Oracle Fusion Cloud Sales Automation

How do I create an application extension for custom objects?

Oracle Fusion Cloud Sales Automation
How do I create an application extension for custom objects?

F88584-18

Author: Jiri Weiss

Contents

Get Help

There are a number of ways to learn more about your product and interact with Oracle and other users.

Get Help in the Applications

Some application pages have help icons to give you access to contextual help. If you don't see any help icons on your page, click your user image or name in the global header and select Show Help Icons. If the page has contextual help, help icons will appear.

Get Support

You can get support at *My Oracle Support*. For accessible support, visit *Oracle Accessibility Learning and Support*.

Get Training

Increase your knowledge of Oracle Cloud by taking courses at *Oracle University*.

Join Our Community

Use *Cloud Customer Connect* to get information from industry experts at Oracle and in the partner community. You can join forums to connect with other customers, post questions, suggest *ideas* for product enhancements, and watch events.

Learn About Accessibility

For information about Oracle's commitment to accessibility, visit the *Oracle Accessibility Program*. Videos included in this guide are provided as a media alternative for text-based topics also available in this guide.

Share Your Feedback

We welcome your feedback about Oracle Applications user assistance. If you need clarification, find an error, or just want to tell us what you found helpful, we'd like to hear from you.

You can email your feedback to *oracle_fusion_applications_help_ww_grp@oracle.com*.

Thanks for helping us improve our user assistance!

1 Before You Begin

Before You Create an Application Extension

Before your team can start creating application extensions, you must first set up Oracle Visual Builder Studio. You only need to set up VB Studio once for every implementation.

Complete VB Studio implementation steps are documented in the *Oracle Cloud Administering Visual Builder Studio* guide. See the topic: *How Do I Set Up VB Studio?*

Required: Set the Extension ID for Sales

When using VB Studio to extend Sales pages, your extension must use the extension ID: `site_cxsales_Extension`. You set this extension ID when you first set up your project.

A project collects all the people, tools, and processes you need to complete a discrete software effort in VB Studio. Oracle best practice dictates that you use a single project for all the extension work you do within the Oracle Cloud Application environment family.

You can create this project using one of two methods discussed in the following video: *Create the Visual Builder Studio Project*.

Each method requires a different way to set the extension ID:

- Create a project from a Sales page by clicking the **Edit Page in Visual Builder Studio** link in the **Settings and Actions** menu. This is the recommended method to create a project because it automates the creation of key VB Studio components. See the topic: *Create a Simple Extension*.

  If you choose this method, then you'll update your project's extension ID to `site_cxsales_Extension` by editing the extension-level settings. See the topic: *Establish Extension-Level Settings*.
- Create a project from the Organization home page. See the topic: *Manually Create a Project for Extensions*.

  If you choose this method, then you'll enter the required extension ID when you create your own workspace. See the topic: *Create an Extension*.

Note: Be sure to publish your extension so that the updated extension ID becomes the default going forward for everyone else working on the extension.

Tip: Create Additional Workspaces

At some point in your extension lifecycle, you might need to create a new workspace in an existing project. You may want to create a new workspace from the main branch if you forget what changes a particular workspace contains, for example. Follow the instructions in the topic: *Clone an Existing Repository*.

You can also view the following video: *Create a Workspace*.

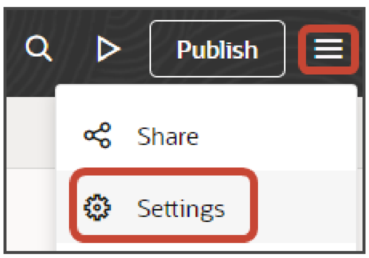
How can I change my project's Extension ID?

When working with a Sales application extension in Oracle Visual Builder Studio, your project's extension ID must be `site_cxsales_Extension`. This topic illustrates how to correct the extension ID, if required.

To change the extension ID for a project:

1. In Visual Builder Studio, from the left navigator, click **Environments** > **Deployments**.
2. Undeploy any deployments to target servers.

   See *Delete an Extension*.
3. Navigate to your workspace and, from the upper menu, click **Settings**.

4. In the **Extension ID** field, enter `site_cxsales_Extension`.
5. Build and deploy your extension once more.

   See *Build Your Applications* and *Package, Deploy, and Manage Extensions*.

2 Add a Custom Top Level Object

Prerequisites for Using the CX Extension Generator

The CX Extension Generator is a tool that automates many of the manual tasks required to build an application extension from scratch. Before you can build an application using the CX Extension Generator, complete these prerequisite steps.

Setup Prerequisites Before Using the CX Extension Generator

Setup Step	Setup Location	More Information
1. Create and activate a sandbox.	Sandboxes work area	*Create and Activate Sandboxes*
2. Create custom top level objects, as well as any child objects, related objects, and relationships.	Application Composer	This chapter provides you with step-by-step instructions for creating a custom application using the CX Extension Generator and Oracle Visual Builder Studio. To build this custom application, you will need to create a custom top level object in Application Composer, as well as a custom child object. The example objects used in this chapter are a Payment object and its child object, Payment Line. We will also add a panel for a related object, Shipment. For more information about creating custom objects, see *Define Objects*.
3. Publish the sandbox.	Application Composer	*Publish Sandboxes* Publish the sandbox so that you can enable all custom objects for Adaptive Search, in the next step. **Note:** If you're already running Visual Builder Studio, then sign out and sign back in before continuing to configure your application extension. Doing this ensures that Visual Builder picks up the latest published changes from Application Composer.
4. Enable all custom objects that you created for Adaptive Search and publish your changes. In addition, create at least one saved search.	Setup and Maintenance work area • Offering: Sales • Functional Area: Sales Foundation	*Enable Business Objects for Adaptive Search* This step is required because the list page is dependent on Adaptive Search.

Setup Step	Setup Location	More Information
	• Show: All Tasks • Task: Configure Adaptive Search	
5. Grant the Custom Objects Administration (ORA_CRM_EXTN_ROLE) role to the user who will create the user interface pages for the custom object. (All custom top level objects are given access to this role by default.)	Setup and Maintenance work area • Offering: Sales • Functional Area: Users and Security • Task: Manage HCM Role Provisioning Rules	*Enable Sales Administrators to Test Configurations in the Sandbox*
6. Create your project and workspace.	Oracle Visual Builder Studio	For instructions about how to create a project and workspace, refer to the Before You Begin chapter.
7. Create a translation bundle.	Oracle Visual Builder Studio	*Create a Translation Bundle, If You Don't Have One Already*

Related Topics

- Create a New Application Using the CX Extension Generator
- Modify an Existing Custom Application Using the CX Extension Generator

Create a New Application Using the CX Extension Generator

The CX Extension Generator is your shortcut to creating applications that extend the functionality of Oracle Sales in the Redwood User Experience. With just a few quick selections, the CX Extension Generator can create an application extension with panels, subviews, details pages, and smart actions, that you can download as a single .zip file and then upload to Oracle Visual Builder Studio. After you upload the files to Visual Builder Studio, you can continue to build out the extension in Visual Builder Studio and then publish it to your users.

Using CX Extension Generator you can add panels and subviews for child, 1:M (one-to-many), and M:M (many-to-many) relationships. CX Extension Generator also creates the Details (edit) pages for each object and the required smart actions that make it possible for users to create and edit individual records.

Prerequisites

- See *Prerequisites for Using the CX Extension Generator*.
- If you're following along with the examples in this chapter, then create these objects and relationships, as well:
  - Objects
    - Payment (top-level object)
    - Shipment (top-level object)
  - Relationships

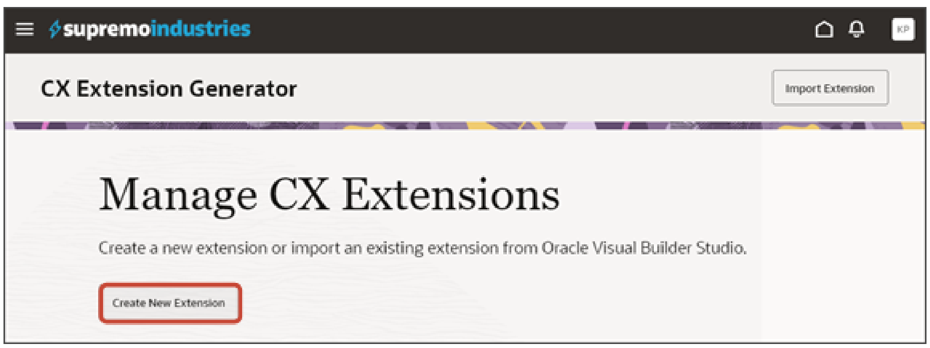
- PaymentLead1M (one-to-many relationship)
- PaymentShipment1M (one-to-many relationship)
- PaymentContactMM (many-to-many relationship)

Create a New Application

To create an application:

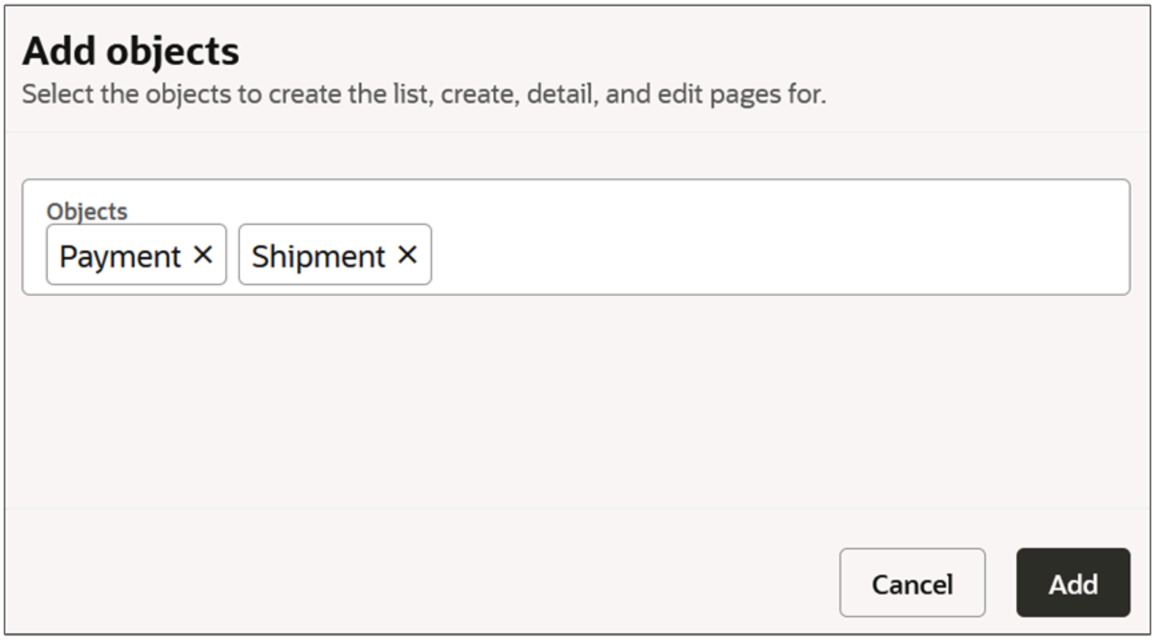
1. In a sandbox, navigate to **Application Composer** > **CX Extension Generator**.
2. Click **Create New Extension**.

> **CAUTION:** You can use the **Create New Extension** button only the first time you configure your application in the environment. If you use the CX Extension Generator to make further changes after your initial upload to VBS, then you must import the files back from VBS using the **Import Extension** button before you start. If you create a new extension using the **Create New Extension** button and import your changes to VBS, then your upload will overwrite all your previous changes.

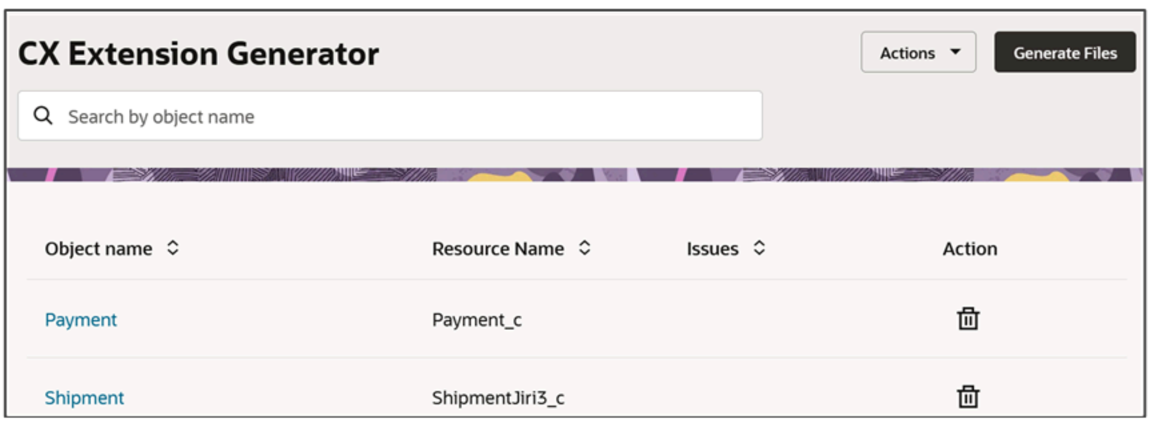


3. In the Add objects drawer, select the objects you're using to create the application, and then click **Add**.

   In this example, select **Payment** and **Shipment**.



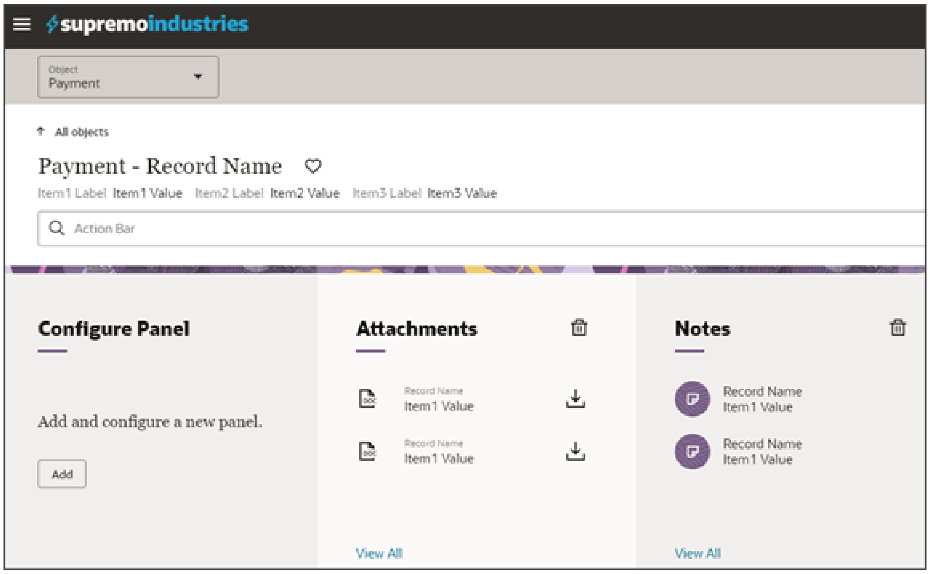
The selected objects display on the list page.



4. Drill down on each object to configure the detail page.

   > **Note:** In the runtime application, the detail page is called the Overview page.

   The page displays automatically-generated panels for attachments and notes. You can optionally delete them.

   Use the default Configure Panel, which always displays as the first panel, to add new panels.

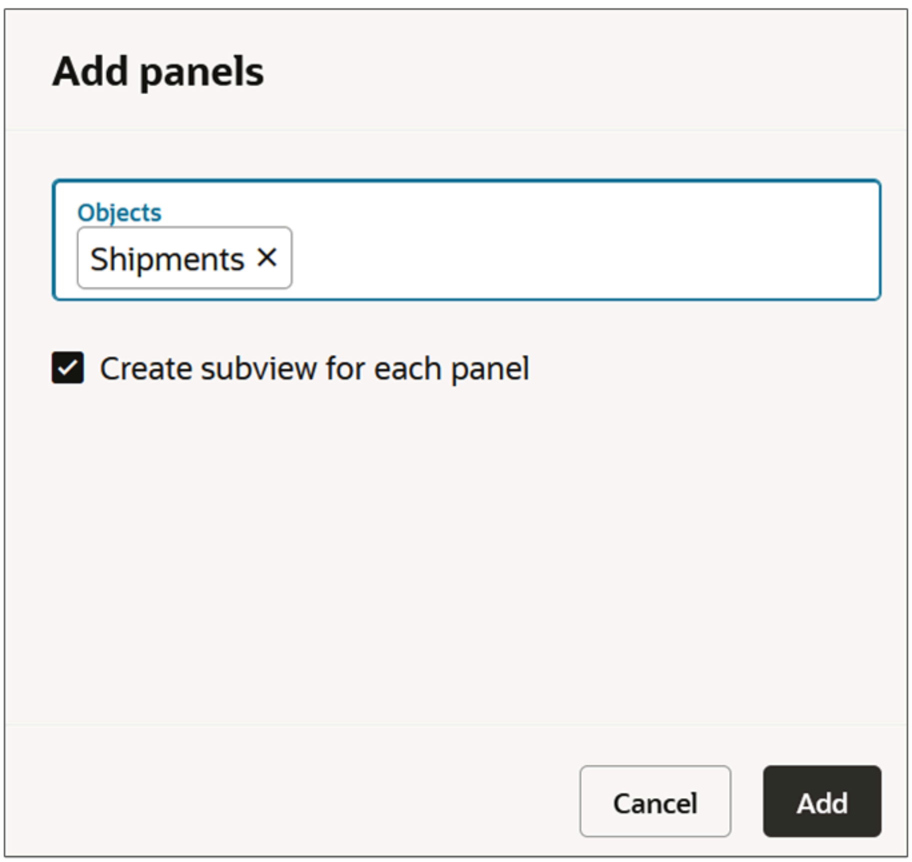
   > **Note:** When you add a panel for an object with a M:M relationship, the generator creates the panel for the intersection object you created as part of the M:M relationship rather than the object itself.

5. To add a panel:

   a. On the default Configure Panel, click **Add**.

   b. In the Add Panels drawer, select the custom related objects that you want to create the panels for. These can be objects with either a 1:M or M:M relationship.

   For example, select **Shipments**.

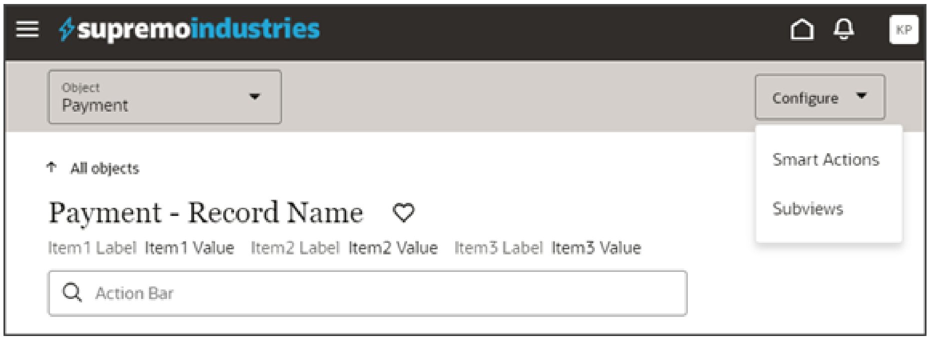


   c. Select the **Create subview for each panel** checkbox to automatically create a subview along with each panel.

> **Note:** If you don't select this checkbox, then you can add subviews later. See the next step.

d. Click **Add**.

6. Optionally, click **Configure** > **Subviews** to add and remove subviews for each panel.



7. Click **Configure** > **Smart Actions** to review the smart actions that the Extension Generator will automatically create for the objects that you selected.

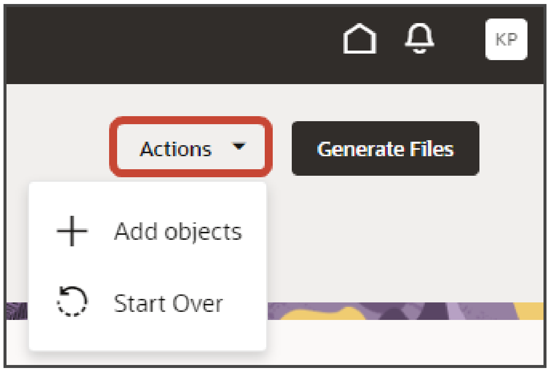
> **Tip:** You can optionally enhance a smart action's configuration after the Extension Generator creates them. You do this by editing the smart action in Application Composer (**Common Setup** > **Smart Actions**).

> **Note:** If you previously created custom smart actions for a non-fragments implementation of an object, then you don't need to create new smart actions for use with fragments. Instead, update existing UI-based custom smart actions to specify the action type, either **Add** or **Create**, as well as the target object and any required field mapping. For existing REST-based or object function-based custom smart actions, edit the action and then save without making any changes. These steps ensure that your custom smart actions still work with new fragment-based extensions.

8. If your application includes more than one object, then use the **Object** drop-down button to switch between objects to configure multiple detail pages.

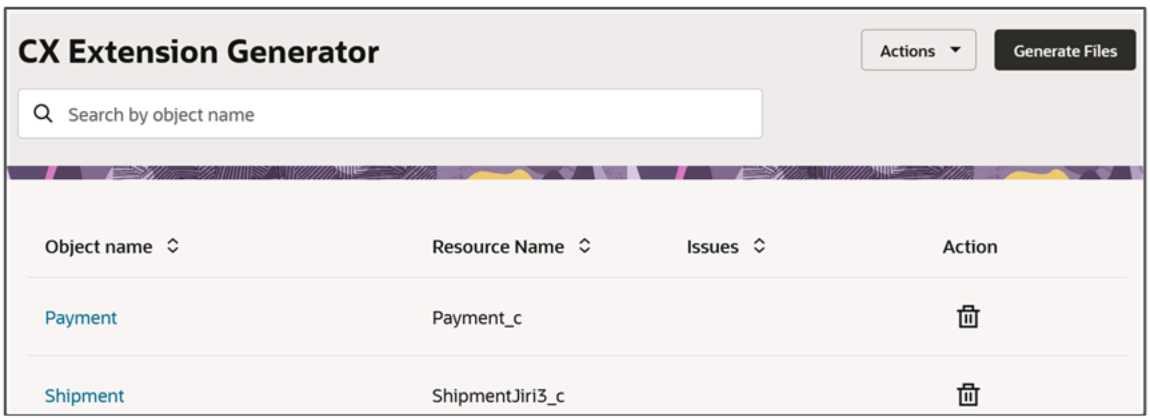
9. After you've completed your changes, you can generate and download the .zip file.

> **Note:** At any time, you can delete your configuration choices from the tool by clicking **Actions** > **Start Over**.



Generate and Download Files

When you're done with your application extension changes, navigate back to the CX Extension Generator list page and click **Generate Files**.



The CX Extension Generator generates and downloads a .zip file that includes the pages and layouts for your selected objects.

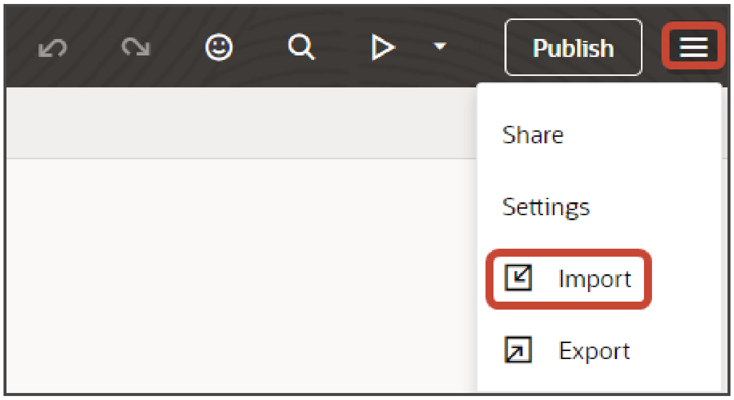
In addition, the process to create the smart actions is launched.

> **Note:** The process of creating smart actions might take some time to complete. After smart actions are created, you can manage them in Application Composer and create additional smart actions, if required.

Import the Files into Visual Builder Studio

1. Use the Navigator to navigate to Visual Builder Studio: **Configuration** > **Visual Builder**.

2. Click the Menu icon at the top of the page, then click **Import**.



3. In the Import Resources dialog, add your .zip file and click **Import**.
4. Click the Preview button to see your newly created application.

5. The resulting preview link will be:

```
https://<servername>/fscmUI/redwood/cx-custom/<object_name>_c
```

6. Change the preview link as follows:

```
https://<servername>/fscmUI/redwood/cx-custom/application/container/<object_name>_c
```

> **Note:** You must add `/application/container` to the preview link.

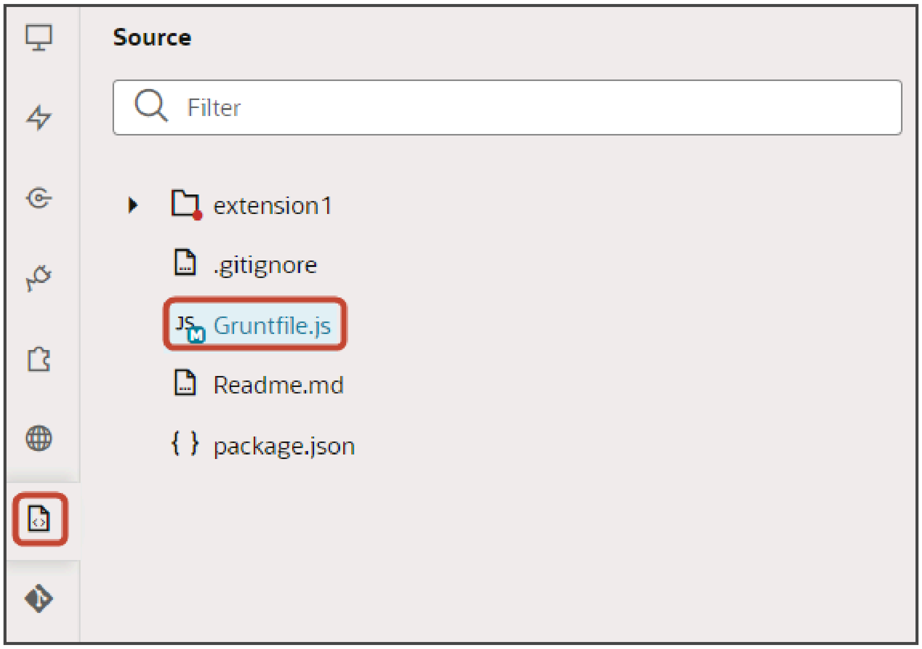
7. You can now continue to make changes to your application extension in Visual Builder Studio.

   For example, you can modify the fields that display in the detail page's header region, or on a subview or create page. The Extension Generator adds some default fields, but you will most likely want to add and remove fields depending on your business needs.
8. If you need more smart actions, you can create them in Application Composer. For example, if you keep the Notes panel, then you must create a Create Note smart action.

Modify the Gruntfile.js File

Once your extension is available in Visual Builder Studio, review the Gruntfile.js and make the following change if the code doesn't match the below sample. You must make this change before publishing your extension.

1. On the Source tab, edit the Gruntfile.js.



2. Replace the existing JavaScript with the following:

```
'use strict';

/**
 * Visual Builder application build script.
 * For details about the application build and Visual Builder-specific grunt tasks
 * provided by the grunt-vb-build npm dependency, please refer to
 * https://www.oracle.com/pls/topic/lookup?ctx=en/cloud/paas/app-builder-cloud&id=visual-application-
build
 */
module.exports = (grunt) => {
 require('load-grunt-tasks')(grunt);
 grunt.initConfig({
 // disable images minification
 "vb-image-minify": {
 options: {
 skip: true,
 },
 },
 // configure requirejs modules bundling
 "vb-require-bundle": {
 options: {
 transpile: false,
 minify: true,
 emptyPaths: [
 "vx/oracle_cx_fragmentsUI/ui/self/resources/js/utils/contextHelper",
 "vx/oracle_cx_fragmentsUI/ui/self/resources/js/utils/actionsHelper",
 "vx/oracle_cx_fragmentsUI/ui/self/resources/js/utils/callbackHelper",
 ],
 },
```

```
         },
         });
        };
```

Create the Row Variable

Create a variable for the detail page. For example:

1. In Visual Builder Studio, click the **App UIs** tab.
2. Expand cx-custom > payment_c, then click the payment_c-detail node.
3. On the payment_c-detail tab, click the **Variables** subtab.
4. Click **+ Variable**.
5. In the Create Variable dialog, make sure the Variable option is selected and, in the ID field, enter `row`.
6. In the Type field, select **Object**.
7. Click **Create**.

Add the Row Variable to the Detail Page

1. On the payment_c-detail tab, click the **Page Designer** subtab.
2. Click the **Code** button.

3. Add the following parameter to the cx-detail code.

```
<oj-vb-fragment-param name="row" value="{{ $page.variables.row }}"></oj-vb-fragment-param>
```

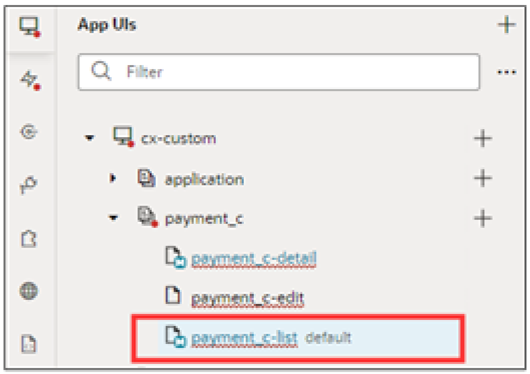
Related Topics

- Overview of Smart Actions

Enable the List Page of Custom Objects Not Yet Enabled for Adaptive Search

Each custom object includes a list page that displays the records for that object. The list page becomes automatically visible in the UI after you enable the custom object for Adaptive Search. If you haven't yet enabled the object for Adaptive Search, then you can make the list page visible by adding a line of code in Visual Builder Studio (VBS).

