verify that the password in the second field is equal to the password entered in the first field.

```
<div id="custom-validator-example" class="oj-form oj-md-odd-cols-4 oj-md-labels-
inline oj-sm-odd-cols-12">
  <div class="oj-flex">
    <div class="oj-flex-item">
      <oj-label show-required for="password">Password</oj-label>
    </div>
    <div class="oj-flex-item">
            <oj-input-password id="password" name="password" required
value="{{password}}"
                help.instruction="Enter at least 6 characters including a
number, one uppercase and lowercase letter"
                validators= '[{
                    "type": "regExp",
                    "options" : {
                      "pattern": "(?=.*\\d)(?=.*[a-z])(?=.*[A-Z]).{6,}",
                      "label": "Password",
                      "messageSummary" : "{label} too Weak",
                      "messageDetail": "You must enter a password that meets our
minimum security requirements."}}]'>
            </oj-input-password>
        </div>
    </div>
   <div class="oj-flex">
    <div class="oj-flex-item">
      <oj-label for="cpassword">Confirm Password</oj-label>
    </div>
    <div class="oj-flex-item">
      <oj-input-password id="cpassword" name="cpassword"
value="{{passwordRepeat}}"
        validators="[[[equalToPassword]]]"></oj-input-password>
```

```
    </div>
  </div>
</div>
```

For the complete code sample used in this section, see Validators (Custom).

11

Working with User Assistance

The Oracle JET user assistance framework includes support for user assistance on the editable components in the form of help, hints, and messaging that you can customize as needed for your application. Editable components include `oj-checkboxset`, `oj-color*`, `oj-combobox*`, `oj-input*`, `oj-radioset`, `oj-select*`, `oj-slider`, `oj-switch`, and `oj-text-area`.

> **Note:**
>
> The `oj-input*` mentioned above refers to the family of input components such as `oj-input-date-time`, `oj-input-text`, and `oj-input-password`, among others. `oj-color*`, `oj-combobox*`, and `oj-select*` each represent two components.

Topics:

- Typical Workflow for Working with User Assistance
- Understanding Oracle JET's Messaging APIs on Editable Components
- Understanding How Validation and Messaging Works in Oracle JET Editable Components
- Using Oracle JET Messaging
- Configuring an Editable Component's oj-label Help Attribute
- Configuring an Editable Component's help.instruction Attribute
- Controlling the Display of Hints, Help, and Messages

> **Tip:**
>
> To add tooltips to plain text or other non-editable components, use `oj-popup`. See Working with oj-popup.

Typical Workflow for Working with User Assistance

Understand Oracle JET's user assistance capabilities and how to add user assistance to your Oracle JET application.

To add user assistance to your Oracle JET application, refer to the typical workflow described in the following table:

Task	Description	More Information
Identify Oracle JET's user assistance capabilities on editable components	Identify the editable components and their messaging APIs.	Understanding Oracle JET's Messaging APIs on Editable Components

Task	Description	More Information
Understand Oracle JET editable component validation and messaging process	Understand the normal, deferred, and mixed validation processes and the messaging associated with each.	Understanding How Validation and Messaging Works in Oracle JET Editable Components
Use Oracle JET validation and messaging on editable components	Learn how to track the validity of a group of editable components and configure an Oracle JET application to notify editable components of business validation failures and to receive notification of an editable component's events and messages.	Using Oracle JET Messaging
Configure the `help` attribute on the `oj-label` component	Add a help icon to your `oj-label`that includes a URL for additional information or adds explanatory text that appears when the user hovers over the icon.	Configuring an Editable Component's oj-label Help Attribute
Configure the `help.instruction` attribute on editable components	Configure the `help.instruction` attribute to add advisory information to an input field that appears when the field has focus.	Configuring an Editable Component's help.instruction Attribute
Control the display of hints, help, and messages	Use the editable component's `display-options` attribute to control the display type of converter and validator hints, messaging, and help instruction properties.	Controlling the Display of Hints, Help, and Messages

Understanding Oracle JET's Messaging APIs on Editable Components

Oracle JET provides a messaging API to support messaging on Oracle JET editable components.

Editable components include the following:

- `oj-checkboxset`
- `oj-color-palette`
- `oj-color-spectrum`
- `oj-combobox-many`
- `oj-combobox-one`
- `oj-input-date`
- `oj-input-date-time`
- `oj-input-number`
- `oj-input-password`
- `oj-input-text`
- `oj-input-time`
- `oj-radioset`

- `oj-select-many`
- `oj-select-one`
- `oj-slider`
- `oj-switch`
- `oj-text-area`

Topics:

- Oracle JET Editable Component Messaging Attributes
- Oracle JET Component Messaging Methods

The Oracle JET Cookbook also includes descriptions and examples for working with each component at: Form Controls.

Oracle JET Editable Component Messaging Attributes

The following attributes impact messaging on editable elements.

Element Attribute	Description
`converter`	Default `converter` hint displayed as placeholder text when a placeholder attribute is not already set.
`display-options`	JSON object literal that specifies the location where the element should display auxiliary content such as `messages`, `converterHint`, `validatorHint`, and `helpInstruction` in relation to itself. Refer to the element's API documentation for details.
`help`	Help message displayed on an `oj-label` element when the user hovers over the Help icon. No formatted text is available for the message. The `oj-label` has two exclusive attributes, `help.definition` and `help.source`. The `help.definition` attribute's value appears and the attribute's value is read by a screen reader when you hover with a mouse, when you tab into the Help icon, or when you press and hold on a device. The default value is `null`. The `help.source` attribute's value is a link, which is opened when you click with a mouse or tap on a device. The default value is `null`.
`help.instruction`	Displays text in a note window that displays when the user sets focus on the input field. You can format the text string using standard HTML formatting tags.
`messages-custom`	List of messages that the application provides when it encounters business validation errors or messages of other severity type.
`placeholder`	The placeholder text to set on the element.
`translations`	Object containing all translated resources relevant to the component and all its superclasses. Use sub-properties to modify the component's translated resources.
`validators`	List of validators used by element when performing validation. Validator hints are displayed in a note window by default.

See the `Attributes` section of the element's API documentation in JavaScript API Reference for Oracle® JavaScript Extension Toolkit (JET) for additional details about its messaging properties. Select the component you're interested in viewing from the API list.

Oracle JET Component Messaging Methods

Editable value components support the following method for messaging purposes.

Component Event	Description
`showMessages`	Takes all deferred messages and shows them. If there were no deferred messages this method simply returns. When the user sets focus on the component, the deferred messages will display inline.

See the `Methods` section of the component's API documentation in the JavaScript API Reference for Oracle® JavaScript Extension Toolkit (JET) for details on how to call the method, its parameters, and return value. Select the component you're interested in viewing from the list.

Understanding How Validation and Messaging Works in Oracle JET Editable Components

The actions performed on an Oracle JET component, the properties set on it, and the methods called on it, all instruct the component on how it should validate the value and what content it should show as part of its messaging.

Editable components always perform either normal or deferred validation in some situations. In other situations, the editable component decides to perform either normal or deferred validation based on the component's state. Understanding the normal and deferred validation process may be helpful for determining what message properties to set on your components.

- Normal Validation: During normal validation, the component clears all `messages` properties (`messages-custom`), parses the UI value, and performs validation. Validation errors are reported to the user immediately. If there are no validation errors, the `value` attribute is updated, and the value is formatted and pushed to the display.

  The editable component always runs normal validation when:

  – The user interacts with an editable component and changes its value in the UI.

  – The application calls `validate()` on the component.

> **Note:**
>
> When the application changes certain properties, the component might decide to run normal validation depending on its current state. See Mixed Validation below for additional details.

- Deferred Validation: Uses the `required` validator to validate the component's `value`. The `required` validator is the only validator that participates in deferred validation. During deferred validation all `messages` properties are cleared unless specified otherwise. If the `value` fails deferred validation, validation errors are not shown to the user immediately.

  The editable component always runs deferred validation when:

  – A component is created. None of the `messages` properties are cleared.

  – The application calls the `reset()` method on the component.

  - The application changes the `value` property on the component programmatically.

> **Note:**
>
> When the application changes certain properties programmatically, the component might decide to run deferred validation depending on its current state. See Mixed Validation below for additional details.

- Mixed Validation: Runs when the following properties are changed or methods are called by the application. Either deferred or normal validation is run based on the component's current state, and any validation errors are either hidden or shown to the user. Mixed validation runs when:
  - `converter` property changes
  - `disabled` property changes
  - `readOnly` properties change
  - `required` property changes
  - `validators` property changes
  - `refresh()` method is called

Topics:

- Understanding How an Oracle JET Editable Component Performs Normal Validation
- Understanding How an Oracle JET Editable Component Performs Deferred Validation

The Oracle JET Cookbook includes additional examples that show normal and deferred validation at Validators (Component). For additional information about the validators and converters included with Oracle JET, see Validating and Converting Input.

Understanding How an Oracle JET Editable Component Performs Normal Validation

An Oracle JET editable component runs normal validation when the user changes the value in the UI or when the application calls the component's `validate()` method. In both cases, error messages are displayed immediately.

Topics:

- Normal Validation Process When User Changes Value of an Editable Component
- Normal Validation Process When Validate() is Called on Editable Component

Normal Validation Process When User Changes Value of an Editable Component

When a user changes an editable value:

1. All `messages-custom` messages are cleared. An `onMessagesCustomChanged` event is triggered if applicable and if the change in value is obvious.
2. If a converter is set on the component, the UI value is parsed. If there is a parse error, then processing jumps to step **5**.
3. If one or more validators are set on the component:
   a. The parsed (converted) value is validated in sequence using the specified validators, with the implicit required validator being the first to run if present. The value that is passed to the implicit required validator is trimmed of white space.
   b. If the validator throws an error, the error is remembered, and the next validator runs if it exists.

      After all validators complete, if there are one or more errors, processing jumps to step **5**.
4. If all validations pass:
   a. The parsed value is written to the component's `value` attribute, and an `onValueChanged` event is triggered for the `value` attribute.
   b. The new value is formatted for display using the converter again and displayed on the component.

> **Note:**
>
> If the component's value property happens to be bound to a Knockout observable, then the value is written to the observable as well.

5. If one or more errors occurred in an earlier step:
   a. The component's `value` attribute is not changed.
   b. The component's `messages-custom` attribute is updated, and an `onMessagesCustomChange` event is triggered for the `messages-custom` attribute.
   c. Errors are displayed on the component. The user can also view the details of the error by setting focus on the component. By default, this will open a note window.

> **Note:**
>
> If the component's `value` attribute is bound to a Knockout observable, then the value is written to the observable array as well.

6. When the user fixes the error, the validation process begins again.

Normal Validation Process When Validate() is Called on Editable Component

The `validate()` method validates the component's current display value using the converter and all validators registered on the component and updates the `value` attribute if validation passes.

For details about the `validate()` method, see validate().

Understanding How an Oracle JET Editable Component Performs Deferred Validation

An Oracle JET editable component runs deferred validation when the component is created, when its `value` or `required` property is changed programmatically, or when the component's `reset()` method is called. This section provides additional detail about the deferred validation process when an Oracle JET editable component is created and when the `value` property is changed programmatically.

Topics:

- Deferred Validation Process When an Oracle JET Editable Component is Created
- Deferred Validation Process When value Property is Changed Programmatically

You can also find additional detail in the JavaScript API Reference for Oracle® JavaScript Extension Toolkit (JET). Select the component you're interested in from the navigation list.

Deferred Validation Process When an Oracle JET Editable Component is Created

When a editable element is created, as one of the last steps, it runs deferred validation on the component's initialized value.

1. The required validator is run, and a validation error raised if the value is empty or null.
2. If a validation error is raised, the component updates the messaging framework. No messaging themes are applied on the component nor does it show the error message in the note window because the validation error message is deferred.

> **Note:**
>
> Page authors can call `showMessages()` at any time to reveal deferred messages.

Deferred Validation Process When value Property is Changed Programmatically

An Oracle JET editable element's `value` property can change programmatically if:

- The page has code that changes the element's value attribute, or
- The page author refreshes the ViewModel observable with a new server value.

In both cases, the element will update itself to show the new value as follows:

1. All messages properties are cleared on the editable element and `onMessagesCustomChanged` events triggered if applicable.
2. An `onValueChanged` event is triggered for the `value` attribute if applicable.

3. If a converter is set on the element, the `value` attribute is retrieved and formatted before it's displayed. If there is a format error, then processing jumps to step 5. Otherwise the formatted value is displayed on the element.
4. Deferred validators are run on the new value. Any validation errors raised are communicated to the messaging framework, but the errors themselves are not shown to the user.
5. If there was a formatting error, the validation error message is processed and the component's `messages-custom` attribute updated. Formatting errors are shown right away.

> **Note:**
>
> Page authors should ensure that the value you set is the expected type as defined by the component's API and that the value can be formatted without any errors for display.

Using Oracle JET Messaging

Use the Oracle JET messaging framework to notify an Oracle JET application of a component's messages and validity as well as notify an Oracle JET component of a business validation failure.

Notifying an Oracle JET Editable Component of Business Validation Errors

You can notify Oracle JET editable elements of business validation errors using the `messages-custom` attribute and the `showMessages()` method.

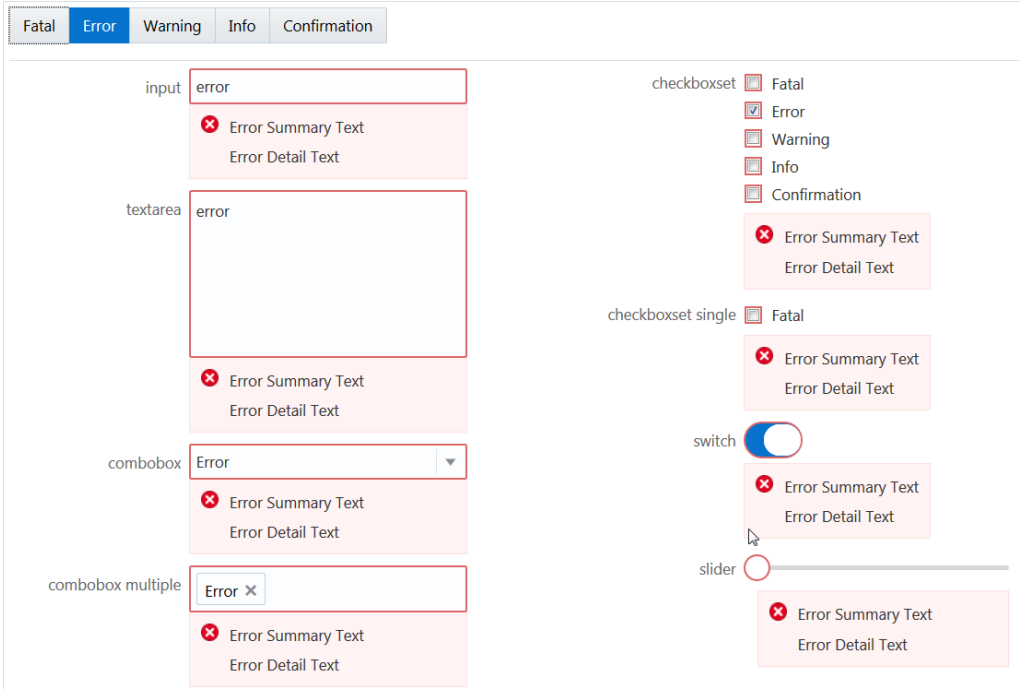
Topics:

- Using the messages-custom Attribute
- Using the showMessages() Method on Editable Components

Using the messages-custom Attribute

Your application can use this attribute to notify Oracle JET components to show new or updated custom messages. These could be a result of business validation that originates in the viewModel layer or on the server. When this property is set, the message shows to the user immediately. The `messages-custom` attribute takes an Object that duck-types `oj.Message` with detail, summary, and severity fields.

In this example, the severity type button is toggled and a message of the selected severity type is pushed onto the `messages-custom` array. The `messages-custom` attribute is set on every form control in this example. When the `messages-custom` attribute is changed, it is shown immediately. In this example, the user selected the `Error` severity type, and the associated messages are shown for the various text input and selection components.



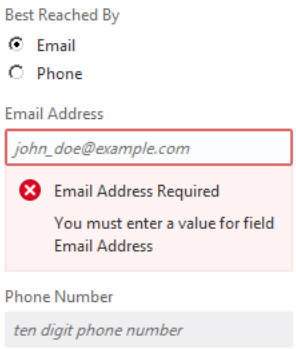
In this example an observable, `appMessages`, is declared and is bound to the `messages-custom` attribute for various elements. The following code describes how you can associate the observable with the `messages-custom` attribute for the `oj-switch` element .

```
<oj-switch id="switch" value="{{switchValue}}"
           messages-custom="{{appMessages}}">
</oj-switch>
```

In the corresponding JavaScript file, set the severity type and pass it to the observable, `appMessages`, to display associated messages.

```
if (summary && detail)
{
 msgs.push({summary: summary, detail: detail, severity: type});
}
self.appMessages(msgs);
```

In the example below, an instance of a cross-field custom validator is associated with the `emailAddress` observable, ensuring that its value is non-empty when the user chooses **Email** as their contact preference.



In the corresponding JavaScript file you must set the `messages-custom` attribute as `emailAddressMessages`.

```
<oj-input-text id="emailId" type="email"  name="emailId"
               placeholder="john_doe@example.com" value="{{emailAddress}}"
               messages-custom="{{emailAddressMessages}}"
               disabled="[[contactPref() !== 'email']]">
</oj-input-text>
```

For the complete example and code used to create the custom validator, see Cross-Field Validation. The demo uses a custom validator to validate an observable value against another. If validation fails the custom validator updates the `messages-custom` attribute.

Using the showMessages() Method on Editable Components

Use this method to instruct the component to show its deferred messages. When this method is called, the Oracle JET editable component automatically takes all deferred messages and shows them. This causes the component to display the deferred messages to the user.

For an example, see App Level Validation.

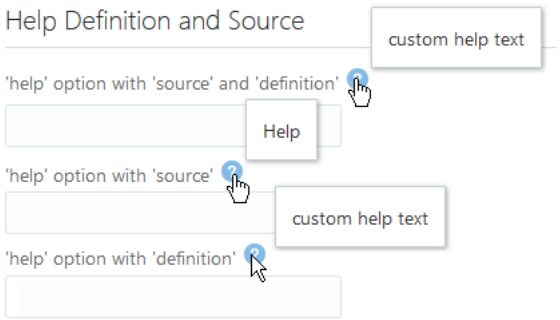
Configuring an Editable Component's oj-label Help Attribute

Configure an `oj-label` element's `help` attribute to add a **Help** icon that displays descriptive text, includes a clickable link to a URL for additional information, or displays both the help text and clickable link when the user hovers over it.

The `help` attribute includes two sub-properties that control the help definition text and help icon:

- `definition`: Contains the help definition text that displays when the user does one of the following:
  - hovers over the help icon
  - tabs into the help icon with the keyboard
  - presses and holds the help icon on a mobile device
- `source`: Contains the URL to be used in the help icon's link

The following image shows three `oj-label` components configured to use the help attribute. The top component is configured with both a `definition` and `source` help sub-property, and the image shows the text and clickable pointer that displays when the user hovers over the help icon. In the middle image, the `oj-label` component includes a help icon that links to a URL when the user clicks it. In the bottom image, the `oj-label` displays custom help text when the user hovers over the label or help icon.



To configure an `oj-label` element's `help` property:

1. Add the `oj-label` to your page.

   If you need help, see Adding an Oracle JET Custom Element to Your Page.

2. In the markup, add the `help` attribute and the `definition` or `source` sub-property.

   The markup for the `ojInputText` components is shown below. Each `ojInputText` component uses the `ojComponent` binding to define the component and set the help sub-properties. In this example, the user will be directed to `http://www.oracle.com` after clicking **Help**.

```
<div id="form-container" class="oj-form">
  <h3 class="oj-header-border">Help Definition and Source</h3>
  <oj-label id="ltext10" for="text10" help.definition="custom help text"
    help.source="http://www.oracle.com">'help' property with 'source' and
'definition'</oj-label>
  <oj-input-text id="text10" name="text10" value="{{text}}"></oj-input-
text>

  <oj-label id="ltext11" for="text11"
    help.source="http://www.oracle.com">'help' property with 'source'</oj-label>
  <oj-input-text id="text11" name="text11" value="{{text}}"></oj-input-text>

  <oj-label id="ltext12" for="text12"
    help.definition="custom help text">'help' property with 'definition'</oj-
label>
  <oj-input-text id="text12" name="text12" value="{{text}}"></oj-input-text>
</div>
```

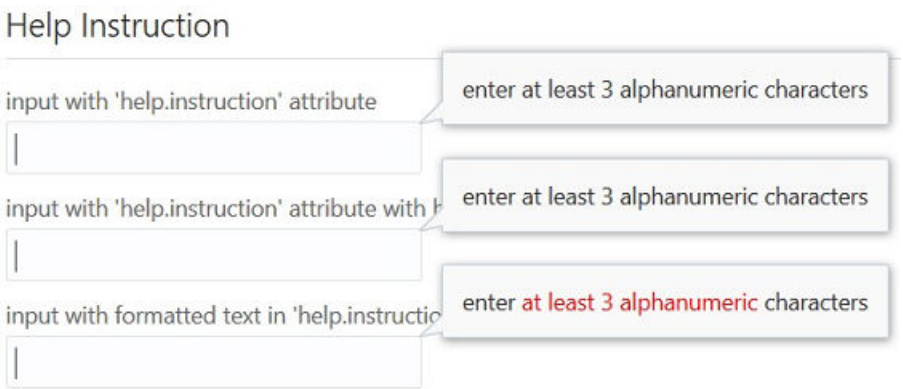
See the Oracle JET Cookbook at Help and Title for the complete example to configure the `help` property on the ojInputText component.

Configuring an Editable Component's help.instruction Attribute

Configure an editable component's help text using the `help.instruction` attribute. This will display a note window with advisory text (often called a tooltip) when the input component has focus.

The following image shows two `oj-input-text` elements configured to display text when the user sets focus in the input field. In the first example, the `help.instruction`

attribute is defined for the `oj-input-text` element without formatting. In the second example, the attribute value has HTML formatting added to the advisory text.



To configure an editable element's `help.instruction` attribute:

1. Add the editable element to your page.

   If you need help, see Adding an Oracle JET Custom Element to Your Page.

2. In the markup, add the `help.instruction` attribute in the element tag.

   The following code sample shows the markup for defining the three `oj-input-text` components.

```
<div id="form-container" class="oj-form">
  <h3 class="oj-header-border">Help Instruction</h3>
    <oj-label for="text20">input with 'help.instruction' attribute</oj-label>
    <oj-input-text id="text20" name="text20"  autocomplete="off"
validators="[[validators]]"
        help.instruction="enter at least 3 alphanumeric characters"
value="{{text}}"></oj-input-text>

    <oj-label for="text21">input with 'help.instruction' attribute with
binding</oj-label>
    <oj-input-text id="text21" name="text21"  autocomplete="off"
validators="[[validators]]"
        help.instruction="{{helpInstruction}}" value="{{text}}"></oj-input-text>

    <oj-label for="text22">input with formatted text in 'help.instruction'
attribute</oj-label>
    <oj-input-text id="text22" name="text22"  autocomplete="off"
validators="[[validators]]"
        help.instruction="<html>enter <span style='color:red'>at least 3
alphanumeric</span> characters</html>"
        value="{{text}}"></oj-input-text>
</div>
```

3. In your application script, bind the component's `value` to a Knockout observable.

   In this example, each `oj-input-text` element's `value` attribute is defined as `text` which is set to a Knockout observable in the following script. The script also defines regular expression validators to ensure that the user enters at least three letters or numbers.

```
  require(['ojs/ojcore', 'knockout', 'jquery', 'ojs/ojknockout', 'ojs/
ojinputtext', 'ojs/ojlabel'],
```

```
  function(oj, ko, $)
  {
    function MemberViewModel()
    {
      var self = this;
      self.text = ko.observable();

      self.validators = ko.computed(function()
        {
          return [{
             type: 'regExp',
             options: {
               pattern: '[a-zA-Z0-9]{3,}',
               messageDetail: 'You must enter at least 3 letters or
numbers'}}];
        });
      self.helpInstruction = "enter at least 3 alphanumeric characters";
    };

    $(
      function()
      {
        ko.applyBindings(new MemberViewModel(), document.getElementById('form-
container'));
      }
    );
  });
```

For the complete example, see Help and Title in the Oracle JET Cookbook. For additional detail about the `oj-input-text` component, see the ojInputText API documentation.

For additional information about the regular expression validator, see About Oracle JET Validators and Converters.

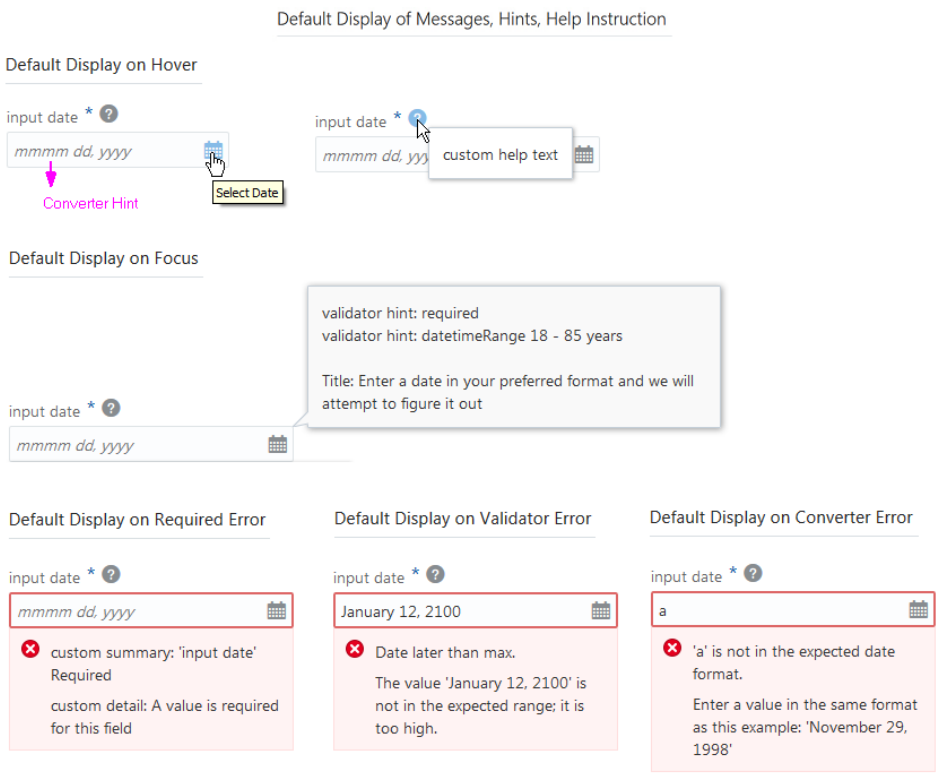
Controlling the Display of Hints, Help, and Messages

Use the `display-options` attribute to control the placement and visibility of converter and validator hints, messages, and help instructions.

The following image shows the default placement and visibility of help, converter and validator hints, and messages. This example uses the `oj-input-date` component, but the behavior is the same on all editable components where it makes sense:

- validator `hint`: Displays in a note window on focus
- converter `hint`: Used as the input field's placeholder, or displays in a note window if the `placeholder` attribute is defined.
- `messages`: Displays inline on error
- `help.instruction`: Displays in a note window on focus

The `oj-label` exclusive attribute `help.definition` displays in a note window on hover.



The code sample below shows the markup for the `oj-input-date` component used in this example. The example includes definitions for `help.instruction`, validator hints, and a data value for custom messages on validation failure. The sample also shows the markup for a `oj-label` element with the `help` attribute.

```
<div id="form-container" class="oj-form">
 <h3 class="oj-header-border">Default Display of Messages, Hints, Help Instruction</
h3>
 <oj-label for="date10" help.definition="custom help text"> Input Date</oj-label>
 <oj-input-date id="date10" size="30" name="date10" required placeholder="month day,
year"
                help.instruction='enter a date in your preferred format and we
will attempt to figure it out'
                converter="[[longDateConverter]]"
                value="{{birthdate}}" validators="[[validators]]"
                translations='{
                  "required": {
                  "hint": "validator hint: required",
                  "messageSummary": "<html>custom summary: {label}
<b>Required</b></html>",
                  "messageDetail": "<html>custom detail: A value is required for
this field</html>"}}'>
 </oj-input-date>
```

The code sample below shows the custom messages on validation failure set in the application's script.

```
function MemberViewModel()
    {
      var self = this;
      self.validators = ko.computed(function()
```

```
        {
          return [{
              type: 'datetimeRange',
              options: {
                min: oj.IntlConverterUtils.dateToLocalIso(new Date(1930, 00, 01)),
                max: oj.IntlConverterUtils.dateToLocalIso(new Date(1995, 11,31)),
                hint: {
                  inRange: 'Validator hint: datetimeRange: January 1, 1930 -
November 30, 1995 years'},
                messageSummary:{
                  rangeOverflow: 'Date later than max.',
                  rangeUnderflow: 'Date earlier than min.'},
                messageDetail: {
                  rangeOverflow: 'The value \'{value}\' is not in the expected
range; it is too high.',
                  rangeUnderflow: 'The value \'{value}\' is not in the expected
range; it is too low.'}
                  }}];
        });
        //...Contents Omitted
  }
$(function(){
        ko.applyBindings(new MemberViewModel(), document.getElementById('form-
container'));
 });
```

Using the `display-options` element attribute in your markup, you can change the default behavior of the hints, help, and messaging properties of a single editable component on your page. To control the behavior of all editable components on the page, you can use the `oj.Component.setDefaultOptions()` method in your application script to set `displayOptions` values.

`display-options` allows you to change the default behavior as follows:

- `helpInstruction`: Set to `none` to turn off the help instruction display.
- `converterHint`: Set to `none` to turn off the display or set to `notewindow` to change the default placement from placeholder text to a note window.
- `validatorHint`: Set to `none` to turn off the display.
- `messages`: Set to `none` to turn off the display or set to `notewindow` to change the default placement from inline to a note window.

To change the default display type (inline or note window) and display options for hints, help, and messages:

1. Add the editable element to your page.

   If you need help, see Adding an Oracle JET Custom Element to Your Page.

2. To change the default display type (inline or note window) for an individual editable component, add the `display-options` attribute to your component definition and set it as needed.

   For example, to turn off the display of hints and `help.instruction` and to display messages in a note window, add the highlighted markup to your component definition:

   ```
   <oj-input-date id="date12" required value="{{birthdate}}"
     converter="[[longDateConverter]]" validators="[[validators]]"
     help.instruction="enter a date in your preferred format and we will attempt to
   figure it out"
   ```

```
  display-options='{"converterHint": "none", "validatorHint": "none",
"helpInstruction": "none", "messages": "notewindow"}'
    ... contents omitted
}"
```

3. To change the default display and location for all editable components in your application, add the `oj.Component.setDefaultOptions()` method to your application's script and specify the desired `displayOptions`.

   For example, to turn off the display of hints and help and to display messages in a note window, add the `ojComponent.setDefaultOptions()` method with the arguments shown below.

```
oj.Components.setDefaultOptions({
  'editableValue':
  {
    'displayOptions':
    {
      'converterHint': ['none'],
      'validatorHint': ['none'],
      'messages': ['notewindow'],
      'helpInstruction': ['none']
    }
  }});
```

The Oracle JET cookbook contains the complete code for this example at User Assistance. You can also find additional examples that illustrate hints, help, and messaging configuration.

12
Developing Accessible Applications

Oracle JET and Oracle JET components have built-in accessibility features for persons with disabilities. Use these features to create accessible Oracle JET web and hybrid mobile application pages.

Topics:

- Typical Workflow for Developing Accessible Oracle JET Applications
- Oracle JET and Accessibility
- Using the Accessibility Features of Oracle JET Components
- Creating Accessible Oracle JET Pages

Typical Workflow for Developing Accessible Oracle JET Applications

Understand accessibility features included in Oracle JET and the components that it provides. You should also understand which tasks are needed for specific components and which tasks are needed at the page level to ensure accessibility.

To develop accessible Oracle JET applications, refer to the typical workflow described in the following table:

Task	Description	More Information
Identify accessibility features included with Oracle JET	Identify the accessibility features included in Oracle JET and Oracle JET components.	Oracle JET and Accessibility
Create accessible Oracle JET components	Identify tasks needed for specific components to ensure accessibility.	Using the Accessibility Features of Oracle JET Components
Create accessible Oracle JET pages	Identify tasks needed at the page level to ensure accessible Oracle JET pages.	Creating Accessible Oracle JET Pages

Oracle JET and Accessibility

Oracle JET components have built-in accessibility support that conforms with the Web Content Accessibility Guidelines version 2.0 at the AA level (WCAG 2.0 AA), developed by the World Wide Web Consortium (W3C).

Accessibility involves making your application usable for persons with disabilities such as low vision or blindness, deafness, or other physical limitations. This means, for example, creating applications that can be:

- Used without a mouse (keyboard only).
- Used with assistive technologies such as screen readers and screen magnifiers.

- Used without reliance on sound, color, animation, or timing.

Oracle JET components provide support for:

- Keyboard and touch navigation

  Oracle JET components follow the Web Accessibility Initiative - Accessible Rich Internet Application (WAI-ARIA) Developing a Keyboard Interface guidelines. The API documentation for each Oracle JET component lists its keyboard and touch end user information when applicable, including a few deviations from the WAI-ARIA guidelines.

- Zoom

  Oracle JET supports browser zooming up to 200%. For example, on the Firefox browser, you can choose **View -> Zoom -> Zoom In**.

- Screen reader

  Oracle JET supports screen readers such as JAWS, Apple VoiceOver, and Google Talkbalk by generating content that complies with WAI-ARIA standards, and no special mode is needed.

- Oracle JET component roles and names

  Each Oracle JET component has an appropriate role, such as `button`, `link`, and so on, and each component supports an associated name (label), if applicable.

- Sufficient color contrast

  Oracle JET provides the Alta theme which is designed to provide a luminosity contrast ratio of at least 4.5:1.

Oracle documents the degree of conformance of each product with the applicable accessibility standards using the Voluntary Product Accessibility Template (VPAT). You should review the appropriate VPAT for the version of Oracle JET that you are using for important information including known exceptions and defects, if any.

While Oracle JET is capable of rendering an application that conforms to WCAG 2.0 AA to the degree indicated by the VPAT, it is the responsibility of the application designer and developer to understand the applicable accessibility standards fully, use JET appropriately, and perform accessibility testing with disabled users and assistive technology.

Using the Accessibility Features of Oracle JET Components

Oracle JET components are designed to generate content that conforms to the WCAG 2.0 AA standards. In most cases, you don't need to do anything to add accessibility to the Oracle JET component. However, there are some components where you may need to supply a label or other property.

For those components, the component's API documentation contains an Accessibility section that provides the information you need to ensure the component's accessibility.

> **Note:**
>
> Some Oracle products have run-time accessibility modes that render content optimized for certain types of users, such as users of screen readers. For the most part, Oracle JET renders all accessibility-related content all of the time. There is only a mode for users that rely on the operating system's high contrast mode, which is described in Creating Accessible Oracle JET Pages.

Oracle JET components that provide keyboard and touch navigation list the keystroke and gesture end user information in their API documentation. Since the navigation is built into the component, you do not need to do anything to configure it.

You can access an individual Oracle JET component's accessibility features and requirements in the JavaScript API Reference for Oracle® JavaScript Extension Toolkit (JET). Select the component you're interested in from the list on the left. You can also find the list of supported keystrokes and gestures for each Oracle JET component that supports keystrokes and gestures in the Oracle® JavaScript Extension Toolkit (JET) Keyboard and Touch Reference.

Creating Accessible Oracle JET Pages

Content generated by Oracle JET is designed to conform to the WCAG 2.0 AA standards. However, many standards are not under the complete control of Oracle JET, such as overall UI consistency, the use of color, the quality of on-screen text and instructions, and so on.

A complete product development plan that addresses accessibility should include proper UI design, coding, and testing with an array of tools, assistive technology, and disabled users.

> **Note:**
>
> In most cases, end-user documentation for your application must describe information about accessibility, such as example keystrokes needed to operate certain components.

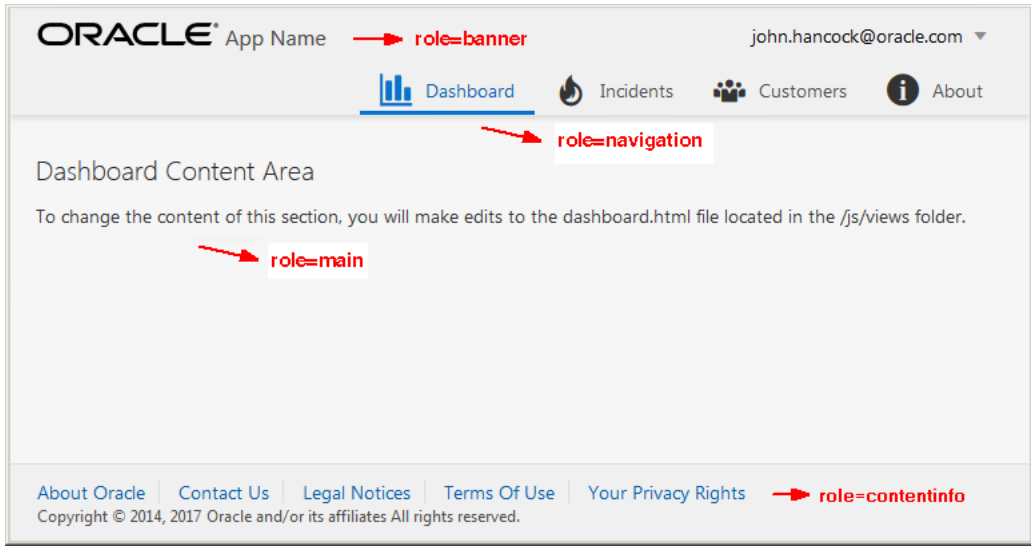
Topics:

- Configuring WAI-ARIA Landmarks
- Configuring High Contrast Mode
- Hiding Screen Reader Content

Configuring WAI-ARIA Landmarks

WAI-ARIA landmarks provide navigational information to assistive technology users. Landmark roles identify the purpose of a page region and allow the user to navigate directly to a desired region. Without landmarks, assistive technology users must use the TAB key to navigate through a page.

The Oracle JET team recommends the use of WAI-ARIA landmarks to ensure page accessibility and provides examples you can use in the Oracle JET Starter Template collection. The following figure shows the run-time view of the Oracle JET Web Nav Drawer Starter Template. In this example, the page is organized into regions compatible with WAI-ARIA landmark regions, including regions for the `banner`, `navigation`, `main`, and `contentinfo` landmarks.



The highlighted code in the following example shows the landmarks for the Web Nav Drawer Starter Template. Each landmark is placed on the HTML element that defines the landmark region: `div` for the `navigation` regions, `header` for the `banner` region, `div` for the `main` region, and `footer` for the `contentinfo` region.

```
<!DOCTYPE html>
<html lang="en-us">
  <head>
    <title>Oracle JET Starter Template - Web Nav Drawer</title>
    ... contents omitted
  </head>
  <body class="oj-web-applayout-body">
    ... contents omitted
    <div id="globalBody" class="oj-offcanvas-outer-wrapper oj-offcanvas-page">
      <div id="navDrawer" role="navigation" class="oj-contrast-marker oj-web-
applayout-offcanvas oj-offcanvas-start">
        <oj-navigation-list data="[[navDataSource]]"
                            edge="start"

item.renderer="[[oj.KnockoutTemplateUtils.getRenderer('navTemplate', true)]]"
                            on-click="[[toggleDrawer]]"
                            selection="{{router.stateId}}">
        </oj-navigation-list>
      </div>
      <div id="pageContent" class="oj-web-applayout-page">
        <header role="banner" class="oj-web-applayout-header">
          <div class="oj-web-applayout-max-width oj-flex-bar oj-sm-align-items-
center">
            ... contents omitted
          </div>
          <div role="navigation" class="oj-web-applayout-max-width oj-web-applayout-
```

```
navbar">
            <oj-navigation-list class="oj-sm-only-hide oj-md-condense oj-md-justify-
content-flex-end"
                                data="[[navDataSource]]"
                                edge="top"

item.renderer="[[oj.KnockoutTemplateUtils.getRenderer('navTemplate', true)]]"
                                selection="{{router.stateId}}">
            </oj-navigation-list>
          </div>
        </header>
        <div role="main" class="oj-web-applayout-max-width oj-web-applayout-content"
data-bind="ojModule: router.moduleConfig">
        </div>
        <footer class="oj-web-applayout-footer" role="contentinfo">
          ... contents omitted
        </footer>
      </div>
    </div>
    <script type="text/javascript" src="js/libs/require/require.js"></script>
    <script type="text/javascript" src="js/main.js"></script>
  </body>
</html>
```

If your application includes a complementary region, add `role="complementary"` to the HTML `div` element:

```
<div role="complementary"></div>
```

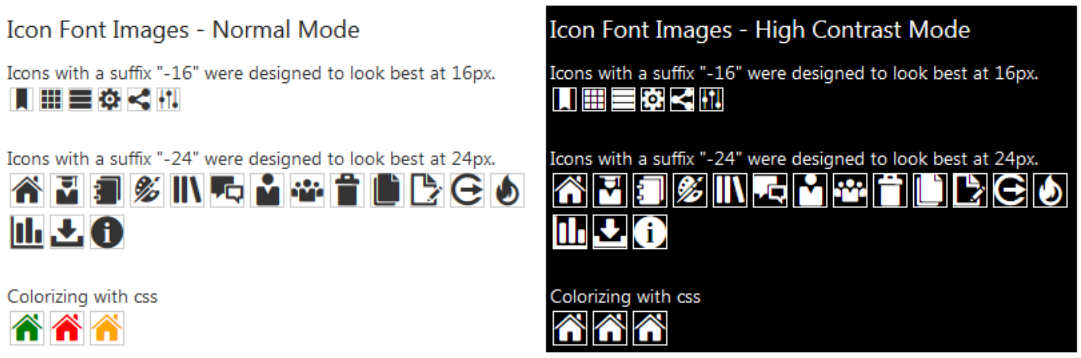
For information about downloading the Oracle JET Starter Templates, see Downloading Oracle JET with a Starter Template. For details about working with the Oracle JET Starter Templates, see Create a Web Application Using the Oracle JET Starter Templates.

For additional information about WAI-ARIA landmark roles, see landmark_roles.

Configuring High Contrast Mode

High contrast mode is for people who require a very high level of contrast in order to distinguish components on the page. Operating systems such as Microsoft Windows and MacOS provide methods for users to run in high contrast mode.

The graphic below shows the effect of changing to high contrast mode on Oracle JET icon font images.



Oracle JET provides the `oj-hicontrast` class that you can use to configure high contrast mode in your application.

Topics:

- Understanding Color and Background Image Limitations in High Contrast Mode
- Adding High Contrast Mode to Your Oracle JET Application
- Adding High Contrast Images or Icon Fonts
- Testing High Contrast Mode

Understanding Color and Background Image Limitations in High Contrast Mode

There are color and background image limitations in high contrast mode that your application may need to work around.