1. In VBS, select the **App UIs** tab open the custom object in the left pane and click the list page (Payment_c-list in this example).



2. Click **Code** on the list page tab.
3. Add the following line of code before the <oj-vb-fragment> end tag:

```
<oj-vb-fragment-param name="query" value="[[ [{'provider':'adfRest', 'params':[]}] ]]"></oj-vb-fragment-param>
```

Modify an Existing Custom Application Using the CX Extension Generator

Once you have a working application extension in Oracle Visual Builder Studio (VBS), you can use the CX Extension Generator to add additional custom objects and custom panels to custom objects. The CX Extension Generator automatically generates their custom subviews and required smart actions. To use the tool, download your workspace as a .zip file from Visual Builder Studio and then import it into the CX Extension Generator.

The process of adding objects and panels is the same as when you're created the application after you import from Visual Builder Studio. Here's an overview of the steps detailed in this topic:

1. Export the files from Visual Builder Studio.
2. Import the files into CX Extension Generator.
3. Add objects, panels, and subviews.

4. Generate the modified files for export.
5. Import the files back into Visual Builder Studio.
6. Add the panels and subviews to the panel and subview layouts.
7. Preview the updated application.

Prerequisites

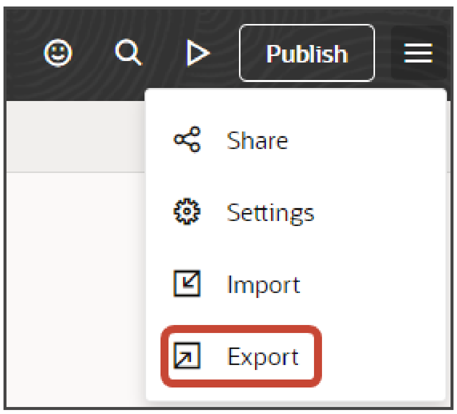
In Application Composer:

- Create the new custom objects and child objects that you want to add to your existing application.

Export Files from Visual Builder Studio

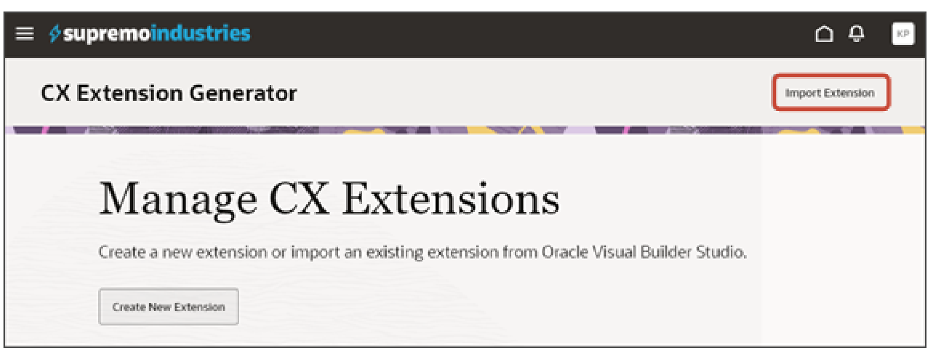
To update an existing custom application, you must first download the application from your Visual Builder Studio workspace.

In Visual Builder Studio, click the **Menu** icon at the top of the page, then click **Export**.



Import the .Zip File into CX Extension Generator

1. In a sandbox, navigate to **Application Composer** > **CX Extension Generator**.
2. Click **Import Extension**.



3. In the Import Application drawer, select your .zip file and click **Import**.

   The existing objects in your application are now visible in the CX Extension Generator.

Add Objects, Panels, and Subviews

Using CX Extension Generator, you can add additional objects, panels, and subviews. And you can generate smart actions for them. To delete existing panels, change their order, and configure them, you must use Visual Builder Studio.

1. To add objects, click **Actions** > **Add Objects** and add any of the objects you want to configure.
2. Drill down on each object to configure the detail page. In the runtime application, the detail page is called the Overview page.

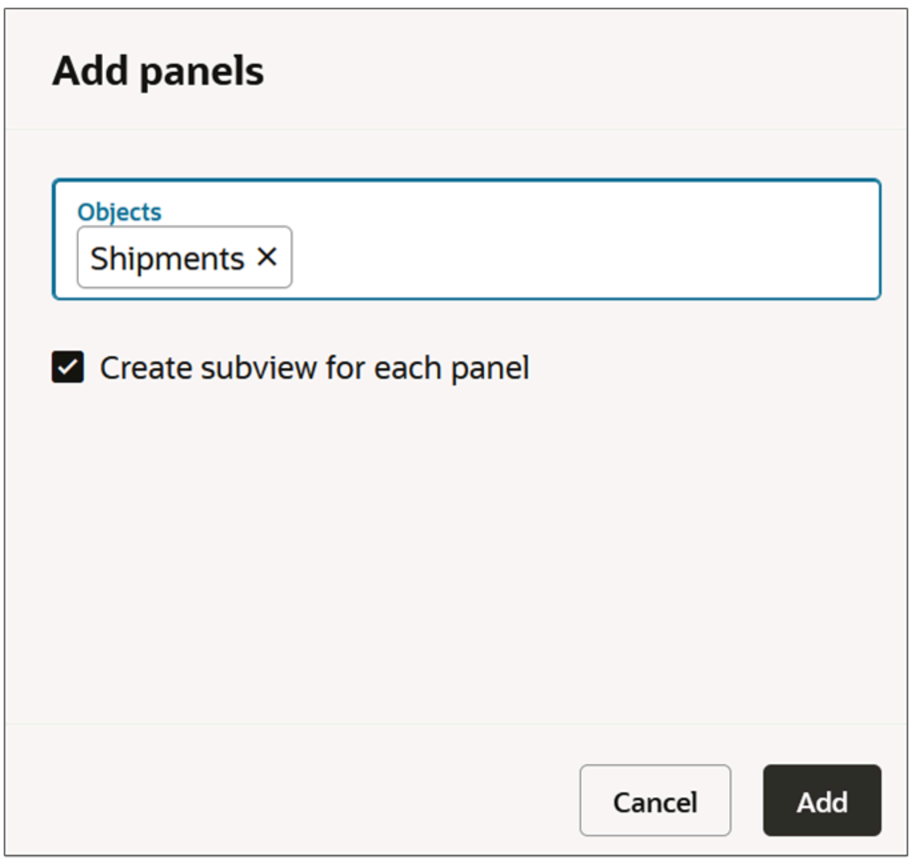
> **Note:** The CX Extension Generator displays only the Configure Panel and the panels you add during your configuration. It doesn't display any of the panels you've added previously.

3. To add a panel:
    a. On the default Configure Panel, click **Add**.

    b. In the Add Panels drawer, select the custom related objects that you want to create the panels for. These can be child objects and objects with either a 1:M or M:M relationship.

    For example, select **Shipments**.

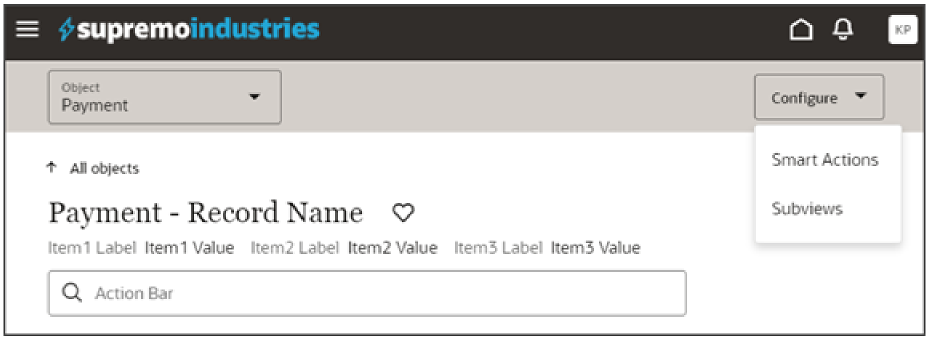
    c. Select the **Create subview for each panel** checkbox to automatically create a subview along with each panel.

    > **Note:** If you don't select this check box, then you can add subviews later. See the next step.

d. Click **Add**.

> **Note:** When you add a panel for an object with a M:M relationship, the generator creates the panel for the intersection object you created as part of the M:M relationship rather than the object itself.

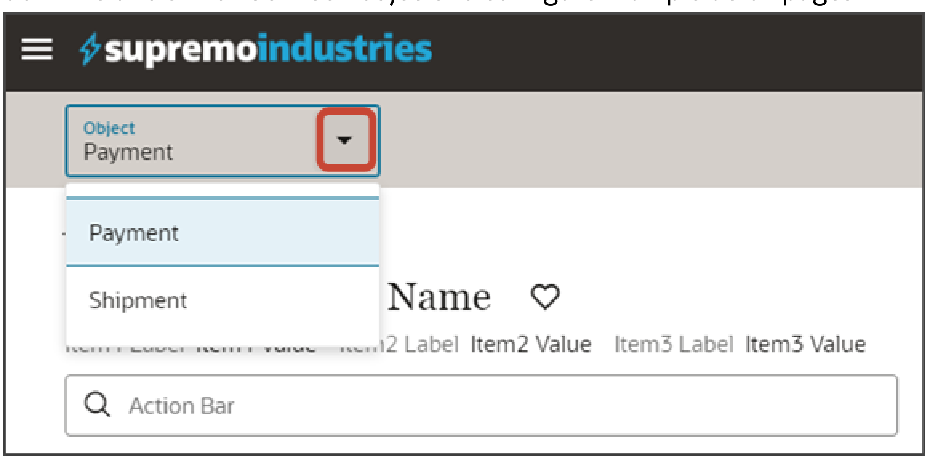
e. Optionally, click **Configure** > **Subviews** to add and remove subviews for each panel.



f. Click **Configure** > **Smart Actions** to review the smart actions that the Extension Generator will automatically create for the objects that you selected.

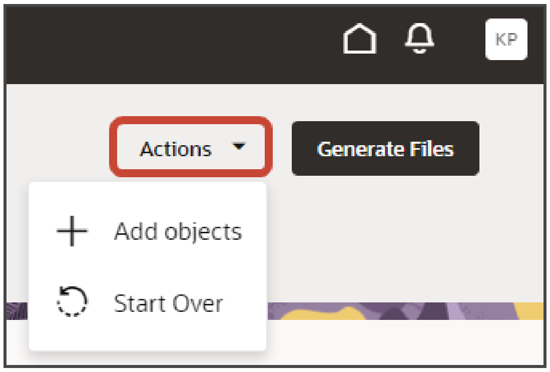
> **Tip:** You can optionally enhance a smart action's configuration after the Extension Generator creates them. You do this by editing the smart action in Application Composer (**Common Setup** > **Smart Actions**). See *Overview of Smart Actions*.

g. If your application includes more than one object, then use the Object drop-down field to switch between objects to configure multiple detail pages.



h. After you've completed your changes, you can generate and download the .zip file.

> **Note:** At any time, you can delete your configuration choices from the tool by clicking **Actions** > **Start Over**.



Generate Files

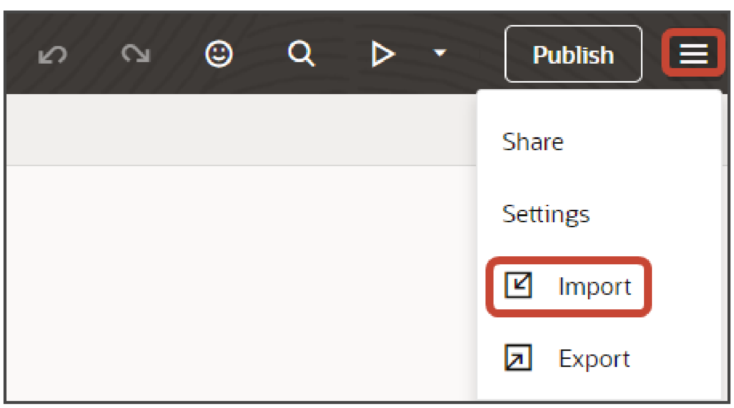
When you're done with your application extension changes, navigate back to the CX Extension Generator list page and click **Generate Files**.

The CX Extension Generator generates and downloads a .zip file that includes the pages and layouts for your selected objects.

Import the Files Back into Visual Builder Studio

The CX Extension Generator generates a .zip file that you can import into Visual Builder Studio.

1. Use the Navigator to navigate to Visual Builder Studio: **Configuration** > **Visual Builder**.
2. In Visual Builder Studio, navigate to the workspace that contains your existing application.
3. Click the Menu icon at the top of the page, then click **Import**.



4. In the Import Resources dialog, add your .zip file and click **Import**.

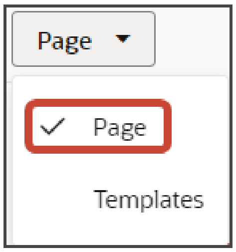
   Your workspace is updated with the newly added objects and related artifacts, without disturbing the existing objects in the application.

For Custom Objects You Imported, Add the Panels and Subviews to the Layout

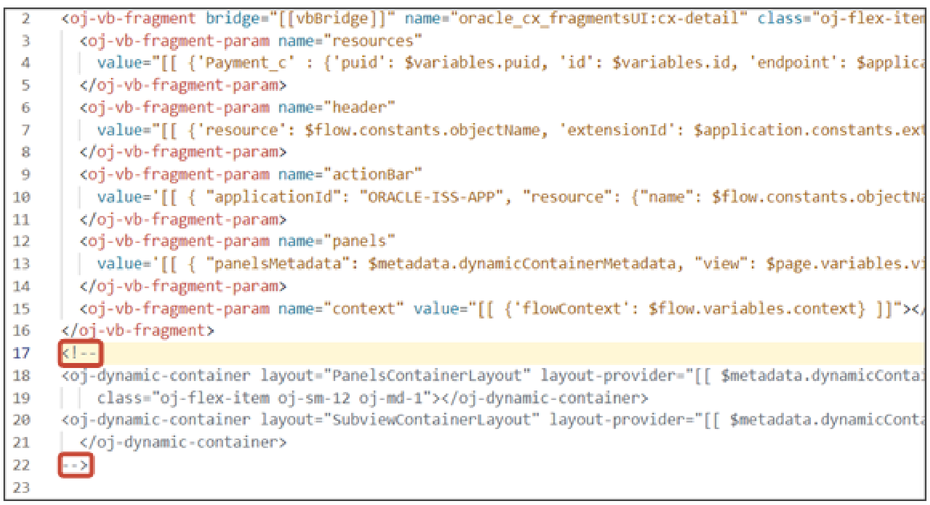
If you imported custom objects, then you must add the imported panels and subviews to the custom Panel Container and Subview Container layouts. This step isn't required for new customer objects you create.

1. Click the **Design** button.

2. Confirm that you're viewing the page in Page Designer.



3. Remove the comment tags for the dynamic container components that contains the panels and any subviews.

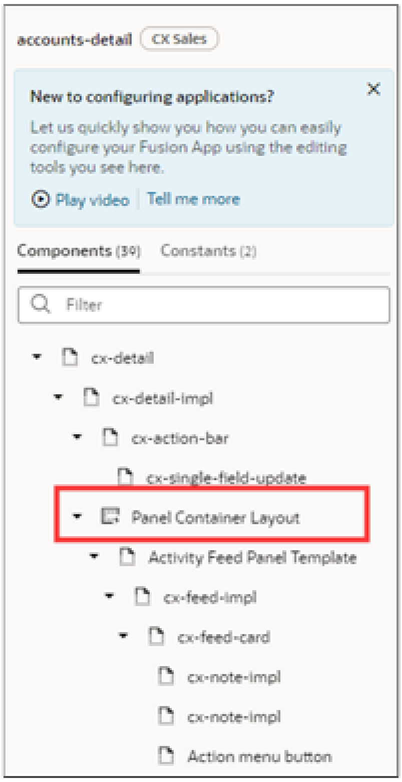


```
<oj-vb-fragment bridge="[[vbBridge]]" name="oracle_cx_fragmentsUI:cx-detail" class="oj-flex-iter
  <oj-vb-fragment-param name="resources"
    value="[[ {'Payment_c' : {'puid': $variables.puid, 'id': $variables.id, 'endpoint': $applica
  </oj-vb-fragment-param>
  <oj-vb-fragment-param name="header"
    value="[[ {'resource': $flow.constants.objectName, 'extensionId': $application.constants.ext
  </oj-vb-fragment-param>
  <oj-vb-fragment-param name="actionBar"
    value='[[ { "applicationId": "ORACLE-ISS-APP", "resource": {"name": $flow.constants.objectNa
  </oj-vb-fragment-param>
  <oj-vb-fragment-param name="panels"
    value='[[ { "panelsMetadata": $metadata.dynamicContainerMetadata, "view": $page.variables.vi
  </oj-vb-fragment-param>
  <oj-vb-fragment-param name="context" value="[[ {'flowContext': $flow.variables.context} ]]"></
</oj-vb-fragment>
<!--
<oj-dynamic-container layout="PanelsContainerLayout" layout-provider="[[ $metadata.dynamicContai
    class="oj-flex-item oj-sm-12 oj-md-1"></oj-dynamic-container>
<oj-dynamic-container layout="SubviewContainerLayout" layout-provider="[[ $metadata.dynamicConta
  </oj-dynamic-container>
-->
```

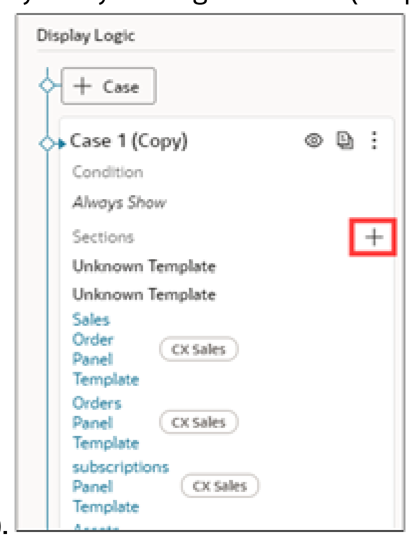
4. Highlight the `<oj-dynamic-container>` tags for the panels.

```
<oj-dynamic-container layout="PanelsContainerLayout" layout-provider="[[ $metadata.dynamicContai
    class="oj-flex-item oj-sm-12 oj-md-1"></oj-dynamic-container>
```

5. Add the panels you imported to the **Panel Container Layout**:
    a. On the Structure panel, click the **Panel Container Layout** node.

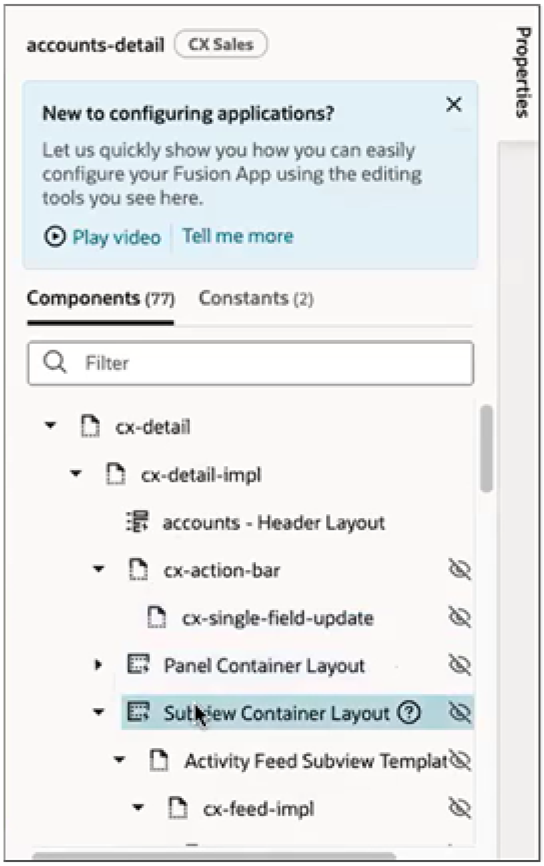


    b. On the Properties pane, add the imported panels to the custom layout by clicking **Add Panel** (the plus



icon next to the Sections heading highlighted in the screenshot).

6. Now repeat the same process for the **Subview Container Layout**.



   a. On the Structure panel, click the **Subview Container Layout** node.
   b. On the Properties pane, add the subview layouts to the custom subview layout, by clicking **Add Panel** (the plus icon next to the Sections heading).

Preview

1. Click the **Preview** button to see the newly added objects in the application.

2. The resulting preview link will be:

```
https://<servername>/fscmUI/redwood/cx-custom/<object_name>_c
```

3. Change the preview link as follows:

```
https://<servername>/fscmUI/redwood/cx-custom/application/container/<object_name>_c
```

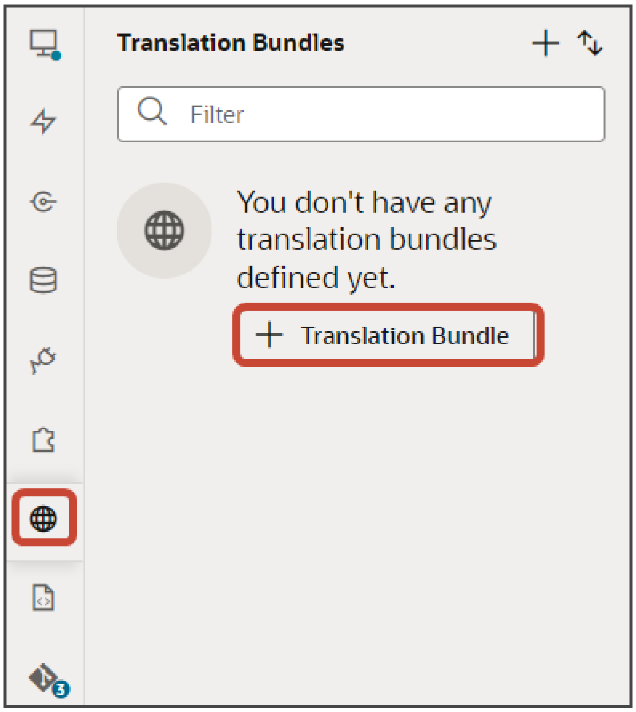
   Note: You must add `/application/container` to the preview link.

4. You can now continue to make changes to your application extension, if needed.
5.

Create a Translation Bundle, If You Don't Have One Already

Create a translation bundle where you can later store custom application strings for translation. If you plan to follow the examples in this chapter, then create the translation bundle and string as indicated below.

1. In Oracle Visual Builder Studio, click the **Translation Bundles side tab** > **+ Translation Bundle**.



2. In the Create Bundle dialog, in the Bundle Name field, enter the name of your translation bundle.

   For example, enter `customBundle`.
3. Click **Create**.
4. Add any required strings to the translation bundle. For example, in the examples in this chapter, you'll use a `Contacts` string.
   a. On the CustomBundle tab, click **+ String**.
   b. In the Key field, enter `contacts`.
   c. In the String field, enter `Contacts`.
   d. Click **Create**.
5. Also add a string for `Contact Name`.
   a. On the CustomBundle tab, click **+ String**.
   b. In the Key field, enter `contactName`.
   c. In the String field, enter `Contact Name`.
   d. Click **Create**.

Configure a Child Object

Add a panel for a child object to the parent and configure the subview for the child object.

Configure the Panel for the Child Object

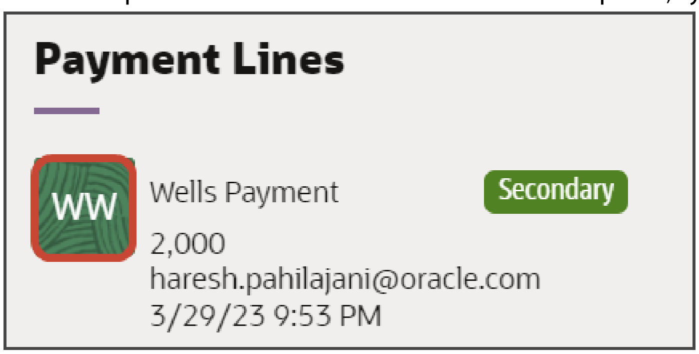
Configure the panel for the child object using the **cx-panel** fragment.

To configure the panel region:

1. In Visual Builder Studio, click the App UIs tab.
2. Expand cx-custom > payment_c, then click the payment_c-detail node.
3. On the payment_c-detail tab, click the Page Designer subtab.
4. Click the Code button.

1. Before we update the rule set and layout that you just created, let's create a custom field that will be used on the panel itself.
   The panels on the detail page use the `avatar-card` style. This means that the panel will have an avatar, which is a visual representation of one of the fields on the panel, typically a name.

   In this step, let's create a custom field, `Avatar`, as well as the `avatarFieldTemplate` field template, which defines the style of the avatar.
   Create the custom field first.
   a. On the PaymentLineCollection_c tab, click the Fields subtab.
   b. Click **+ Custom Field**.
   c. In the Create Field dialog, in the Label field, enter `Avatar`.
   d. In the ID field, the value should be `avatar`.
   e. In the Type field, select **Object (Virtual Field)**.
   f. Click **Create**.
   g. On the Properties pane for the new virtual field, next to the Fields section, click **Add**.
   h. Select **RecordName**.

Pick any field from which initials can be derived. For example, we can use the RecordName field for custom objects. For the Account object, you would select the PartyName field.

i. Click **Add**.

Create the field template next.

a. On the PaymentLinesCollection_c tab, click the Templates subtab.
b. Click **+ Template**.
c. In the Create Template dialog, for Type, select the Field option.
d. In the Label field, enter `avatarFieldTemplate`.
e. Click **Create**.

Map the custom field to the field template.

a. On the PaymentLinesCollection_c tab, click the JSON subtab.
b. In the `PanelCardLayout` section, add this code:

```
,
"fieldTemplateMap": {
"avatar": "avatarFieldTemplate"
```

```
}
```

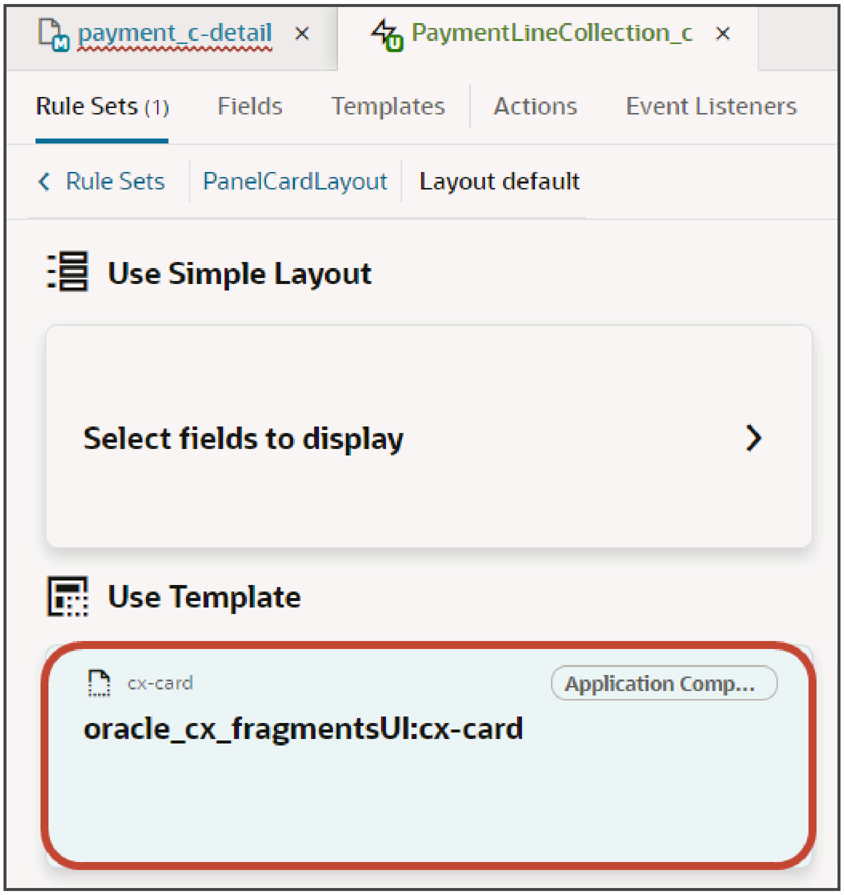
The resulting code will look like this:

```
"PanelCardLayout": {
  "type": "cx-custom",
  "layoutType": "form",
  "label": "PanelCardLayout",
  "rules": [
    "isDefault"
  ],
  "layouts": {
    "default": {
      "layoutType": "form",
      "layout": {
        "displayProperties": [],
        "templateId": "defaultTemplate",
        "labelEdge" : "none"
      },
      "usedIn": [
        "isDefault"
      ]
    }
  },
"fieldTemplateMap": {
    "avatar": "avatarFieldTemplate"
  }
}
```

2. Next, let's add the fields that you want to display on the panel.

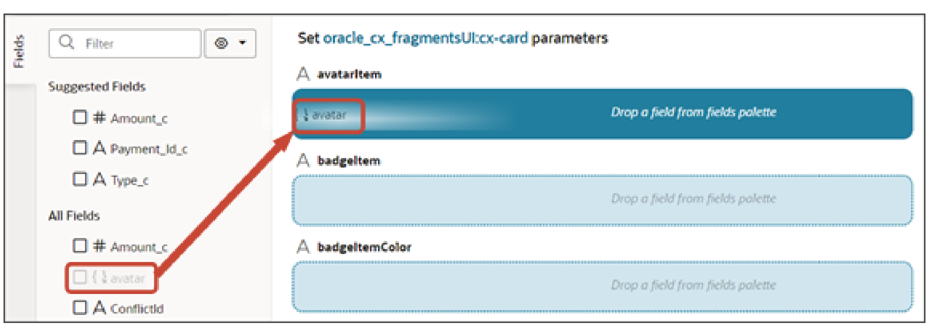
    a. On the PaymentLineCollection_c tab, click the Rule Sets subtab.
    b. Click the Open icon next to the **default** layout.
    c. Click the **cx-card** fragment.