In high contrast mode the colors in the CSS may be ignored or overridden, including background, border, and text colors. Therefore, in high contrast mode you may need to find an alternative way to show state. For example, you might need to add or change the border to show that something is selected.

Also, your application may need to show alternate high contrast images that work on dark or light background color. Some operating systems, like Microsoft Windows 7, offer multiple display profiles for high contrast mode, including a black-on-white and white-on-black mode.

Consider providing an image that includes a background, so either black on a white background or white on a black background. That way it won't matter what the background color is on the page since the contrast is in the image itself.

Finally, on Windows, background images don't appear in high contrast mode. Therefore, you cannot use background images to communicate anything informative. You can use a background image to make something look nice, but don't use it to communicate information like the status of a process or whether something is required.

Adding High Contrast Mode to Your Oracle JET Application

In most cases, you do not need to do anything to enable high contrast mode in your Oracle JET application. If you're using RequireJS to load Oracle JET component modules, Oracle JET will load a script that attempts to detect if a user is running in high contrast mode. If the script succeeds at detection, it will place the `oj-hicontrast` class on the `body` element.

There is a serious limitation to this method, however. There is no standard way to detect high contrast mode, and we can't guarantee that the script works in all cases on all browsers. For example, on Windows, the script does detect high contrast mode on Internet Explorer, Microsoft Edge, and Firefox browsers, but it does not detect high contrast mode on Chrome.

To be guaranteed that the `.oj-hicontrast` styles are applied, add a user preference setting for high contrast to your application and configure the application to add the `oj-hicontrast` class to the `body` element when the preference is set.

When the class is added, the `.oj-hicontrast` CSS styles are applied to the page where defined. The code below shows an excerpt from the Alta CSS which changes the `outline-width` to `3` on the `ojButton` component when the button has focus.

```
.oj-hicontrast .oj-button.oj-focus {
  outline-width: 3px; }
```

> **Note:**
>
> For disabled content, JET supports an accessible luminosity contrast ratio, as specified in WCAG 2.0 - Section 1.4.3 Contrast (Minimum), in the themes that are accessible.
>
> Section 1.4.3 says that text or images of text that are part of an inactive user interface component have no contrast requirement. Because disabled content may not meet the minimum contrast ratio required of enabled content, it cannot be used to convey meaningful information. For example, a checkbox may still appear checked in disabled mode, but since the colors may not pass contrast ratio, you cannot rely on all users being able to see that it's checked. If the information is needed to interact with the page correctly, you must convey it using a different method, for example as plain text.

Adding High Contrast Images or Icon Fonts

To support high contrast image files, Oracle JET provides Sass mixins that you can use to generate the correct CSS in high contrast mode to:

- Use an alternate image.
- Use images without using background images.

The Oracle JET cookbook provides examples that you can use at: CSS Images.

You can also use icon fonts instead of image files to support high contrast mode. The limitation is that icon fonts use a single color. Since these icons are text, they will be guaranteed to contrast against the background color on systems that ignore colors in the CSS. However, if you use color to show state (for example, changing an icon to blue when selected), the colors may be ignored in high contrast modes. You may need to find an alternative like setting a border instead. The Oracle JET cookbook provides icon font demos at: Icon Fonts.

Testing High Contrast Mode

The recommended way to test high contrast mode in Oracle JET applications is to set high contrast mode at the operating system level on a Microsoft Windows platform. The Windows platform is recommended because Windows turns off background colors and images in high contrast mode. Also, the Google Chrome browser does not remove background images in high contrast mode, and unless this is the only browser you plan to support, you should test high contrast with Microsoft Internet Explorer or Firefox.

To turn high contrast mode on and off in Microsoft Windows, use the following key combination: Left Alt+Left Shift+PrtScn. You may need to refresh your browser to see the new mode.

Hiding Screen Reader Content

Sometimes you want to have some text on the page that is read to the screen reader user, but the sighted user doesn't see. Oracle JET provides the `oj-helper-hidden-accessible` class that you can use to hide content.

You can find the `.oj-helper-hidden-accessible` style defaults in the Oracle JET CSS file. For the Alta theme, the CSS file is: `css/libs/oj/v*xxx*/alta/oj-alta.css`. For additional information about theming and Oracle JET, see Theming Applications.

13

Internationalizing and Localizing Applications

Oracle JET supports internationalization and globalization of Oracle JET web and hybrid mobile applications. Configure your Oracle JET application so that the application can be used in a variety of locales and international user environments.

Topics:

- Typical Workflow for Internationalizing and Localizing Oracle JET Applications
- About Internationalizing and Localizing Oracle JET Applications
- Internationalizing and Localizing Oracle JET Applications
- Working with Oracle JET Translation Bundles

Typical Workflow for Internationalizing and Localizing Oracle JET Applications

Understand Oracle JET's internationalization and localization support. Optionally, learn how to merge translations into the Oracle JET translation bundle.

To add internationalization and localization to your Oracle JET application, refer to the typical workflow described in the following table:

Task	Description	More Information
Understand internationalization and localization support in Oracle JET	Understand internationalization and localization support and how it applies to Oracle JET applications.	About Internationalizing and Localizing Oracle JET Applications
Internationalize and localize an Oracle JET application	Use Oracle JET's internationalization features to add internationalization support to an Oracle JET application.	Internationalizing and Localizing Oracle JET Applications
Add translations to an Oracle JET application	Merge your translations into the Oracle JET translation bundle.	Working with Oracle JET Translation Bundles

About Internationalizing and Localizing Oracle JET Applications

Oracle JET includes support for internationalization (I18N) , localization (L10N), and Oracle National Language Support (NLS) translation bundles.

Internationalization (I18N) is the process of designing software so that it can be adapted to various languages and regions easily, cost effectively, and, in particular,

without engineering changes to the software. Localization (L10N) is the use of locale-specific language and constructs at run time. Oracle has adopted the industry standards for I18N and L10N such as World Wide Web Consortium (W3C) recommendations, Unicode technologies, and Internet Engineering Task Force (IETF) specifications to enable support for the various languages, writing systems, and regional conventions of the world.

Languages and locales are identified with a standard language tag and processed as defined in BCP 47.

Oracle JET includes Oracle National Language Support (NLS) translation support for the languages listed in the following table.

Language	Language Tag	Language	Language Tag
Arabic	`ar`	Korean	`ko`
Brazilian Portuguese	`pt`	Norwegian	`no`
Canadian French	`fr-CA`	Polish	`pl`
Czech	`cs`	Portuguese	`pt-PT`
Danish	`da`	Romania	`ro`
Dutch	`nl`	Russian	`ru`
Finnish	`fi`	Simplified Chinese	`zh-Hans` (or `zh-CN`)
French	`fr`	Slovak	`sk`
German	`de`	Spanish	`es`
Greek	`el`	Swedish	`sv`
Hebrew	`he`	Thai	`th`
Hungarian	`hu`	Traditional Chinese	`zh-Hant` (or `zh-TW`)
Italian	`it`	Turkish	`tr`
Japanese	`ja`		

Oracle JET translations are stored in a resource bundle. You can add your own translations to the bundles. For additional information see Adding Translation Bundles to Oracle JET.

Oracle JET also includes formatting support for over 180 locales. Oracle JET locale elements are based upon the Unicode Common Locale Data Repository (CLDR) and are stored in locale bundles. For additional information about Unicode CLDR, see http://cldr.unicode.org. You can find the supported locale bundles in the Oracle JET distribution:

```
js/libs/oj/vxxx/resources/nls
```

It is the application's responsibility to determine the locale used on the page. Typically the application determines the locale by calculating it on the server side from the browser locale setting or by using the user locale preference stored in an identity store and the supported translation languages of the application.

Once the locale is determined, your application must communicate this locale to Oracle JET for its locale sensitive operations such as loading resource bundles and formatting date-time data. Oracle JET determines the locale for locale sensitive operations in the following order:

1. Locale specification in the RequireJS configuration.

2. `lang` attribute of the `html` tag.
3. `navigator.language` browser property or `navigator.userLanguage` Internet Explorer property.

Setting the `lang` attribute on the `html` tag is the recommended practice because, in addition to setting the locale for Oracle JET, it sets the locale for all HTML elements as well. Oracle JET automatically loads the translations bundle for the current language and the locale bundle for the locale that was set. If you don't set a locale, Oracle JET will default to the browser property.

> **Note:**
>
> The locale and resource bundles are loaded automatically only when your application uses the RequireJS ojl10n plugin. For information about using RequireJS in your Oracle JET application, see Using RequireJS for Modular Development.

Finally, Oracle JET includes validators and converters that use the locale bundles. When you change the locale on the page, an Oracle JET component has built in support for displaying content in the new locale. For additional information about Oracle JET's validators and converters, see Validating and Converting Input.

Internationalizing and Localizing Oracle JET Applications

Configure your application to use Oracle JET's built-in support for internationalization and localization.

Topics:

- Using Oracle JET's Internationalization and Localization Support
- Enabling Bidirectional (BiDi) Support in Oracle JET
- Setting the Locale Dynamically
- Working with Currency, Dates, Time, and Numbers

Using Oracle JET's Internationalization and Localization Support

To use Oracle JET's built-in internationalization and localization support, you can simply specify one of the supported languages or locales on the `lang` attribute of the `html` element on your page. For example, the following setting will set the language to the French (France) locale.

```
<html lang="fr-FR">
```

If you want to specify the French (Canada) locale, you would specify the following instead:

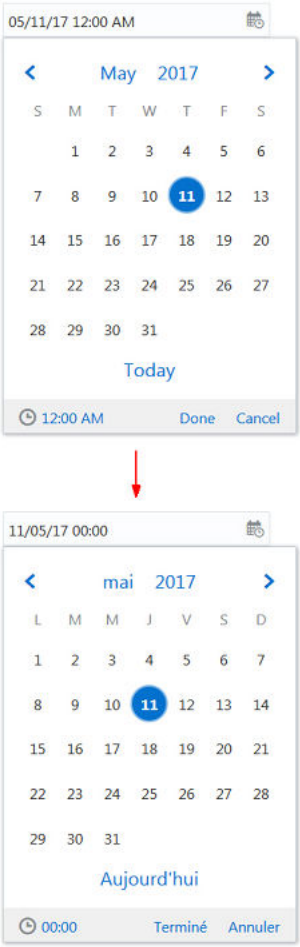
```
<html lang="fr-CA">
```

> **Tip:**
>
> The `lang` specification isn't case sensitive. Oracle JET will accept `FR-FR`, `fr-fr`, etc., and map it to the correct resource bundle directory.

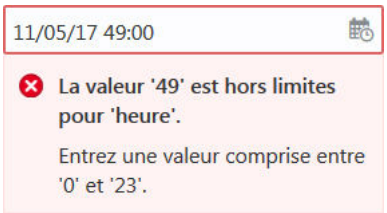
When you specify the locale in this manner, any Oracle JET component on your page will display in the specified language and use locale constructs appropriate for the locale.

If the locale doesn't have an associated resource bundle, Oracle JET will load the lesser significant language bundle. If Oracle JET doesn't have a bundle for the lesser significant language, it will use the default root bundle. For example, if Oracle JET doesn't have a translation bundle for `fr-CA`, it will look for the `fr` resource bundle. If the `fr` bundle doesn't exist, Oracle JET will use the default root bundle and display the strings in English.

In the image below, the page is configured with the `oj-input-date-time` component. The figure shows the effect of changing the `lang` attribute to `fr-FR`.



If you type an incorrect value in the `oj-input-date-time` field, the error text is displayed in the specified language. In this example, the error is displayed in French.



Enabling Bidirectional (BiDi) Support in Oracle JET

If the language you specify uses a right-to-left (RTL) direction instead of the default left-to-right (LTR) direction, such as Arabic and Hebrew, you must specify the `dir` attribute on the `html` tag in addition to the `lang` attribute. The code below shows an example that specifies the Hebrew Israel (`he-IL`) locale with BiDi support enabled.

```
<html lang="he-IL" dir="rtl">
```

The image below shows the same `oj-input-date-time` field that displays if you specify the Hebrew Israel locale and change the `dir` attribute to `rtl`.

Once you have enabled BiDi support in your Oracle JET application, you must still ensure that your application displays properly in the desired layout and renders strings as expected.

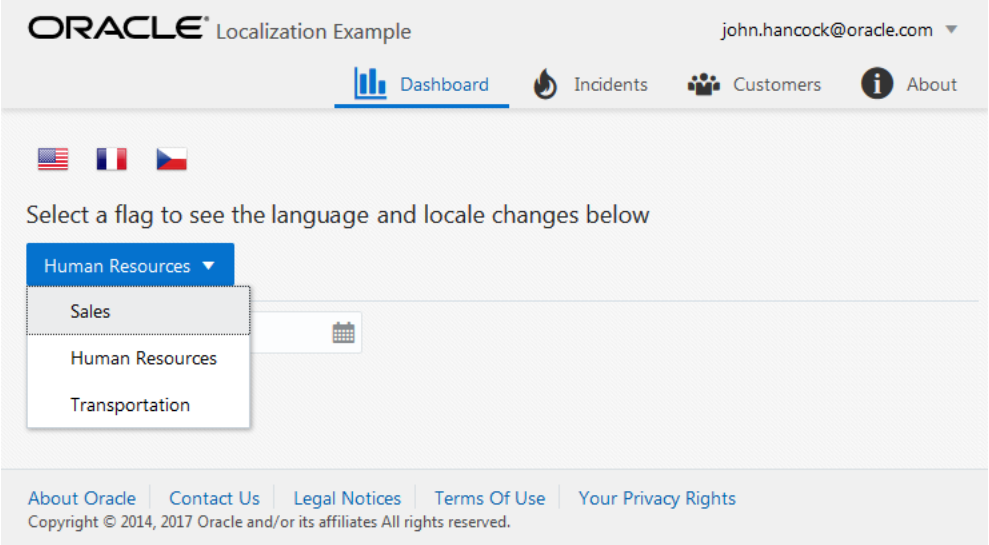
> **Note:**
>
> JET does not support the setting of the `dir` attribute on individual HTML elements which would cause a page to show mixed directions. Also, if you programmatically change the `dir` attribute after the page has been initialized, you must reload the page or refresh each JET component.

Setting the Locale Dynamically

You can configure your application to change the page's locale dynamically by using the `oj.Config.setLocale()` function:

```
setLocale(locale, callback)
```

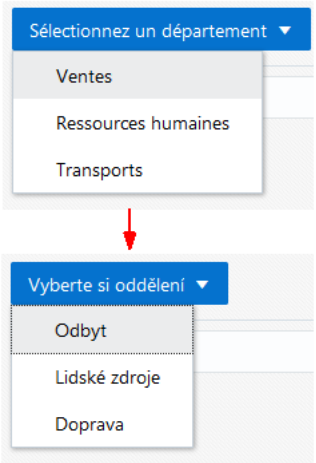
The image below shows the Oracle JET JET-Localization.zip application configured to display a menu that displays a department list when clicked and a date picker. By default, the page is set to the `en-US` locale. Both the menu and date picker are displayed in English.



The application also includes a button set which shows the United States of America, France, and Czech Republic flags. When the user clicks one of the flags, the page locale is set to the locale represented by the flag: `en-US`, `fr-FR`, or `cs-CZ`.

> **Note:**
>
> The flags used in this example are for illustrative use only. Using national flags to select a UI language is strongly discouraged because multiple languages are spoken in one country, and a language may be spoken in multiple countries as well. In a real application, you can use clickable text instead that indicates the preferred language to replace the flag icons.

The figure below shows the updated department list after the user clicks the French and Czech Republic flags.



The code that sets the locale in this example uses the `oj.Config.setLocale()` function call highlighted below. The menu is refreshed in the ViewModel to reload the department list for the chosen locale.

```
// When the country flags are clicked we get a new language to set as the current
locale
self.setLang = function(evt) {
  var newLang = '';
  var lang = evt.currentTarget.id;
  switch (lang){
    case '?eština':
      newLang = 'cs-CZ';
      break;
    case 'français':
      newLang = 'fr-FR';
      break;
    default:
      newLang = 'en-US';
  }
  oj.Config.setLocale(newLang,
        function() {
          $('html').attr('lang', newLang);
          // In this callback function we can update whatever is needed with the
          // new locale. In this example, we reload the menu items.
          loadMenu();
        }
  );
}
```

When the application changes the locale by calling `refresh()` on the `oj-input-date` component, the page will automatically update to use the new locale and display the menu and date in the new locale. However, you must explicitly define the strings that appear in the menu items and then retrieve those strings using the `oj.Translations.getTranslatedString()` method.

```
getTranslatedString(key, var_args)
```

> **Note:**
>
> Do not use this functionality unless the application itself can switch the UI locale dynamically. Dynamically changing the UI locale often ends up with the UI in mixed languages or locales because the application may have cached data that are locale sensitive.

The code that loads the menu is shown below. The menu items and menu button labels are defined with a call to `getTranslatedString()`. The `refresh()` method of both the menu and date component are called after the translations and locale bundles are loaded for the new locale to refresh the display.

```
function DashboardViewModel() {
  var self = this;

  // Setting up knockout observables for the button label and the menu items
  self.localeLabel = ko.observable();
  self.menuNames = ko.observableArray([]);

  self.changeLabel = function (evt) {
    self.localeLabel(oj.Translations.getTranslatedString(evt.target.value));
  }

  self.setLang = function (evt) {
      ...contents omitted
  }


  // This function loads the menu items.
  function loadMenu() {
    // These two lines are pulling the translated values for the menu items from the
    // appropriate resource file in the /resources/nls directory
    self.menuNames(
      [
        {'itemName': oj.Translations.getTranslatedString('menu1')},
        {'itemName': oj.Translations.getTranslatedString('menu2')},
        {'itemName': oj.Translations.getTranslatedString('menu3')}
      ]
    );
    self.localeLabel(oj.Translations.getTranslatedString('label'));

    // Since we've modified the children of a jqueryUI component, we need
    // to refresh it to get all of the build in styling again
    document.getElementById('buttonMenu').refresh();
    document.getElementById('dateInput').refresh();
  }
```

For information about defining your own translation strings and adding them to the Oracle JET resource bundle, see Adding Translation Bundles to Oracle JET.

When you use this approach to internationalize and localize your application, you must consider every component and element on your page and provide translation strings where needed. If your page includes a large number of translation strings, the page can take a performance hit.

Also, if SEO (Search Engine Optimization) is important for your application, be aware that search engines normally do not run JavaScript and access static text only.

> **Tip:**
>
> To work around issues with performance or SEO, you can add pages to your application that are already translated in the desired language. When you use pages that are already translated, the Knockout bindings are executed only for truly dynamic pieces.

Working with Currency, Dates, Time, and Numbers

When you use the converters included with Oracle JET, dates, times, numbers, and currency are automatically converted based on the locale settings. You can also provide custom converters if the Oracle JET converters are not sufficient for your application. For additional information about Oracle JET converters, see Using Oracle JET Converters. For information about adding custom converters to your application, see Using Custom Converters in Oracle JET.

Working with Oracle JET Translation Bundles

Oracle JET includes a translation bundle that translates strings generated by Oracle JET components into all supported languages. Add your own translation bundle by merging it with the Oracle JET bundle.

Topics

- About Oracle JET Translation Bundles
- Adding Translation Bundles to Oracle JET

About Oracle JET Translation Bundles

Oracle JET includes a translation bundle that translates strings generated by Oracle JET components into all supported languages. You can add your own translation bundle following the same format used in Oracle JET.

The Oracle JET translation bundle follows a specified format for the content and directory layout but also allows some leniency regarding case and certain characters.

Translation Bundle Location

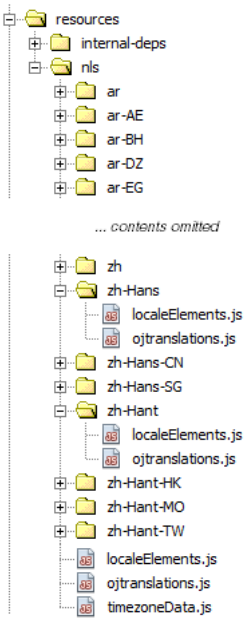
The RequireJS bootstrap file (usually `main.js`) contains the location of the Oracle JET translation bundle, which is named `ojtranslations.js`.

```
libs/oj/vxxx/resources/nls/ojtranslations
```

Each supported language is contained in a directory under the `nls` directory. The directory names use the following convention:

- lowercase for the language sub-tag (`zh`, `sr`, etc.)
- title case for the script sub-tag (`Hant`, `Latn`, etc.)
- uppercase for the region sub-tag (`HK`, `BA`, etc.)

The language, script, and region sub-tags are separated by hyphens (-). The following image shows a portion of the directory structure.



Top Level Module

The `ojtranslations.js` file contains the strings that Oracle JET translates and lists the languages that have translations. This is the top level module or root bundle. In the root bundle, the strings are in English and are the runtime default values when a translation isn't available for the user's preferred language.

Translation Bundle Format

Oracle JET expects the top level root bundle and translations to follow a specified format. The root bundle contains the Oracle JET strings with default translations and the list of locales that have translations. `//` indicates a comment.

```
define({
// root bundle
  root: {
    "oj-message":{
        fatal:"Fatal",
        error:"Error",
        warning:"Warning",
        info:"Info",
        confirmation:"Confirmation",
        "compact-type-summary":"{0}: {1}"
     },
     // ... contents omitted
  },

// supported locales.
  "fr-CA":1,
   ar:1,
   ro:1,
   "zh-Hant":1,
   nl:1,
   it:1,
   fr:1,
   //  ... contents omitted
```

```
   tr:1,fi:1
});
```

The strings are defined in nested JSON objects so that each string is referenced by a name with a prefix: `oj-message.fatal`, `oj-message.error`, etc.

The language translation resource bundles contain the Oracle JET string definitions with the translated strings. For example, the following code sample shows a portion of the French (Canada) translation resource bundle, contained in `nls/fr-CA/ojtranslations.js`.

```
define({
  "oj-message":{
     fatal:"Fatale",
     error:"Erreur",
     warning:"Avertissement",
     info:"Infos",
     confirmation:"Confirmation",
     "compact-type-summary":"{0}: {1}"
     },
    //  ... contents omitted
});
```

When there is no translation available for the user's dialect, the strings in the base language bundle will be displayed. If there are no translations for the user's preferred language, the root language bundle, English, will be displayed.

Named Message Tokens

Some messages may contain values that aren't known until run time. For example, in the message `"User foo was not found in group bar"`, the `foo` user and `bar` group is determined at run time. In this case, you can define `{username}` and `{groupname}` as named message tokens as shown in the following code.

```
"MyUser":"User {username} was not found in group {groupname}."
```

At run time, the actual values are replaced into the message at the position of the tokens by calling `oj.Translations.getTranslatedString()` with the key of the message as the first argument and the parameters to be inserted into the translated pattern as the second argument.

```
var params = {'username': 'foo', 'groupname': 'bar'};
oj.Translations.getTranslatedString("MyKey", params);
```

Numeric Message Tokens

Alternatively, you can define numeric tokens instead of named tokens. For example, in the message `"This item will be available in 5 days"`, the number `5` is determined at run time. In this case, you can define the message with a message token of `{0}` as shown in the following code.

```
"MyKey": "This item will be available in {0} days."
```

A message can have up to 10 numeric tokens. For example, the message `"Sales order {0} has {1} items"` contains two numeric tokens. When translated, the tokens can be reordered so that message token `{1}` appears before message token `{0}` in the translated string, if required by the target language grammar. The Javascript code that calls `getTranslatedString()` remains the same no matter how the tokens are reordered in the translated string.

> **Tip:**
>
> Use named tokens instead of numeric tokens to improve readability and reuse.

Escape Characters in Resource Bundle Strings

The dollar sign, curly braces and square brackets require escaping if you want them to show up in the output. Add a dollar sign ($) before the characters as shown in the following table.

Escaped Form	Output
$$	$
${	{
$}	}
$[	[
$]	]

For example, if you want your output to display `[Date: {01/02/2015}, Time: {01:02 PM}, Cost: $38.99, Book Name: John's Diary]`, enter the following in your resource bundle string:

```
"productDetail": "$[Date: ${01/02/2015$}, Time: ${01:02 PM$}, Cost: $$38.99, Book
Name: John's Diary$]"
```

Adding Translation Bundles to Oracle JET

You can add a translation bundle to Oracle JET by merging your new bundle with the existing Oracle JET translation bundle.

To add translation bundles to Oracle JET:

1. Define the translations.

   For example, the following code defines a translation set for a menu containing a button label and three menu items. In this example, the default language is set to English, and the default label and menu items will be displayed in English. The root object in the file is the default resource bundle. The other properties list the supported locales, `fr` and `cs`.

   ```
   define({
     "root": {
       "label": "Select a department",
       "menu1": "Sales",
       "menu2": "Human Resources",
       "menu3": "Transportation"
   },
     "fr": true,
     "cs": true
   });
   ```

   To add a prefix to the resource names (for example, `department.label`, `department.menu1`, and so on), add it to your bundles as shown below.

   ```
   define({
     "root": {
   ```

```
    "department": {
      "label": "Select a department",
      "menu1": "Sales",
      "menu2": "Human Resources",
      "menu3": "Transportation"
    }
  }
},
  "fr": true,
  "cs": true
});
```

When the locale is set to a French locale, the French translation bundle is loaded. The code below shows the definition for the label and menu items in French.

```
define({
  "label": "Sélectionnez un département",
  "menu1": "Ventes",
  "menu2": "Ressources humaines",
  "menu3": "Transports"
})
```

You can also provide regional dialects for your base language bundle by just defining what you need for that dialect.

```
define({
    "label": "Canadian French message here"
});
```

> **Note:**
>
> When there is no translation available for the user's dialect, the strings in the base language bundle will be displayed. In this example, the menu items will be displayed using the French translations. If there are no translations for the user's preferred language, the root language bundle, whatever language it is, will be displayed.

2. Include each definition in a file located in a directory named `nls`.

   For example, the default translation in the previous step is placed in a file named `menu.js` in the `js/resources/nls` directory. The French translation is located in a file named `menu.js` in the `js/resources/nls/fr` directory.

3. In the application's requireJS bootstrap file (typically `main.js`), add the bundle to the `merge` option in the `ojL10n` definition.

```
// This section configures the i18n plugin.
// It is merging the Oracle JET built-in translation resources
// with a custom translation file for the menu items used in this example.
// Any resource file added, must be placed under a directory named "nls".
// You can use a path mapping or you can define
// a path that is relative to the location of this main.js file.

config: {
  ojL10n: {
    merge: {
      'ojtranslations/nls/ojtranslations': 'resources/nls/menu'
    }
```

```
  }
}
```

> **Note:**
>
> Oracle JET supports only one custom bundle to be merged with the default JET translation bundle. If your application has more than one translation bundle, combine them into one and then add the bundle to the `merge` option.

14
Theming Applications

Oracle JET includes themes that provide styling across a web or hybrid mobile application. You can use these themes as provided, or you can customize them manually and through the tooling.

Topics:

- Typical Workflow for Theming an Oracle JET Application
- CSS Files Included With Oracle JET
- Customizing Themes Using the Tooling
- Sass Files, Variables, and Tools
- Oracle JET Selectors
- Working with Images

Typical Workflow for Theming an Oracle JET Application

Oracle JET comprises CSS files and Sass files, variables, and tools. Understand themes included in Oracle JET and how to customize or specify an alternate theme using the tooling. Optionally, understand Oracle JET's use of selectors and support for images.

To understand and customize the themes included in Oracle JET, refer to the typical workflow described in the following table:

Task	Description	More Information
Identify the CSS files included with Oracle JET	Identify the CSS files included for web, Android, iOS and Windows mobile development.	CSS Files Included With Oracle JET
Customize an Oracle JET application's theme	Use the tooling to customize or specify an alternate theme.	Customizing Themes Using the Tooling
Understand Oracle JET's Sass support	Identify the Sass files, variables and tools included in Oracle JET.	Sass Files, Variables, and Tools
Work with selectors	Understand Oracle JET's use of selectors.	Oracle JET Selectors
Work with images	Understand Oracle JET's support for images.	Working with Images

CSS Files Included With Oracle JET

Oracle JET includes CSS files designed for display on web browsers and hybrid mobile applications. Each theme includes minified and readable versions of the CSS and source maps.

Oracle JET themes are based on Oracle Alta UI, a mobile and browser application design system. There are two types of themes included in Oracle JET, both based on Oracle Alta UI:

- `alta` web theme

  Web themes are designed to be used in a browser on all platforms, and the same theme can be used regardless of whether you are looking at a web page on a desktop Firefox, Android Chrome, or iOS Safari browser.

- `android` , `ios`, and `windows` hybrid mobile themes for Android, iOS, and Windows

  Hybrid themes are designed to be used with Cordova to create a hybrid mobile application. The colors use Oracle's Alta look and feel, but otherwise these themes try to match the look and feel of a native mobile application.

The Alta CSS included with the Oracle JET distribution are generated by the Sass preprocessor and include the following files:

- `oj-alta*.css`: Readable version of the CSS
- `oj-alta*-min.css`: Minified version of the CSS
- `oj-alta*.css.map`: CSS source map

In addition, the Alta web theme includes the following generated CSS:

- `oj-alta-notag.css`: Readable version of the CSS generated without tag selectors

  For additional details about Oracle JET theming and tag selectors, see Using Tag Selectors or Classes.

- `oj-alta-notag-min.css`: Minified version of the CSS generated without tag selectors.

If the CSS files provided by Oracle JET with your application are sufficient and you only want to add a few application-specific styles, you may find that adding the classes to `app.css` in your application's `src/css` folder will meet your needs.

If, however, you want to use a different theme or add more than a few application-specific classes, then you can use the Oracle JET tooling to generate your own CSS. For instructions, see Customizing Themes Using the Tooling.

> **Important:**
>
> Do not override Oracle JET CSS. Oracle JET's CSS is considered private and can change at any time.

DOCTYPE Requirement

In order for Oracle JET's theming to work properly, you must include the following line at the top of all HTML5 pages:

```
<!DOCTYPE html>
```

If you don't include this line, the CSS in your application may not function as expected. For example, you may notice that some elements aren't properly aligned.

> **Tip:**
>
> If you create an Oracle JET application using the tooling or one of the sample applications, this line is already added for you, and you do not need to add it yourself.

ThemeUtils

Oracle JET provides the `oj.ThemeUtils` class that you can use to obtain information about the current theme and use that information to generate and apply a class.

The `oj.ThemeUtils` class provides the following services:

- Return the name of the current theme.
- Return the target platform of the current theme, usually `web`, `ios`, `android`, or `windows`.
- Parse JSON data to use as the font family.

For an example that shows how to vary background color based on the current theme's name and target platform, see Theme Info. For information about the specific services that the `oj.ThemeUtils` class provides, see oj.ThemeUtils.

Setting Text Direction

If the language you specify uses a right-to-left (RTL) direction instead of the default left-to-right (LTR) direction, such as Arabic and Hebrew, you must specify the `dir` attribute on the `html` tag: `<html lang=name dir="rtl">`.

For example, the following code specifies the Hebrew Israel (he-IL) locale with right-to-left direction enabled:

```
<html lang=he-IL dir="rtl">
```

You can find out more about localizing your application and adding bidirectional support in Internationalizing and Localizing Oracle JET Applications.

Customizing Themes Using the Tooling

Use the Oracle JET command-line interface (CLI) to add Sass support and customize your application's theme or specify an alternate theme.

By default, the SCSS files are not included with applications that you scaffold with the Oracle JET CLI. However, you can use tooling commands as described in the procedure below to add node-sass and skeleton theme files to your application. You can then modify the skeleton theme files while the application is running and observe the effect of the change immediately using the live reload feature.

To customize an Oracle JET theme using the tooling:

If needed, scaffold your application as described in Scaffold a Web Application with the Oracle JET CLI or Scaffold a Hybrid Mobile Application with the Oracle JET CLI.

1. In your application's top level directory, enter the following command at a terminal prompt to add `node-sass` to your application:

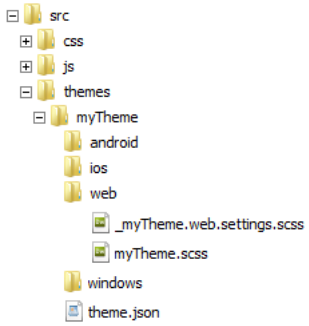
   ```
   ojet add sass
   ```

2. Add the skeleton theme files to your application.

   ```
   ojet create theme themeName
   ```

   For example, enter the following command to create the skeleton theme files for a custom theme named `myTheme`: `ojet create theme myTheme`.

   The command creates a new folder with the custom theme name in the application's `src` directory. The web folder is expanded to show the .scss files that you can modify. The `myTheme.scss` is the aggregating file for the given platform. The `android`, `ios`, and `windows` folders contain similar files with the name of the platform in the title: `_myTheme.`**`android`**`.settings.scss`, `_myTheme.`**`web`**`.settings.scss`, and so on.



   You'll find out more about modifying these files in a later step.

   The `theme.json` file contains the version number of the theme, starting with `0.0.1`.

> **Note:**
>
> The file locations will vary if you modified your directory structure as described in Modify the Web Application's File Structure or Modify the Hybrid Mobile Application's File Structure.

3. Use `ojet build` with the `--theme` option to build a development version of your application with the new theme.

   ```
   ojet build --theme=themeName[:android|ios|web|windows] [--platform=android|ios|web|windows]
   ```

   For example, the following command builds a web application with the `myTheme` custom theme.

   ```
   ojet build --theme=myTheme
   ```

4. To build more than one theme, add the `--themes` option with the names of the desired themes separated by commas:

   ```
   ojet build --theme=myTheme --themes=myTheme,myTheme1,myTheme2
   ```

5. To run your application with live reload enabled, enter `ojet serve` with the `theme` option and optional platform.

   ```
   ojet serve --theme=themeName[:android|ios|web|windows] [--platform=android|ios|web|windows]
   ```

   For example, the following command serves a web application with the `myTheme` custom theme in the default browser.

   ```
   ojet serve --theme=myTheme
   ```

6. Edit the *`themeName`*`.`*`platform`*`.settings.scss` file to set variables as needed for your application.

   For detailed information about Oracle JET and SCSS variables, see SCSS Variables.

   For example, to turn off tag selectors, remove the comment from the `$allowTagSelectors` variable if needed and set it to `false`.

   ```
   $allowTagSelectors: false;
   ```

   For information about Oracle JET's use of tag selectors, see Using Tag Selectors or Classes.

   Other high impact variables that you can modify to see their effect include:

   - Palette variables: `$interaction1Color`, `$background1Color`, and `$border1Color`

     These variables impact color across the application. For example, change the `$interaction1Color` to control the color that indicates a selected state.

   - Text variables: `$fontFamily`, `$rootFontSize`, `$textColor`, and `$linkTextColor`

     Change these variables to affect the font type, size, and color of text in your application.

   When you use the Oracle JET CLI to serve to serve your application, live reload will reload the page, and any changes you make to your theme files will display in the browser.

7. To optimize performance, consider the use of variables to include only the CSS that your application needs. For help, see Using Variables to Control CSS Content.

> **Tip:**
>
> You can observe the effect of any changes immediately. This can be handy if you're not sure whether your application needs a given variable.

8. To exit the application, save your changes, and press Ctrl+C at the terminal prompt.

   You may need to press Ctrl+C a few times for the command to take effect.

> **Tip:**
>
> To make working with themes easier, Oracle JET provides Theme Builder, an example application and tutorial. You can work with it online to see the effect of changes before you apply them to your own application. You can also download the sample and create your own themes offline for sharing.

Sass Files, Variables, and Tools

When you add a theme to your scaffolded application, Oracle JET adds SCSS files that you can modify as needed to customize your application's theme. Typically, you modify one or more SCSS variables and generate custom CSS. Oracle JET also provides Sass source maps for working with partial files.

Topics

- SCSS File Organization and Naming Convention
- SCSS Variables
- SCSS Tools

SCSS File Organization and Naming Convention

Oracle JET follows the Sass underscore naming convention. Files that start with an underscore are called partials and are not compiled independently with Sass. Instead, the partials are imported into an aggregating file. There is one partial file per module, for example, `_oj.table.scss`, `_oj.formcontrol.inputnumber.scss`, and so on.

The following table lists the directory structure for the files contained within the `alta` directory.

File	Description
`widgets/_oj.alta.formcontrol.inputnumber.scss`	Alta widget partial files
`widgets/_oj.alta.table.scss`	
etc.	
`_oj.alta.variables.scss`	Contains the SCSS variables

File	Description
`_oj.alta.settings.scss`	Contains a commented version of the variables file that you can remove the comments from and modify
`_oj.alta.widgets.scss`	Imports widget partial files
`oj-alta.scss`	Generates the `oj-alta.css` and `oj-alta-min.css` files
`oj-alta-notag.scss`	Generates the `oj-alta-notag.css` and `oj-alta-notag-min.css` files

SCSS Variables

The SCSS variables that Oracle JET uses to generate the application use the same naming convention and are grouped together according to function. Oracle JET also uses SCSS variables to default some component options, such as the ojButton's chroming option.

Variable Naming Convention

Oracle JET variables names use a camel case naming structure: <widgetName><Element><Property><State>.

Type	Description	Examples
widgetName	Component or pattern name	`dataGrid`, `button`
Element (Optional)	Component subsection	`Cell`, `Header`
Property	Property affected	`BgColor`, `BorderColor`, `TextColor`
State (Optional)	Widget's state when applied	`Hover`, `Active`, `Focus`, `Selected`, `Disabled`

Using this convention, the following are all valid Oracle JET SCSS variables: `$buttonHeight`, `$sliderThumbWidth`, `$dataGridCellBgColorHover`, `$treeNodeBorderColorSelected`

Variable Documentation

Variable names are typically self-documenting. In cases where additional documentation is needed, the SCSS file will contain comments above the variable definition.

Variable Categories

The following table lists the high level variable categories.

Category	Description
Logistical	Affects logistics, such as the path to an image directory or text direction.
Performance	Set these variables to include only the CSS your application uses.

Category	Description
Palette	Application's color palette. Examples include `$interaction1Color`, `$background1Color`, and `$border1Color`.
Text	Controls text appearance. Widget files use these variables extensively. Examples include `$fontFamily`, `$rootFontSize`, `$textColor`, and `$linkTextColor`.
General	Sets general theme properties such as drag and drop colors, border radius defaults, and animation duration defaults.
Widget	Affects specific widgets or widget groups, such as button or form control variables. Some component default options are included here. For example, you can change the `ojButton` component's chroming option using `$buttonChromingOptionDefault`.

Using Variables to Control CSS Content

You can use the `$includeAllClasses` variable in your SCSS settings file to control content for the entire application. By default, the variable is set to true as shown below.

```
// by default everything is included, but you can also start by setting
// $includeAllClasses to false and then just add in what you want.
$includeAllClasses: true !default;
```

Some partials also allow you to control their content with a variable. For example, the `ojButton` component has the `$includeButtonClasses` variable near the top of the settings file.

```
$includeButtonClasses: $includeAllClasses !default;
```

To exclude the `ojButton` style classes, you can set `$includeButtonClasses` to `false` as shown below.

```
$includeButtonClasses: false;
```

Oracle JET also uses several higher level groupings that let you control the CSS for a logical group of components. The table lists the groups that are available for your use.

Group	SCSS Variable	File Names
Tags (headers, links, and so on)	`$includeTagClasses: $includeAllClasses !default;`	Files with `tag` in name, as in: `_oj.alta.tags.typography`
Data visualizations (charts, gauges, and so on)	`$includeDvtClasses: $includeAllClasses !default;`	Files with `dvt` in name, as in: `_oj.alta.dvt.chart`
Form controls (labels, combo box, and so on)	`$includeFormControlClasses: $includeAllClasses !default;`	Files with `formcontrol` in name, as in: `_oj.alta.formcontrol.label`

You can include or exclude classes and groups as shown in the following examples.

```
// Example: Exclude the dvt classes
$includeDvtClasses: false;

// Example: Include only the dvt classes, exclude everything else
$includeAllClasses: false;
$includeDvtClasses: true;

// Example: Include the chart and sunburst classes, exclude everything else
$includeAllClasses:      false;
$includeChartClasses:    true;
$includeSunburstClasses: true;
```

> **Note:**
>
> Some components depend on others. For example, the `ojInputNumber` uses `ojButton` internally, so if you include the `ojInputNumber` classes, you will also get the `ojButton` classes automatically.

Understanding Right-to-Left Behavior

By default, the same CSS file is used for right-to-left behavior when you set `dir=rtl` on the HTML element as described in Enabling Bidirectional (BiDi) Support in Oracle JET. If you prefer to generate one CSS file for each direction, you can set the `$textDirection` variable to `rtl` or `ltr`.

Oracle JET does not support multiple directions on a page. The reason this is not supported is that the proximity of elements in the document tree has no effect on the CSS specificity, so switching directions multiple times in the page may not work the way you might expect. The code sample below shows an example.

```
<style>
  [dir=ltr] .foo {color: blue}
  [dir=rtl] .foo {color: red}
</style>
<span dir="rtl">
  <span dir="ltr">
    <span class="foo">You might think I will be blue because dir=ltr is on,
                      a closer ancestor than dir=rtl. But css doesn't care
                      about proximity, so instead I am red
                      (because [dir=rtl] .foo {color: red} was defined last).
                      Isn't that surprising?
    </span>
  </span>
</span>
```

For additional information about Oracle JET and bidirectional support, see Enabling Bidirectional (BiDi) Support in Oracle JET. For more information about CSS specificity, see https://developer.mozilla.org/en-US/docs/Web/CSS/Specificity.

Understanding Oracle JET Theming For Compatibility

There may be cases where you do not want to use aspects of the default theming because it causes compatibility issues. For example, you may be embedding JET components or regions in a page controlled by another technology, such as Oracle

ADF. Oracle JET provides several options for controlling use of tag selectors, REM, and normalize.css.

Topics:

- Using Tag Selectors or Classes
- Using REM
- Using Normalize

Using Tag Selectors or Classes

By default, Oracle JET applies styles directly to HTML tag elements such as `a`, `h1`, `h2`, and so on. This feature makes it easier to code a page since you do not have to apply selectors to each occurrence of the element.

The following code shows a portion of a custom SCSS file that defines styles for link and header tags. When the CSS file is generated, the styles will be applied directly to the link and header tags.

```
/* links */
a {
    color: $linkTextColor;
    line-height: inherit;
    text-decoration: none;
}

a:visited {
    color: $linkTextColorVisited;
}

/* headers */
h1, h2, h3, h4 {
    color: $headerTextColor;
    font-family: inherit;
    font-style: normal;
    font-weight: $headerFontWeight;
    margin: 10px 0;
}
```

If you do not want to apply styles directly to the tags, you can specify that Oracle JET use classes instead of tag selectors in your *`custom`*`.scss` file:

```
$allowTagSelectors: false;

// theme import
@import "../oj.alta/oj-alta";
```

The code below shows the classes that Oracle JET will use for the Alta theme's links and headers when you set `$allowTagSelectors` to `false`. To use the class on your page, specify the class in the tag element on your page (`<a class=oj-link>`).

```
/* links */
.oj-link {
    color: $linkTextColor;
    line-height: inherit;
    text-decoration: none;
}

.oj-link:visited {
```

```
    color: $linkTextColorVisited;
}

/* headers */
.oj-header {
    color: $headerTextColor;
    font-family: inherit;
    font-style: normal;
    font-weight: $headerFontWeight;
    margin: 10px 0;
}
```

The following table lists the HTML tags with default Oracle JET tag styles and the corresponding Oracle JET class.