    This fragment provides the format of the panel.



    d. Each panel includes specific slots. From the list of fields, drag each field to the desired slot.

    For example, drag and drop the Avatar field to the avatarItem slot.



    Drag and drop other desired fields to the appropriate slots.

    e. On the Properties pane, click **Go to Template**.

3. On the Templates subtab, click the Code button.

4. Add the following parameters to the template code.

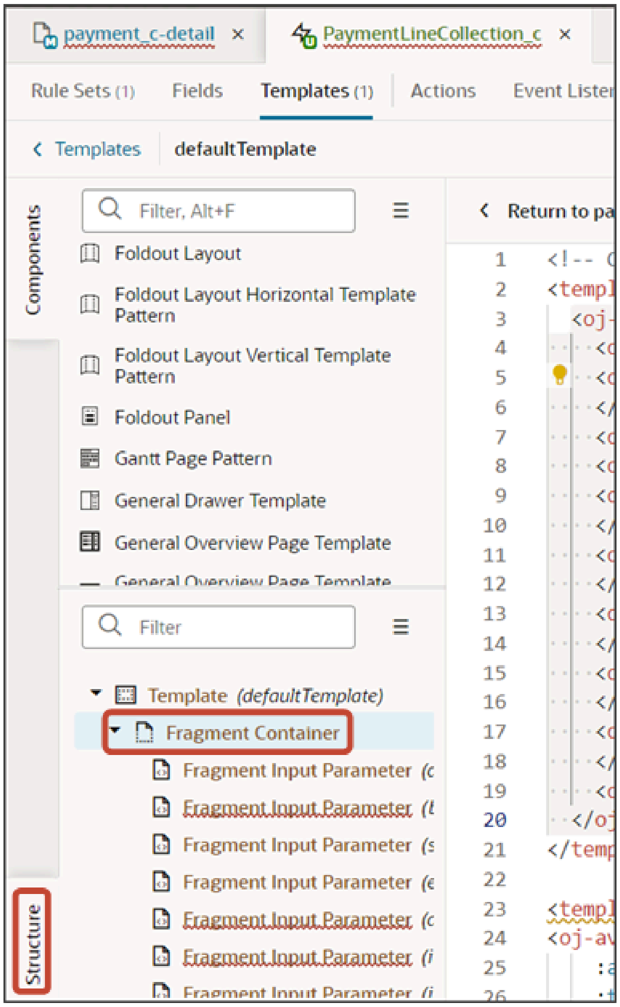
```
<oj-vb-fragment-param name="dynamicLayoutContext" value="{{ $dynamicLayoutContext }}"></oj-vb-fragment-
param>
 <oj-vb-fragment-param name="style" value="avatar-card"></oj-vb-fragment-param>
 <oj-vb-fragment-param name="enableActions" value="false"></oj-vb-fragment-param>
 <oj-vb-fragment-param name="badgeItemColor" value="oj-badge-success"></oj-vb-fragment-param>
```

The template code should look like the below sample. (The below sample also includes style instructions for the avatarFieldTemplate field template.)

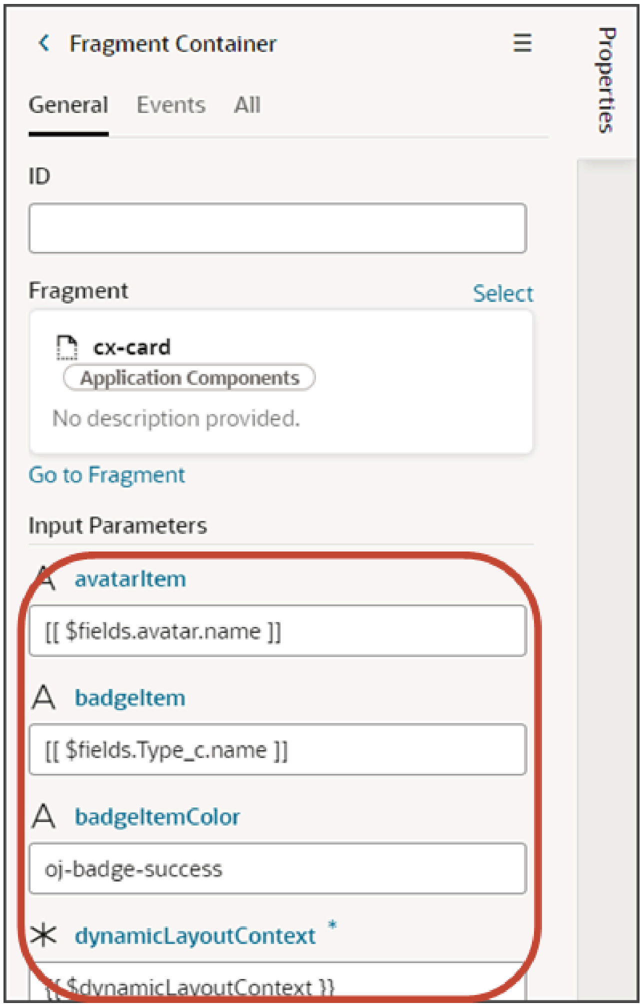
```
<!-- Contains Dynamic UI layout templates -->
<template id="defaultTemplate">
<oj-vb-fragment name="oracle_cx_fragmentsUI:cx-card" bridge="[[ vbBridge ]]">
 <oj-vb-fragment-param name="dynamicLayoutContext" value="{{ $dynamicLayoutContext }}"></oj-vb-fragment-
param>
 <oj-vb-fragment-param name="style" value="avatar-card"></oj-vb-fragment-param>
 <oj-vb-fragment-param name="enableActions" value="false"></oj-vb-fragment-param>
 <oj-vb-fragment-param name="badgeItem" value="[[ $fields.Type_c.name ]]">
 </oj-vb-fragment-param>
 <oj-vb-fragment-param name="avatarItem" value="[[ $fields.avatar.name ]]">
 </oj-vb-fragment-param>
 <oj-vb-fragment-param name="item1" value="[[ $fields.RecordName.name ]]">
 </oj-vb-fragment-param>
 <oj-vb-fragment-param name="item4" value="[[ $fields.CreationDate.name ]]">
 </oj-vb-fragment-param>
 <oj-vb-fragment-param name="item2" value="[[ $fields.Amount_c.name ]]">
 </oj-vb-fragment-param>
 <oj-vb-fragment-param name="item3" value="[[ $fields.CreatedBy.name ]]">
 </oj-vb-fragment-param>
 <oj-vb-fragment-param name="badgeItemColor" value="oj-badge-success"></oj-vb-fragment-param>
 </oj-vb-fragment>
</template>

<template id="avatarFieldTemplate">
<oj-avatar role="img" size="sm" initials="[[ oj.IntlConverterUtils.getInitials($value.RecordName(),
 $value.RecordName()) ]]"
 :aria-label="[['Avatar of ' + $value.RecordName()]]"
 :title="[['Avatar of ' + $value.RecordName()]]" background="green">
 </oj-avatar></template>
```

You can add more fields by returning to the **default** layout and dragging and dropping, as you previously did.

Alternatively, you can add fields to the panel using the Properties pane. To do this, click the Fragment Container node in the Structure pane.



Then, add your desired custom object fields using the Input Parameter fields on the Properties pane.



This table describes some of the parameters that you can provide for a custom panel.

Parameters for Custom Panel

Parameter Name	Description
style	Specify `"avatar-card"` for the panels that you add to the detail page.
enableActions	Panels can have actions when they display on a dashboard. Panels that display on a detail page typically don't need actions, however, so set this parameter to `"false"`.

5. Manually update the template's JSON with the labelEdge property.

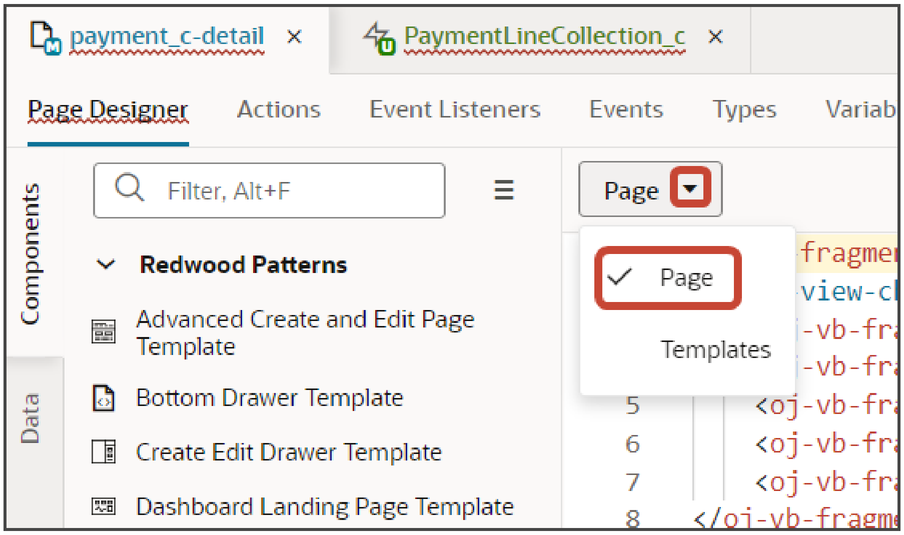
   a. On the PaymentLineCollection_c tab, click the JSON subtab.
   b. In the section for the PanelCardLayout's **default** layout, add the labelEdge property with a value of `"none"`.

      In our example, this is what the labelEdge property should look like:

```
"PanelCardLayout": {
  "type": "cx-custom",
  "layoutType": "form",
  "label": "PanelCardLayout",
  "rules": [
    "isDefault"
  ],
  "layouts": {
    "default": {
      "layoutType": "form",
      "layout": {
        "displayProperties": [],
        "templateId": "defaultTemplate",
        "labelEdge" : "none"
      },
      "usedIn": [
        "isDefault"
      ]
    }
  },
```

6. Finally, comment out the dynamic container component from the payment_c-detail page.

   a. Click the payment_c-detail tab, then click the Page Designer subtab.
   b. Click the Code button.
   c. Select **Page** from the dropdown list.



   d. Comment out the dynamic container component that you added for the panel region.

> **Note:** To add more panels to the panel region, you must first un-comment the dynamic container component so that you can add a new section for each desired panel.

7. Test the panel that you added by previewing your application extension from the payment_c-list page.

    a. From the payment_c-list page, click the Preview button to see your changes in your runtime test environment.

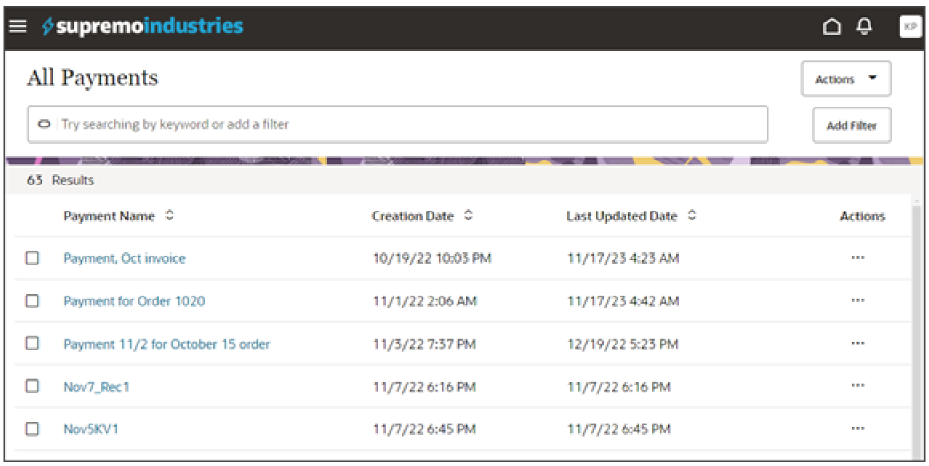
    b. The resulting preview link will be:

    `https://<servername>/fscmUI/redwood/cx-custom/payment_c/payment_c-list`

    c. Change the preview link as follows:

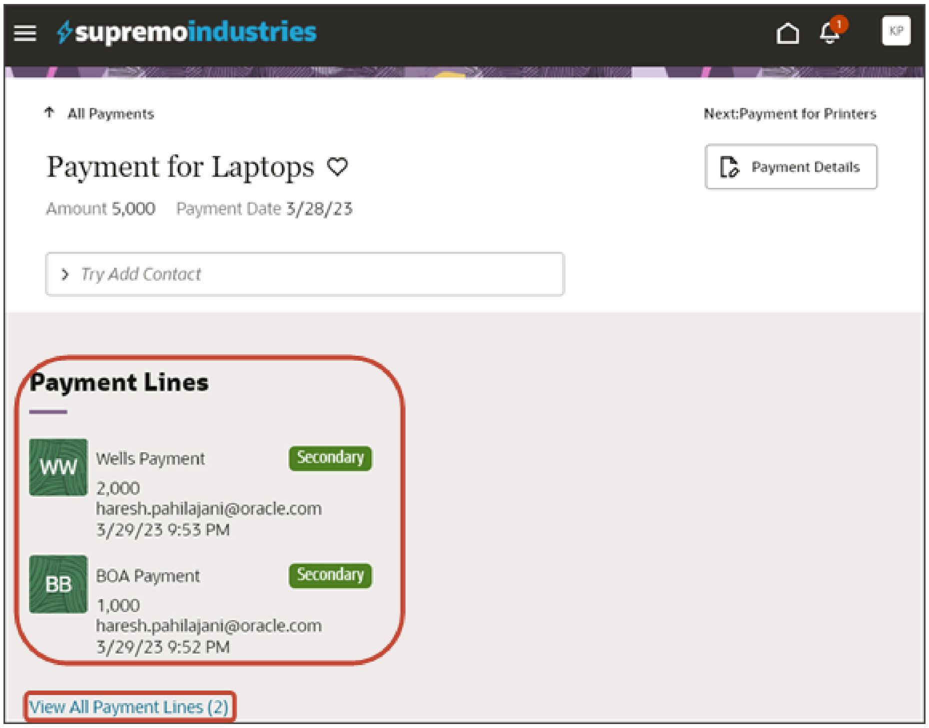
    `https://<servername>/fscmUI/redwood/cx-custom/application/container/payment_c/payment_c-list`

    The screenshot below illustrates what the list page looks like with data.

    d. If data exists, you can click any record on the list page to drill down to the detail page. The detail page, including header region and panels, should display.

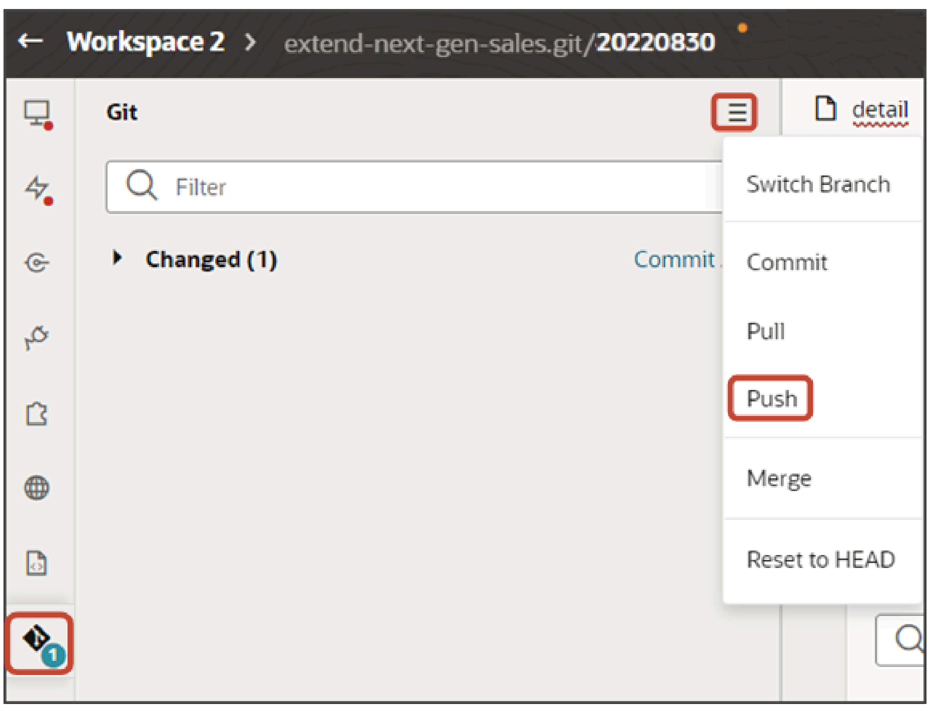
    > **Note:** The screenshot below illustrates what a panel looks like with data. In your testing, the panel might be empty.



The link at the bottom of the panel navigates the user to the subview. Learn how to configure the subview in *Configure the Subview for Child Objects*.

8. Save your work by using the Push Git command.

   Navigate to the Git tab, review your changes, and do a Git push (which does both a commit and a push to the Git repository).



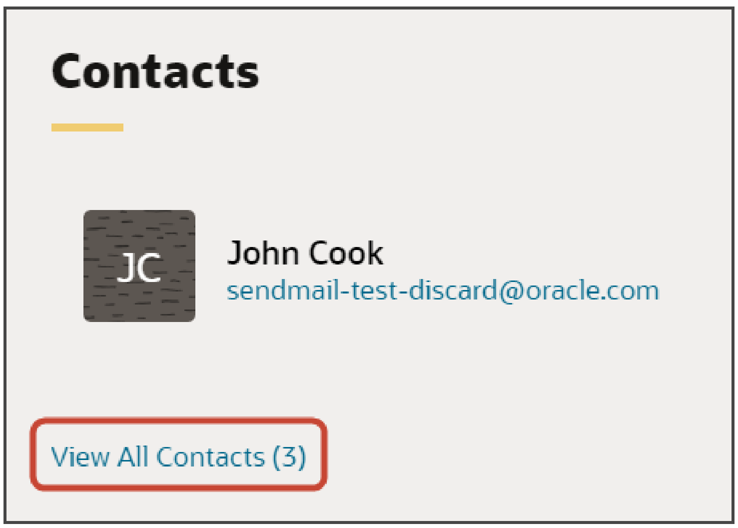
Configure the Subview for Child Objects

You can use fragments in Oracle Visual Builder Studio to create a custom object’s user interface pages: the list page, create page, and detail page with panels. A panel can display only a few records, however, due to limited real estate. To see all records, users must navigate from the panel to a subview. This topic explains how to build the subview using fragments.

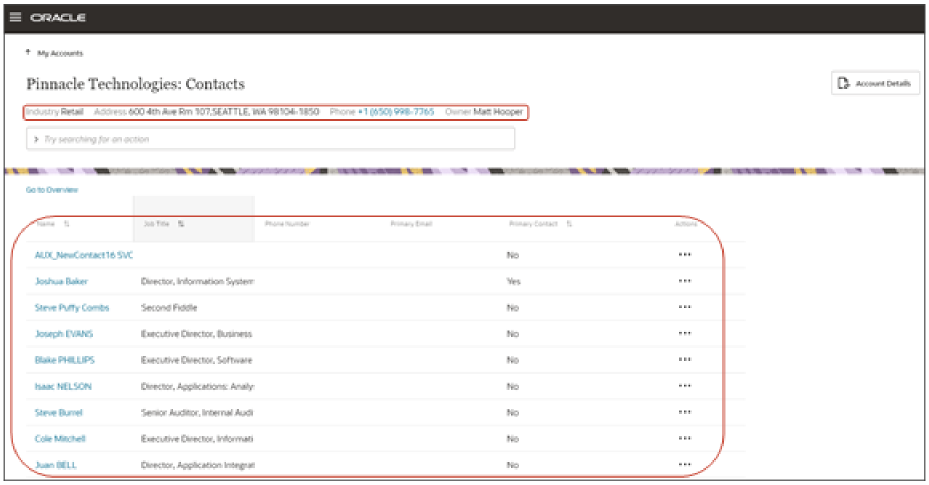
What's a Subview?

Since the real estate of a panel is small, users can click a View All link to navigate to a second page that displays all records.

Here's a screenshot of a View All Contacts link on a panel. Notice that, in this example, the panel itself has room to display only one contact, John Cook, although a total of three records exist. Users can click the View All Contacts link to see all three contacts.



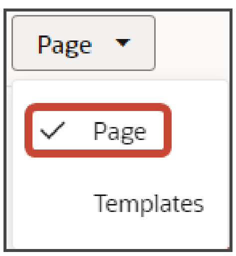
Here's a screenshot of a subview. A subview includes a basic information region at the top and a table. If you create a custom panel for a child or related object, then you must create this page, as well. You can create this page using a fragment.



Create the Payment Lines Subview

Let's create the subview for our payment records. To do this, we'll add a new dynamic container to the detail page in Page Designer.

1. In Visual Builder Studio, click the App UIs tab.
2. Expand cx-custom > payment_c, then click the payment_c-detail node.
3. On the payment_c-detail tab, click the Page Designer subtab.
4. Confirm that you are viewing the page in Page Designer.



5. Click the Code button.

6. In the Filter field, enter `dynamic container`.
7. Drag and drop the dynamic container component to the detail page canvas, outside the previous dynamic container that holds the panels. This dynamic container will hold the subview.
8. In the code for the dynamic container, replace `containerLayout1` with `SubviewContainerLayout`.

9. On the payment_c-detail tab, click the JSON subtab.
10. In the detail page's JSON, rename the two instances of `containerLayout1` to `SubviewContainerLayout`.

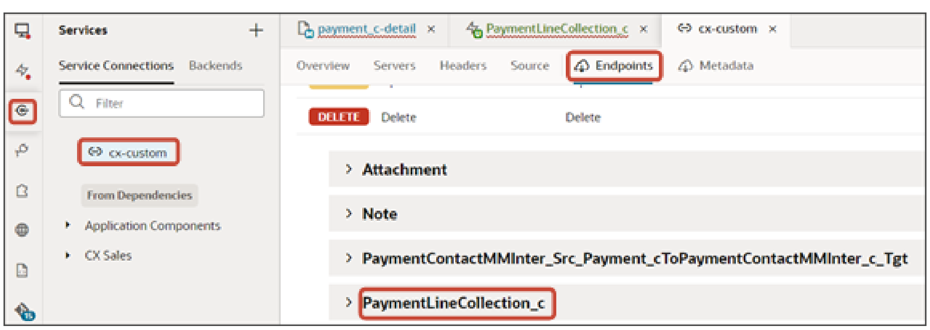
    The two instances appear in the `"layouts"` section and in the `"templates"` section. Here's where the instance appears in the `"templates"` section.

```
"templates": {
  "leads": {
    "title": "Leads",
    "description": "",
    "extensible": "byReference",
    "@dt": {
      "type": "section",
      "layout": "PanelsContainerLayout"
    }
  },
  "paymentLines": {
    "title": "Payment Lines",
    "description": "",
    "extensible": "byReference",
    "@dt": {
      "type": "section",
      "layout": "PanelsContainerLayout"
    }
  },
  "template1": {
    "title": "Default Section",
    "extensible": "byReference",
    "@dt": {
      "type": "section",
      "layout": "containerLayout2"
    }
  }
```

Create the section template that will be used for the subview.

1. On the payment_c-detail tab, click the Page Designer subtab.
2. On the Properties pane, in the Case 1 region, click the Add Section icon, and then click **New Section**.
3. In the Title field, enter a title for the section using the REST child object name, such as `PaymentLineCollection_c`.
4. In the ID field, change the value to `PaymentLineCollection_c`.

> **Note:** You can retrieve the REST child object name from the service connection endpoint.



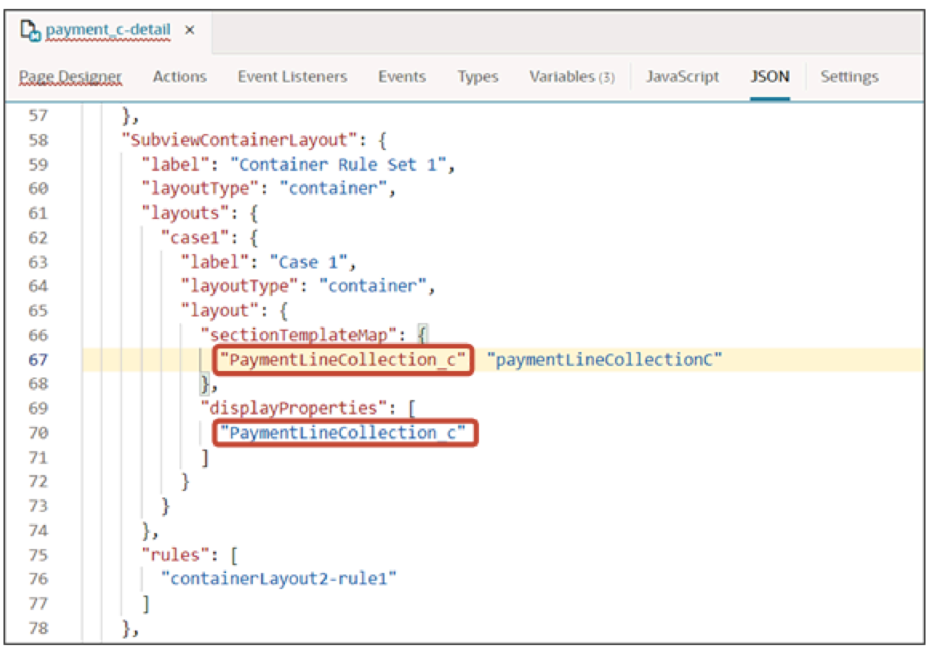
5. Click **OK**.
6. Delete the `Default Section`.

7. Manually update the template's JSON with the correct subview name.

    a. On the Payment_c-detail tab, click the JSON subtab.

    b. In the section for the SubviewContainerLayout's section template layout, replace the `sectionTemplateMap` and `displayProperties` values to match the subview's name, `PaymentLineCollection_c`.

    In our example, this is what the SubviewContainerLayout's `sectionTemplateMap` and `displayProperties` should look like:



Configure the Subview Layout

We previously added the subview dynamic container to the page, as well as the section template.

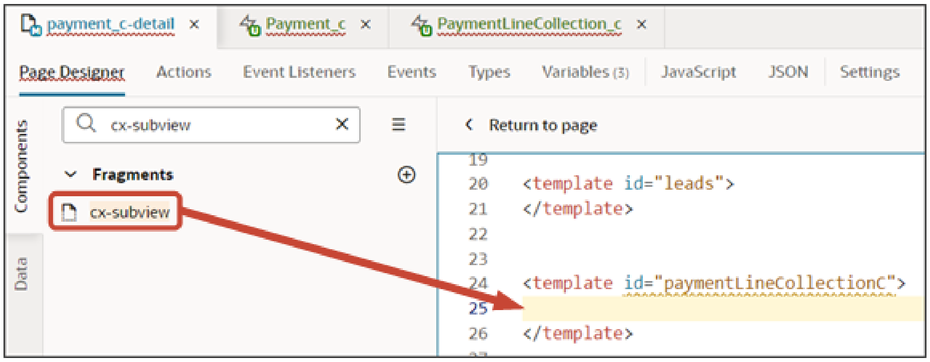
Build the structure of the subview using the **cx-subview** fragment.

1. On the Properties pane, click the PaymentLineCollection_c section that you just added.

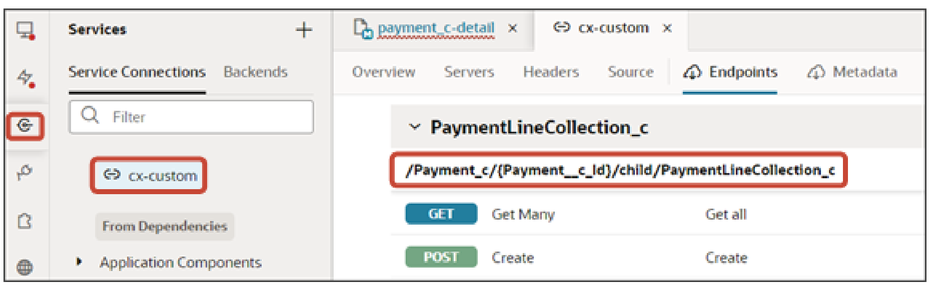
Page Designer navigates you to the template editor, still on the payment_c-detail tab, where you can design the PaymentLineCollection_c template.

2. Click the Code button.

3. On the Components palette, in the Filter field, enter **`cx-subview`**.
4. Drag and drop the cx-subview fragment to the template editor, between the paymentLineCollectionC template tags.



5. Add the following parameters to the fragment code so that the code looks like the below sample. Be sure to replace **`Payment__c_Id`** and **`PaymentLineCollection_c`** with the appropriate values for your custom top level and child objects. Retrieve these values from the service connection endpoint.




```
<template id="paymentLineCollectionC">
 <oj-vb-fragment bridge="[[vbBridge]]" name="oracle_cx_fragmentsUI:cx-subview">
<oj-vb-fragment-param name="resource"
 value='[[ {"name": $flow.constants.objectName, "primaryKey": "Id", "endpoint":
 $application.constants.serviceConnection } ]]'>
 </oj-vb-fragment-param>
 <oj-vb-fragment-param name="sortCriteria" value='[[ [{"attribute": "LastUpdateDate","direction":
 "desc" }] ]]'>
 </oj-vb-fragment-param>
 <oj-vb-fragment-param name="query"
 value='[[ [{"type": "selfLink", "params": [{"key": "Payment__c_Id", "value": $variables.id }]}] ]]'></
oj-vb-fragment-param>
 <oj-vb-fragment-param name="child" value='[[ {"name": "PaymentLineCollection_c", "primaryKey": "Id",
 "relationship": "Child"} ]]'></oj-vb-fragment-param>
 <oj-vb-fragment-param name="context" value="[[ {} ]]"></oj-vb-fragment-param>
 <oj-vb-fragment-param name="extensionId" value="[[ $application.constants.extensionId ]]"></oj-vb-
fragment-param>

 </oj-vb-fragment>
```

```
</template>
```

This table describes the parameters that you can provide for the subview:

Parameters for Subview

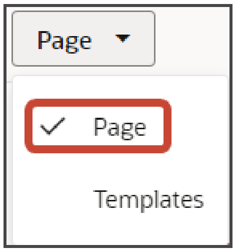
Parameter Name	Description
sortCriteria	Specify how to sort the data on the subview, such as sort by last updated date and descending order.
query	Include criteria for querying the data on the subview.
child	Enter the REST child object name for the child object that the panel is based on.