HTML Tag	Oracle JET Style Class
`html`	`oj-html`
`body`	`oj-body`
`a`	`oj-link`
`h1`, `h2`, `h3`, `h4`	`oj-header`
`hr`	`oj-hr`
`p`	`oj-p`
`ul`, `ol`	`oj-ul`, `oj-ol`

If you also do not want to use the Oracle JET tag classes, you can set `$includeTagClasses` to `false` in your *`custom`*`.scss` file as shown below.

```
$includeTagClasses:false;
```

Using REM

By default, Oracle JET uses REM (root em) for font sizing. This means that a specific font size is set on the root (`html`) element, and other font sizes are defined relative to that root size. The following example shows the default font sizing for the Alta theme.

```
// The browser usually uses 16px and the Alta default is 14px.
// Therefore set $rootFontValue to .875em to get 14px;
$rootFontSize:     .875em !default;   // 14px

$fontSize:         1rem !default;     // 14px
$smallestFontSize: .7857rem !default; // 11px when root 14px
$smallFontSize:    .857rem !default;  // 12px when root 14px
$mediumFontSize:   1.143rem !default; // 16px when root 14px
$largeFontSize:    1.286rem !default; // 18px when root 14px
$largestFontSize:  1.429rem !default; // 20px when root 14px
```

To use REM, the font size must be set on the `html` element. If you want to use REM but do not want to set a style directly on the html tag, you can reference the `oj-html` class as described in Using Tag Selectors or Classes.

If you do not want to use REM, you can specify an alternate font size by modifying a *`custom`*`.scss` file to use the units your application requires. The code sample below shows how you could use pixels instead of rems to set font sizes.

```
// use px instead of rem for fonts
$rootFontSize:     12px;
```

```
$fontSize:         $rootFontSize;
$smallestFontSize: $rootFontSize - 2px;
$smallFontSize:    $rootFontSize - 1px;
$mediumFontSize:   $rootFontSize + 2px;
$largeFontSize:    $rootFontSize + 4px;
$largestFontSize:  $rootFontSize + 6px;
```

For additional information about REM, see http://snook.ca/archives/html_and_css/font-size-with-rem.

Using Normalize

By default, Oracle JET uses normalize.css to promote consistency when rendering across browsers. If your application also uses normalize.css, add the import in your *custom*.scss:

```
@import "../3rdparty/normalize/normalize";
```

If you do not want to use normalize, you can set the following variable to false in your *custom*.scss:

```
$includeNormalize: false;
```

Alternatively, you can set `$allowTagSelectors` to `false` as described in Using Tag Selectors or Classes.

For additional information about normalize.css, see http://necolas.github.io/normalize.css.

SCSS Tools

Oracle JET provides its own Sass mixins and doesn't use Sass tools such as Autoprefixer or Compass. This reduces the number of third-party tools required to work with Oracle JET's SCSS.

However, Oracle JET does support Sass source maps which can make it easier to work with a large number of partial files. The toolkit includes source maps for the default and no tag selector versions of the generated CSS for the Alta theme: `oj-alta.css.map`, and `oj-alta-notag.css.map`.

For additional information about source maps, see https://developers.google.com/chrome-developer-tools/docs/css-preprocessors.

Oracle JET Selectors

Oracle JET provides component and pattern selectors that you can use to apply styling to an Oracle JET component or pattern. The toolkit also provides selectors to use instead of CSS3 and CSS4 pseudo classes that apply the same style on any DOM element.

Topics:

- Component and Pattern Selectors
- Marker Classes

Component and Pattern Selectors

Component and pattern selector names use lower case and follow the naming convention: `.oj-[widgetname]-[element-name]`.

The following list describes the naming convention in greater detail:

- `oj`: Oracle JET classes use the reserved `oj` namespace prefix.
- *`widgetname`*: Component or pattern names are one word and lower case, such as `datagrid`.
- *`element-name`*: A widget's element is hyphenated: `header-cell-text`.

For example, the following class applies to data grid header cell text: `.oj-datagrid-header-cell-text`. To reference this selector, add it to the HTML and omit the leading period.

```
<!-- ko if: $key=='FIRST_NAME' -->
        <span class="oj-datagrid-header-cell-text"
            data-bind="text: 'First Name'"></span>
<!-- /ko -->
```

Marker Classes

Oracle JET provides selectors that you can use instead of CSS3 and CSS4 pseudo classes. The advantage of using an Oracle JET selector is that you can specify the same style on any DOM element.

For example, you can use the `oj-hover` instead of the `:hover` pseudo class. The following table lists the Oracle JET marker classes and their CSS3 and jQuery UI equivalent.

Marker Type	Oracle JET Marker Class	CSS3 Inspiration	jQuery UI Equivalent	Description
Accessibility	`oj-hicontrast`			This marker is placed on the body tag by either a script or the application. For details, see Configuring High Contrast Mode.

Alignment	`oj-right`
	`oj-center`
	`oj-left`
	`oj-start`
	`oj-end`
				Indicates the horizontal position, such as the tail of a popup.

Alignment	`oj-top`
	`oj-middle`
	`oj-bottom`
				Indicates the vertical position, such as the tail of a popup.

Collapsible	`oj-expanded`			Used when something expandable is expanded.
Collapsible	`oj-collapsed`			Used when something expandable is collapsed, meaning that it's not expanded.
Component	`oj-component`		`ui-widget`	Marks the root DOM element of a widget.

Marker Type	Oracle JET Marker Class	CSS3 Inspiration	jQuery UI Equivalent	Description
Component	`oj-component-initmode`			Mark the `init` DOM element of a widget. This can be different than the root DOM element if the `init` node was wrapped by other DOM. For example, in an `ojInputNumber` component, the `init` node is an input element, but this is wrapped by other DOM when the component is created.
Input	`oj-disabled`	:disabled		
Input	`oj-enabled`	:enabled	`ui-state-default`	
Input	`oj-read-only`			
Input	`oj-invalid`			
Input	`oj-warning`			
Input	`oj-required`			
Input	`oj-indeterminate`	:indeterminate		
Link	`oj-visited`	:visited		Applied when user visits the link or item.
User Action	`oj-active`	:active	`ui-state-active`	jQuery UI uses `ui-state-active` for both active and selected states. Oracle JET provides separate styling markers for active and selected/checked/toggled states so that you can style these states separately.
User Action	`oj-selected`		`ui-state-active`	Used when something is selected, checked, or toggled on. See previous explanation.
User Action	`oj-hover`	:hover	`ui-state-hover`	`:hover` works on any DOM element but only on the closest DOM element currently hovered over. `oj-hover` can be placed on an element which is an ancestor of the element being hovered over. For example if you're hovered over an icon in a button, `oj-hover` may be placed on the button tag, which is the ancestor of the tag with the icon.
User Action	`oj-focus`	:focus	`ui-state-focus`	
User Action	`oj-focus-only`			Used when something is focused, but you only want to set the style when `oj-selected`, `oj-hover`, `oj-active` are not set.

Marker Type	Oracle JET Marker Class	CSS3 Inspiration	jQuery UI Equivalent	Description
Visibility	`oj-complete`			Added to a DOM element when a composite component or page is ready for styling. To prevent a flash of unstyled content before the component is ready, add the following to the component or page CSS: `component:not(.oj-complete) { visibility: hidden }`

Oracle JET also provides the following drag and drop marker classes that can be used instead of CSS4 pseudo classes.

Oracle JET Marker Class	CSS4 Inspiration	jQuery UI Equivalent	Description
`oj-draggable`			Used on draggable elements.
`oj-drag`			Used on element when dragged.
`oj-drop`	:drop		Applies to all elements that are drop targets during a drag.
`oj-active-drop`	:drop (active)		Used on element that will receive the item currently being dragged.
`oj-valid-drop`	:drop (valid)		Used on elements that can receive the item currently being dragged.
`oj-invalid-drop`	:drop (invalid)		Used on elements that can't receive the item currently being dragged.

Internally, Oracle JET follows these rules for clickable elements:

Oracle JET Marker Class	Used if Widget is Disabled or Read Only?
`oj-enabled` or `oj-disabled` or `oj-readonly`	`oj-enabled`: no
`oj-selected`	Yes
`oj-active`	No
`oj-hover`	No
`oj-focus`	No
`oj-focus-only`	No
`oj-default`	No

For additional information about the CSS3 selectors, see http://www.w3.org/TR/css3-selectors/. For more information about the CSS4 selectors, see http://dev.w3.org/csswg/selectors4/.

Working with Images

Oracle JET uses icon fonts whenever possible to render images provided by the Alta theme. When icon fonts are not possible, Oracle JET uses image files. In either case, framework images support high contrast mode. Oracle JET also provides CSS classes for most framework images.

For examples of framework images, see Framework Images.

You may also find the following topics helpful when working with images.

Topics:

- Image Considerations
- Icon Fonts
- Image Files

Image Considerations

There are a variety of ways to load icons, such as sprites, data URIs, icon fonts, and so on. Factors to consider when choosing an image strategy include:

- Themable: Can you use CSS to change the image? Can you replace a single image easily?
- High contrast mode: Does the image render properly in high contrast mode for accessibility?
- High resolution support: Does the image look acceptable on high resolution (retina) displays?
- Browser support: Do you require support for all browsers? Some browsers may not support certain image formats, such as SVG.
- Image limitations: Are there limitations that impact your use case? For example, icon fonts are a single color, and small SVG images often do not render well.
- Performance: Is image size a factor? Do you need alternate versions of an image for different resolutions or states such as disabled, enabled, hover, and active?

Icon Fonts

Oracle JET uses icon fonts whenever possible because icon fonts have certain advantages over other formats.

- Themable: You can use style classes to change their color instead of having to replace the image, making them very easy to theme.
- High contrast mode: Icon fonts are optimal for high contrast mode as they are considered text. However, keep in mind that you can't rely on color in high contrast mode, and you may need to indicate state (active, hover, and so on) using another visual indicator. For example, you can add a border or change the icon font's size. For additional information about Oracle JET and high contrast mode, see Configuring High Contrast Mode.
- High resolution: Icon fonts look good on a high resolution (retina) display without providing alternate icons.

- Performance: You can change icon font colors using CSS so alternate icons are not required to indicate state changes. Alternate images are also not required for high resolution displays.

Icon fonts also have disadvantages. It can be difficult to replace a single image, and they only show one color. You can use text shadows to provide some depth to the icon font.

Oracle JET supports two generic classes for setting the icon font colors in the `$iconColorDefault`, `$iconColorHover`, `$iconColorActive`, `$iconColorSelected`, and `$iconColorDisabled` variables.

- `oj-clickable-icon`
- `oj-clickable-icon-nocontext`

These classes, when used in conjunction with an anchor tag and/or marker classes like `oj-default`, `oj-hover`, `oj-focus`, `oj-active`, `oj-selected`, and `oj-disabled`, will use the `$iconColor*` variables.

The `oj-clickable-icon` class is optionally contextual, meaning the anchor or marker style can be on an ancestor as shown in the example below. The example assumes that a JavaScript method is replacing `oj-default` as needed with `oj-hover`, `oj-focus`, `oj-active`, `oj-selected`, and `oj-disabled`.

```
<div class="oj-default">
  <span class="oj-clickable-icon demo-icon-font demo-icon-gear"></span>
</div>
<a href="http://www.oracle.com">
  <span class="oj-clickable-icon demo-icon-font demo-icon-gear"></span>
</a>
```

The `oj-clickable-icon-nocontext` class is not contextual and must be placed on the same tag as shown in the example below. `oj-clickable-icon` would also work in this example.

```
<span class="oj-default oj-clickable-icon-nocontext demo-icon-font demo-icon-gear">
</span>
<a href="http://www.oracle.com"
   class="oj-clickable-icon-nocontext demo-icon-font demo-icon-gear">
</a>
```

For an example that illustrates icon font classes on a link, see Icon Fonts.

Image Files

Oracle JET uses images when icon fonts can't be used. Informational images must appear in high contrast mode for accessibility, as described in Configuring High Contrast Mode.

To improve performance, the SVG images used by JET components are sprited. JET does not support creating custom image sprites, and you must use a third-party tool to create them. However, if you use Oracle JET tooling, you can overwrite some of the default SVG images and use the tooling to combine them into a single `sprite.svg` file.

You can replace any of the SVG images in the following application directories. Note that your new image must use the same name as the original image.

```
themes/alta/android/images
themes/alta/common/images
```

```
themes/alta/ios/images
themes/alta/web/images
themes/alta/windows/images
```

When you build your application using `ojet build` or `ojet serve` commands, JET will automatically combine the SVG images into the `sprite.svg` file in the appropriate `themes/alta/*/images/sprites/` directory.

You can also use custom images that are not sprited using CSS, but the improved performance gained by using sprited images will be lost. To do so, you must override the image style classes with `{background-position: 0% 0%;}`.

Oracle JET provides Sass mixins that you can use to create CSS for your own icons. For examples, see CSS Images.

15
Securing Applications

Oracle JET follows security best practices for Oracle JET components and provides the `oj.OAuth` class to help you manage access to users' private data.

Topics:

- Typical Workflow for Securing Oracle JET Applications
- About Securing Oracle JET Applications
- Using oj.OAuth in Your Oracle JET Application
- About Securing Hybrid Mobile Applications
- Dealing With Cross-Origin Resource Sharing (CORS)

Typical Workflow for Securing Oracle JET Applications

Understand how to secure your Oracle JET application. Optionally, understand how to use the oj.OAuth plugin to manage access to client (end user) private data.

To develop secure Oracle JET applications, refer to the typical workflow described in the following table:

Task	Description	More Information
Understand Oracle JET application security	Identify Oracle JET security features and which tasks you should take to secure your Oracle JET application.	About Securing Oracle JET Applications
Use `oj.OAuth`	Use Oracle JET oj.OAuth plugin.	Using oj.OAuth in Your Oracle JET Application

About Securing Oracle JET Applications

Oracle JET applications are client-side HTML applications written in JavaScript, and you should follow best practices for securing your Oracle JET applications.

There are a number of Internet resources available that can assist you, including the Open Web Application Security Project (OWASP), Web Application Security Project (WASP), Web Application Security Working Group (WASWG), and various commercial sites.

Topics:

- Oracle JET Components and Security
- Oracle JET Security and Developer Responsibilities
- Oracle JET Security Features

Oracle JET includes components that follow best practices for security and provides the `oj.OAuth` plugin for providing secure access to a user's private data. However, the application developer is expected to perform tasks that are not included in Oracle JET.

Oracle JET Components and Security

Oracle JET components follow best practices for security. In particular:

- All JavaScript code is executed in `strict` mode using the `use strict` directive.

  Strict mode changes warnings about poor syntax, such as using undeclared variables, into actual errors that you must correct. For more information, see http://www.w3schools.com/js/js_strict.asp.

- Oracle JET code does not use inline `script` elements.

  Because browsers can't tell where the inline script originated, the World Wide Web Consortium (W3C) Content Security Policy prohibits the use of inline scripts. For additional information, see https://w3c.github.io/webappsec/specs/content-security-policy.

- Oracle JET code does not generate random numbers.
- Any HTML generated by an Oracle JET component is ether escaped or sanitized.

  For information about why this is needed, see https://www.owasp.org/index.php/DOM_based_XSS_Prevention_Cheat_Sheet#Guidelines_for_Developing_Secure_Applications_Utilizing_JavaScript.

Oracle JET Security and Developer Responsibilities

Oracle JET components follow established security guidelines and ensure that strings provided as options and user input will never be executed as JavaScript to prevent XSS attacks. However, Oracle JET does not include a mechanism for sanitizing strings, and you should consult established guidelines for dealing with XSS attacks in your own code and content.

You can find more information about securing JavaScript applications at https://www.owasp.org/index.php/DOM_based_XSS_Prevention_Cheat_Sheet#Guidelines_for_Developing_Secure_Applications_Utilizing_JavaScript.

Oracle JET Security Features

The Oracle JET API provides the `oj.OAuth` authorization plugin which supports the OAuth 2.0 open protocol. OAuth standardizes the way desktop and web applications access a user's private data. It provides a mechanism for users to grant access to private data without sharing their private username and password credentials.

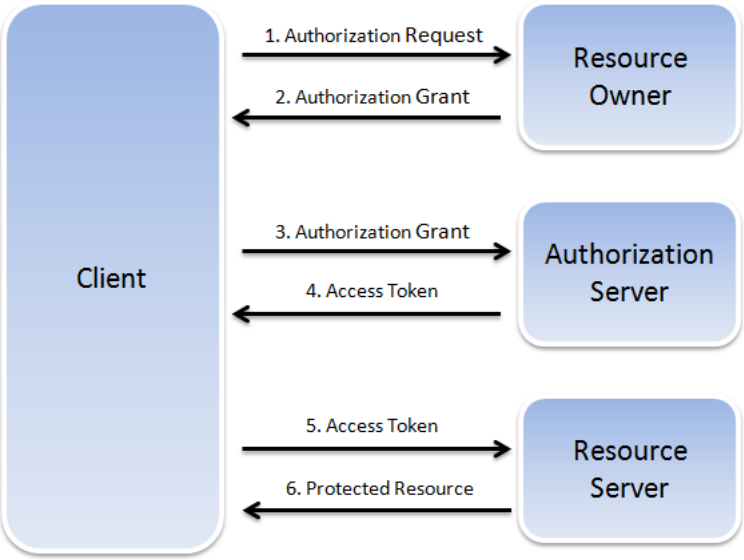
OAuth 2.0 defines the following roles:

- Resource owner: An entity that can grant access to a protected resource, such as the end user.
- Client: Application making protected and authorized resource requests on behalf of the resource owner.
- Resource server: Server hosting the protected resources that can accept and respond to protected resource requests using access tokens.

- Authorization server: Server that issues access tokens to the client after it successfully authenticates the resource owner and obtains authorization.

> **Note:**
>
> The authorization server can be the same server as the resource server. In addition, an authorization server can issue access tokens accepted by multiple resource servers.

OAuth 2.0 Request for Comments (RFC) 6749 describes the interaction between the four roles as an abstract flow.



1. The client requests authorization from the resource owner, either directly or through the authorization server.

> **Note:**
>
> The RFC specifies that the authorization server is preferred.

2. The client receives an authorization grant, which is defined as the credential representing the resource owner's authorization.
3. The client requests an access token from the authorization server by authenticating with the server and presenting the authorization grant.
4. The authorization server issues the access token after authenticating the client and validating the authorization grant.
5. The client presents the access token to the resource server and requests the protected resource.

6. The resource server validates the access token and serves the request if validated.

The access token is a unique identifier issued by the server and used by the client to associate authenticated requests with the resource owner whose authorization is requested or has been obtained by the client.

The Oracle JET `oj.OAuth` plugin provides functions for the following tasks:

- Getting access token credentials if initialized by client credentials.
- Caching access token credentials.
- Creating the header array with bearer token.

For details about using the `oj.OAuth` plugin, see Using oj.OAuth in Your Oracle JET Application. For additional information about OAuth 2.0, see http://tools.ietf.org/html/rfc6749.

Using oj.OAuth in Your Oracle JET Application

You can use the `oj.OAuth` plugin to manage access to client (end user) private data. The Oracle JET API includes the `OAuth` class which provides the methods you can use to initialize the `oj.OAuth` object, verify initialization, and calculate the authorization header based on client credentials or access token.

Topics:

- Initializing oj.OAuth
- Verifying oj.OAuth Initialization
- Obtaining the OAuth Header
- Using oj.OAuth with Oracle JET Common Model
- Integrating oj.OAuth with Oracle Identity Management (iDM) Server

Initializing oj.OAuth

You can create an instance of a specific `oj.OAuth` object using the `oj.OAuth` constructor:

```
new OAuth(header, attributes)
```

The `attributes` and `header` parameters are optional.

Parameter	Type	Description
`header`	String	MIME Header name. Defaults to `Authorization`

Parameter	Type	Description
`attributes`	Object	Contains client credentials or access/bearer token. Client credentials contain: • `client_id` (required): public client Credentials • `client_secret` (required): secret client credentials • `bearer_url` (required): URL for token bearer and refresh credentials • Additional attributes as needed (optional) Access/bearer tokens contain: • `access_token` (required): Bearer token • Additional attributes as needed (optional)

The code sample below shows three examples for initializing `oj.OAuth`.

```
// Initialize oj.OAuth with client credentials
var myOAuth = new oj.OAuth('X-Header', {...Client credentials...});

// Initialize oj.OAuth with token credentials
var myOAuth = new oj.OAuth('X-Header', {...Access/Bearer token...});

// Initialize oj.OAuth manually
var myOAuth = new oj.OAuth();
```

If you choose to initialize oj.OAuth manually, you can add the client credentials or access/bearer token using methods shown in the following code sample.

```
// Initializing client credentials manually
myOAuth.setAccessTokenRequest({...Client Credentials ...});
myOAuth.clientCredentialGrant();

// Initializing access bearer token manually
myOAuth.setAccessTokenResponse({...Access Token...});
```

The OAuth API also includes methods for getting and cleaning the client credentials or access tokens. For additional information, see the oj.OAuth API documentation.