Configure the subview layout.

1. Click the Layouts tab, then click **PaymentLineCollection_c**.
2. On the PaymentLineCollection_c tab, click **+ Rule Set** to create a new rule set for the layout.
    a. In the Create Rule Set dialog, in the Component field, select **Dynamic Table**.
    b. In the Label field, enter `SubViewLayout`.
    c. In the ID field, change the value to `SubViewLayout`.
    d. Click **Create**.
3. Add the fields that you want to display in the layout.
    a. Click the Open icon next to the **default** layout.
    b. From the list of fields, select the fields that you want to display on the subview table. The fields display as columns in the order that you click them, but you can rearrange them.

4. Create an event so that users can be automatically navigated back to the subview after editing the payment.

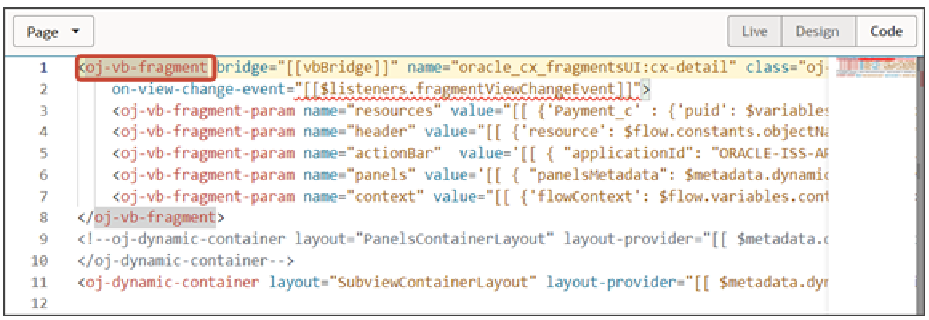
   a. On the payment_c-detail tab, click the Page Designer subtab.
   b. Confirm that you are viewing the page in Page Designer.

   c. Click the Code button.

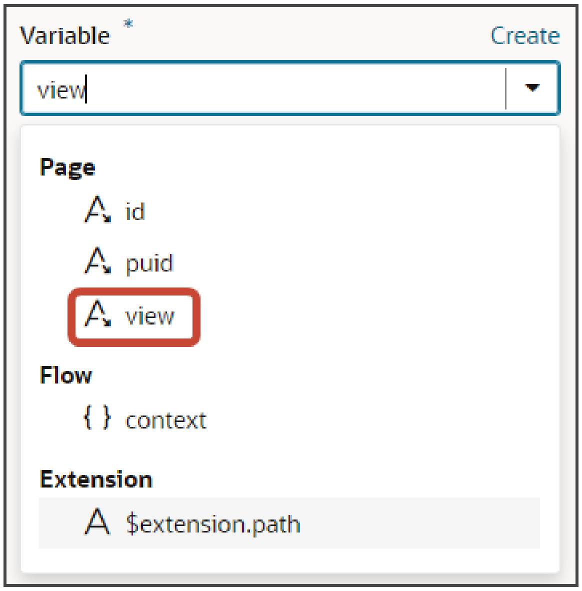
   

   d. In the code for the detail page, click the `oj-vb-fragment` tag.

   e. On the Properties pane for the cx-detail fragment, click the Events subtab.

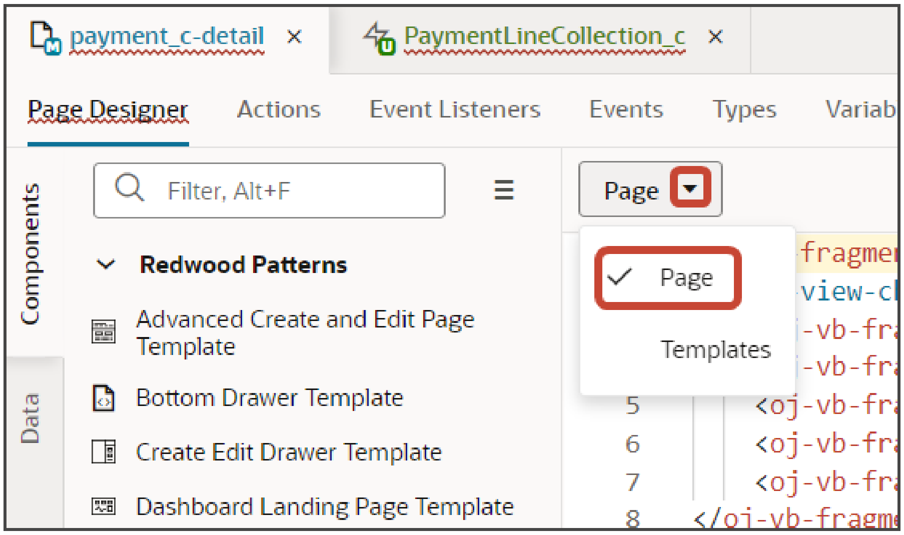
      i. Click **+ New Event** > **On 'viewChangeEvent'**.
      ii. Drag an Assign Variables action onto the canvas.
      iii. On the Properties pane, next to the Variable field, click the Select Variable icon.
      iv. In the Variables popup, under the Page heading, click **view**.



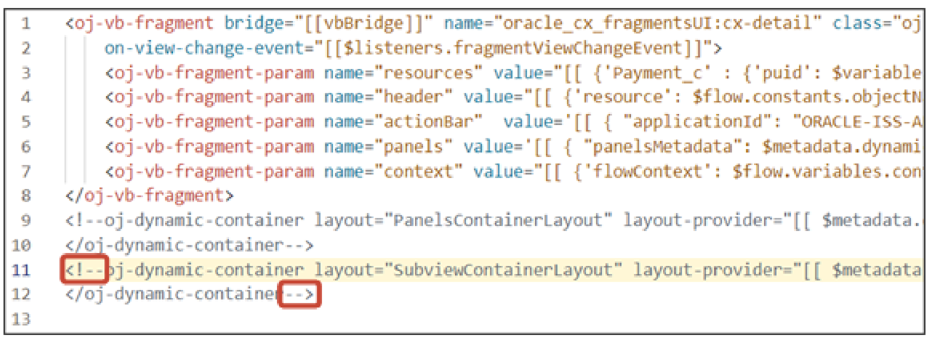
v. In the Value field, enter `{{ payload.view }}`.

5. Comment out the subview's dynamic container component from the payment_c-detail page.

    a. Click the payment_c-detail tab, then click the Page Designer subtab.
    b. Click the Code button.
    c. Select **Page** from the dropdown list.



    d. Comment out the dynamic container component that you added for the subview.



6. Add an Actions menu to the subview.

    To do this, create a custom field, `Actions Menu`.

    a. On the PaymentLineCollection_c tab, click the Fields subtab.
    b. Click **+ Custom Field**.
    c. In the Create Field dialog, in the Label field, enter `Actions Menu`.
    d. In the ID field, the value should be `actionsMenu`.
    e. In the Type field, select **String**.
    f. Click **Create**.

    Map the custom field to the field template.

    a. On the PaymentLinesCollection_c tab, click the JSON subtab.
    b. In the `subViewLayout` section, add this code:

```
,
```

```
"fieldTemplateMap": {
"actionsMenu" : "actionMenuTemplate"
}
```

The resulting code will look like this:

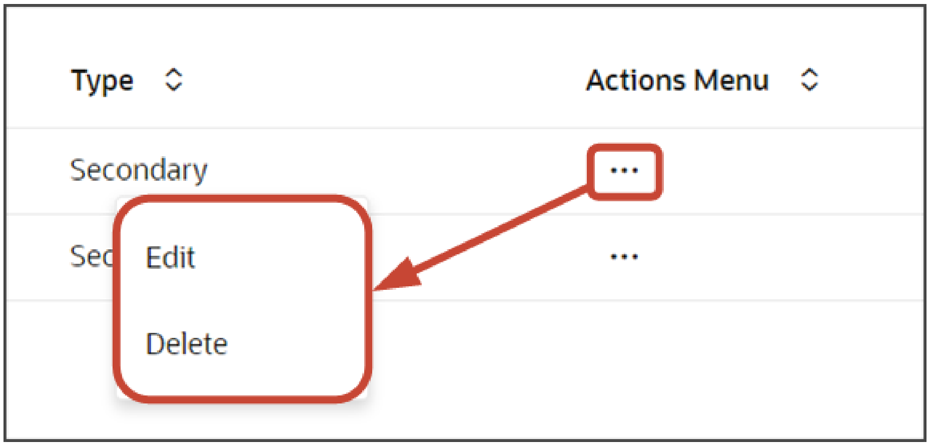
```
"SubViewLayout": {
  "type": "cx-custom",
  "layoutType": "table",
  "label": "SubViewLayout",
  "rules": [
    "isDefault2"
  ],
  "layouts": {
    "default": {
      "layoutType": "table",
      "layout": {
        "displayProperties": [
          "RecordName",
          "Amount_c",
          "Type_c"
        ]
      },
      "usedIn": [
        "isDefault2"
      ]
    }
  },
  "fieldTemplateMap": {
    "actionsMenu" : "actionMenuTemplate"
  }
}
```

Add the custom field to the subview table.

a. Click the PaymentLineCollection_c tab > Rule Sets subtab.
b. Click **SubViewLayout**.
c. Click the Open icon next to the **default** layout.
d. From the list of fields, click the actionsMenu field to add it to the subview table.

7. The Actions menu provides both an Edit and Delete action. Users click **Edit** to edit a payment line.

   Create a layout for the edit payment line page.



   a. On the PaymentLineCollection_c tab, click **< Rule Set** to return to the main Rule Sets page where you can create a new rule set.

      i. Click **+ Rule Set**.
      ii. In the Create Rule Set dialog, in the Component field, select **Dynamic Form**.
      iii. In the Label field, enter `EditLayout`.
      iv. In the ID field, change the value to `EditLayout`.
      v. Click **Create**.

   b. Add the fields that you want to display in the layout.

      i. Click the Open icon next to the **default** layout.
      ii. Click **Select fields to display**.
      iii. From the list of fields, select the fields that you want to display on the edit payment line page. The fields display as columns in the order that you click them, but you can rearrange them.

   c. On the Properties pane for this layout, in the Max Columns field, enter `2`.

      You might need to click **< Form** to see the properties for the layout.

8. Test the subview by previewing your application extension from the payment_c-list page.

   a. From the payment_c-list page, click the Preview button to see your changes in your runtime test environment.

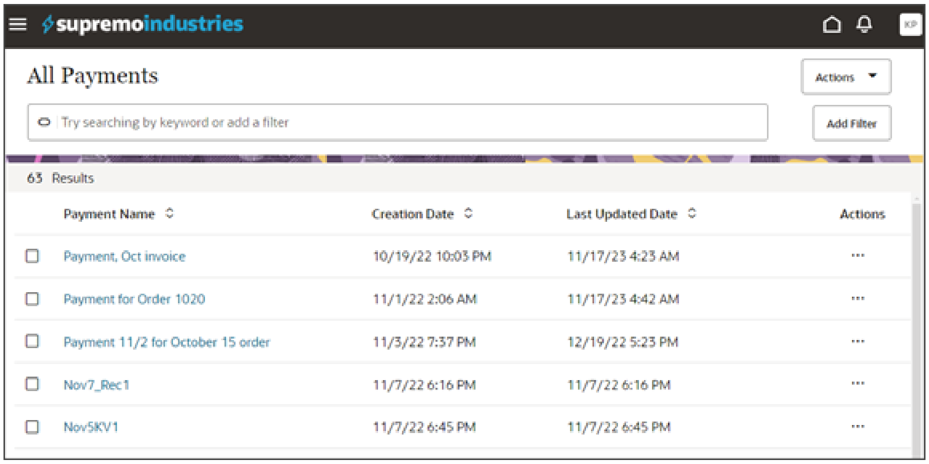
   b. The resulting preview link will be:

   ```
   https://<servername>/fscmUI/redwood/cx-custom/payment_c/payment_c-list
   ```

   c. Change the preview link as follows:

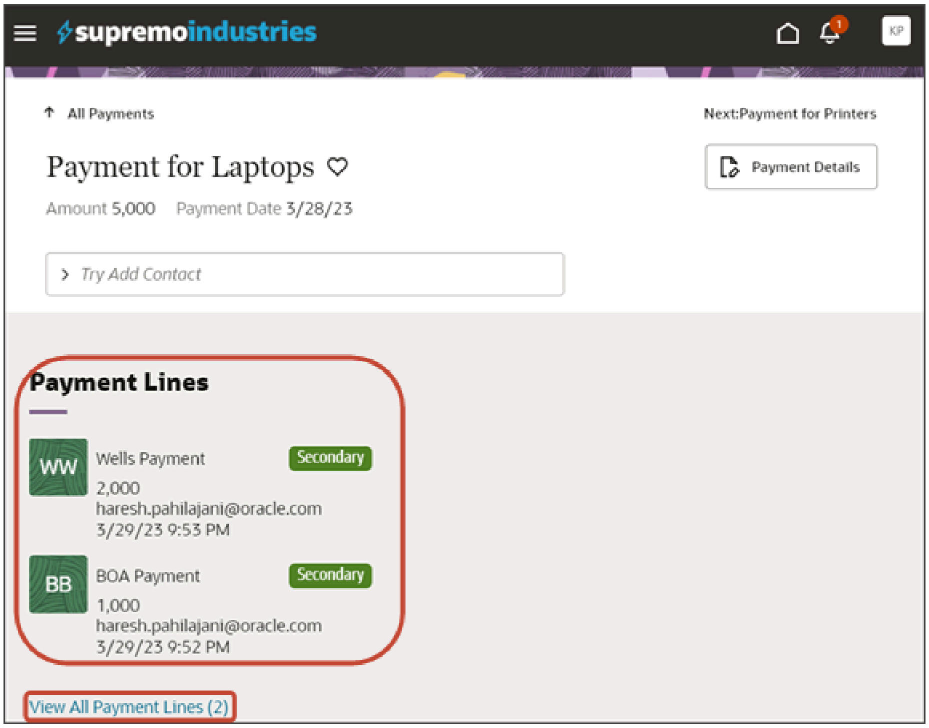
   ```
   https://<servername>/fscmUI/redwood/cx-custom/application/container/payment_c/payment_c-list
   ```

   The screenshot below illustrates what the list page looks like with data.

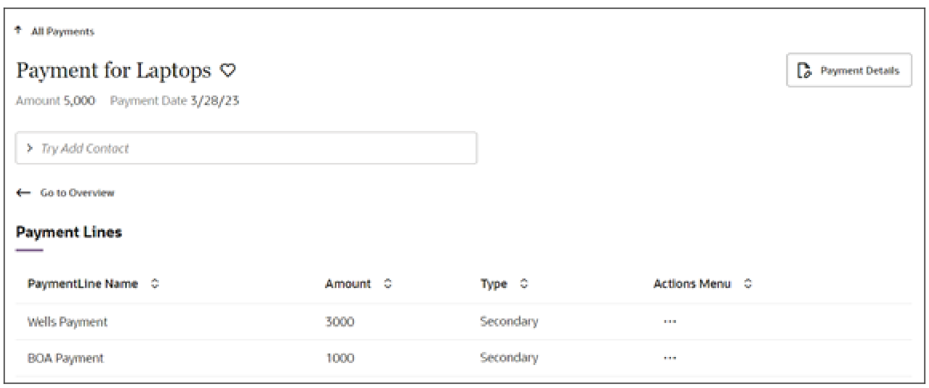
   

   d. If data exists, you can click any record on the list page to drill down to the detail page. The detail page, including header region and panels, should display.

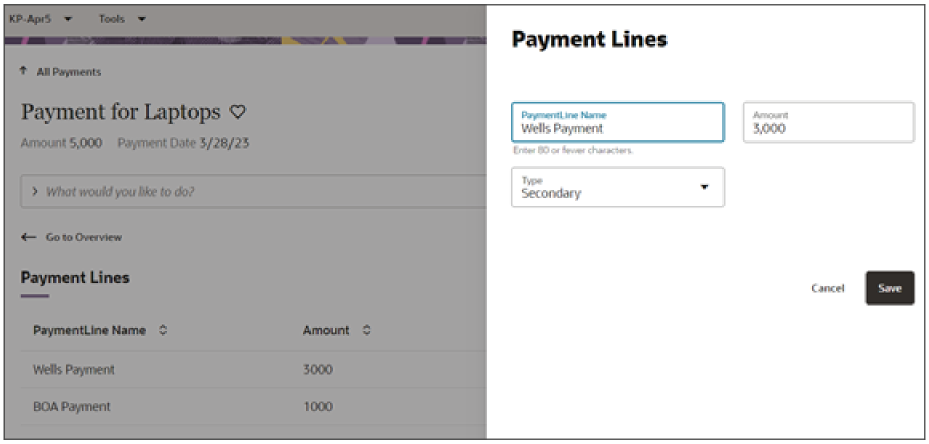
   > **Note:** The screenshot below illustrates what a panel looks like with data. In your testing, the panel might be empty.



e. Click the View All link to drill down to the subview.

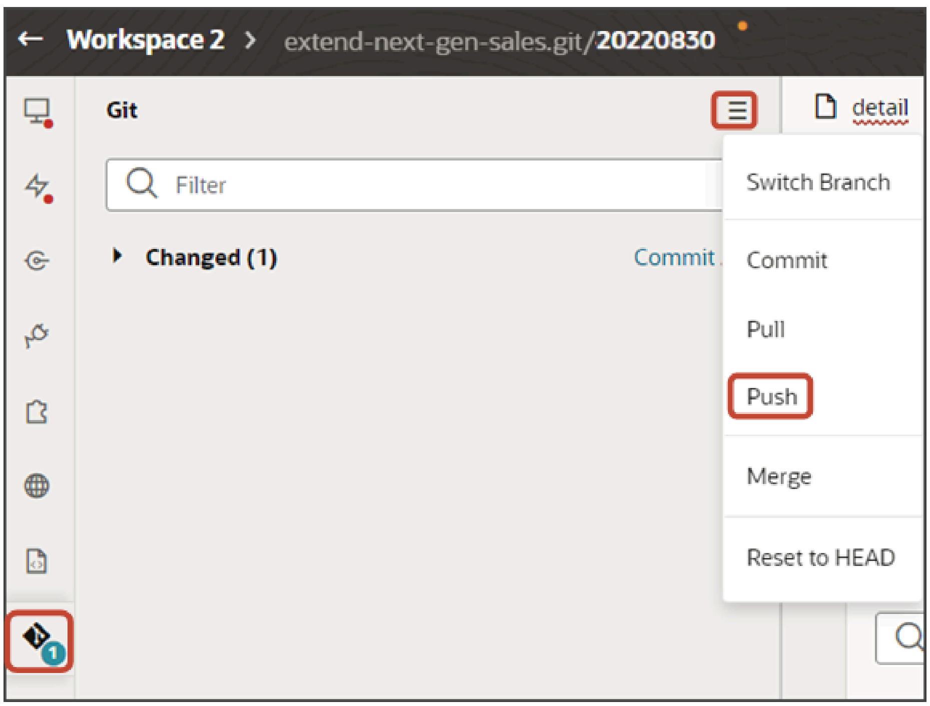


f. Click the Actions Menu > Edit to edit the payment line.



9. Save your work by using the Push Git command.

   Navigate to the Git tab, review your changes, and do a Git push (which does both a commit and a push to the Git repository).

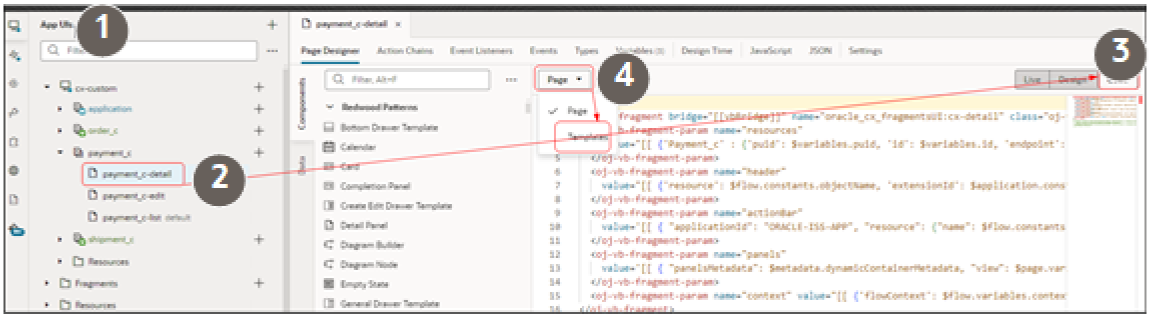


Display a Smart Action on a Child Object Subview Only for a Specific Parent Object

When you create a subview for a child object, you might need to create smart actions that apply specifically to that subview.

Suppose, for example, that you used the CX Extension Generator to display a list of shipments on payments, with Shipments being a child object of Payments. CX Extension Generator automatically creates the Add smart action that users enter in the Action Bar to add shipments to payments, but it doesn't create any smart action to remove the entry. Here are the steps to create the smart action and make it available only on the Payment subview.

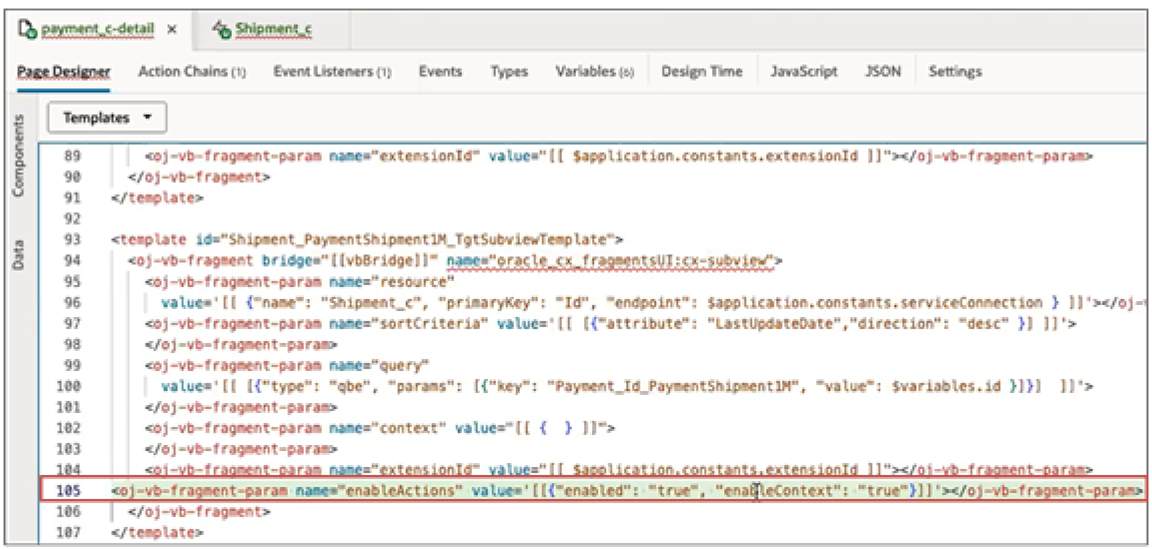
1. Create the Remove smart action as a REST-based smart action as described in the topic *Create REST-Based Smart Actions*. Be sure to enter the name of the parent object in the Context region of the Availability page.
2. In VB Studio open the parent object's detail page and view the Template code (**App UIs** > **Payment_c_detail** > **Code** > **Page** > **Template**



3. Find the template for the subview in the code, and add the following parameter:

```
<oj-vb-fragment-param name="enableActions" value='[[{"enabled": "true", "enableContext": "true"}]]'></
oj-vb-fragment-param>
```

Insert it right before the **</oj-vb-fragment>** tag.



Add a Standard Object Panel for Related Objects (One-to-Many)

You can configure an object's detail page by adding panels for related objects. This makes it easy for users to see – on a single page – all pertinent information related to a record. You can add custom object panels or standard object panels.

This topic illustrates how to add a standard object panel to an object's detail page (when the panel object is related via a one-to-many relationship).

What's the Scenario?

Let's look at an example. In this example, the Payment object has a one-to-many relationship with the Lead object. At runtime, users should be able to create leads for a payment, and view those leads on the Payment detail page.

Setup Overview

To enable users to create leads for a payment, we'll add a Leads panel and subview to the Payment detail page.

1. First, create the Create Lead smart action in Application Composer.

   See *Prerequisite: Create Smart Action*.
2. Add a new Leads panel to the Payment detail page.

   See *Add the Leads Panel to the Payment Detail Page*.
3. After adding the Leads panel, you can then add the subview.

   See *Add a Subview for the Leads Panel*.

Prerequisite: Create Smart Action

The Create Lead smart action displays from the Action Bar on both the Payment detail page and Leads subview. Users can select the Create Lead smart action to navigate to a create lead page.

> **Note:** If you previously created a Create Lead smart action for a non-fragments implementation, then you don't need to create a new smart action for this use case. Instead, update your existing smart action to specify the **Create** action type, object, and field mapping. This ensures that your custom smart action still works with this new fragment-based extension.

If you haven't yet created a Create Lead smart action, then create one now:

1. Create a sandbox.
2. In Application Composer, under the **Common Setup** menu, click **Smart Actions**.
3. At the top of the page, click **Create**.
4. On the Kind of Action page, click **UI-based action** and then click **Continue**.
5. On the Basic Details page, in the Name field, enter the smart action name.

   For example, enter `Create Lead`.
6. In the Object field, select the one-to-many relationship's source object.

   In this case, select **Payment** and then click **Continue**.
7. On the Availability page, in the Application field, select **Sales**.
8. In the UI Availability field, select **List Page** and click **Continue**.
9. On the Action Type page, in the Type field, select **Create**.
10. In the Target Object field, under the Top Level Object heading, select the one-to-many relationship's target object.

    For example, select **Sales Lead**.
11. In the Field Mapping region, click **Add**.

12. In the Actions column, click the Edit icon and then set these field values:

Attribute Defaults

Column	Value
Name	Select the field on the one-to-many relationship's target object that holds the source object's ID and relationship name. This is a standard field on the target object (Sales Lead). The format of the field name is always **<Source object name>_Id_<Relationship name>**. For example, select **Payment ID PaymentLead1M (Payment_Id_PaymentLead1M)**. **Note:** You won't see this field on the target object in Application Composer.
Type	Attribute
Value	Select **Record ID (Id)**. This is a standard field on the source object (Payment). This means that when users create a lead, the **create** smart action defaults the payment's ID into the lead record's **Payment ID PaymentLead1M (Payment_Id_PaymentLead1M)** attribute.

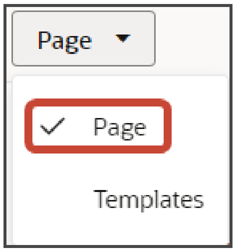
13. Click **Done**.
14. Click **Continue**.
15. On the Action Details page, in the Navigation Target field, select **Local** and then click **Continue**.
16. On the Review and Submit page, click **Submit**.

Add the Leads Panel to the Payment Detail Page

To add a new Leads panel to the Payment detail page:

1. In Visual Builder Studio, click the App UIs tab.
2. Expand cx-custom > payment_c, then click the payment_c-detail node.
3. On the payment_c-detail tab, click the Page Designer subtab.
4. Click the Code button.

5. Confirm that you are viewing the page in Page Designer.

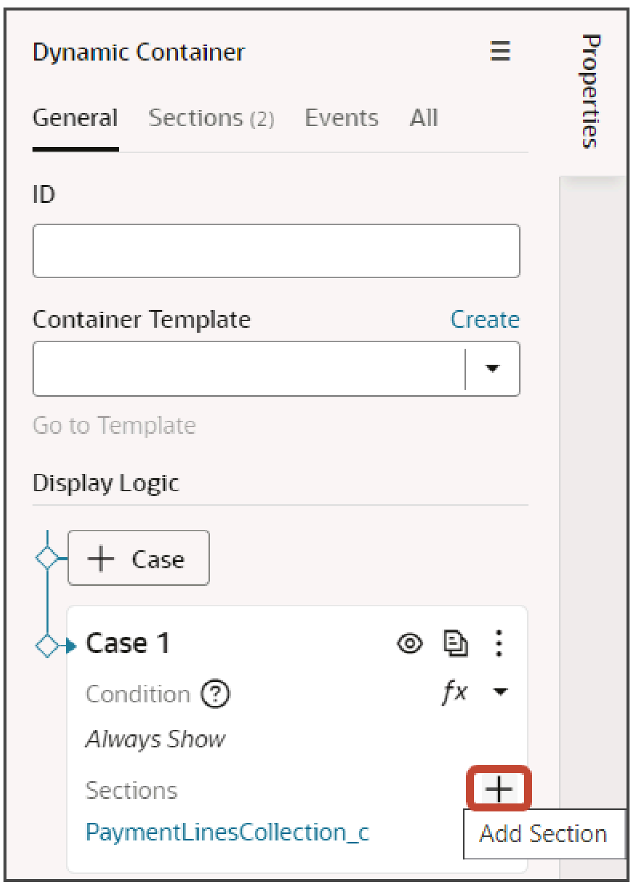


6. Add the following code to the canvas, just below the closing `</oj-vb-fragment>` tag of the **cx-detail** fragment:

```
<oj-dynamic-container layout="PanelsContainerLayout" layout-
provider="[[ $metadata.dynamicContainerMetadata.provider ]]"
 class="oj-flex-item oj-sm-12 oj-md-1"></oj-dynamic-container>
<oj-dynamic-container layout="SubviewContainerLayout" layout-
provider="[[ $metadata.dynamicContainerMetadata.provider ]]">
 </oj-dynamic-container>
```

7. Highlight the `<oj-dynamic-container>` tags for the panels.

8. On the Properties pane, in the Case 1 region, click the **Add Section** icon, and then click **New Section**.



9. In the Title field, enter a title for the section, such as `Leads Panel`.

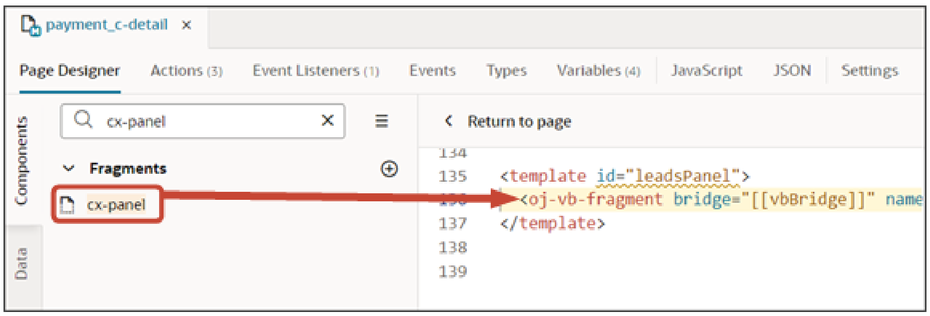
10. In the ID field, keep the value of `leadsPanel`.