Verifying oj.OAuth Initialization

Use the `isInitialized()` method to verify that the initialization succeeded.

```
var initFlag = myOAuth.isInitialized();
```

Obtaining the OAuth Header

Use the `getHeader()` method to get the OAuth header. The method calculates the authorization header based on the client credentials or access token.

```
// Client credentials
var myOAuth = new oj.OAuth('New-Header', {...Client credentials...});
var myHeaders =  myOAuth.getHeader();

// Access token
var myOAuth = new oj.OAuth('New-Header', {...Access/Bearer token...});
var myHeaders =  myOAuth.getHeader();

// Manual initialization, client credentials
var myOAuth = new oj.OAuth();
```

```
myOAuth.setAccessTokenRequest({...Client credentials...});
var myHeaders =  myOAuth.getHeader();

// Manual initialization, access token
var myOAuth = new oj.OAuth('New-Header', {...Access/Bearer token...});
var myHeaders =  myOAuth.getHeader();
```

Using oj.OAuth with Oracle JET Common Model

You can add the `oj.OAuth` object to your ViewModel, either embedded or as an external plugin.

Topics:

- Embedding oj.OAuth in Your Application's ViewModel
- Adding oj.OAuth as a Plugin in Your ViewModel

For information about Oracle JET's Common Model, see Using the Common Model and Collection API.

Embedding oj.OAuth in Your Application's ViewModel

The code sample below shows how you could embed the `oj.OAuth` object in your ViewModel. This example initializes `oj.OAuth` with client credentials.

```
function viewModel() {
  var self = this;
  ...
  self.myOAuth = new oj.OAuth('X-Authorization', {...Client credentials...});

  var tweetModel = oj.Model.extend({
     ...
  });
  var myTweet = new tweetModel();
  ...
  var tweetCollection = oj.Collection.extend({
     model: myTweet,
     oauth: self.myOAuth, // using embedded feature
     ...
  });
  self.myTweetCol = new tweetCollection();
  ...
  self.myTweetCol.fetch({
     success: function(collection, response, options) {
        ...
     },
     error: function(jqXHR, textStatus, errorThrown) {
        ... // process errors
     }
  });
}
```

To embed the `oj.OAuth` object in your ViewModel and initialize it with a bearer/access token:

```
function viewModel() {
  var self = this;
  ...
  self.myOAuth = new oj.OAuth('X-Authorization', {...Access/Bearer token...});
```

```
  var tweetModel = oj.Model.extend({
     ...
  });
  var myTweet = new tweetModel();
  ...
  var tweetCollection = oj.Collection.extend({
     model: myTweet,
     oauth: self.myOAuth, // using embedded feature
     ...
  });
  self.myTweetCol = new tweetCollection();
  ...
  self.myTweetCol.fetch({
     success: function(collection, response, options) {
        ...
     },
     error: function(jqXHR, textStatus, errorThrown) {
        ... // process errors or insert new access_token and re-fetch
     }
  });
}
```

Adding oj.OAuth as a Plugin in Your ViewModel

The code sample below shows how you could add the `oj.OAuth` object as a plugin in your ViewModel. This example initializes `oj.OAuth` with client credentials.

```
var viewModel() {
  var self = this;
  ...
  self.myOAuth = new oj.OAuth('X-Authorization', {...Client credentials...});

  var tweetModel = oj.Model.extend({
     ...
  });
  var myTweet = new tweetModel();
  ...
  var tweetCollection = oj.Collection.extend({
     model: myTweet,
     ...
  });
  self.myTweetCol = new tweetCollection();
  ...
  self.preFetch = function() {
     var header = self.myOAuth.getHeader();
     $.ajaxSetup({
        beforeSend: function (xhr){
           for(var hdr in header ) {
              if(header.hasOwnProperty(hdr))
                 xhr.setRequestHeader(hdr, header[hdr]);
              }
           }
        }
     });
     self.myTweetCol.fetch({
        success: function(collection, response, options) {
           ...
        },
        error: function(jqXHR, textStatus, errorThrown) {
           ... // process errors
```

```
        }
      });
    }
  }
```

Integrating oj.OAuth with Oracle Identity Management (iDM) Server

Oracle iDM servers use a two-legged authorization (Resource Owner Password Credentials Grant). In addition, the iDM servers require that you do the following:

- Keep client credentials on your own proxy server. If you don't have one, you must create one.
- iDM servers use a non standard authorization header and require that `Authorization:access_token` be used instead of `Authorization: Bearer access_token`. To supply the custom header, you must rewrite the OAuth header for specific Authorization using the `getHeader()` method.

  The code except below shows an example that adds the `oj.OAuth` object with a modified header to the application's viewModel.

```
function viewModel() {
  var self = this;
  self.bearer = {
    access_token: ...,
    token_type: "Bearer",
    expires_in: ...,
    ...
  }
  ...
  self.myOAuth = new oj.OAuth();
  // Rewrite oAuth header for specific Authorization
  self.myOAuth.getHeader = function() {
    var headers = {};
    headers['X-Authorization']=self.bearer.access_token;
    return headers;
  }
  var idmModel = oj.Model.extend({...  });
  var myIDM = new idmModel();
  ...
  var idmCollection = oj.Collection.extend ({
    model: myIDM, oauth: self.myOAuth,
    // using embedded feature
    ...
  });
  self.myIDMCol = new idmCollection();
  ...
  self.myIDMCol.fetch({
    success: function(collection, response, options) {
      ...
    },
    error: function(jqXHR, textStatus, errorThrown) {
      ...
      // process errors or insert new access_token and re-fetch
    }
  });
}
```

About Securing Hybrid Mobile Applications

Since hybrid mobile applications are JavaScript HTML5 applications, many of the same security practices that apply to web applications apply to hybrid mobile applications. However, there are additional considerations when you're deploying to mobile devices.

The Cordova documentation includes a Security Guide that provides some security best practices for Cordova applications. You can use this guide as a starting point to secure your hybrid mobile application. However, as Cordova points out, security is a complicated topic, and its guide is not exhaustive.

JET adopts the AppConfig approach to manage mobile applications in the enterprise. AppConfig is an industry-sponsored community that provides tools and best practices to configure and secure mobile applications. Oracle JET provides the `cordova-plugin-emm-app-config plugin`, described in Managing App Configuration for JET Hybrid Mobile Apps, to facilitate this process.

Managing App Configuration for JET Hybrid Mobile Apps

Many enterprises make use of Enterprise Mobility Management (EMM) software to manage mobile devices, applications, wireless networks, and other mobile computing services.

EMM software allows administrators to configure mobile apps via a web console and apply the configuration to the app on every managed device as part of the app installation process. This configuration typically consists of generic name-value pairs defined by the app developer, such as URI, port numbers, tenant IDs, or CSS skin configurations.

The Android and iOS platforms support this functionality natively, but retrieving app configuration data and being alerted when it has changed is just as important for Cordova-based hybrid mobile apps. To make this possible, Oracle JET provides the `cordova-plugin-emm-app-config` Cordova plugin that can be used to implement app configuration in JET mobile apps on the Android and iOS platforms. Using this plugin, you can retrieve the app configuration data at any point in the app lifecycle, including when the EMM server changes the app configuration.

You can add the `cordova-plugin-emm-app-config` to your hybrid mobile app using the Oracle JET CLI as follows:

```
ojet add plugin cordova-plugin-emm-app-config
```

For more information about this plugin, see https://github.com/oracle/cordova-plugin-emm-app-config. This plugin can be used in any Cordova-based hybrid mobile app, not just JET hybrid apps.

Vendors and other stakeholders in the application configuration software sector formed the AppConfig community to provide tools and best practices. For more information, see http://appconfig.org/.

Dealing With Cross-Origin Resource Sharing (CORS)

CORS is a mechanism that allows restricted resources on a web page to be requested from another domain outside the domain from which the first resource was served.

The same-origin security policy of JavaScript forbids certain cross-domain requests, notably Ajax requests, by default.

Rejected resource requests due to CORS can affect web or hybrid mobile applications. Applications that encounter a rejection receive messages such as the following example in response to resource requests:

```
No 'Access-Control-Allow-Origin' header is present on the requested resource.
```

Server-side administrators can specify the origins allowed to access their resources by modifying the policy used by their remote server to allow cross-site requests from trusted clients. For example, to access a remote service managed by Oracle's Mobile Cloud Service (MCS), an MCS administrator configures MCS's `Security_AllowOrigin` environment policy with a comma-separated list of URL patterns that identify the remote services that serve resources from different domains.

If you serve your web or hybrid mobile application to the local browser for testing, you may encounter CORS rejections. Some browsers provide options to disable CORS, such as Chrome's `--disable-web-security` and Firefox's `security.fileuri.strict_origin_policy` and some browsers support plugins that work around CORS.

Only use these options when testing your application and ensure that you complete further testing in a production-like environment without these options to be sure that your application will not encounter CORS issues in production.

The default web views used by hybrid mobile applications do not implement CORS and therefore hybrid mobile applications will not encounter CORS issues when run on a device in the default web view. However, if you use an alternative web view, such as WKWebView on iOS, you may encounter CORS issues. To work around this on iOS, consider using the `cordova-plugin-wkwebview-file-xhr` plugin in your hybrid mobile application as an alternative to WKWebView. For additional information, see Using a Different Web View in your JET Hybrid Mobile App.

16
Optimizing Performance

Oracle JET applications are client-side HTML5 applications. Most performance optimization recommendations relating to client-side HTML applications also apply to applications developed using Oracle JET or to Oracle JET components. In addition, some Oracle JET components have performance recommendations that are specific to the component.

Topics:

- Typical Workflow for Optimizing Performance of Oracle JET Applications
- About Performance and Oracle JET Applications
- Adding Performance Optimization to an Oracle JET Application

Typical Workflow for Optimizing Performance of Oracle JET Applications

Understand performance optimization recommendations related to Oracle JET applications and components.

To optimize performance of an Oracle JET application, refer to the typical workflow described in the following table:

Task	Description	More Information
Identify general performance optimization goals	Identify performance optimization goals for client-side HTML5 applications.	About Performance and Oracle JET Applications
Apply performance optimization to your Oracle JET application	Take specific steps to optimize performance of your application.	Adding Performance Optimization to an Oracle JET Application

About Performance and Oracle JET Applications

In general, you can optimize an Oracle JET application the same way that you would optimize performance for any client-side HTML5 application.

There are many online resources that provide tips for performance optimization. For example, the Google Developers website describes processes for:

- Optimizing caching
- Minimizing round trip times between the client and the server
- Minimizing the payload size (downloads, responses, and cached pages)
- Minimizing request overhead (upload size)
- Improving browser page layout

- Optimizing performance on mobile devices

Most of these recommendations are up to you to implement, but Oracle JET includes features that can reduce the payload size and the number of trips to retrieve the Oracle JET application's CSS.

Adding Performance Optimization to an Oracle JET Application

Most tips for optimizing performance of web and hybrid mobile applications also apply to Oracle JET applications. However, there are some steps you can take that apply specifically to Oracle JET applications to optimize JavaScript, CSS, Oracle JET components, REST calls, and images.

JavaScript Performance Tips

Performance Tip	Details
Send only the JavaScript code that your application needs.	Oracle JET includes modules that you can load with RequireJS. For additional information, see Using RequireJS for Modular Development.

Send minified/obfuscated JavaScript.	Oracle JET provides minified versions of the Oracle JET library as well as third-party libraries when available. By default, these libraries are specified in the Oracle JET RequireJS bootstrap file included with all Oracle JET distributions.
	
	`requirejs.config({`
	`  // Path mappings for the logical module names`
	`  paths: {`
	`  'knockout': 'libs/knockout/knockout-3.4.0',`
	`  **'jquery': 'libs/jquery/jquery-3.1.1.min',**`
	`  **'jqueryui-amd': 'libs/jquery/jqueryui-amd-1.12.0.min',**`
	`  **'promise': 'libs/es6-promise/es6-promise.min',**`
	`  **'hammerjs': 'libs/hammer/promise-2.0.8.min',**`
	`  **'ojdnd': 'libs/dnd-polyfill/dnd-polyfill-1.0.0.min'**`
	`  **'ojs': 'libs/oj/v4.*x*.*x*/min',**`
	`  'ojL10n': 'libs/oj/v4.*x*.*x*/ojL10n',`
	`  'ojtranslations': 'libs/oj/v4.*x*.*x*/resources',`
	`  **'signals': 'libs/js-signals/signals.min',**`
	`  'text': 'libs/require/text',`
	`  'css': 'libs/require-css/css.min',`
	`  'customElements': 'libs/webcomponents/custom-elements.min',`
	`  'proj4': 'libs/proj4js/dist/proj4'`
	`  },`
	
	For additional information about using the RequireJS bootstrap file in your Oracle JET application, see Use RequireJS to Manage Library, Link, and Script References.
	
	If your application isn't using RequireJS, you can specify the links to the minified libraries in your main application file. For additional information, see Specify Library, Link, and Script References Without RequireJS.

Minimize the number of trips to retrieve the JavaScript.	Oracle JET doesn't provide support for minimizing the number of trips, but RequireJS has an optimization tool that you can use to combine modules. For additional detail, see r.js.
Use lazy loading for JavaScript not needed on first render.	You can lazy load content that is not needed on first render. For example, you can configure the `oj-film-strip` component to retrieve child node data only when requested. For an example, see the Lazy Loading (oj-film-strip) Oracle JET Cookbook example.

Performance Tip	Details
Compress or zip the payload.	Oracle JET has no control over the server, and this recommendation is up to you to implement. For some additional information and tips, see https://developers.google.com/speed/docs/best-practices/payload#GzipCompression.
Set cache headers.	JET has no control over the server, and this recommendation is up to you to implement. For additional information about cache optimization, see https://developers.google.com/speed/docs/best-practices/caching.

CSS Performance Tips

Performance Tip	Details
Send only the CSS that your application needs.	You can control the CSS content that goes into your application. For additional information, see Using Variables to Control CSS Content. If you're using the Oracle JET grid system, you can also control which responsive classes get included in the CSS. For details, see Controlling the Size and Generation of the CSS.
Send minified/obfuscated CSS.	By default, Oracle JET includes minified CSS. However, if you want to modify the CSS to send only what your application needs, you can use Sass to minimize your output. For additional information, see the `:compressed` option at: http://sass-lang.com/documentation/file.SASS_REFERENCE.html#output_style.

Oracle JET Component Performance Tips

Performance Tip	Details
Use Oracle JET components only when needed.	Build your page in HTML, and add only the Oracle JET components you need. If your application only needs the component briefly, add it and then remove it as soon as your application no longer need it.
Follow Oracle JET component best practices.	Consult the API documentation for the Oracle JET component. The JavaScript API Reference for Oracle® JavaScript Extension Toolkit (JET) includes a performance section for a component when applicable.
Limit number of Oracle JET components to 50 per page.	For optimal performance, limit the number of Oracle JET components on your page to 50. In addition, some Oracle JET components impose additional limitations: • 10 components per page: `oj-*-gauge`, `oj-input-date`, `oj-buttonset-*`, `oj-radioset`, `oj-checkboxset`, `oj-paging-control`, and `oj-accordion` • 5 components per page: `oj-tab-bar`, and `oj-toolbar` • 2 components per page: `oj-*-chart` with fewer than 100 data points, `oj-table`, and `oj-tree*` • 1 component per page: `oj-*-chart` with more than 100 data points and fewer than 1000. If your data contains more than 1000 data points, Oracle JET components may suffer a performance or usability hit.

REST Request Performance Tips

Performance Tip	Details
Limit number of REST requests.	Limit the number of REST requests to 10 on a page and the number of dependent REST calls to at most 2.

Image Optimization

Performance Tip	Details
Reduce image size.	Reducing the size of the images will result in faster downloads and reduce the time it takes to render the content on the screen. For example, Scalable Vector Graphics (SVG) images are usually smaller than Portable Network Graphics (PNG) images and scale on high resolution devices. There are also a number of third-party tools that you can use to reduce the size of your images. The tool that you select will depend on the image type, for example: • pngcrush: Utility to compress PNG images • svgomg: Utility to compress SVG images. You can use this tool online or download svgo to work with the images on your own system.
Reduce the number of round trips between client and server.	There are a number of techniques that you can use to reduce the number of round trips, and here are some examples: • Icon fonts Icon fonts are useful when your icon uses a single color. Oracle JET uses icon fonts, and you can see examples of them at: Icon Fonts. You can find utilities on the Internet such as IcoMoon that you can use to generate icon fonts. • Image Sprites An image sprite is a collection of images combined into a single image, reducing the number of server requests. You can find examples of them at http://www.w3schools.com/css/css_image_sprites.asp. • Lazy loading You can use lazy loading to defer the loading of images not in the user's viewport. You can find many examples and utilities on the Internet that use this technique. • Base64 Encoding You can use Base64 Encoding to inline image data. They are commonly used in data Uniform Resource Indicators (URIs), and you can find additional information about them at https://developer.mozilla.org/docs/Web/HTTP/data_URIs.

For additional performance tips, see the Google Developers documentation at: https://developers.google.com/speed/docs/best-practices/rules_intro.

17
Testing and Debugging

Test and debug Oracle JET web applications using your favorite testing and debugging tools for client-side JavaScript applications. You can use a similar process to debug hybrid mobile applications when run in the browser, and you can also test and debug hybrid mobile applications on a simulator, emulator, or device.

Topics:

- Typical Workflow for Testing and Debugging an Oracle JET Application
- Testing Oracle JET Applications
- Debugging Oracle JET Applications

Typical Workflow for Testing and Debugging an Oracle JET Application

Understand testing tools and debugging options for Oracle JET web and hybrid mobile applications.

To test and debug an Oracle JET application, refer to the typical workflow described in the following table:

Task	Description	More Information
Test Oracle JET applications	Identify testing tools for web and hybrid mobile applications.	Testing Oracle JET Applications
Debug Oracle JET applications	Identify debugging options for Oracle JET web and hybrid mobile applications.	Debugging Oracle JET Applications

Testing Oracle JET Applications

Use third-party tools such as QUnit or Selenium WebDriver to test your Oracle JET application. Google, Apple, and Microsoft also provide instructions for testing Andoid, iOS, and Windows applications.

Topics:

- Testing Applications
- Testing Hybrid Mobile Applications
- Using oj.BusyContext API in Automated Testing

Testing Applications

You can use virtually any testing tool that tests client-side HTML applications written in JavaScript for testing Oracle JET applications.

For internal development, Oracle JET uses the following tools for testing Oracle JET components and toolkit features:

- QUnit: JavaScript unit testing framework capable of testing any generic JavaScript project and used by the jQuery, jQuery UI, and jQuery Mobile projects.

  QUnit requires configuration on your test page to include library and CSS references. You must also add the HTML `div` element to your page. In the example below, the highlighted code shows additions required by QUnit. The actual paths will vary, depending upon where you install QUnit.

  ```
  <!doctype html>
  <html lang="en">
    <head>
      <link rel="stylesheet" href="../../../../code/css/libs/oj/vx.x.x/alta/oj-
  alta-min.css"></link>
      <link rel="stylesheet" href="../../../css/qunit.css">
      <script type="text/javascript" src="../../../js/qunit.js"></script>
      <script>
        QUnit.config.autoload = false;
        QUnit.config.autostart = false;
      </script>
      <script data-main="js/main" src="../../../../code/js/libs/require/
  require.js"></script>
      </head>
    <body>
      <div id="qunit"></div>
      <div id="qunit-fixture">
        <oj-slider id="slider1"></oj-slider>
      </div>
    </body>
  </html>
  ```

  For more information about QUnit, see http://qunitjs.com.

- Selenium WebDriver: Alternative method of testing applications that you do not want to modify for QUnit or that contain multiple HTML pages.

  For additional information, see http://docs.seleniumhq.org/projects/webdriver.

Testing Hybrid Mobile Applications

You can test your hybrid mobile application in a local browser, using an emulator or simulator, and on an attached device.

- Testing in a local browser

  When you invoke `ojet serve` with the `--browser` option to serve your hybrid mobile application in a local browser, you can use the same method for testing Android, iOS, and Windows applications that you would use for testing any web application. However, this method can only approximate the end user experience and is most useful early in the development process.

Depending upon your use case, you may need to add Cordova plugins to add functionality to your mobile hybrid app. Many Cordova plugins provide mock data when deploying to the browser platform. If, however, you add a Cordova plugin that doesn't have browser platform support, you can add objects that represent mock data to `cordovaMock.js` in `src/js`.

- Testing in an emulator or simulator

  You can invoke `ojet serve` with the `--emulator` option to test the functionality of your application in the iOS Simulator, Windows emulator, or Android Virtual Devices (AVDs) using the Android emulator. These methods can approximate the look and feel of an actual device, but you won't be able to test performance or responsiveness to touch reliably. For additional information, see

  – iOS Simulator
  – Android Emulator
  – Windows Emulator

- Testing on attached physical devices

  You can invoke `ojet serve` with the `destination=device` option to test the functionality on attached physical devices. This provides the most reliable form of testing, but you may not have access to all the devices that your users might use to run your application.

  If you want to serve your application to an iOS device, you must take additional steps as described in Packaging and Publishing Hybrid Mobile Applications.

- Working around cross-origin resource sharing (CORS) issues

  Hybrid mobile applications that communicate with remote services may encounter issues if they request resources that originate in difference domains. An example includes a response such as the following:

  ```
  No 'Access-Control-Allow-Origin' header is present on the requested resource.
  ```

  To work around these types of issue, your server-side administrator may need to modify the policy used by the remote server to allow cross-site requests. For example, to access a remote service managed by Oracle's Mobile Cloud Service (MCS), an MCS administrator configures MCS's `Security_AllowOrigin` environment policy with a comma separated list of URL patterns that identify the remote services that serve a resource from different domains.

For additional information about testing hybrid mobile applications on specific platforms, see the following resources:

- Android Testing
- iOS Testing
- Windows Debugging, testing, and performance

Using oj.BusyContext API in Automated Testing

Use `oj.BusyContext` to wait for a component or other condition to complete some action before interacting with it.