> **Note:** Don't use the REST object name for this ID because you'll use the REST object name when you create the subview.

11. Click **OK**.
12. On the Properties pane, click the **Leads Panel** section that you just added.

    Page Designer navigates you to the template editor, still on the payment_c-detail tab, where you can design the Leads panel template.
13. Click the Code button.

14. On the Components palette, in the Filter field, enter `cx-panel`.
15. Drag and drop the cx-panel fragment to the template editor, between the leadsPanel template tags.



16. Add the following parameters to the fragment code so that the code looks like the below sample. Be sure to replace `leads` and `Payment_Id_PaymentLead1M` with the appropriate values for your related object name and foreign key field.

> **Note:** The format of the foreign key field's name is always `<Source object name>_Id_<Relationship name>`. You can also retrieve the field name by doing a REST describe of the target object (leads).

```
<template id="leadsPanel">
 <oj-vb-fragment bridge="[[vbBridge]]" class="oj-sp-foldout-layout-panel"
 name="oracle_cx_fragmentsUI:cx-panel">
<oj-vb-fragment-param name="resource" value='[[ {"name": "leads", "primaryKey": "Id", "endpoint":
 "cx" } ]]'>
 </oj-vb-fragment-param>
 <oj-vb-fragment-param name="sortCriteria" value='[[ [{"attribute": "LastUpdateDate","direction":
 "desc" }] ]]'>
 </oj-vb-fragment-param>
 <oj-vb-fragment-param name="query"
 value='[[ [{"type": "qbe", "params": [{"key": "Payment_Id_PaymentLead1M", "value":
 $variables.id }]}] ]]'>
 </oj-vb-fragment-param>
 <oj-vb-fragment-param name="context" value="[[ {} ]]"></oj-vb-fragment-param>
 <oj-vb-fragment-param name="extensionId" value="{{ $application.constants.extensionId }}"></oj-vb-
fragment-param>
 </oj-vb-fragment>
```

```
</template>
```

This table describes some of the parameters that you can provide for a custom panel.

Parameters for Custom Panel

Parameter Name	Description
sortCriteria	Specify how to sort the data on the panel, such as sort by last updated date and descending order.
query	Include criteria for querying the data on the panel.

17. Click **< Return to page**.
18. Click the Code button.
19. You're ready to add the subview next.

> **Tip:** Once you add the panel to the panel region, that's all that's required. The standard object panel comes configured with a set of attributes to display by default. If you want to configure the panel, however, then you can do so. See *Configure the Contents of a Panel*.

You can test the panel after you add the subview. Let's do that next.

Add a Subview for the Leads Panel

After adding a related object panel to your custom object's detail page, add the subview next.

1. On the payment_c-detail page, highlight the `<oj-dynamic-container>` tags for the subviews.

```
<div class="oj-flex">
  <oj-dynamic-container layout="SubviewContainerLayout" layout-provider="[[ $metadata.dynamicCo
    class="oj-flex-item oj-sm-12 oj-md-12"></oj-dynamic-container>
</div>
```

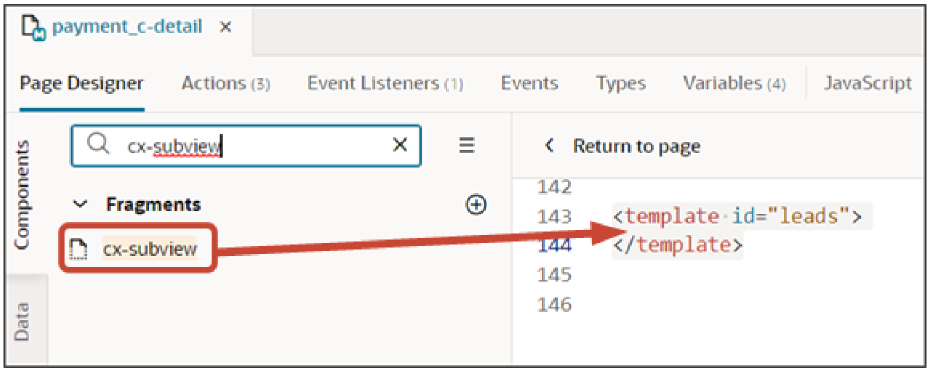
2. On the Properties pane, in the Case 1 region, click the **Add Section** icon, and then click **New Section**.
3. In the Title field, enter a title for the section, such as `Leads`.
4. In the ID field, keep the value of `leads`.

   > **Note:** Use the REST API object name for this ID.

5. Click **OK**.
6. On the Properties pane, click the Leads section that you just added.

   Page Designer navigates you to the template editor, still on the payment_c-detail tab, where you can design the leads template.
7. Click the Code button.

8. On the Components palette, in the Filter field, enter `cx-subview`.
9. Drag and drop the cx-subview fragment to the template editor, between the leads template tags.



10. Add the following parameters to the fragment code so that the code looks like the below sample. Be sure to replace `leads` and `Payment_Id_PaymentLead1M` with the appropriate values for your object and foreign key field.

> **Note:** The format of the foreign key field's name is always `<Source object name>_Id_<Relationship name>`. You can also retrieve the field name by doing a REST describe of the target object (leads).

```
<template id="leads">
 <oj-vb-fragment bridge="[[vbBridge]]" name="oracle_cx_fragmentsUI:cx-subview">
 <oj-vb-fragment-param name="resource" value='[[ {"name": "leads", "primaryKey": "Id", "endpoint":
 "cx" } ]]'>
 </oj-vb-fragment-param>
 <oj-vb-fragment-param name="sortCriteria" value='[[ [{"attribute": "LastUpdateDate","direction":
 "desc" }] ]]'>
 </oj-vb-fragment-param>
 <oj-vb-fragment-param name="query"
 value='[[ [{"type": "qbe", "params": [{"key": "Payment_Id_PaymentLead1M", "value":
 $variables.id }]}] ]]'>
 </oj-vb-fragment-param>
 <oj-vb-fragment-param name="context" value="[[ {} ]]"></oj-vb-fragment-param>
 <oj-vb-fragment-param name="extensionId" value="{{ $application.constants.extensionId }}"></oj-vb-
fragment-param>
 </oj-vb-fragment>
</template>
```

This table describes some of the parameters that you can provide for the subview:

Parameters for Subview

Parameter Name	Description
sortCriteria	Specify how to sort the data on the subview, such as sort by last updated date and descending order.
query	Include criteria for querying the data on the subview.

11. Comment out the dynamic container components from the payment_c-detail page.

    a. Click **< Return to page**.

    b. Click the Code button.

    c. Comment out the dynamic container components that contain the panels and subviews.

> **Note:** To add more panels and subviews, you must first un-comment the dynamic container components.

> **Tip:** Once you add the subview, that's all that's required. The subview for a standard object comes configured with a set of attributes to display by default. If you want to configure the subview, however, then you can do so. See *Configure the Subview Layout*.

Test Your Panel and Subview

Test the subview by previewing your application extension from the payment_c-list page.

1. From the payment_c-list page, click the Preview button to see your changes in your runtime test environment.

2. The resulting preview link will be:

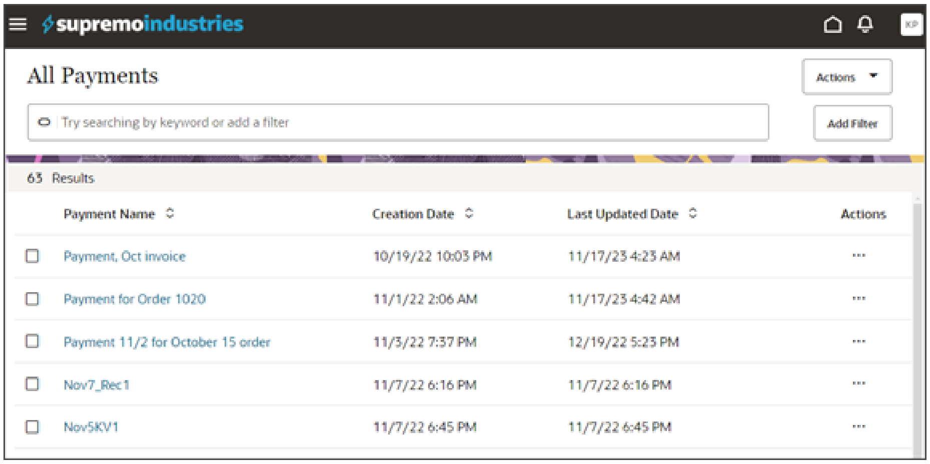
```
https://<servername>/fscmUI/redwood/cx-custom/payment_c/payment_c-list
```

3. Change the preview link as follows:

```
https://<servername>/fscmUI/redwood/cx-custom/application/container/payment_c/payment_c-list
```

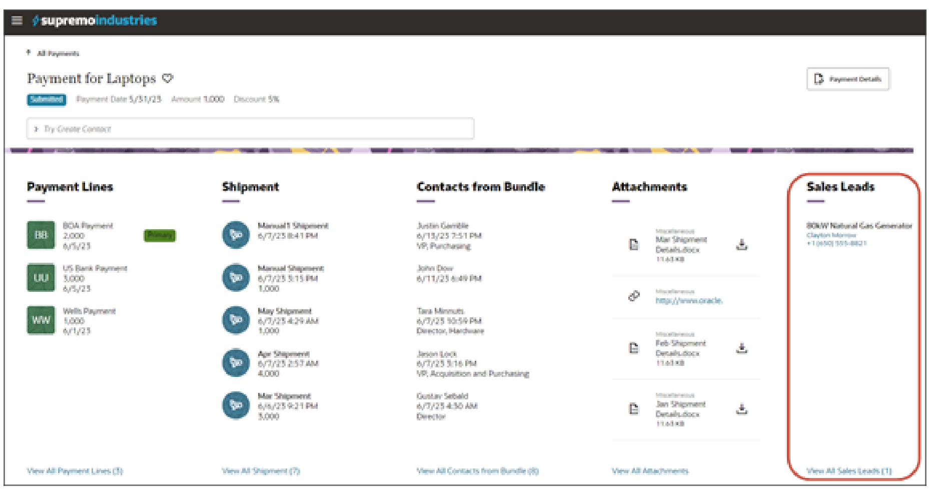
> **Note:** You must add `/application/container` to the preview link.

The screenshot below illustrates what the list page looks like with data.

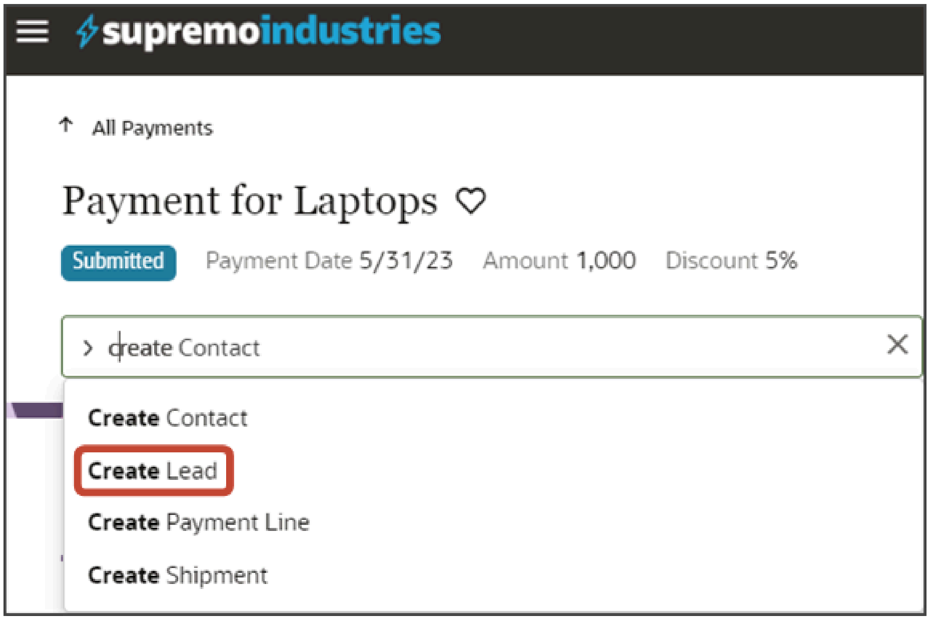


4. If data exists, you can click any record on the list page to drill down to the detail page. The detail page, including header region and panels, should display.

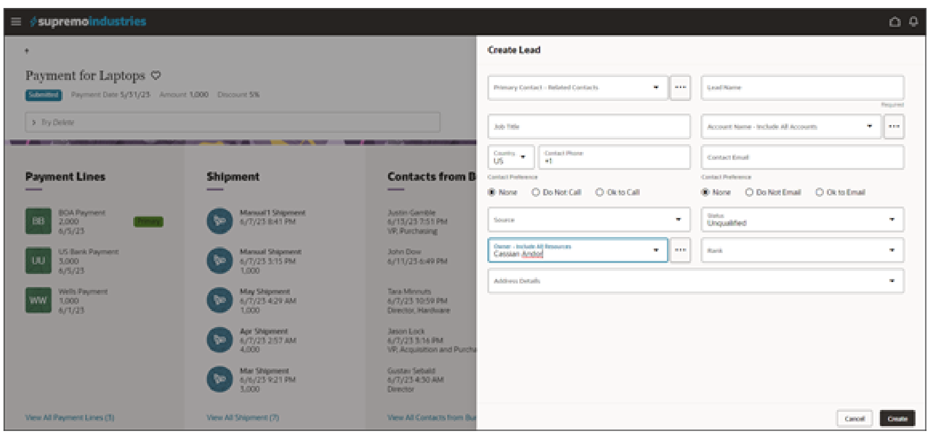
   You should now see a Sales Leads panel.



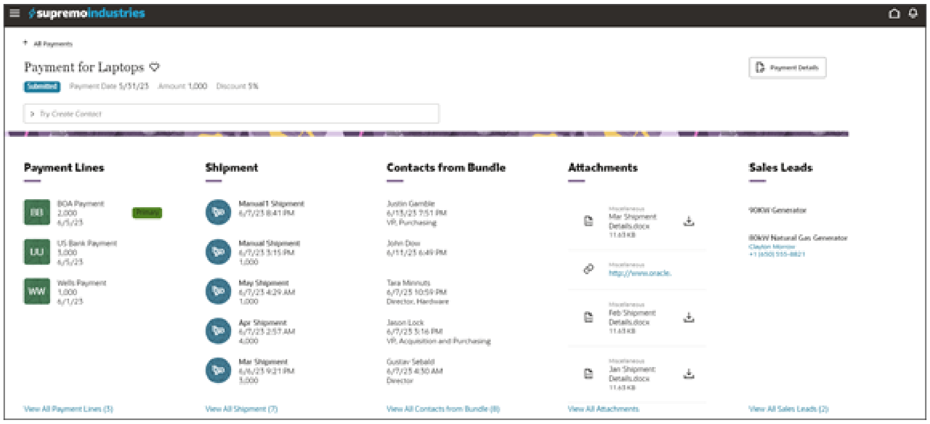
5. In the Action Bar, select the Create Lead action.



The Create Lead page displays. Here's an example of a general Create Lead page:

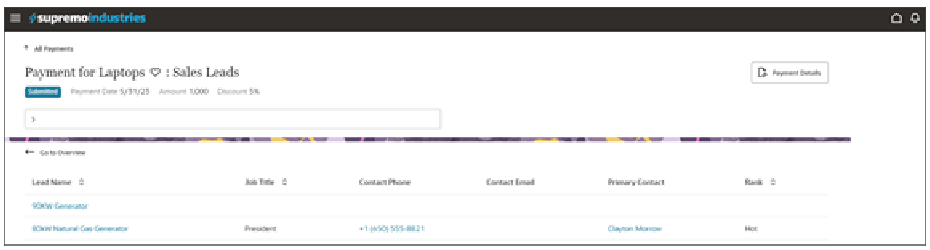


After creating a lead, you should be navigated to the lead's detail page. Click the browser back button to return to the Payment detail page where the new lead displays in the Sales Leads panel.



6. On the Sales Leads panel, click the link for the lead you just created to navigate to the lead's detail page.

   Click the browser back button.
7. Click the View All link to drill down to the subview.

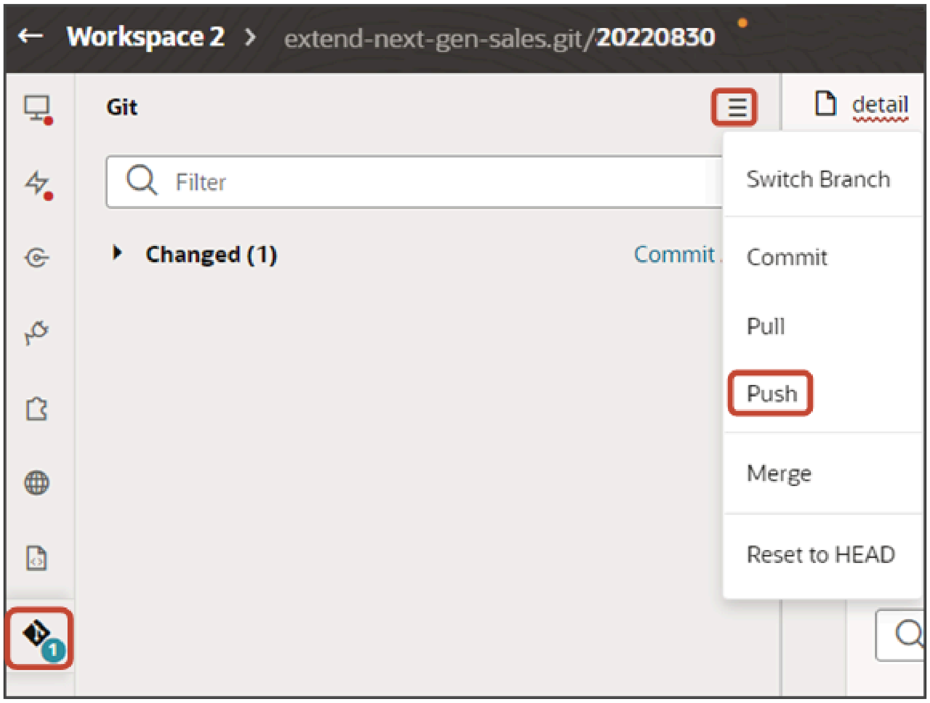


8. On the Sales Leads subview, click a lead to navigate to the lead's detail page.

   Click the browser back button.

9. Save your work by using the Push Git command.

   Navigate to the Git tab, review your changes, and do a Git push (which does both a commit and a push to the Git repository).



Add a Custom Object Panel for Related Objects (One-to-Many)

You can enhance an object's detail page by adding panels for related objects. This makes it easy for users to see – on a single page – all pertinent information related to a record. You can add custom object panels or standard object panels. This topic illustrates how to add a custom object panel to an object's detail page (when the panel object is related via a one-to-many relationship).

What's the Scenario?

In our example relationship, the Payment object has a one-to-many relationship with the Shipment object. At runtime, users should be able to create shipments for a payment, and view those shipments on the Payment detail page. To enable this, we need to add a Shipments panel to the Payment detail page.

Setup Overview

To add a related object panel to a custom object's detail page, you must complete a few steps first. Here's an overview of the required steps.

1. Complete these steps for your related object:
    a. Make sure that the CX Extension Generator generated the pages and layout for the related object, in this case, for the Shipment object.

       See *Create a New Application Using the CX Extension Generator*.
    b. Create the required **create** smart action for the related object in Application Composer.

       For example, create a Create Shipment smart action.

       The Create Shipment smart action displays from the Action Bar on both the Payment detail page and Shipments subview. Users can select the Create Shipment smart action to navigate to a create shipment page.
2. You can then add a new related object panel to the custom object's detail page.

   See *Add the Shipment Panel to the Payment Detail Page*.
3. After adding the panel, you can then create and configure the subview.

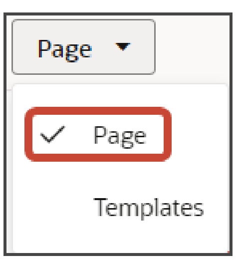
   See *Configure the Subview for Related Objects (One-to-Many)*.

Add the Shipment Panel to the Payment Detail Page

To add a new panel for shipments to the Payment detail page:

1. In Visual Builder Studio, click the App UIs tab.
2. Expand cx-custom > payment_c, then click the payment_c-detail node.
3. On the payment_c-detail tab, click the Page Designer subtab.
4. Click the Code button.

5. Confirm that you are viewing the page in Page Designer.



6. Remove the comment tags for the dynamic container components that contains the panels and any subviews.

```
<oj-vb-fragment bridge="[[vbBridge]]" name="oracle_cx_fragmentsUI:cx-detail" class="oj-flex-iter
  <oj-vb-fragment-param name="resources"
    value="[[ {'Payment_c' : {'puid': $variables.puid, 'id': $variables.id, 'endpoint': $applic
  </oj-vb-fragment-param>
  <oj-vb-fragment-param name="header"
    value="[[ {'resource': $flow.constants.objectName, 'extensionId': $application.constants.ext
  </oj-vb-fragment-param>
  <oj-vb-fragment-param name="actionBar"
    value='[[ { "applicationId": "ORACLE-ISS-APP", "resource": {"name": $flow.constants.objectNa
  </oj-vb-fragment-param>
  <oj-vb-fragment-param name="panels"
    value='[[ { "panelsMetadata": $metadata.dynamicContainerMetadata, "view": $page.variables.vi
  </oj-vb-fragment-param>
  <oj-vb-fragment-param name="context" value="[[ {'flowContext': $flow.variables.context} ]]"></
</oj-vb-fragment>
<!--
<oj-dynamic-container layout="PanelsContainerLayout" layout-provider="[[ $metadata.dynamicConta
    class="oj-flex-item oj-sm-12 oj-md-1"></oj-dynamic-container>
<oj-dynamic-container layout="SubviewContainerLayout" layout-provider="[[ $metadata.dynamicCont
  </oj-dynamic-container>
-->
```

7. Highlight the `<oj-dynamic-container>` tags for the panels.

```
<oj-dynamic-container layout="PanelsContainerLayout" layout-provider="[[ $metadata.dynamicContai
    class="oj-flex-item oj-sm-12 oj-md-1"></oj-dynamic-container>
```

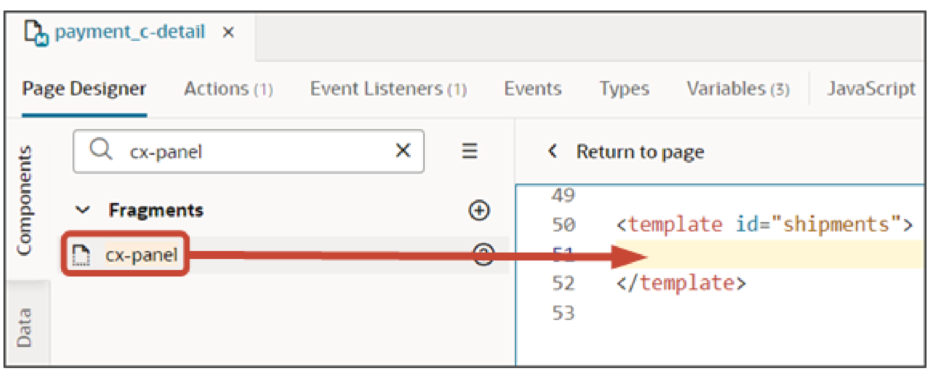
8. On the Properties pane, in the Case 1 region, click the **Add Section** icon, and then click **New Section**.
9. In the Title field, enter a title for the section, such as `Shipments`.
10. In the ID field, change the value to `ShipmentsPanel`.

> **Note:** Don't use the REST object name for this ID because you'll use the REST object name when you create the subview.

11. Click **OK**.
12. On the Properties pane, click the **Shipments** section that you just added.

    Page Designer navigates you to the template editor, still on the payment_c-detail tab, where you can design the Shipments panel template.
13. Click the Code button.

14. On the Components palette, in the Filter field, enter `cx-panel`.
15. Drag and drop the cx-panel fragment to the template editor, between the shipments template tags.



16. Add the following parameters to the fragment code so that the code looks like the below sample. Be sure to replace `Shipment_c` and `Payment_Id_PaymentShipment1M` with the appropriate values for your related object name and foreign key field, and retain the proper capitalization of the custom object name.

    > **Note:** The format of the foreign key field's name is always `<Source object name>_Id_<Relationship name>`. You can also retrieve the field name by doing a REST describe of the target object (Shipment).

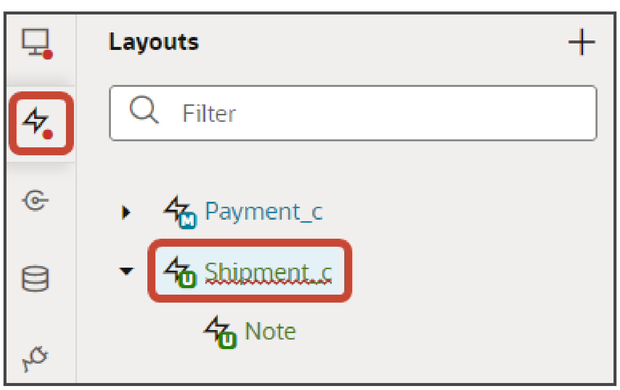
```
<template id="shipments">
 <oj-vb-fragment bridge="[[vbBridge]]" class="oj-sp-foldout-layout-panel"
 name="oracle_cx_fragmentsUI:cx-panel">
<oj-vb-fragment-param name="resource" value='[[ {"name": "Shipment_c", "primaryKey": "Id", "endpoint":
 $application.constants.serviceConnection } ]]'>
 </oj-vb-fragment-param>
 <oj-vb-fragment-param name="sortCriteria" value='[[ [{"attribute": "LastUpdateDate","direction":
 "desc" }] ]]'>
 </oj-vb-fragment-param>
 <oj-vb-fragment-param name="query"
 value='[[ [{"type": "qbe", "params": [{"key": "Payment_Id_PaymentShipment1M", "value":
 $variables.id }]}] ]]'>
 </oj-vb-fragment-param>
 <oj-vb-fragment-param name="context" value="[[ {} ]]"></oj-vb-fragment-param>
 <oj-vb-fragment-param name="extensionId" value="{{ $application.constants.extensionId }}"></oj-vb-
fragment-param>
 </oj-vb-fragment>
</template>
```

    This table describes some of the parameters that you can provide for a custom panel.

    Parameters for Custom Panel

Parameter Name	Description
sortCriteria	Specify how to sort the data on the panel, such as sort by last updated date and descending order.
query	Include criteria for querying the data on the panel.

17. In the previous step, you configured the panel template. Next, let's configure the layout for the panel.
18. Click the Layouts tab, then click the Shipment_c node.