The purpose of the oj.BusyContext API is to accommodate sequential dependencies of asynchronous operations. Typically, you use `oj.BusyContext` in test automation

when you want to wait for an animation to complete, a JET page to load, or a data fetch to complete.

Wait Scenarios

The Busy Context API will block until all the busy states resolve or a timeout period lapses. There are four primary wait scenarios:

- Components that implement animation effects
- Components that fetch data from a REST endpoint
- Pages that load bootstrap files, such as the Oracle JET libraries loaded with RequireJS
- Customer-defined scenarios that are not limited to Oracle JET, such as blocking conditions associated with application domain logic

Determining the Busy Context's Scope

The first step for waiting on a busy context is to determine the wait condition. You can scope the granularity of a busy context for the entirety of the page or limit the scope to a specific DOM element. Busy contexts have hierarchical dependencies mirroring the document's DOM structure with the root being the page context. Depending on your particular scenario, target one of the following busy context scopes:

- Scoped for the page

  Choose the page busy context to represent the page as a whole. Automation developers commonly need to wait until the page is fully loaded before starting automation. Also, automation developers are usually interested in testing the functionality of an application having multiple Oracle JET components rather than a single component.

  ```
  var busyContext = oj.Context.getPageContext().getBusyContext();
  ```

- Scoped for the nearest DOM element

  Choose a busy context scoped for a DOM node when your application must wait until a specific component's operation completes. For example, you may want to wait until an `ojPopup` completes an open or close animation before initiating the next task in the application flow. Use the `data-oj-context` marker attribute to define a busy context for a DOM subtree.

  ```
  <div id="mycontext" data-oj-context>
    ...
    <!-- JET content -->
    ...
  </div>

  var node = $("#mycontext");
  var busyContext = oj.Context.getContext(node[0]).getBusyContext();
  ```

Determining the Ready State

After obtaining a busy context, the next step is to inquire the busy state. `oj.BusyContext` has two operations for inquiring the ready state: isReady() and whenReady(). The `isReady()` method immediately returns the state of the busy context. The `whenReady()` method returns a Promise that resolves when the busy states resolve or a timeout period lapses.

The following example shows how you can use `isReady()` with WebDriver.

```
public static void waitForJetPageReady(WebDriver webDriver, long timeoutInMillis)
{
  try
  {
    final WebDriverWait wait = new WebDriverWait(webDriver, timeoutInMillis /
_THOUSAND_MILLIS);
    // Eat any WebDriverException
    // "ExpectedConditions.jsReturnsValue" will continue to be called if it doesn't
return a value.
    // /ExpectedConditions.java#L1519
    wait.ignoring(WebDriverException.class);
    wait.until(ExpectedConditions.jsReturnsValue(_PAGE_WHEN_READY_SCRIPT));
  }

  catch (TimeoutException toe)
  {
    String evalString = "return
oj.Context.getPageContext().getBusyContext().getBusyStates().join('\\n');";
    Object busyStatesLog = ((JavascriptExecutor)webDriver).executeScript(evalString);
    String retValue = "";
    if busyStatesLog != null)
      retValue = busyStatesLog.toString();
      Assert.fail("waitForJetPageReady failed - !
oj.Context.getPageContext().getBusyContext().isReady() - busyStates: " +
        retValue);  }
}

// The assumption with the page when ready script is that it will continue to
execute until a value is returned or
// reached the timeout period.
//
// There are three areas of concern:
// 1) Has the application opt'd in on the whenReady wait for bootstrap?
// 2) If the application has opt'd in on the jet whenReady strategy for bootstrap
"('oj_whenReady' in window)",
// wait until jet core is loaded and have a ready state.
// 3) If not opt-ing in on jet whenReady bootstrap, make the is ready check if jet
core has loaded. If jet core is
// not loaded, we assume it is not a jet page.

// Check to determine if the page is participating in the jet whenReady bootstrap
wait period.

static private final String _BOOTSTRAP_WHEN_READY_EXP = "(('oj_whenReady' in window)
&& window['oj_whenReady'])";

// Assumption is we must wait until jet core is loaded and the busy state is ready.
static private final String _WHEN_READY_WITH_BOOTSTRAP_EXP =
"(window['oj'] && window['oj']['Context'] &&
oj.Context.getPageContext().getBusyContext().isReady() ?" +
" 'ready' : '')";

// Assumption is the jet libraries have already been loaded. If they have not, it's
not a Jet page.
// Return jet missing in action "JetMIA" if jet core is not loaded.
static private final String _WHEN_READY_NO_BOOTSTRAP_EXP =
"(window['oj'] && window['oj']['Context'] ? " +
"(oj.Context.getPageContext().getBusyContext().isReady() ? 'ready' : '') :
'JetMIA')";

// Complete when ready script
```

```
static private final String _PAGE_WHEN_READY_SCRIPT =
"return (" + _BOOTSTRAP_WHEN_READY_EXP + " ? " + _WHEN_READY_WITH_BOOTSTRAP_EXP +
" : " +
_WHEN_READY_NO_BOOTSTRAP_EXP + ");";
```

The following example shows how you can use `whenReady()` with QUnit.

```
// Utility function for creating a promise error handler
function getExceptionHandler(assert, done, busyContext)
{
  return function (reason)
    {
      if (reason && reason['busyStates'])
      {
        // whenReady timeout
        assert.ok(false, reason.toString());
      }
      else
      {
        // Unhandled JS Exception
        var msg = reason ? reason.toString() : "Unknown Reason";
        if (busyContext)
          msg += "\n" + busyContext;
        assert.ok(false, msg);
      }

      // invoke done callback
      if (done)
        done();
    };
};

QUnit.test("popup open", function (assert)

{
  // default whenReady timeout used when argument is not provided
  oj.BusyContext.setDefaultTimeout(18000);

  var done = assert.async();
  assert.expect(1);

  var popup = document.getElementById("popup1");

  // busy context scoped for the popup
  var busyContext = oj.Context.getContext(popup).getBusyContext();
  var errorHandler = getExceptionHandler(assert, done, busyContext);

  popup.open("#showPopup1");

  busyContext.whenReady().then(function ()
  {
    assert.ok(popup.isOpen(), "popup is open");
    popup.close();
    busyContext.whenReady().then(function ()
    {
      done();
    }).catch(errorHandler);
  }).catch(errorHandler);
});
```

Creating Wait Conditions

Jet components utilize the busy context to communicate blocking operations. You can add busy states to any scope of the busy context to block operations such as asynchronous data fetch.

The following high level steps describe how to add a busy context:

1. Create a Scoped Busy Context.
2. Add a busy state to the busy context. You must add a description that describes the purpose of the busy state. The busy state returns a resolve function which is called when it's time to remove the busy state.

> **Note:**
>
> Busy Context dependency relationships are determined at the point the first busy state is added. If the DOM node is re-parented after a busy context was added, the context will maintain dependencies with any parent DOM contexts.

3. Perform the operation that needs to be guarded with a busy state. These are usually asynchronous operations that some other application flow depends on its completion.
4. Resolve the busy state when the operation completes.

> **Important:**
>
> The application is responsible for releasing the busy state. The application must manage a reference to the resolve function associated with a busy state, and it must be called to release the busy state. If the DOM node that the busy context is applied to is removed in the document before the busy state is resolved, the busy state will be orphaned and will never resolve.

Debugging Oracle JET Applications

Since Oracle JET web applications are client-side HTML5 applications written in JavaScript, you can use your favorite browser's debugging facilities. The process for debugging hybrid mobile applications differs, depending on whether you're debugging your application in a web browser, emulator, or device.

Topics:

- Debugging Web Applications
- Debugging Hybrid Mobile Applications

Debugging Web Applications

Use your favorite browser's debugging facilities for debugging.

Before you debug your application, you should verify that your application is using the debug version of the Oracle JET libraries. If you used the tooling to build your

application in development mode, then your application should be using the debug library. If, however, you are using one of the sample applications, it may be configured to use the minified version of the Oracle JET libraries.

To verify that you are using the debug version of the library, check your RequireJS bootstrap file (typically `apphome/js/main.js`) for the Oracle JET library path, highlighted in **bold** below.

```
requirejs.config(
{
  baseUrl: 'js',

  // Path mappings for the logical module names
  paths:
  //injector:mainReleasePaths
  {
    ... contents omitted

    'ojs': 'libs/oj/vx.x.x/min',
    'ojL10n': 'libs/oj/vx.x.x/ojL10n',
    'ojtranslations': 'libs/oj/vx.x.x/resources',
    'text': 'libs/require/text',
    'signals': 'libs/js-signals/signals'
  }
  ... contents omitted
);
```

In this example, the path is pointing to the minified version of the Oracle JET libraries. To change to the debug version, edit the bootstrap file and replace `min` with `debug` as shown.

```
'ojs': 'libs/oj/vx.x.x/debug',
```

Debugging facilities vary by browser. For example, with Google Chrome you can:

- do normal debugging, including setting breakpoints and inspecting CSS using Chrome Inspector (`<a target="_blank" href="chrome://inspect">chrome://inspect</a>`) without having to install any 3rd party add-ons.
- add the Knockoutjs Context Debugger extension to examine the Knockout context and data for a DOM node in a sidebar.
- add the Window Resizer extension to resize the browser window to emulate common screen sizes or set your own. You can find the extensions and other developer tools at the Chrome Web Store.

If you're using an Integrated Development Environment (IDE) for development, you can use debugging tools such as the one provided by the NetBeans IDE to set break points and watches. For details, see Debugging and Testing JavaScript in an HTML5 Application.

Debugging Hybrid Mobile Applications

The process for debugging hybrid mobile applications differs, depending on whether you're debugging your application in a web browser, emulator, or device.

There are several ways to debug a hybrid mobile application, depending on the environment where you are running the application:

- Debugging in a local web browser

Since Cordova applications are HTML5 applications written in JavaScript, you can use the same debugging facilities that you would use for web applications as described in Debugging Web Applications. With this approach, however, you might find it difficult to debug code that depends on device services and Cordova plugins.

To run an Oracle JET hybrid mobile application in your desktop Safari or Chrome browser, serve it with the `--browser` option.

```
ojet serve ios|android|windows --browser
```

Once your app loads in the browser, you can use the browser's developer tools to view the source code, debug JavaScript using breakpoints and watches, or change the source code in the browser, which can be especially useful when you want to experiment with CSS changes that you intend to make in your app. For more information about the Chrome browser's developer tools, see the Chrome DevTools documentation and for the iOS Safari browser's developer tools, see the Web Inspector Tutorial.

- Debugging in the emulator

  You can run the app in the emulator which eliminates the need for an actual device. However, emulators can run slowly and include limited support for emulating native device services.

  To run an Oracle JET hybrid mobile application in the default emulator, serve it with the `--emulator` option.

  ```
  ojet serve ios|android|windows --emulator
  ```

  You can also specify a named emulator or simulator on Android or iOS devices:

  ```
  ojet serve ios|android|windows --emulator=emulator-name
  ```

  To determine the emulator or simulator name:

  – For Android emulators, run `android avd` at a command prompt.

  – For iOS simulators, at a command prompt in the application's hybrid directory, enter the following: `ios-sim showdevicetypes`.

- Debugging on the device

  Debugging the application on a real device provides the most accurate experience, but the development cycle can take longer since you must package the application, deploy it to the device, and use a desktop browser or other development tool to connect to it running on the device.

  To run an Oracle JET hybrid mobile application on the device, serve it with the `--device` option.

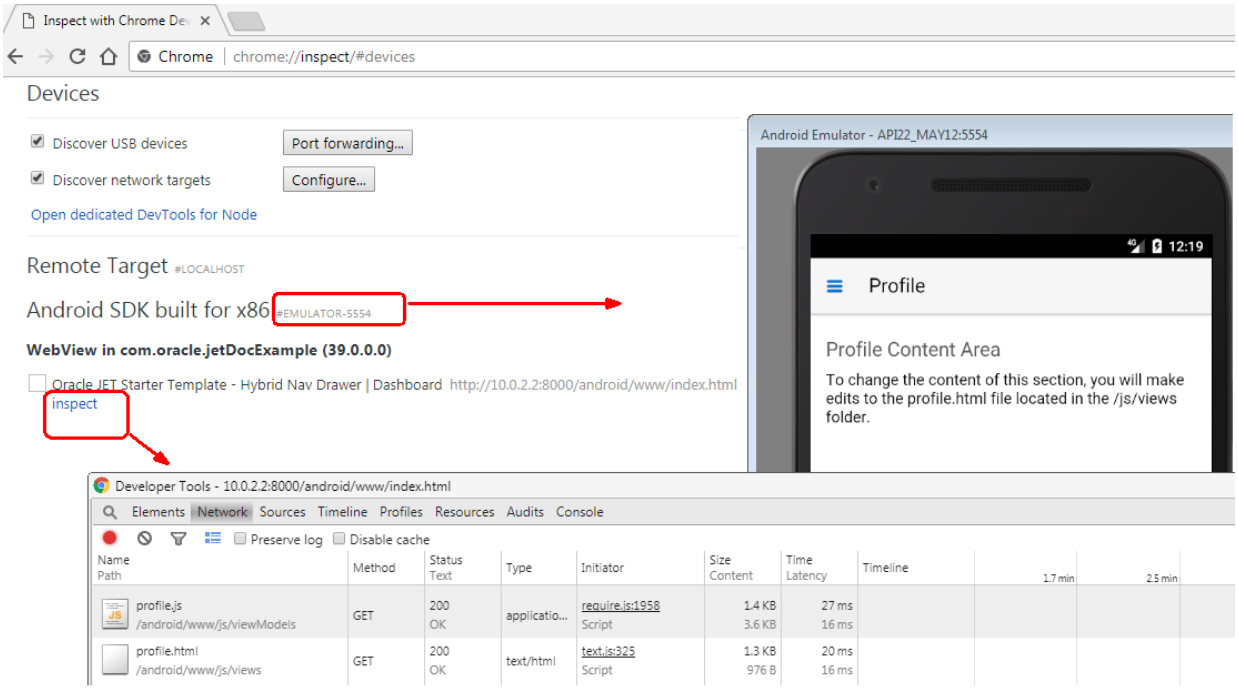
  ```
  ojet serve ios|android|windows --device
  ```

Once the app installs and runs on the device or on the emulator, you can use your browser's developer tools to debug and inspect your application's code as it executes. The browser developer tools that you use depend on the platform where your application runs.

Use the Safari browser's Web Inspector, described in the Safari Web Inspector Guide, if you run your app an iOS device or simulator.

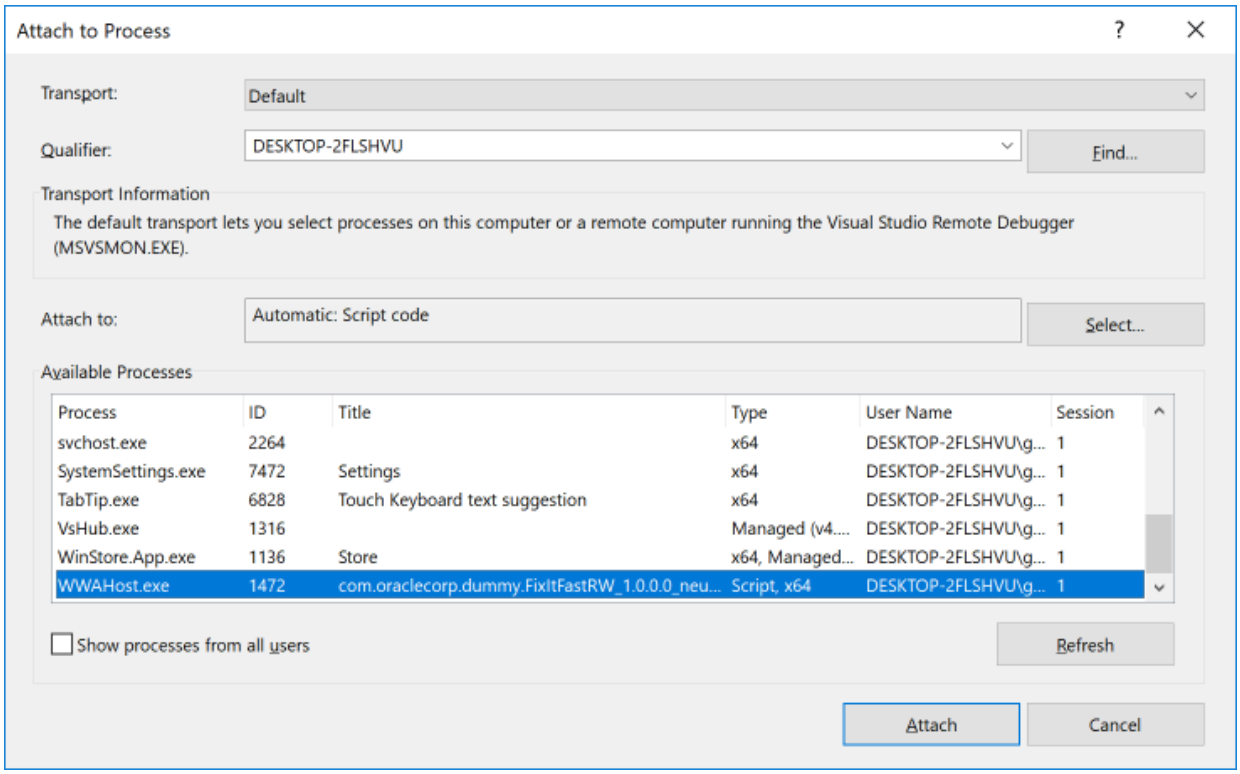
Use the Chrome browser's developer tools if your app runs on an Android emulator or device. For the Android platform, once JET has served the app to the emulator or

device, enter `chrome://inspect/` in the Chrome browser's address bar to access the developer tools. This renders the Inspect with Chrome Developer Tools page that lists the connected devices and the applications on these devices that you can inspect using the developer tools. Click the **inspect** link beside the app you want to inspect, as illustrated by the following figure which shows the developer tools in the foreground for the application running in the emulator.



For more information about debugging apps on an Android device using Chrome DevTools, see Get Started with Remote Debugging Android Devices in Google's documentation.

Use Visual Studio to debug and inspect your application's code if your application runs on the Windows platform. Serve your application on your Windows machine and attach the process for the application to the Visual Studio debugger, as described in Microsoft's documentation. JavaScript apps, such as your application, run in an instance of the `wwahost.exe` process on Windows, so multiple instances of the `wwahost.exe` process appear in the Attach to Process dialog if you have more than one application running. Use the Title column of the Attach to Process dialog to select the correct application to attach, as illustrated by the following image where an instance of the FixItFast sample application is selected.



Once you have attached your app, use the Visual Studio debugger, as described in Microsoft's documentation, to debug your application.

18

Packaging and Deploying Applications

If you used Oracle JET tooling to create your Oracle JET application, you can package web applications for deployment to a web or application server and hybrid mobile applications for deployment to the Google Play or Apple App stores.

Topics

- Typical Workflow for Packaging and Deploying Applications
- Packaging and Deploying Web Applications
- Packaging and Publishing Hybrid Mobile Applications
- Removing Build Output and Extraneous Files from Your JET Application's Source

Typical Workflow for Packaging and Deploying Applications

Understand how to use Oracle JET's tooling to package and deploy your Oracle JET application.

To package and deploy your Oracle JET application, refer to the typical workflow described in the following table:

Task	Description	More Information
Remove Build Output and Extraneous Files from Your JET Application's Source	Use `ojet clean` or `ojet script` to remove extraneous files from the source of your project if you want to package your source code for distribution to colleagues.	Removing Build Output and Extraneous Files from Your JET Application's Source
Package and Deploy Web Application	Use the JET tooling to package your application and ready it for deployment.	Packaging and Deploying Web Applications
Package and Publish Hybrid Mobile Application	Use the JET tooling to package your application and ready it for publishing.	Packaging and Publishing Hybrid Mobile Applications

Packaging and Deploying Web Applications

Use the Oracle JET tooling and third-party tools to package your Oracle JET web application and deploy it to any type of web or application server.

Topics

- Packaging Web Applications
- Deploying Web Applications

Packaging Web Applications

If you created your application using the tooling, use the Oracle JET command-line interface (CLI) to create a release version of your application containing your application scripts and applicable Oracle JET code in minified format.

1. From a terminal prompt in your application's root directory, enter the following command: `ojet build --release`.

   The command will take some time to complete. When it's successful, you'll see the following message: `Build finished!`.

   The command replaces the development version of the libraries and scripts in `web/js/` with minified versions where available.

2. To verify that the application still works as you expect, run `ojet serve` with the `release` option.

   The `ojet serve --release` command takes the same arguments that you used to serve your web application in development mode.

   ```
   ojet serve --release [--serverPort=server-port-number --serverOnly]
   ```

> **Tip:**
>
> For a complete list of options, type `ojet help serve` at the terminal prompt.

Deploying Web Applications

Oracle JET is a collection of HTML, JavaScript, and CSS files that you can deploy to any type of web or application server. There are no unique requirements for deploying Oracle JET applications.

Deployment methods are quite varied and depend upon the type of server environment your application is designed to run in. However, you should be able to use the same method for deploying Oracle JET applications that you would for any other client interface in your specific environment.

For example, if you normally deploy applications as zip files, you can zip the `web` directory and use your normal deployment process.

Packaging and Publishing Hybrid Mobile Applications

You can use the Oracle JET command line tooling to package your hybrid mobile app for publishing to Google Play, Windows Store, or Apple App Store.

Topics

- About Packaging and Publishing Hybrid Mobile Applications
- Packaging a Hybrid Mobile App on Android
- Packaging a Hybrid Mobile App on iOS
- Packaging a Hybrid Mobile App on Windows

About Packaging and Publishing Hybrid Mobile Applications

The steps to package your app for release are broadly the same, irrespective of the platform where your app is going to be installed.

You create a build configuration file that identifies the location of your digital certificate, and then use the build configuration file as an input parameter to the `ojet build` command in order to sign your app with the digital certificate.

Before you proceed to signing and packaging your app for each platform where it will be used, review the entries in your app's `config.xml` file, as described in Review Your Application Settings in the config.xml File. Once you complete that task, proceed to the platform-specific instructions in the following sections.

Packaging a Hybrid Mobile App on Android

To prepare your app for distribution on Android devices, you need to sign it using a digital certificate and package it into an `.APK` file.

To sign an app that you want to release and publish through the Google Play Store or some other distribution mechanism, you need a unique key. Your organization may provide you with one, in which case you specify its location in the build configuration file that you use to sign and package your app. If you don't have a key, you can create one that you use to sign your app. Subsequent updates to your app must be signed using the same key that you create. You can create a self-signed key using the JDK's `keytool` utility or using the dialogs that Android Studio provides from its **Build** > **Generate Signed APK** menu. For more information about the latter option, see the "Generate a key and keystore" section in the Sign Your App page on the Android developer's site.

Once you have generated your key, specify the location of its store, its alias, and its access credentials in your build configuration file, as shown by the following example.

```
{
  "android": {
    "release": {
      "keystore": "/home/pathTo/keystore/android.jks",
      "storePassword": "MyKeystorePassword",
      "alias": "myAndroidKey",
      "password" : "MyKeyPassword",
      "keystoreType": ""
    }
  }
}
```

You can now package your app into an `.APK` file (the installation file type for apps on Android devices).

At a terminal prompt, in your app's top-level directory, enter the following command:

```
ojet build android --release --build-config=/pathTo/yourBuildConfig.json
```

On successful completion, the terminal window displays output similar to the following:

```
...

...
 Oracle JET CLI
```

```
Running "build:release" (build) task
Running "oraclejet-build:release" (oraclejet-build) task
 Oracle JET Grunt plugin
Processing Grunt command...
 Oracle JET Tooling
cleaning staging path.....
...
BUILD SUCCESSFUL

Total time: 4.376 secs

Built the following apk(s):
        /appRootDir/hybrid/platforms/android/build/outputs/apk/android-release.apk
Cordova compile finished....

Done.
```

The build command outputs the `.APK` file that packages your app to the following location:

`/appRootDir/hybrid/platforms/android/build/outputs/apk/android-release.apk`.

Use this file to release your app through a public app marketplace, such as Google Play, a private app store, or some other means. End users who install your app from a location other than Google Play need to configure their device to opt in to install the app from an unknown source. For information about all these options, see https://developer.android.com/studio/publish/index.html#publishing-release.

Packaging a Hybrid Mobile App on iOS

To prepare your app for distribution on iOS devices, you need to sign it and package it into an `.IPA` file.

Joining the Apple Developer Program is the first step to submit your iOS app to the Apple App Store, to distribute an in-house app, or to sign an app that you distribute outside the Apple App Store. As a member, you have access to the resources you need to configure app services and to submit new apps and updates. For more information on the Apple Developer Program, see About the Apple Developer Program in Apple's documentation.

Code signing your app allows users to trust that your app has been created by a source known to Apple and that it hasn't been tampered with. All apps must be code signed and provisioned to deploy to an iOS device, to use certain Apple services, to be distributed for testing on iOS devices, or to be uploaded to the Apple App Store.

Signing identities are used to sign your app or installer package. A signing identity consists of a public-private key pair that Apple issues. The public-private key pair is stored in your keychain, and is used by cryptographic functions to generate the signature. A *certificate* stored in your developer account contains the associated public key. An *intermediate certificate* is also required to be in your keychain to ensure that your certificate is issued by a certificate authority.

Code signing requires that you have both the signing identity and the intermediate certificate installed in your keychain. When you install Xcode, Apple's intermediate certificates are added to your keychain for you.

There are different types of certificates. A *development* certificate identifies a person on your team and is used to run an app on a single device. A *distribution* certificate identifies the team and is used to submit your app to the Apple App store or to distribute it outside of the store. Within an enterprise (organization) distribution certificates can be shared by team members in order to deploy various apps to a number of different devices. Certificates are issued and authorized by Apple.

If you are a member of an organization membership in the Apple Developer Program, you should request your team agent to provide your required signing identifies and certificates. Otherwise, you can create all the types of certificates and signing identities you need using the Apple Developer portal at https://developer.apple.com/account.

For more information on managing signing identities and certificates, see Maintaining Your Signing Identities and Certificates in Apple's documentation.

When you code sign an app, a *provisioning profile* is installed into the package that associates one or more certificates and one or more devices with an *application ID*. Provisioning profiles are created using the Apple Developer portal and must be downloaded and installed into Xcode on your development machine.

An application ID is a two-part string used to identify one or more apps from a single development team. The string consists of a *Team ID* and a *bundle ID search string*, with a period (.) separating the two parts. The Team ID is supplied by Apple and is unique to a specific development team, while the bundle ID search string is supplied by you to match either the bundle ID of a single app or a set of bundle IDs for a group of your apps.

Application IDs and devices are registered with your team account using the Apple Developer portal.

For more information on managing application IDs, devices and provisioning profiles, see Maintaining Identifiers, Devices, and Profiles in Apple's documentation.

Once you have created and installed the necessary code signing artifacts, specify the details in your build configuration file, as shown in the following example:

```
{
  "ios": {
    "debug": {
      "provisioningProfile": "My Developer Provisioning Profile ID",
      "developmentTeam": "My Team ID",
      "codeSignIdentity": "iPhone Developer",
      "packageType": "development"
    },
    "release": {
      "provisioningProfile": "My Distribution Provisioning Profile ID",
      "developmentTeam": "My Team ID",
      "codeSignIdentity": "iPhone Distribution",
      "packageType": "enterprise"
    }
  }
}
```

You can now package your app into an .IPA file (the installation file type for apps on iOS devices).

At a terminal prompt, in your app's top-level directory, enter the following command:

```
ojet build ios --device --release --build-config=/pathTo/yourBuildConfig.json
```

On successful completion, the terminal window displays output similar to the following:

```
...
** ARCHIVE SUCCEEDED **


Exported appName.xcarchive to: appDir/hybrid/platforms/ios/build/device

** EXPORT SUCCEEDED **


Cordova compile finished....

Done.
```

The resulting .IPA file will be located at:

appDir/hybrid/platforms/ios/build/device/appName.ipa

This file can be deployed to an iOS device that matches the installed provisioning profile using iTunes, or you can release your app through the public marketplace, Apple App Store, a private app store, or some other means. For information about submitted your app to the Apple App Store, see Submitting Your App to the Store in Apple's documentation.

Packaging a Hybrid Mobile App on Windows

To prepare your app for distribution on Windows devices or submission to the Windows Store, you need to sign it and package it into an `.APPX` file.

To complete this task, you need access to a digital certificate (a `.PFX` file) to sign your app. You must also create a build configuration file (`build.json`) where you specify the location of the digital certificate and a number of other parameters. You use the `build.json` file as an input parameter to the `ojet build` command that JET uses to package your app for Windows.

How to Create the Build Configuration File to Package Your App on Windows

You need to create a build configuration file in JSON format that passes details, such as the location of your PFX file, to the `ojet build` command that JET uses to package your app for Windows.

The following example build configuration file shows sample entries that you can use to package an app in debug and release mode.

```
{
    "windows": {
        "debug": {
            "packageCertificateKeyFile": "C:\\AppRootDir\hybrid\platforms\windows
\CordovaApp_TemporaryKey.pfx"
        },
        "release": {
            "packageCertificateKeyFile": "c:\\path-to-key\\keycert.pfx",
            "packageThumbprint": "ABCABCABCABC123123123123",
            "publisherId": "CN=Doc Example,OU=JET,O=Oracle,C=US"
        }
    }
}
```

Create your `build.json` file using the previous example as a reference. For information about creating and installing a PFX file, see todo. Obtain the values for the `packageThumbprint` and the `publisherID` entries by running the following command in a PowerShell command window:

```
Get-PfxCertificate -FilePath "c:\path-to-key\keycert.pfx"
```

PowerShell prompts you for the PFX file's password if the file is password protected. Output similar to the following then appears:

```
Thumbprint                                Subject
----------                                -------
702F25BA3FED453A3F8ADCC13900A6353703AB54  CN=Doc Example, OU=JET, O=Oracle, C=US
```

For more information about the Get-PfxCertificate command, see Microsoft's documentation.

How to Build Your App for Windows

Execute the `ojet build` command with the `release` flag from your app's top level directory to build the app.

You must specify the architecture that you want the app to target when you package the app for release, as demonstrated by the following example that targets the x64 and x86 architectures.

```
ojet build windows --release --platform-options="--archs=\"x64 x86\"" --build-
config=pathTo\build.json
```

On successful completion, the following directory contains the application package and other resources for your app:

```
AppRootDir\hybrid\platforms\windows\AppPackages\
```

The AppPackages directory contains another directory (`CordovaApp.Windows10_*_Test`). The actual name of the directory depends on the version number of your app and the architecture that it targets. For example, a version 1 app that targets the x64 architecture will be named `CordovaApp.Windows10_1.0.0.0_x64_Test`.

Submit the application package (`.appx`) in the directory to the Windows App Store or distribute the contents to end users with Windows devices who want to install your app. The directory includes a PowerShell script (`Add-AppDevPackage.ps1`) that end users execute to install the application. In addition to the script, the directory contains dependent packages and the certificate that signed the application. The following example lists the contents:

```
Add-AppDevPackage.ps1
Add-AppDevPackage.resources
CordovaApp.Windows10_1.0.0.0_x64.appx
CordovaApp.Windows10_1.0.0.0_x64.cer
```

For information about how to submit your app to the Windows Store, see https://developer.microsoft.com/en-us/store/publish-apps.

Removing Build Output and Extraneous Files from Your JET Application's Source

The Oracle JET CLI provides commands (clean, strip, and restore) that manage the source code of your JET application by removing extraneous files, such as the build output for the platforms your JET application supports or npm modules installed into your project.

Consider using these commands when you want to package your source code for distribution to colleagues or others who may work on the source code with you. Use of these commands may not be appropriate in all circumstances. Use of the clean and strip commands will, for example, remove the build output for hybrid mobile applications that includes the installation files to install the application on a user's device.

ojet clean

Use the `ojet clean` command to clean the build output of your JET application. Specify the appropriate parameter with the `ojet clean` command to clean the build output on the platform(s) that your JET application supports (`android`, `ios`, `windows`, and `web`). This command can be useful when developing a hybrid mobile application that makes use of staging files as you can make sure that all staging files are removed between builds. These staging files will be regenerated the next time that you build or serve the hybrid mobile application. Run the following command to clean the output of your application that targets the Android platform:

```
ojet clean android
```

Similarly, run `ojet clean web` to remove the contents of your application's root directory's `web` directory.

ojet strip

Use `ojet strip` when you want to remove all non-source files from your JET application. In addition to the build output removed by the `ojet clean` command, `ojet strip` removes additional dependencies, such as Cordova plugins, and npm modules installed into your project. A typical usage scenario for the `ojet strip` command is when you want to distribute the source files of your JET application to a colleague and you want to reduce the number of files to transmit to the minimum.

ojet restore

Use the `ojet restore` command to restore the dependencies, plugins, libraries, and composite components that the `ojet strip` command removes. After the `ojet restore` command completes, use the `ojet build` and/or `ojet serve` commands to build and serve your JET application.

A
Troubleshooting

Follow the same procedure for troubleshooting your Oracle JET application that you would follow for any client-side JavaScript application.

If you're having issues troubleshooting a specific Oracle JET component or toolkit feature, see Oracle JET Support. Before requesting support, be sure to check the product Release Notes.

B

Oracle JET v4.1.0 Tooling Migration

If you used Oracle JET tooling to scaffold your application with Oracle JET `v3.0.0` or higher, you can migrate your application manually to `v4.1.0`.

The steps to migrate depend upon whether you are migrating your application from v3.0.0 or v4.0.0.

Before you migrate your application, be sure to check the Oracle JET Release Notes for any component, framework, or other change that could impact your application.

> **Important:**
>
> This process is not supported for Oracle JET releases prior to `v3.0.0`.

Topics:

- Migrating a v3.x.0 Application to v4.1.0
- Migrating a v4.0.0 Application to v4.1.0

Migrating a v3.x.0 Application to v4.1.0

To migrate your Oracle JET application to `v4.1.0`, you must upgrade npm packages and update theme and library reference paths.

One of the major changes to tooling for `v4.x.0` is the replacement of the Oracle JET generator (`generator-oraclejet`) with the Oracle JET command-line interface (`ojet-cli`). In the steps that follow, you will remove the generator and associated packages and replace them with the `v4.1.0` versions of the tooling packages.

The other major change to tooling for `v4.x.0` is the removal of grunt and Yeoman as tooling dependencies. In the steps that follow, you can also choose to remove them from your system if you no longer need them.

1. At a terminal prompt, enter the following command to clean the npm cache:

   ```
   npm cache clean
   ```

2. Remove `generator-oraclejet` and install `ojet-cli`.

   As Administrator on Windows or using `sudo` as needed on Macintosh and Linux systems, enter the following commands in a terminal window:

   ```
   [sudo] npm uninstall -g generator-oraclejet
   [sudo] npm install -g @oracle/ojet-cli@~4.1.0
   ```

3. Enter the following commands to change to the application's top level directory and upgrade local npm dependencies:

```
cd appDir
npm uninstall oraclejet grunt-oraclejet oraclejet-tooling
npm install @oracle/oraclejet@~4.1.0 @oracle/oraclejet-tooling@~4.1.0 --save
```

4. Review the Theming section in the release notes for SCSS variable or other changes, and update your theming files manually, if needed.

5. In the application's `src` directory, edit `index.html` and replace the existing css reference with the `v4.1.0` version.

```
<link rel="stylesheet" href="css/libs/oj/v4.1.0/alta/oj-alta-min.css" id="css" />
```

6. Update the Oracle JET library configuration paths to reference the `v4.1.0` version of Oracle libraries.

   a. Open *appDir*`/src/js/main.js` and edit `requirejs.config()` paths to point to the updated libraries highlighted below.

```
paths:
 //injector:mainReleasePaths
 {
   'knockout': 'libs/knockout/knockout-3.4.0.debug',
   'jquery': 'libs/jquery/jquery-3.1.1',
   'jqueryui-amd': 'libs/jquery/jqueryui-amd-1.12.0',
   'promise': 'libs/es6-promise/es6-promise',
   'hammerjs': 'libs/hammer/hammer-2.0.8',
   'ojdnd': 'libs/dnd-polyfill/dnd-polyfill-1.0.0',
   'ojs': 'libs/oj/v4.1.0/debug',
   'ojL10n': 'libs/oj/v4.1.0/ojL10n',
   'ojtranslations': 'libs/oj/v4.1.0/resources',
   'text': 'libs/require/text',
   'signals': 'libs/js-signals/signals',
   'customElements': 'libs/webcomponents/custom-elements.min',
   'proj4': 'libs/proj4js/dist/proj4-src',
   'css': 'libs/require-css/css',
 }
 //endinjector
```

   b. Open *appDir*`/src/js/main-release-paths.json` and edit the library version numbers to use the same version numbers you referenced in the previous step.

7. To test the migration, run `ojet build` and `ojet serve` with appropriate options to build and serve the application.

   For a list of available options, enter the following command at a terminal prompt in your application's top level directory: `ojet help`.

   If your application uses a custom theme, be sure to include the `--theme` option to regenerate the CSS:

```
ojet build [options] --theme=myTheme
```

   To specify multiple custom themes, use:

```
ojet build [options] --theme=myTheme --themes=myTheme,myTheme1,myTheme2
```

8. If you no longer need grunt and Yeoman on your system, remove them.

   As Administrator on Windows or using `sudo` as needed on Macintosh and Linux systems, enter the following commands in a terminal window:

```
[sudo] npm uninstall -g yo
[sudo] npm uninstall -g grunt-cli
```

Migrating a v4.0.0 Application to v4.1.0

To migrate your Oracle JET application from v4.0.0 to `v4.1.0`, you must upgrade npm packages and update theme and library reference paths.

1. At a terminal prompt, enter the following command to clean the npm cache:

   ```
   npm cache clean
   ```

2. Remove the existing version of the `ojet-cli` tooling package and install the latest version..

   As Administrator on Windows or using `sudo` as needed on Macintosh and Linux systems, enter the following commands in a terminal window:

   ```
   [sudo] npm uninstall -g @oracle/ojet-cli
   [sudo] npm install -g @oracle/ojet-cli@~4.1.0
   ```

3. Enter the following commands to change to the application's top level directory and upgrade local npm dependencies:

   ```
   cd appDir
   npm uninstall @oracle/oraclejet-tooling @oracle/oraclejet
   npm install @oracle/oraclejet-tooling@~4.1.0 @oracle/oraclejet@~4.1.0 --save
   ```

4. Review the Theming section in the release notes for SCSS variable or other changes, and update your theming files manually, if needed.

5. In the application's `src` directory, edit `index.html` and replace the existing css reference with the `v4.1.0` version.

   ```
   <link rel="stylesheet" href="css/libs/oj/v4.1.0/alta/oj-alta-min.css" id="css" />
   ```

6. Update the Oracle JET library configuration paths to reference the `v4.1.0` version of Oracle libraries.

   a. Open *appDir*`/src/js/main.js` and edit `requirejs.config()` paths to point to the updated libraries highlighted below.

   ```
   paths:
    //injector:mainReleasePaths
    {
      'knockout': 'libs/knockout/knockout-3.4.0.debug',
      'jquery': 'libs/jquery/jquery-3.1.1',
      'jqueryui-amd': 'libs/jquery/jqueryui-amd-1.12.0',
      'promise': 'libs/es6-promise/es6-promise',
      'hammerjs': 'libs/hammer/hammer-2.0.8',
      'ojdnd': 'libs/dnd-polyfill/dnd-polyfill-1.0.0',
      'ojs': 'libs/oj/v4.1.0/debug',
      'ojL10n': 'libs/oj/v4.1.0/ojL10n',
      'ojtranslations': 'libs/oj/v4.1.0/resources',
      'text': 'libs/require/text',
      'signals': 'libs/js-signals/signals',
      'customElements': 'libs/webcomponents/custom-elements.min',
      'proj4': 'libs/proj4js/dist/proj4-src',
      'css': 'libs/require-css/css',
    }
    //endinjector
   ```

   b. Open *appDir*`/src/js/main-release-paths.json` and edit the library version numbers to use the same version numbers you referenced in the previous step.

7. To test the migration, run `ojet build` and `ojet serve` with appropriate options to build and serve the application.

   For a list of available options, enter the following command at a terminal prompt in your application's top level directory: `ojet help`.

   If your application uses a custom theme, be sure to include the `--theme` option to regenerate the CSS:

   ```
   ojet build [options] --theme=myTheme
   ```

   To specify multiple custom themes, use:

   ```
   ojet build [options] --theme=myTheme --themes=myTheme,myTheme1,myTheme2
   ```

C

Oracle JET References

Oracle JET includes third-party libraries and tools that are referenced throughout the guide. Oracle also provides optional tools and libraries to assist with application development.

Topics:

- Oracle Libraries and Tools
- Third-Party Libraries and Tools

Oracle Libraries and Tools

Oracle provides optional tools and libraries to use in conjunction with Oracle JET. Use this reference to locate additional information about the Oracle products referenced throughout the guide.

Name	Description	Additional Information
NetBeans	Integrated Development Environment (IDE) with support for HTML5 application development	https://netbeans.org
Oracle Alta UI	A mobile and browser application design system	Oracle Alta UI

Third-Party Libraries and Tools

Use this reference to locate additional information about the third-party libraries and tools used by Oracle JET and referenced throughout the guide.

Name	Description	Additional Information
Apache Cordova (Mobile only)	Open source mobile development framework that allows you to use HTML5, CSS3, and JavaScript for cross-platform development targeted to multiple platforms with one code base	http://cordova.apache.org/
CSS3 (Cascading Style Sheets)	Used for adding style to Web applications	http://www.w3.org/Style/CSS
Hammer.js	JS library used for multi-touch gestures	http://hammerjs.github.io/getting-started
HTML5 (Hypertext Markup Language 5)	Core language of the World Wide Web	http://www.w3.org/TR/html5
JavaScript	Scripting language used in client-side applications	https://developer.mozilla.org/en/About_JavaScript
js-signals	JS library used for custom event/messaging system	http://millermedeiros.github.io/js-signals

Name	Description	Additional Information
jQuery	JS library designed for HTML document traversal and manipulation, event handling, animation, and Ajax	http://jquery.com
jQuery UI	JS library built on top of jQuery for UI development. Oracle JET includes the UI Core download only.	http://www.jqueryui.com
Knockout	JS library used for two-way data binding	http://www.knockoutjs.com
Node.js	Open source, cross-platform runtime environment for developing server-side web applications, used by Oracle JET for package management	https://nodejs.org
proj4js	JavaScript library designed for transforming point coordinates from one coordinate system to another, including datum transformations.	http://proj4js.org/
QUnit	JavaScript unit testing framework	http://qunitjs.com
RequireJS	JS file and module loader used for managing library references and lazy loading of resources	http://www.requirejs.org
RequireJS CSS	RequireJS CSS plugin that allows to load CSS files.	http://requirejs.org/docs/faq-advanced.html
SASS (Syntactically Awesome Style Sheets)	Extends CSS3 and enables you to use variables, nested rules, mixins, and inline imports	http://www.sass-lang.com
Selenium WebDriver	Alternative method of testing applications	http://docs.seleniumhq.org/projects/webdriver