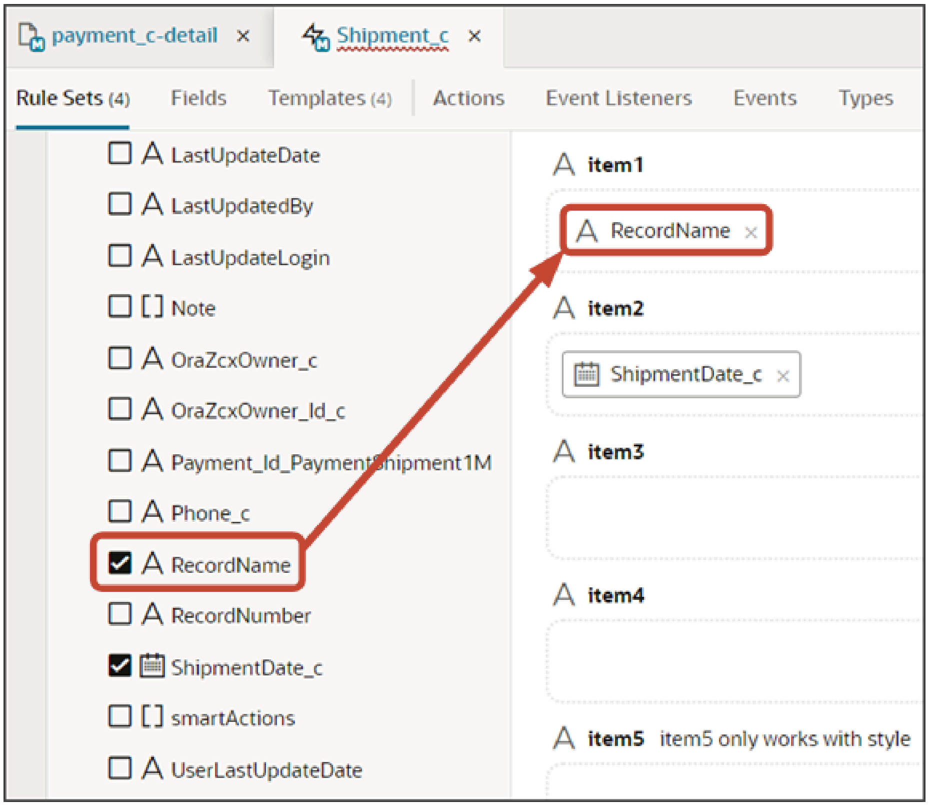


19. On the Shipment_c tab, click the **Panel Card Layout** rule set.

20. Add the fields that you want to display on the panel.

    a. Click the Open icon next to the **default** layout.
    b. Each panel includes specific slots. From the list of fields, drag each field to the desired slot.

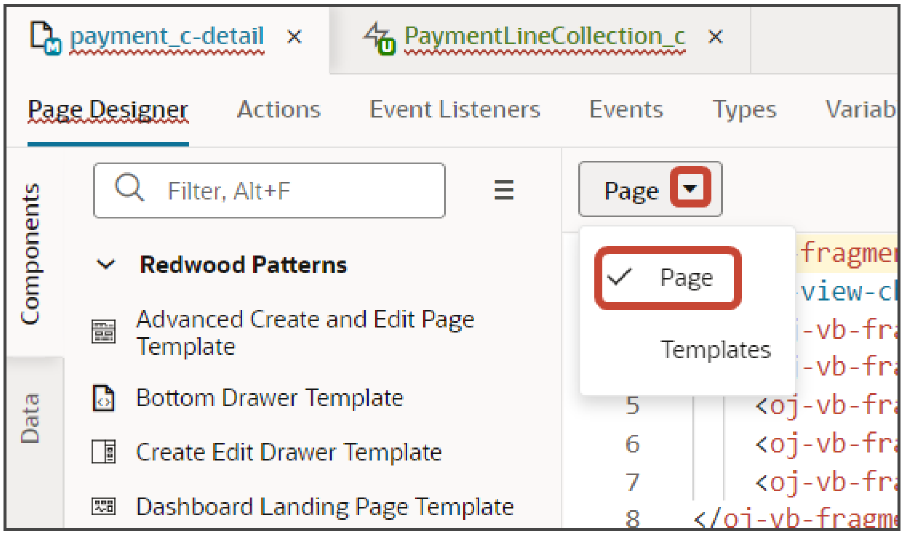
       For example, drag and drop the RecordName field to the item1 slot. If an Id field is present in that slot, you can remove it.



       Drag and drop other desired fields to the appropriate slots. For example, drag the ShipmentDate_c field to the item2 slot, and the Email_c field to the item3 slot.

21. Comment out the dynamic container components from the payment_c-detail page.

    a. Click the payment_c-detail tab, then click the Page Designer subtab.
    b. Click **< Return to page**.
    c. Click the Code button.
    d. Select **Page** from the drop-down list.



    e. Comment out the dynamic container components that contain the panels and subviews.

> **Note:** To add more panels to the panel region, you must first un-comment the dynamic container component so that you can add a new section for each desired panel.

You can test the panel after you add the subview. Let's do that next.

Configure the Subview for Related Objects (One-to-Many)

After adding a related object panel to your custom object's detail page, you can now create and configure the subview. This topic illustrates how to create the subview for a related object (related via a one-to-many relationship).

What's the Scenario?

In our example relationship, the Payment object has a one-to-many relationship with the Shipment object. At runtime, users should be able to create and view shipments for a payment on the Payment detail page, and then drill down to a Shipments subview.

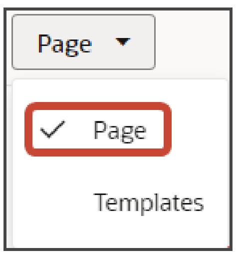
We've already added a Shipments panel to the Payment detail page. Let's create the Shipments subview next.

Create the Shipments Subview

Create a new template for the subview that displays the shipments created for a payment.

1. In Visual Builder Studio, click the App UIs tab.
2. Expand cx-custom > payment_c, then click the payment_c-detail node.
3. On the payment_c-detail tab, click the Page Designer subtab.
4. Click the Code button.

5. Confirm that you're viewing the page in Page Designer.



6. Remove the comment tags for the dynamic container components that contains the panels and any subviews.

```
<oj-vb-fragment bridge="[[vbBridge]]" name="oracle_cx_fragmentsUI:cx-detail" class="oj-flex-iten
  <oj-vb-fragment-param name="resources"
    value="[[ {'Payment_c' : {'puid': $variables.puid, 'id': $variables.id, 'endpoint': $applica
  </oj-vb-fragment-param>
  <oj-vb-fragment-param name="header"
    value="[[ {'resource': $flow.constants.objectName, 'extensionId': $application.constants.ext
  </oj-vb-fragment-param>
  <oj-vb-fragment-param name="actionBar"
    value='[[ { "applicationId": "ORACLE-ISS-APP", "resource": {"name": $flow.constants.objectNa
  </oj-vb-fragment-param>
  <oj-vb-fragment-param name="panels"
    value='[[ { "panelsMetadata": $metadata.dynamicContainerMetadata, "view": $page.variables.vi
  </oj-vb-fragment-param>
  <oj-vb-fragment-param name="context" value="[[ {'flowContext': $flow.variables.context} ]]"></
</oj-vb-fragment>
<!--
<oj-dynamic-container layout="PanelsContainerLayout" layout-provider="[[ $metadata.dynamicContai
    class="oj-flex-item oj-sm-12 oj-md-1"></oj-dynamic-container>
<oj-dynamic-container layout="SubviewContainerLayout" layout-provider="[[ $metadata.dynamicConta
  </oj-dynamic-container>
-->

```

7. Highlight the `<oj-dynamic-container>` tags for the subviews.

```
<div class="oj-flex">
  <oj-dynamic-container layout="SubviewContainerLayout" layout-provider="[[ $metadata.dynamicCo
    class="oj-flex-item oj-sm-12 oj-md-12"></oj-dynamic-container>
</div>
```


ORACLE

8. On the Properties pane, in the Case 1 region, click the **Add Section** icon, and then click **New Section**.
9. In the Title field, enter a title for the section, such as `shipment_c`.
10. In the ID field, change the value to `shipment_c`.

    > **Note:** Use the REST API object name for this ID.

11. Click **OK**.
12. Manually update the template's JSON with the correct subview name.

    a. On the payment_c-detail tab, click the JSON subtab.

    b. In the section for the SubviewContainerLayout's section template layout, replace the `sectionTemplateMap` and `displayProperties` values to match the subview's ID name, `shipment_c`. This is the object's REST API name.

    In our example, this is what the SubviewContainerLayout's `sectionTemplateMap` and `displayProperties` should look like:

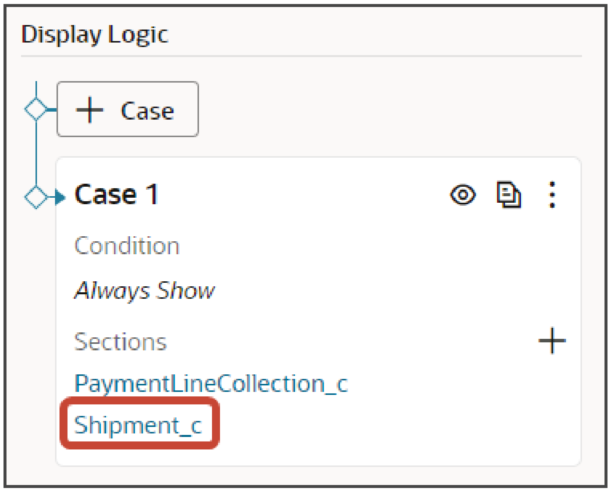
```
"SubviewContainerLayout": {
  "label": "Container Rule Set 1",
  "layoutType": "container",
  "layouts": {
    "case1": {
      "label": "Case 1",
      "layoutType": "container",
      "layout": {
        "sectionTemplateMap": {
          "PaymentLineCollection_c": "paymentLineCollectionC",
          "Shipment_c": "shipmentC"
        },
        "displayProperties": [
          "PaymentLineCollection_c",
          "Shipment_c"
        ]
      }
    }
  },
  "rules": [
    "containerLayout2-rule1"
  ]
},
```

Configure the Subview Layout

Next, let's build the structure of the subview using the **cx-subview** fragment.

1. On the payment_c-detail tab, click the Page Designer subtab.

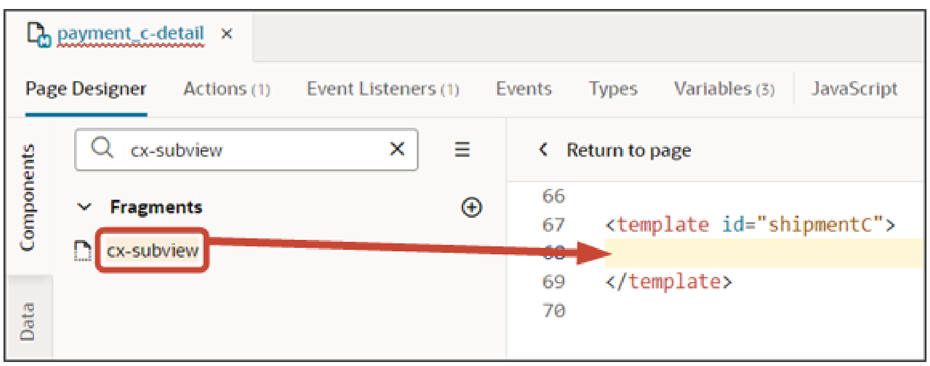
2. On the Properties pane, click the Shipment_c section that you just added.



Page Designer navigates you to the template editor, still on the payment_c-detail tab, where you can design the Shipment_c template.

3. Click the Code button.

4. On the Components palette, in the Filter field, enter `cx-subview`.
5. Drag and drop the cx-subview fragment to the template editor, between the shipmentC template tags.



6. Add the following parameters to the fragment code so that the code looks like the below sample. Be sure to replace `shipment_c` and `Payment_Id_PaymentShipment1M` with the appropriate values for your object and foreign key field.

> **Note:** The format of the foreign key field's name is always `<Source object name>_Id_<Relationship name>`. You can also retrieve the field name by doing a REST describe of the target object (Shipment).

```
<template id="shipmentC">
 <oj-vb-fragment bridge="[[vbBridge]]" name="oracle_cx_fragmentsUI:cx-subview">
 <oj-vb-fragment-param name="resource" value='[[ {"name": "Shipment_c", "primaryKey": "Id", "endpoint":
 $application.constants.serviceConnection } ]]'>
 </oj-vb-fragment-param>
 <oj-vb-fragment-param name="sortCriteria" value='[[ [{"attribute": "LastUpdateDate","direction":
 "desc" }] ]]'>
 </oj-vb-fragment-param>
```

```
 <oj-vb-fragment-param name="query"
 value='[[ [{"type": "qbe", "params": [{"key": "Payment_Id_PaymentShipment1M", "value":
 $variables.id }]}] ]]'>
 </oj-vb-fragment-param>
 <oj-vb-fragment-param name="context" value="[[ {} ]]"></oj-vb-fragment-param>
 <oj-vb-fragment-param name="extensionId" value="{{ $application.constants.extensionId }}"></oj-vb-
fragment-param>

 </oj-vb-fragment>
```

This table describes some of the parameters that you can provide for the subview:

Parameters for Subview

Parameter Name	Description
sortCriteria	Specify how to sort the data on the subview, such as sort by last updated date and descending order.
query	Include criteria for querying the data on the subview.

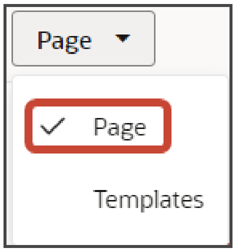
Next, let's configure the subview layout.

1. Click the Layouts tab, then click **Shipment_c** > Rule Sets subtab.
2. Click the **Sub View Layout** rule set.
3. Add the fields that you want to display in the layout.
   a. Click the Open icon next to the **default** layout.
   b. From the list of fields, select the fields that you want to display on the subview table. The fields display as columns in the order that you click them, but you can rearrange them.

4. Create an event that updates the URL when a user navigates to a subview. This enables users to be automatically navigated back to the subview after editing a record.

   > **Note:** If you already created this event when configuring a subview for a different object, then you can skip this step.

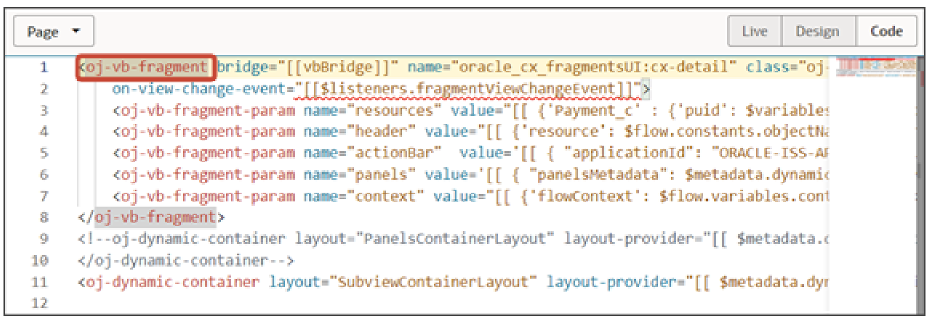
   a. On the payment_c-detail tab, click the Page Designer subtab.
   b. Confirm that you're viewing the page in Page Designer.

   c. Click the Code button.

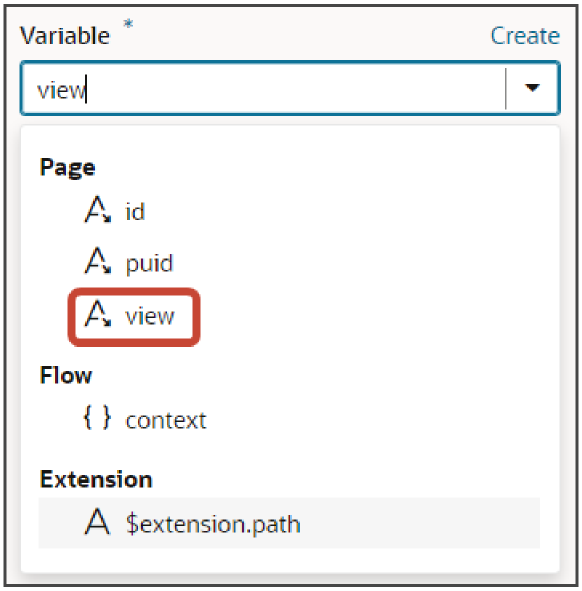
   

   d. In the code for the detail page, click the `oj-vb-fragment` tag.

   e. On the Properties pane for the cx-detail fragment, click the Events subtab.

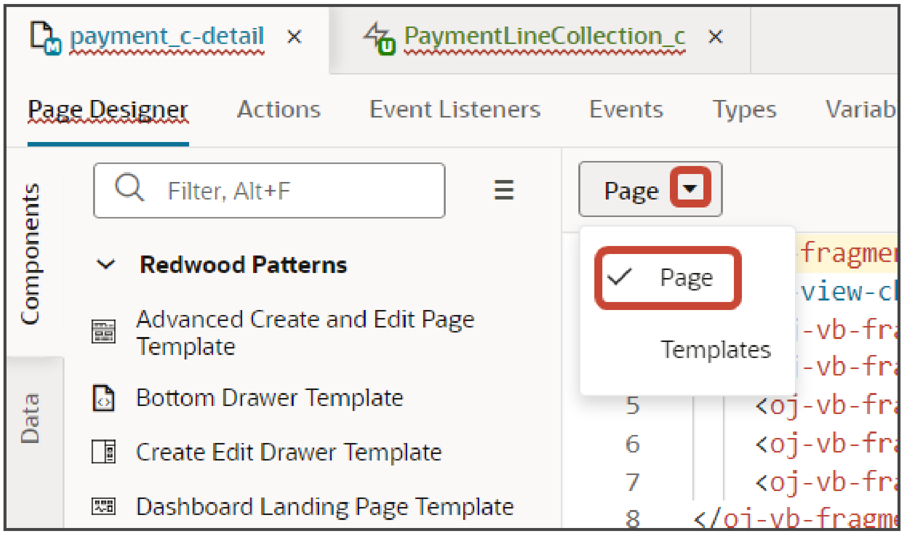
      i. Click **+ New Event** > **On 'viewChangeEvent'**.
      ii. Drag an Assign Variables action onto the canvas.
      iii. On the Properties pane, in the Variable field, click **view** under the Page heading.



iv. In the Value field, enter `{{ payload.view }}`.

5. Comment out the dynamic container components from the payment_c-detail page.

    a. Click the payment_c-detail tab, then click the Page Designer subtab.
    b. Click the Code button.
    c. Select **Page** from the dropdown.



    d. Comment out the dynamic container components that contain the panels and subviews.

Add a Link to the Panel and Subview

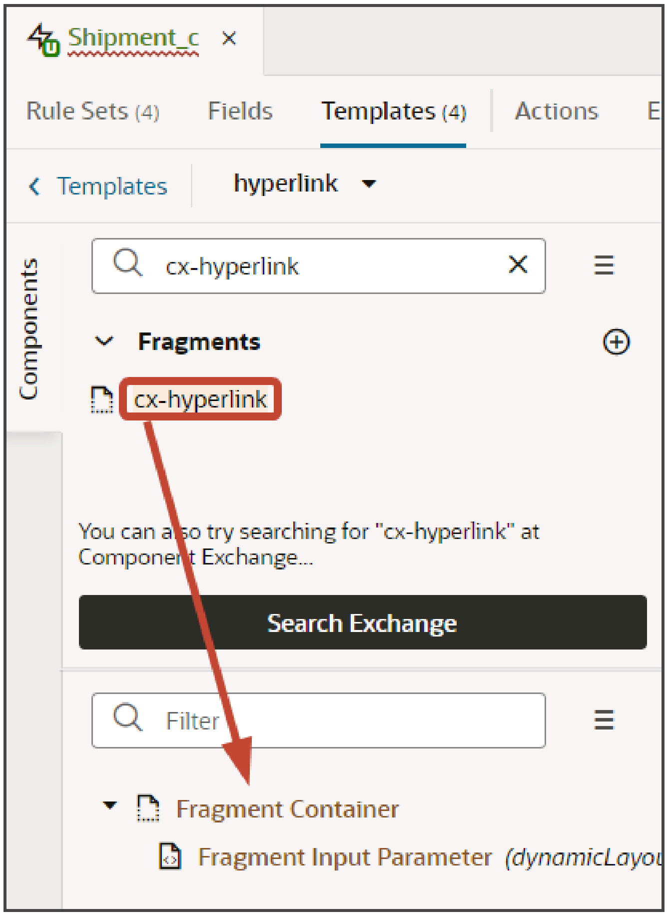
On both the panel and subview, users should be able to click the shipment name to navigate to the Shipment detail page. To enable the shipment name as a hyperlink, you must do the following:

1. Add the RecordName field to the Shipment panel and subview layouts.
2. Create a hyperlink field template.
3. For each layout, associate the RecordName field with the hyperlink field template.

First, create the hyperlink field template.

1. Click the Layouts tab, then click the Shipment_c node.

2. On the Shipment_c tab, click the Templates subtab.
3. Click **+ Template**.
4. In the Create Template dialog, for Type, select **Field**.
5. In the Label field, enter `hyperlinkTemplate`.
6. Click **Create**.
7. In the template editor, click the Design button.
8. On the Components palette, in the Filter field, enter `cx-hyperlink`.
9. Drag and drop the cx-hyperlink fragment to the Structure pane, under the Input Text node.

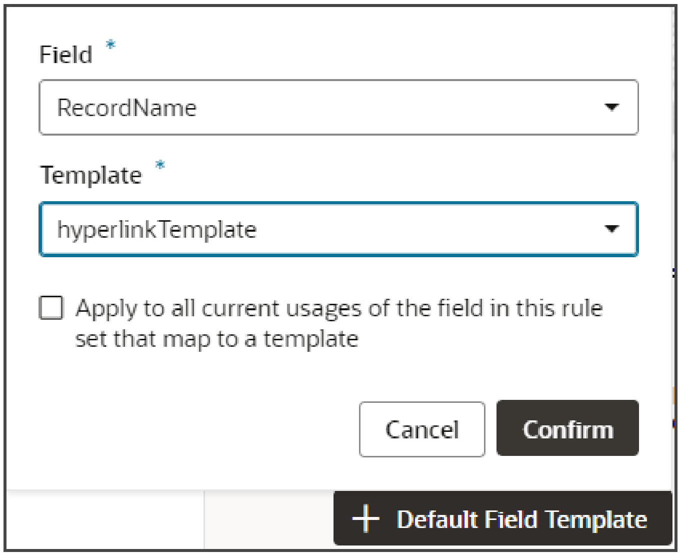


10. Delete the Input Text node.

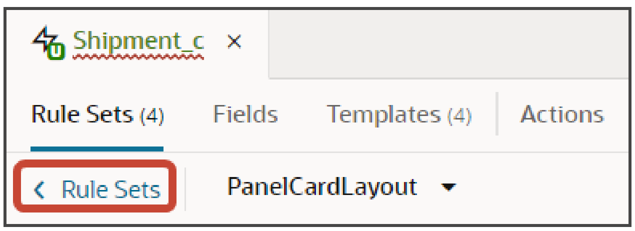
Next, associate the hyperlink field template that you just created with the RecordName field that you already added to the panel and subview layouts.

1. Click the Layouts tab, then click the Shipment_c node.
2. On the Shipment_c tab, click the Rule Sets subtab.
3. Click the **Panel Card Layout** rule set.
4. On the Properties pane, click the Templates subtab.
5. Click **+ Default Field Template**.

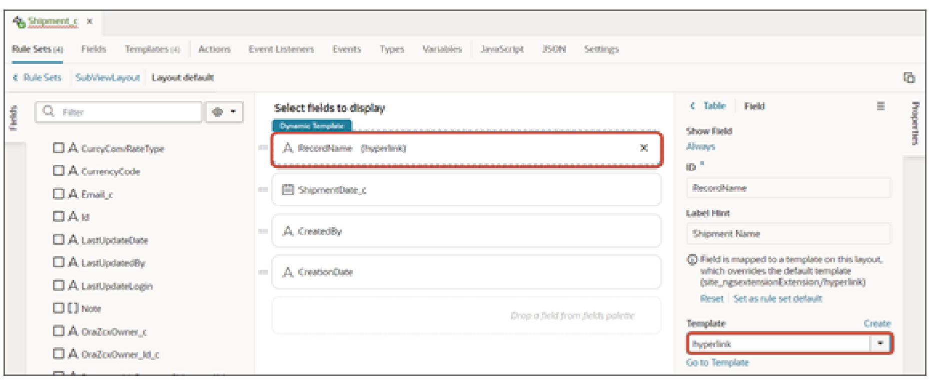
6. Select the **RecordName** field and then select the **hyperlinkTemplate** template. Click **Confirm**.



7. At the top of the page, click **< Rule Sets** to return to the list of rule sets.



8. Click the **Sub View Layout** rule set.
9. Click the Open icon next to the **default** layout.
10. Click the RecordName field to select it.
11. On the Properties pane, in the Template field, select the **hyperlinkTemplate** template.



12. Modify the hyperlink template's JSON entry for the panel card layout.
On the Shipment_c tab, click the JSON subtab.
The following code displays in the PanelCardLayout section:

```
,
 "fieldTemplateMap": {
 "RecordName": "/hyperlinkTemplate"
```

```
}
```

13. Remove the "/" so that the code looks like this:

```
,
"fieldTemplateMap": {
"RecordName": "hyperlinkTemplate"
}
```

Test Your Panel and Subview

Test the subview by previewing your application extension from the payment_c-list page.

1. From the payment_c-list page, click the Preview button to see your changes in your runtime test environment.

2. The resulting preview link will be:

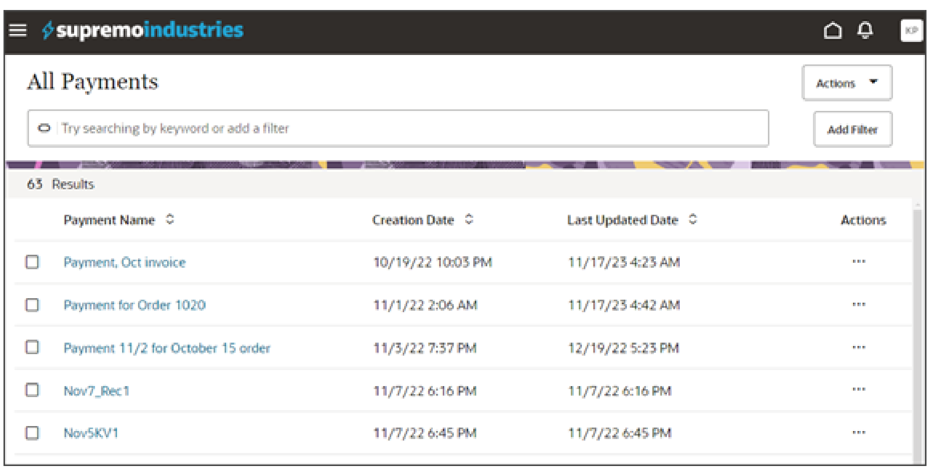
```
https://<servername>/fscmUI/redwood/cx-custom/payment_c/payment_c-list
```

3. Change the preview link as follows:

```
https://<servername>/fscmUI/redwood/cx-custom/application/container/payment_c/payment_c-list
```

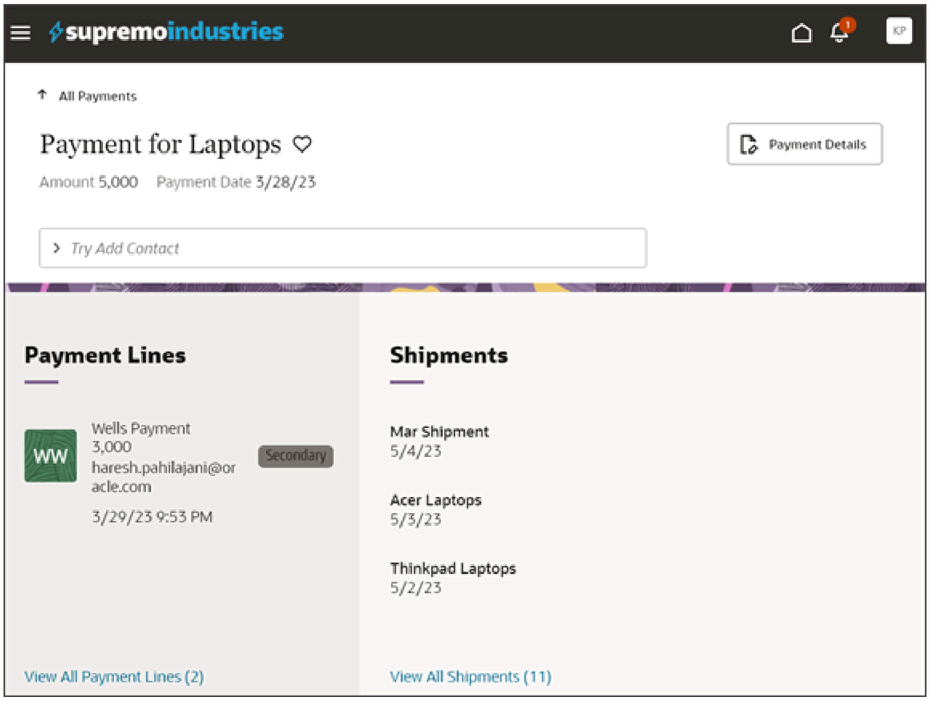
> **Note:** You must add `/application/container` to the preview link.

The screenshot below illustrates what the list page looks like with data.

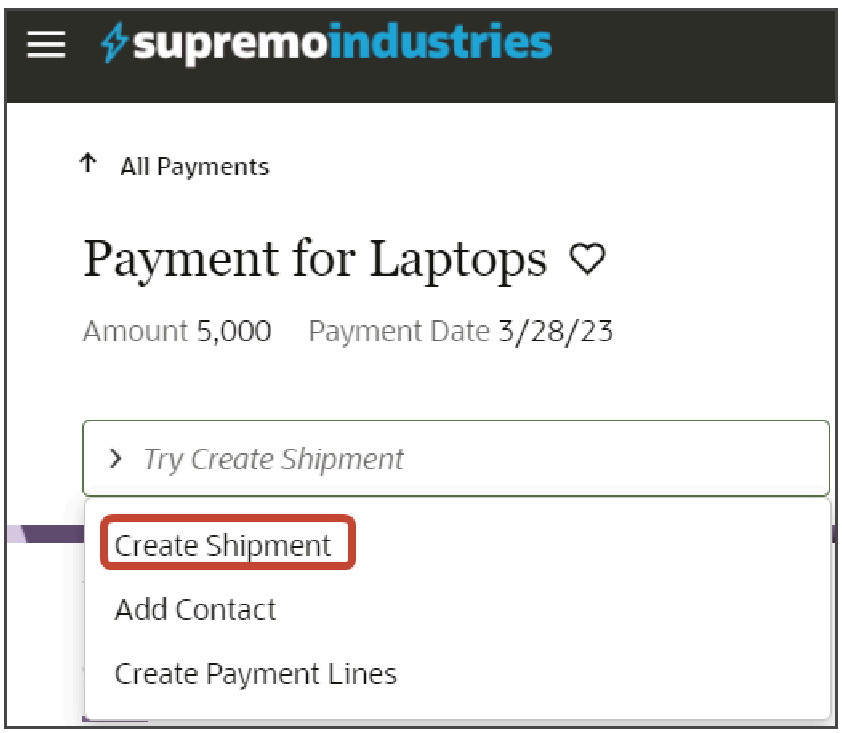


4. If data exists, you can click any record on the list page to drill down to the detail page. The detail page, including header region and panels, should display.

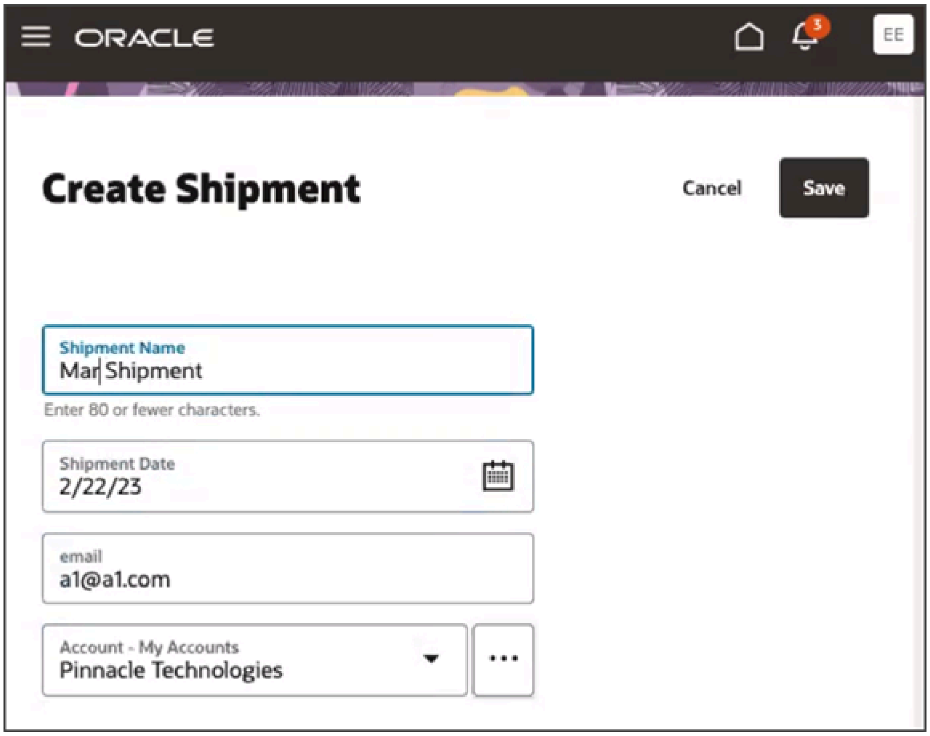
   You should now see a Shipments panel.

5. In the Action Bar, select the Create Shipment action.

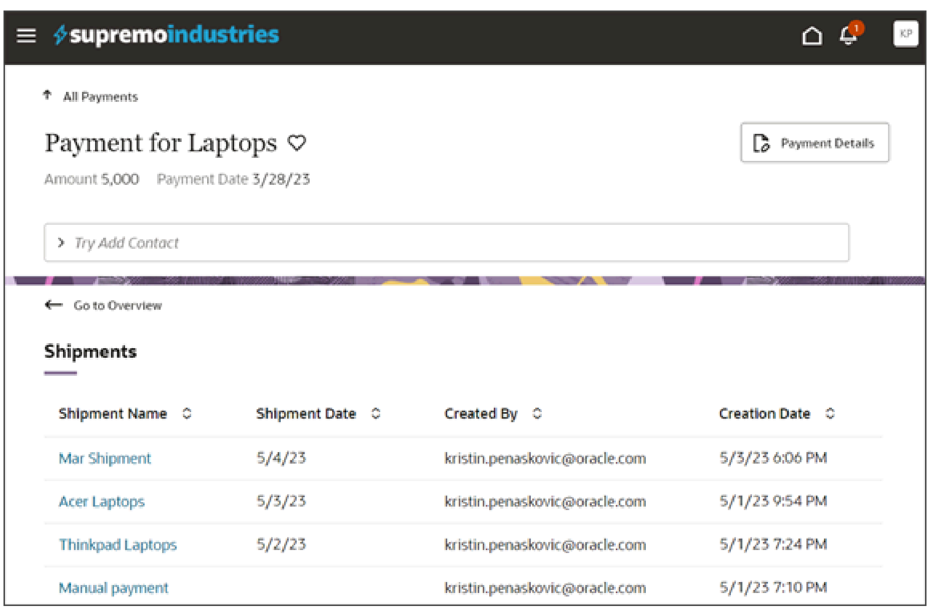
   The Create Shipment page displays. Here's an example of a general Create Shipment page:



After creating a shipment, you should be navigated to the shipment detail page. Click the browser back button to return to the Payment detail page.

6. On the Shipments panel, click the link for the shipment you just created to navigate to the Shipment detail page.

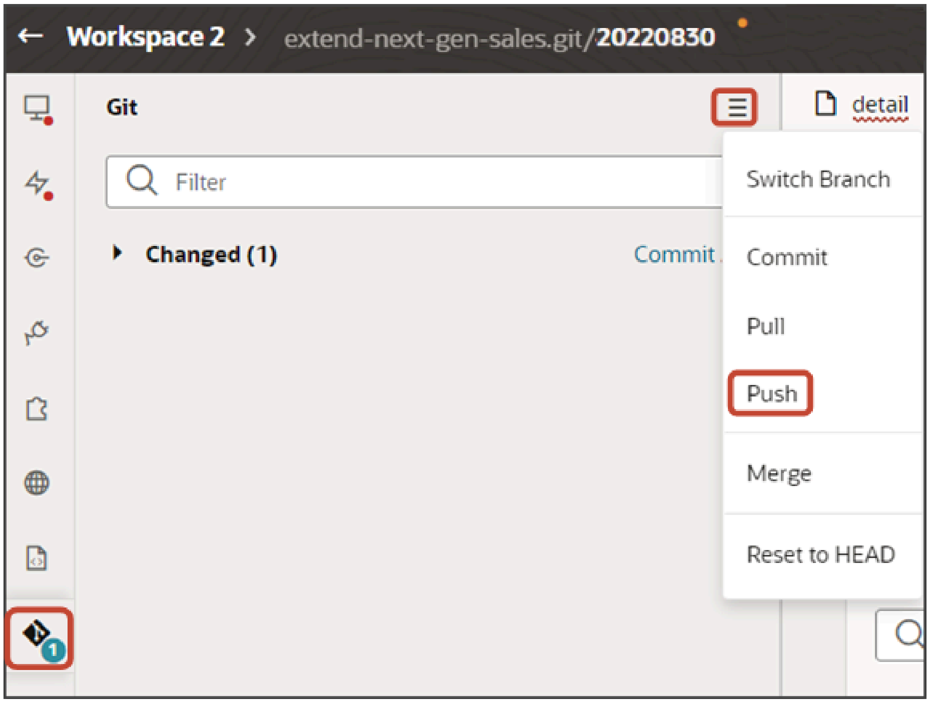
   Click the browser back button.
7. Click the View All link to drill down to the subview.



8. On the Shipments subview, click the Mar Shipment link to navigate to the Shipment detail page.

   Click the browser back button.
9. Save your work by using the Push Git command.

   Navigate to the Git tab, review your changes, and do a Git push (which does both a commit and a push to the Git repository).



Display a Panel and Subview Based on a Field Value

You can display different sets of panels (and their corresponding subviews) based on the value of a field.

To do this, create a panel layout or subview layout, and then add a field value condition. If a record's field matches the specified value, then the associated layout displays. If not, then a different layout displays.

This topic illustrates how an account's type, either Customer or Prospect, changes the panel and subview layout on an account detail page.

Prerequisite

To create a layout condition that references a field value, you must first enable this feature so that panels and subviews are loaded to the page only after evaluating the header.

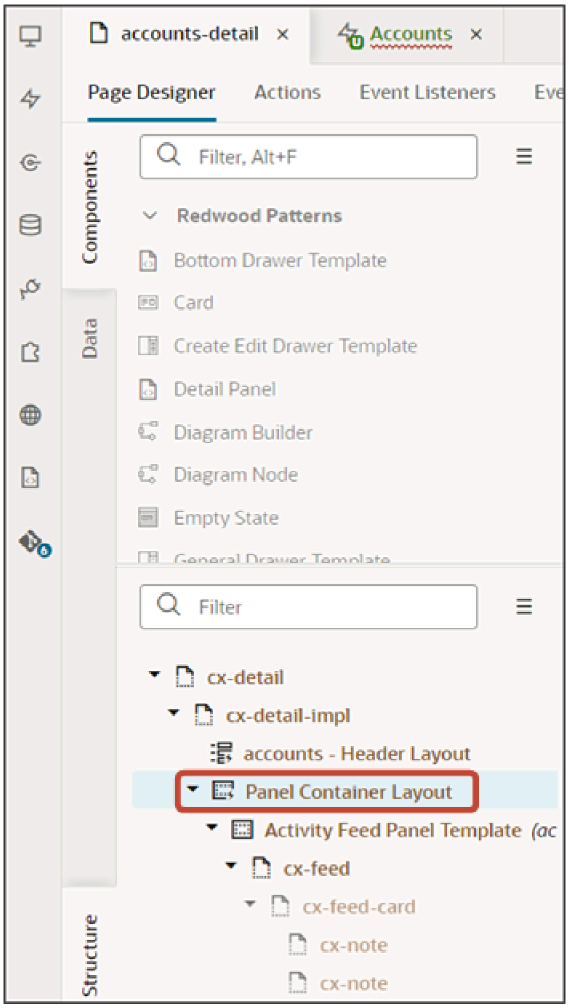
1. In Visual Builder Studio, click the App UIs side tab.
2. Navigate **CX Sales** > **cx-sales** > **accounts** > **accounts-detail**.
3. On the accounts-detail page, click the Variables subtab.
4. In the Constants region, click the **deferRelatedDataLoad** constant.
5. On the Properties pane, in the Default Value field, select **True**.

If you want to add a field value condition to panel and subview container layouts, then you must set this value to true.

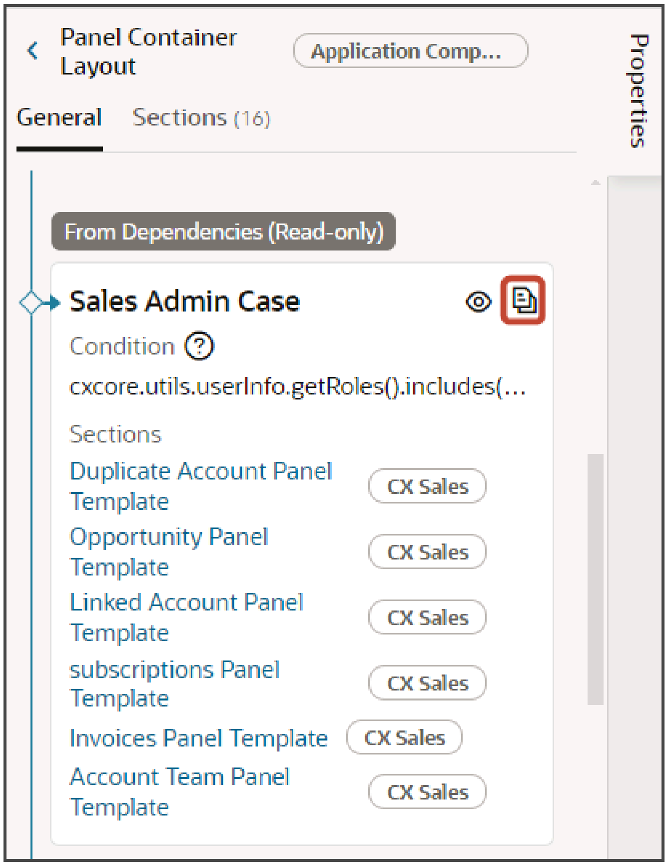
Create a New Panel Layout

Once you have enabled the feature, you can now add a field value condition to a panel layout. Let's add a condition to the account detail page.

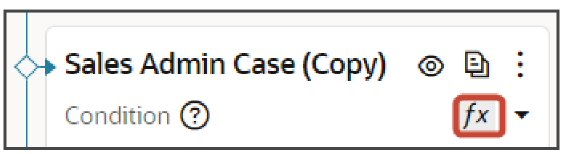
1. Navigate to Visual Builder Studio from an account record.
2. On the accounts-detail page, click the Page Designer subtab.
3. On the Structure panel, click the **Panel Container Layout** node.



4. On the Properties pane, next to **Sales Admin Case**, click the Duplicate icon.



5. Next to the Sales Admin Case (Copy) panel layout's condition, click the Expression Editor icon.



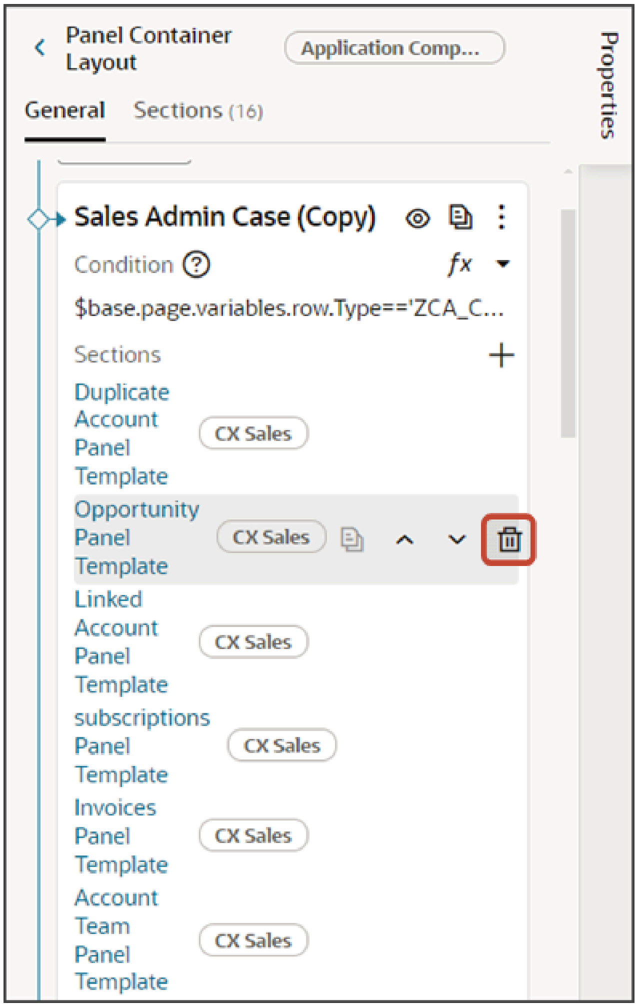
6. In the Expression Editor dialog, replace the existing expression with this new one, just for testing:

```
$base.page.variables.row.Type=='ZCA_CUSTOMER'
```

7. Click **Save**.

8. Delete the Opportunity Panel Template.

   With the field condition specified above, this means that accounts of type Customer won't see the Opportunities panel on the account detail page.



9. Let's test this change.

   From the accounts-list page, click the Preview button to see your changes in your runtime test environment.

10. The resulting preview link will be:

```
https://<servername>/fscmUI/redwood/cx-sales/accounts/accounts-list
```

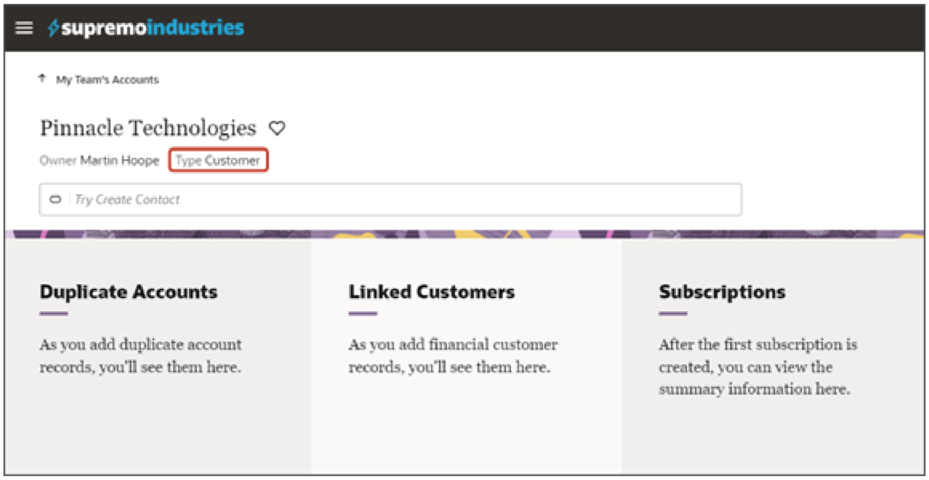
11. Change the preview link as follows:

```
https://<servername>/fscmUI/redwood/cx-sales/application/container/accounts/accounts-list
```

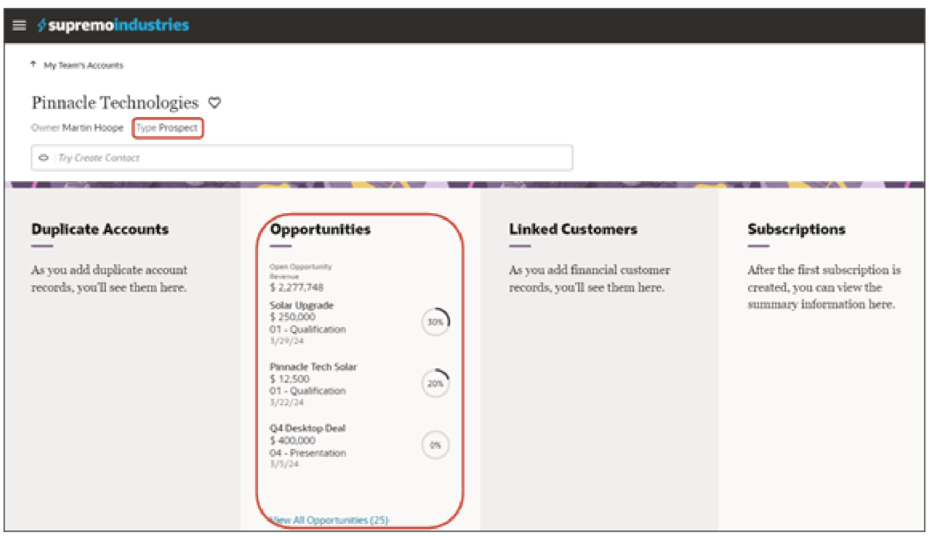
> **Note:** You must add `/application/container` to the preview link.

12. On the My Team's Accounts page, click any account.

    - If the account is of type Customer, then you won't see the Opportunities panel.

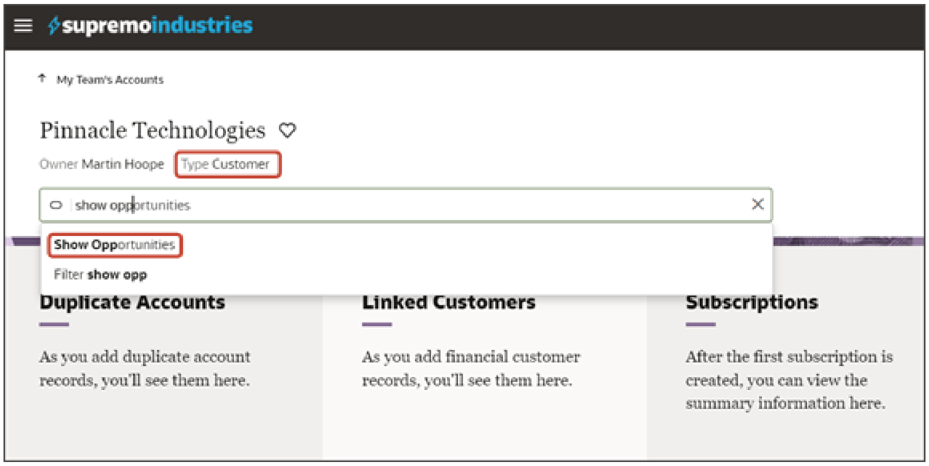


    - If the account is of type Prospect, then the Opportunities panel does display.



Create a New Subview Layout

Next, add the field condition to the subview layout, as well. It's important to add the field condition to the subview layout. Otherwise, the Show Opportunities smart action is still available from the Action Bar even when the account is a customer.



1. Navigate to Visual Builder Studio from any subview page, which you can navigate to from any panel on an account record.
2. On the accounts-detail page, click the Page Designer subtab.
3. On the Structure panel, click the **Subview Container Layout** node.
4. On the Properties pane, next to **Subview Container Layout**, click the Duplicate icon.
5. Next to the Subview Container Layout (Copy) subview layout's condition, click the Expression Editor icon.
6. In the Expression Editor dialog, add this expression:

   ```
   $base.page.variables.row.Type=='ZCA_CUSTOMER'
   ```
7. Click **Save**.
8. Delete the Opportunity Subview Template.

   With the field condition specified above, this means that accounts of type Customer won't see the Show Opportunities smart action on the account detail page.
9. Let's test this change.

   From the accounts-list page, click the Preview button to see your changes in your runtime test environment.

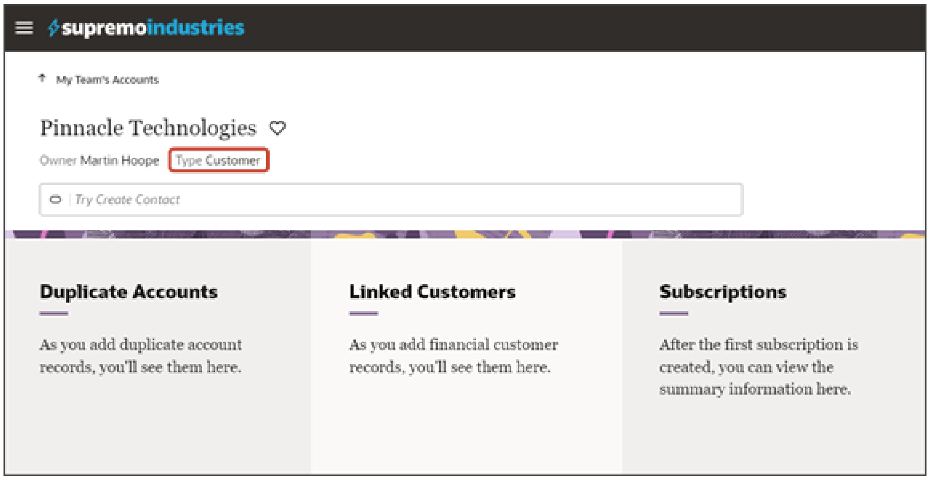
10. The resulting preview link will be:

    ```
    https://<servername>/fscmUI/redwood/cx-sales/accounts/accounts-list
    ```
11. Change the preview link as follows:

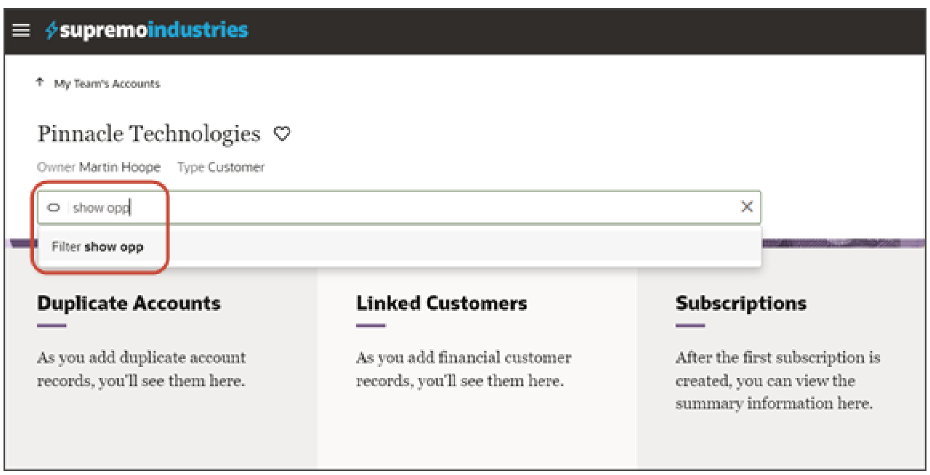
    ```
    https://<servername>/fscmUI/redwood/cx-sales/application/container/accounts/accounts-list
    ```

> **Note:** You must add `/application/container` to the preview link.

12. On the My Team's Accounts page, click any account and make sure that the account is of type Customer. The Opportunities panel shouldn't display.



13. Test the field condition on the subview layout by checking to see if the Show Opportunities smart action is still available from the Action Bar. It shouldn't be visible anymore if the account is a customer.

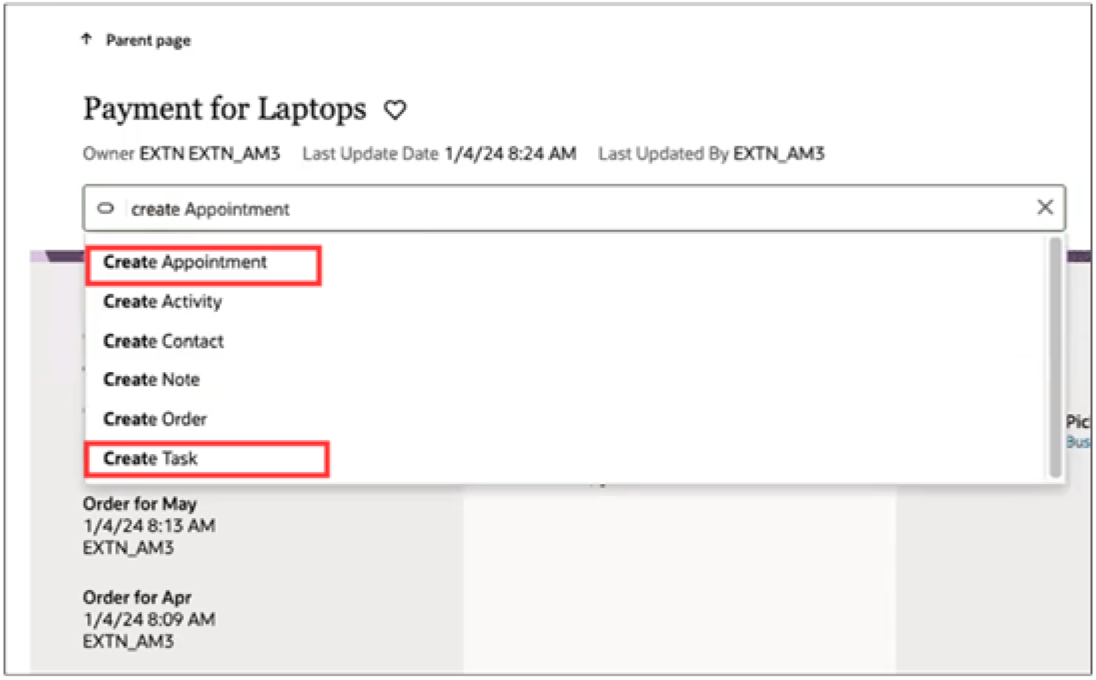


Configure the Subviews for Appointments and Tasks

Using Oracle Visual Builder Studio, you can make it possible for users to create and view appointments and tasks right from a custom object's detail page.

What's the Scenario?

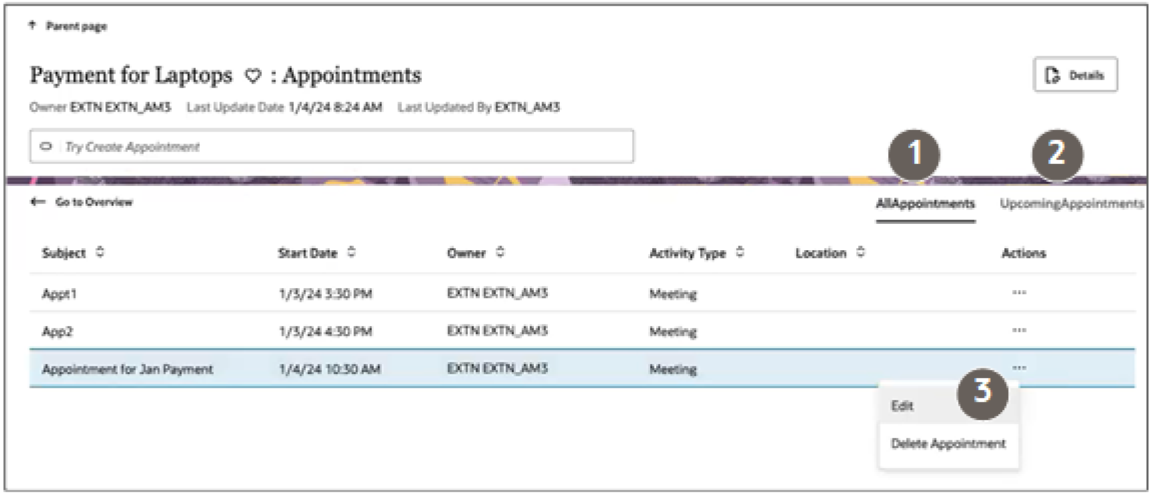
This example shows you how to create subtabs for appointments and tasks on a Payment custom object and how to enable users to view and create tasks and appointments directly from each payment's Action Bar. The view and delete actions are already provided for you, but you must create the Create smart actions for tasks and for appointments.



When you're finished with the setup in this example, you'll end up with separate subtabs for tasks and appointments.

Here's a screenshot of a sample Tasks subtab. The subtab includes 3 separate views highlighted by callouts: All Tasks, Open Tasks, and Overdue Tasks. For each task, there are 3 available actions: Mark Complete, Edit and Delete Task.

The Appointments subtab includes 2 separate views highlighted by callouts in the following screenshot: All Appointments and Upcoming Appointments. The available actions are: Edit and Delete Appointment.



Prerequisites

In Application Composer, Create a 1:M relationship between your custom object and the Activity object. In this example, we're adding the relationship for the Payment custom object.

Create Smart Actions for Appointments and Tasks

1. In a sandbox, open Application Composer.
2. Click **Smart Actions**.
3. Create separate Create smart actions for tasks and appointments.
    a. On the Smart Actions list page, click **Create**.

       The application displays a guided process with 7 steps to complete in order.

       > **Note:** For your entries to be saved when creating smart actions, you must click **Submit** (available in the last step in the guided process) . You can always go back and make changes after submitting.

    b. In the **Kind of Action** step, select **UI-based action**.
    c. Click **Continue**.
    d. In the **Basic Details** step, enter the following on the Payment object.:

Field Name	Entries for Tasks	Entries for Appointments
Name	Create Task	Create Appointment
Object	Payment	Payment

Field Name	Entries for Tasks	Entries for Appointments
Action ID	You can accept the default.	You can accept the default.

e. Click **Continue**.
f. In the **Availability** step, enter the following:

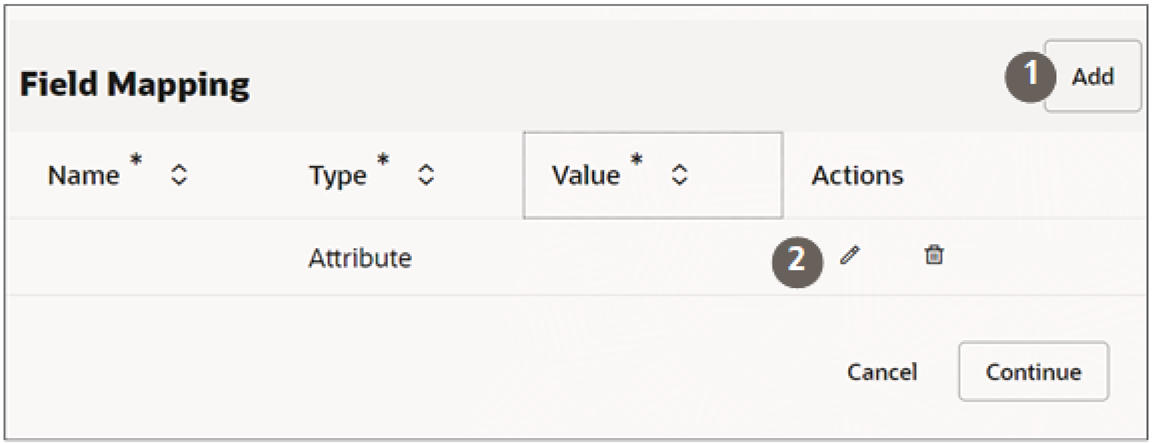
Field Name	Entries for Tasks and Appointments
Application	Sales
UI Availability	List page
Action ID	You can accept the default.
Role Filter	Optionally, specify the job roles that can use this smart action. No entry means that all job roles can use this action.

g. Click **Continue**.
h. In the **Action Type** step, make the following entries:

Field Name	Entries for Tasks	Entries for Appointments
Type	Create	Create
Subtype	Task	Appointment
Target Object	Activity	Activity

Field Name	Entries for Tasks	Entries for Appointments
Object Subtype	Task	Appointment

i. While in the **Action Type** step, in the **Field Mapping** section add two field mapping conditions.
j. Click **Add** (callout 1 in the screenshot)



k. In the **Actions** column, click **Edit** (the pencil icon highlighted by callout 2)
l. Make the following entries:

Field Name	Entries for Tasks	Entries for Appointments
Name	Payment ID Activities (Payment_id_ Activities)	Payment ID Activities (Payment_id_ Activities)
Type	Attribute	Attribute
Value	Record ID (Id)	Record ID (Id)

m. Click **Done** to save the row.
n. Click **Add** again and add the second condition. The entries are the same for tasks and appointments except for the Value where you must type either TASK or APPOINTMENT.

Field Name	Entries for Tasks	Entries for Appointments
Name	Activity (ActivityFunctionCode)	Activity (ActivityFunctionCode)
Type	User-entered	User-entered

Field Name	Entries for Tasks	Entries for Appointments
Value	TASK	APPOINTMENT

o. Click **Done**.
p. Click **Continue** twice to skip over the **UI-Based Action Details** step. This step doesn't apply to Redwood Sales.
q. Optionally enter a confirmation message in the **Confirmation Message** step. The confirmation message appears briefly after the user creates the record.
r. Click **Continue**.
s. On the **Review and Submit** step, click **Submit**.

Create the Subviews

Create new templates for the subviews that display the appointments and tasks created for a payment. You will configure the actual subviews in the next section.

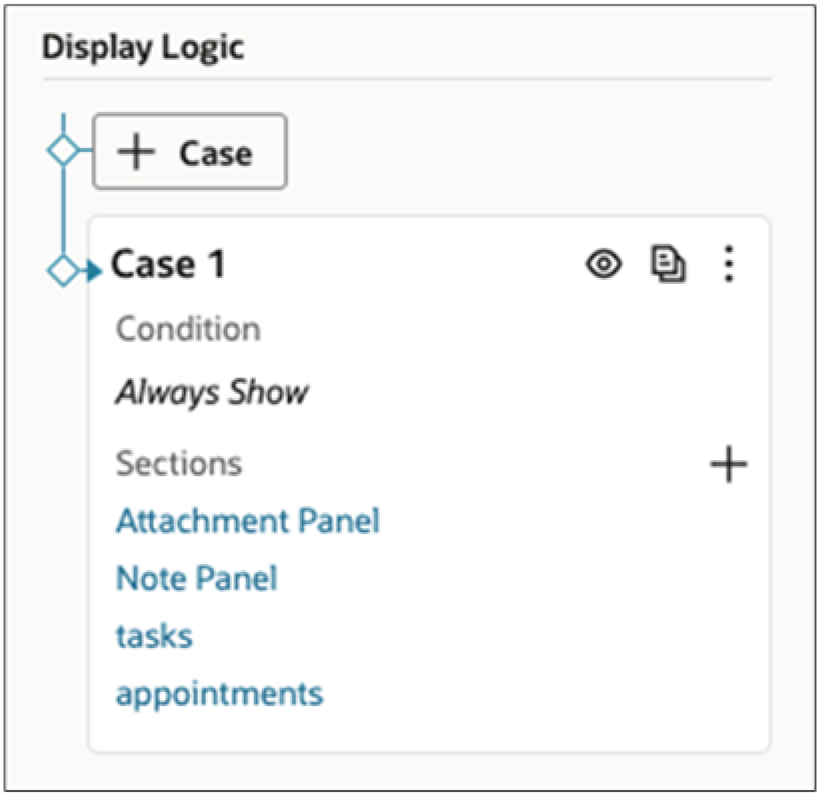
1. In Visual Builder Studio, click the **App UIs** tab.
2. Expand **cx-custom** > **payment_c**, then click the **payment_c-detail** node.
3. On the payment_c-detail tab, click the **Page Designer** subtab.
4. Click the **Code** button.
5. Confirm that you are viewing the page in Page Designer.
6. Remove the comment tags for the dynamic container components that contains the panels and any subviews.
7. Highlight the `<oj-dynamic-container>` tags for the subviews.
8. On the Properties pane, in the Case 1 region, click the **Add Section** icon, and then click **New Section.**
9. In the Title field, enter a title for the section, such as **tasks**.
10. In the **ID** field, accept the value **tasks**.
11. Click **OK**.
12. Repeat steps to create a second section: appointments.
13. Add the following code for activity translations below imports:

```
"translations": {
"activityBundle": {
"path": "faResourceBundle/nls/oracle.apps.crmCommon.activities.resource.activityManagement"
}
},
```

Configure the Subview Layouts

Next, build the structure of the subviews using the cx-subview fragment.

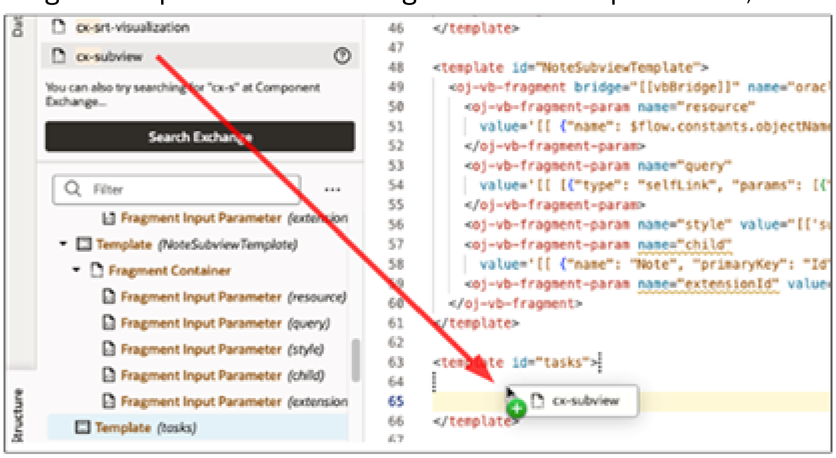
1. On the **payment_c-detail tab**, click the **Page Designer** subtab.
2. On the Properties pane, click the **tasks** section that you just added.



Page Designer navigates you to the template editor, still on the payment_c-detail tab, where you can design the **tasks** template.

3. Click the **Code** button.

4. On the Components palette, in the Filter field, enter `cx-subview`.
5. Drag and drop the cx-subview fragment to the template editor, between the tasks template tags.



6. Add the following parameters to the fragment code so that the code looks like the below sample. For the query parameter, be sure to replace the foreign key `Payment_Id_PaymentToActivities` with the appropriate value.

   > **Note:** The format of the foreign key field's name is always <Source object name>_Id_<Relationship name>.

```
<oj-vb-fragment-param name="resource"
 value='[[ {"name": "activities", "primaryKey": "ActivityId","puid": "ActivityNumber", "endpoint":
 "cx" , "alias" : "tasks"} ]]'></oj-vb-fragment-param>
 <oj-vb-fragment-param name="sortCriteria" value='[[ [{"attribute": "LastUpdateDate","direction":
 "desc" }] ]]'>
 </oj-vb-fragment-param>
 <oj-vb-fragment-param name="query" value='[[ [{"type": "qbe","provider": "adfRest", "params": [{"key":
 "Payment_Id_PaymentToActivities", "value":$variables.id }]}] ]]'></oj-vb-fragment-param>
 <oj-vb-fragment-param name="context" value="[[ {} ]]">
 </oj-vb-fragment-param>
 <oj-vb-fragment-param name="extensionId" value="oracle_cx_salesUI"></oj-vb-fragment-param>
<oj-vb-fragment-param name="types" value='[[ $functions.getTaskSubviewTypesData($page.variables.id,
 $page.translations) ]]'></oj-vb-fragment-param>
 <oj-vb-fragment-param name="title" value="Tasks"></oj-vb-fragment-param>
 <oj-vb-fragment-param name="subviewLayoutId" value="[[ 'SubViewLayoutForTasks' ]]"></oj-vb-fragment-
param>
```

7. Return to step 2 and complete the same steps for the appointments section.

   Add the following parameters to the fragment code so that the code looks like the below sample. For the query parameter, be sure to replace the foreign key `Payment_Id_PaymentToActivities` with the appropriate value.

   > **Note:** The format of the foreign key field's name is always: <Source object name>_Id_<Relationship name>.

```
<oj-vb-fragment-param name="resource"
 value='[[ {"name": "activities", "primaryKey": "ActivityId","puid": "ActivityNumber", "endpoint":
 "cx" , "alias" : "appointments"} ]]'></oj-vb-fragment-param>
 <oj-vb-fragment-param name="sortCriteria" value='[[ [{"attribute": "LastUpdateDate","direction":
 "desc" }] ]]'>
 </oj-vb-fragment-param>
 <oj-vb-fragment-param name="query"
 value='[[ [{"type": "qbe", "params": [{"key": "Payment_Id_PaymentToActivities", "value":
 $variables.id }]}] ]]'>
 </oj-vb-fragment-param>
 <oj-vb-fragment-param name="context" value="[[ {} ]]">
 </oj-vb-fragment-param>
 <oj-vb-fragment-param name="extensionId" value="oracle_cx_salesUI"></oj-vb-fragment-param>
<oj-vb-fragment-param name="types"
 value='[[ $functions.getAppointmentSubviewTypesData($page.variables.id, $page.translations) ]]'></oj-
vb-fragment-param>
 <oj-vb-fragment-param name="title" value="Appointments"></oj-vb-fragment-param>
 <oj-vb-fragment-param name="subviewLayoutId" value="[[ 'SubViewLayoutForAppointments' ]]"></oj-vb-
fragment-param>
```

   This table describes some of the parameters that you can provide for the subview:

Parameter Name	Description
sortCriteria	Specify how to sort the data on the subview, such as sort by last updated date and descending order.
query	Include criteria for querying the data on the subview.
types	Pass a JavaScript function, either getTaskSubviewTypesData or getAppointmentSubviewTypesData. These functions enable the tabs on each subview, such as All Tasks, Open Tasks, and Overdue Tasks, as well as All Appointments and Upcoming Appointments.

8. In the previous step, you added the types parameter to each subview fragment to pass a JavaScript function, either getTaskSubviewTypesData or getAppointmentSubviewTypesData.

   In this step, manually add the functions to the JavaScript:

   a. On the **payment_c-detail** tab, click the **JavaScript** subtab.
   b. Add the below functions. Be sure to replace the foreign key `Payment_Id_PaymentToActivities` with the appropriate value.

```
define(['vx/oracle_cx_salesUI/ui/self/applications/cx-sales/resources/utils/CrmCommonUtils','vx/
oracle_cx_salesUI/ui/self/applications/cx-sales/resources/utils/FormatUtils'],
 (CrmCommonUtils,FormatUtils) => {
 'use strict';

 class PageModule {
 }

PageModule.prototype.getTaskSubviewTypesData = function (id, translation) {
 const typesData = [];

 typesData.push({
 "resource": "activities",
 "query": [{
 "type": "qbe",
 "provider": "adfRest",
 "params": [
 {
 "key": "Payment_Id_PaymentToActivities",
 "value": id
 },
 {
 "key": "ActivityFunctionCode",
 "value": "TASK"
 }
 ]
 }],
 "isDefault": true,
 "sortCriteria": [{
 "attribute": "LastUpdateDate",
 "direction": "descending"
 }],
 "title": "AllTasks",
 "id": "AllTasks"
 });

 typesData.push({
 "resource": "activities",
 "query": [{
 "type": "qbe",
 "provider": "adfRest",
 "params": [
 {
 "key": "Payment_Id_PaymentToActivities",
 "value": id
 },
 {
 "key": "ActivityFunctionCode",
 "value": "TASK"
 },
 {
 "key": "StatusCode",
 "operator": "$in",
 "value": "NOT_STARTED,IN_PROGRESS,ON_HOLD"
 }
 ]
```

```
}],
"isDefault": true,
"sortCriteria": [{
"attribute": "DueDate",
"direction": "ascending"
}],
"title": "OpenTasks",
"id": "OpenTasks"
});

typesData.push({
"resource": "activities",
"query": [{
"type": "qbe",
"provider": "adfRest",
"params": [
{
"key": "Payment_Id_PaymentToActivities",
"value": id
},
{
"key": "ActivityFunctionCode",
"value": "TASK"
},
{
"key": "DueDate",
"operator": "$lt",
"value": FormatUtils.getFormattedDate(new Date())
},
{
"key": "StatusCode",
"operator": "$in",
"value": "NOT_STARTED,IN_PROGRESS,ON_HOLD"
}
]
}],
"isDefault": true,
"sortCriteria": [{
"attribute": "DueDate",
"direction": "descending"
}],
"title":"OverdueTasks",
"id": "OverdueTasks"
});

return { "style": "tab", "items": typesData };
};


PageModule.prototype.getAppointmentSubviewTypesData = function (id, translation) {
const typesData = [];

typesData.push({
"resource": "activities",
"query": [{
"type": "qbe",
"provider": "adfRest",
"params": [
{
"key": "Payment_Id_PaymentToActivities",
"value": id
},
{
"key": "ActivityFunctionCode",
"value": "APPOINTMENT"
}
```

```
]
}],
"isDefault": true,
"sortCriteria": [{
"attribute": "SortDate",
"direction": "ascending"
}],
"title": "AllAppointments",
"id": "AllAppointments"
});

typesData.push({
"resource": "activities",
"query": [{
"type": "qbe",
"provider": "adfRest",
"params": [
{
"key": "Payment_Id_PaymentToActivities",
"value": id
},
{
"key": "ActivityFunctionCode",
"value": "APPOINTMENT"
},
{
"key": "ActivityEndDate",
"operator": "$ge",
"value": new Date().toISOString()
}
]
}],
"isDefault": true,
"sortCriteria": [{
"attribute": "ActivityStartDate",
"direction": "ascending"
}],
"title": "UpcomingAppointments",
"id": "UpcomingAppointments"
});
return { "style": "tab", "items": typesData };
};

return PageModule;
});
```

9. Comment out the dynamic container components from the payment_c-detail page:
   a. Click the **payment_c-detail** tab, then click the **Page Designer** subtab.
   b. Click the **Code** button.
   c. Add the subview label for tasks and appointments in the actionBar param:

   ```
   <oj-vb-fragment-param name="actionBar"
    value='[[ { "applicationId": "ORACLE-ISS-APP", "resource": {"name": "Payment_c", "primaryKey":
    "Id", "puid": "Id", "value": $variables.puid }, "subviewLabel": {"tasks" : "Tasks",
    "appointments" : "Appointments"}} ]]'>
    </oj-vb-fragment-param>
   ```

   d. Comment out the dynamic container components that contain the panels and subviews.

Test Your Subviews

Test the subview by previewing your application extension from the payment_c-list page.

1. From the payment_c-list page, click the Preview button to see your changes in your runtime test environment.

2. The resulting preview link will be:

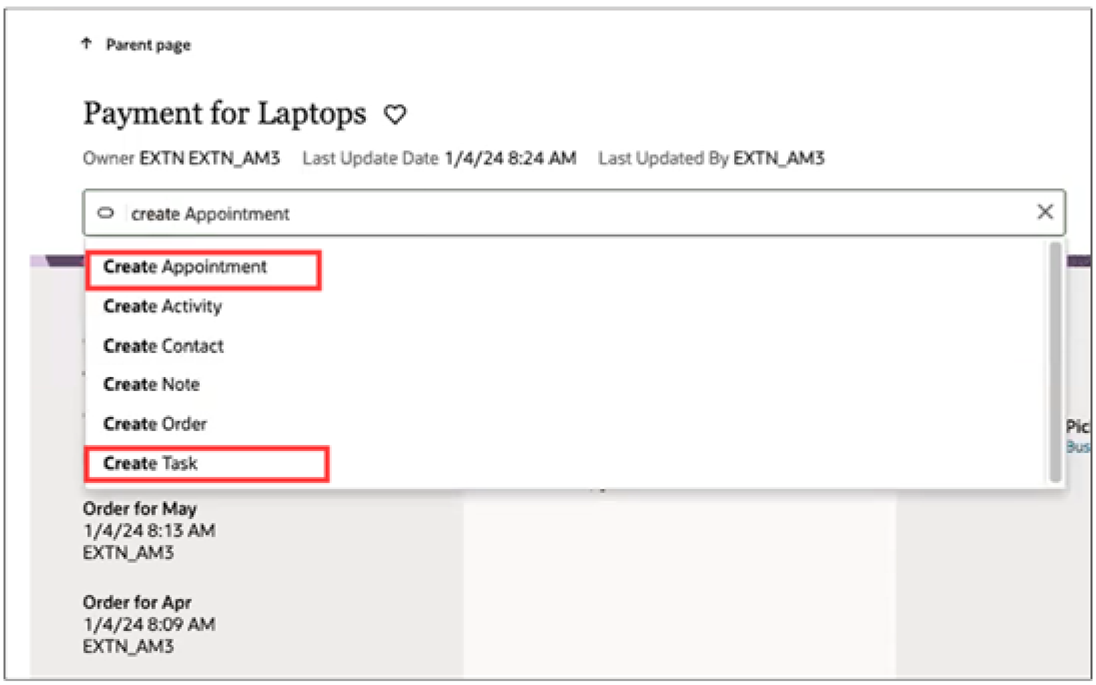
   `https://<servername>/fscmUI/redwood/cx-custom/payment_c/payment_c-list`
3. Change the preview link as follows:

   `https://<servername>/fscmUI/redwood/cx-custom/application/container/payment_c/payment_c-list`

   Note: You must add `/application/container` to the preview link.

   The screenshot below illustrates what the list page looks like with data.

4. Create a task and an appointment by entering **Create Task** and **Create Appointment** in the Action Bar.



   After creating a task and an appointment, view the records you created by entering **Show Tasks** and **Show Appointments** in the Action Bar.
5. In the list page, drill down into the record you created to view the subtabs and actions.

Save Your Work to Git

Save your work by using the Push Git command:

Navigate to the Git tab, review your changes, and do a Git push (which does both a commit and a push to the Git repository).

Create Navigation Menu Entry

After you create a custom application in Oracle Visual Builder Studio, you must create a Navigator entry for your custom application. This topic explains how to add an entry to the Navigator for your custom object.

Prerequisites

1. In Oracle Visual Builder Studio, create the list page for your custom application.
2. In Fusion Applications, create a sandbox with the Structure tool enabled.

Create the Navigator Menu Entry

1. From the sandbox menu bar, click **Tools** > **Structure**.
2. Click **Create** > **Create Page Entry**.
3. Enter these details:
    a. In the Name field, enter the Navigator menu text, such as `Payments`.
    b. In the Icon field, click the search icon to pick an icon for this navigator entry.
    c. Click **OK**.
    d. In the Group field, select the group that makes sense for your business needs, such as **Redwood Sales**.
    e. In the Show on Navigator field, keep the default: **Yes**.
    f. In the Show on Springboard field, keep the default: **Yes**.
    g. In the Mobile Enabled field, keep the default: **No**.
    h. In the Link Type field, select **VB Studio Page**.
    i. In the Focus View ID field, enter `/index.html`.
    j. In the Web Application field, search for and select: **ORA_FSCM_UI**.
    k. Click **OK**.
    l. In the Application Stripe field, enter `crm`.
    m. In the VB Studio Flow field, enter the flow name, `application`.
    n. In the VB App UI field, enter the App UI, `cx-custom`.
    o. In the VB Page Name field, enter the page name including the `container/flow name` prefix, such as `container/payment_c/payment_c-list`.
4. Click **Save and Close**.
5. Open your list page in Visual Builder Studio.
6. Click the Preview button to see your changes in your runtime test environment.

    a. From the list page, click the Home icon at the top of the page.
    b. From the Navigator, click the new Payments entry under Digital Sales.
    c. You should be navigated back to your custom object's list page.

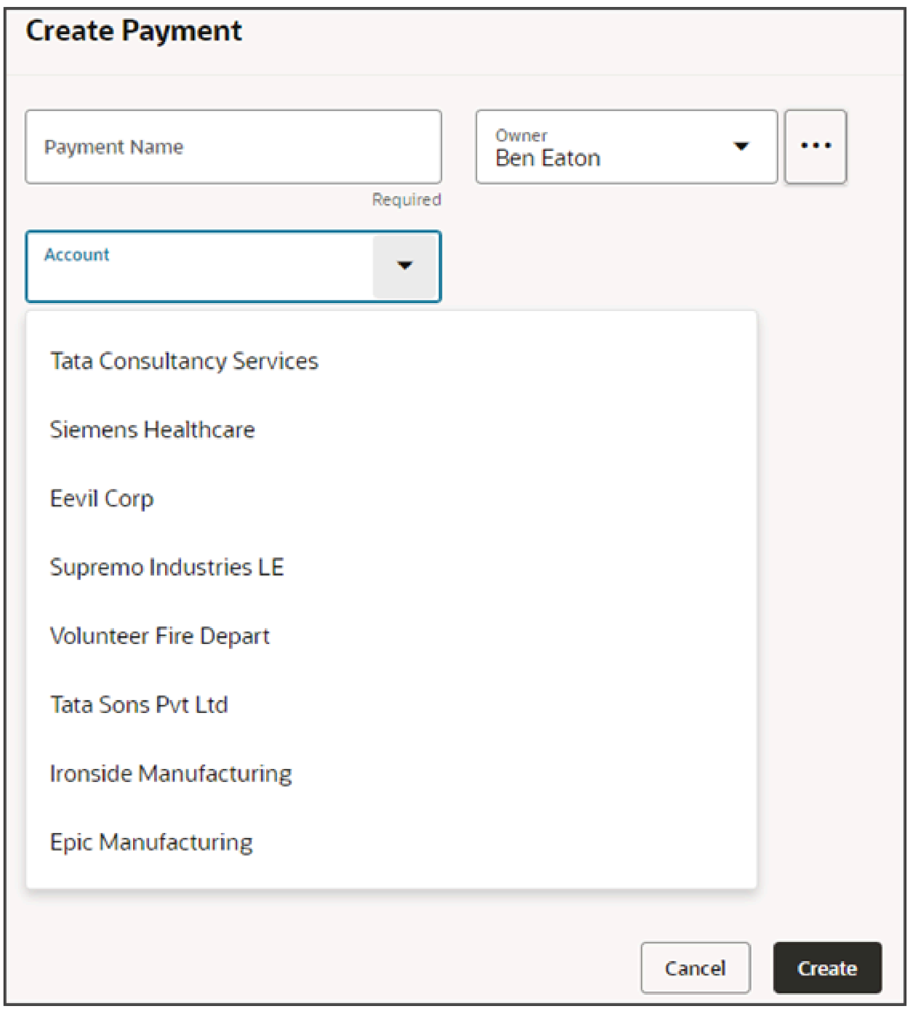
7. Publish your sandbox.

> **Note:** If you need to make changes to this Navigator menu entry in the future, you can do so from a new sandbox.

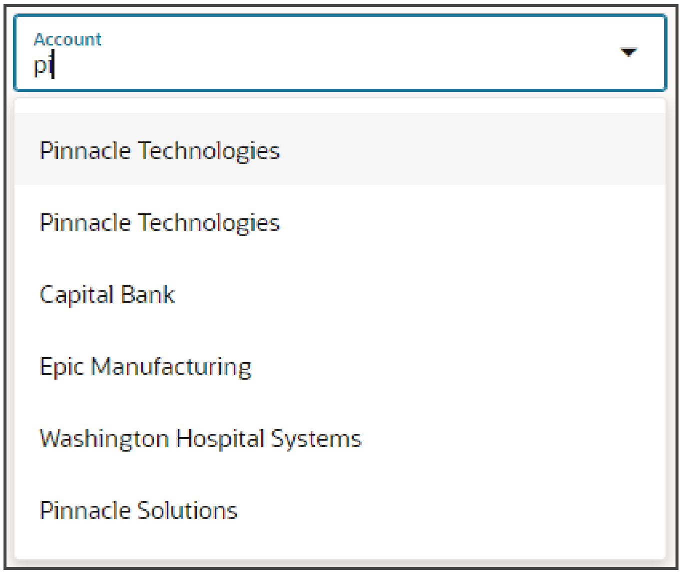
Configure the Picker

A picker enhances a regular list of values field so that users can quickly find the record they need. Depending on setup, pickers can display either a list of saved searches to pick from, or a list of results most relevant to the user's context. Pickers are already available on certain standard fields and can't be modified, although you can configure new pickers for those fields, if needed. You can also configure pickers for custom list of values fields. Use the **cx-picker** fragment in Oracle Visual Builder Studio to configure new pickers.

Here's an example of a field without a picker. Without a picker, the field has a button that users can click to view a list of values.



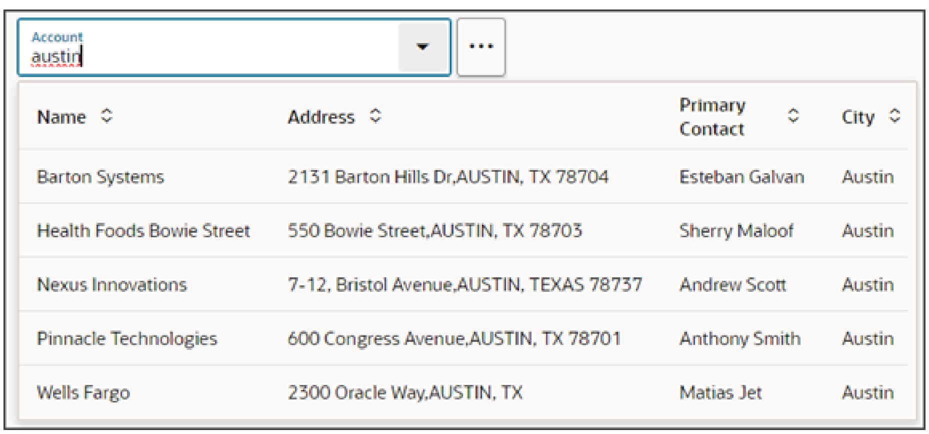
This basic list of values includes the ability to filter on a value that the user enters into the field. For example, if the user enters `pi`, then a list of accounts whose names include `pi` display for selection.



This basic filtering functionality is helpful, but the best practice is to add a picker to enable a wider range of search features on a field.

What's a Picker?

A picker is a special kind of search on a dynamic choice list field. With pickers, users can search on more than one attribute of a record, not just on a single attribute. For example, in an Account picker, users can search on account name, address details, and contact name. The screenshot below illustrates a search on city name.

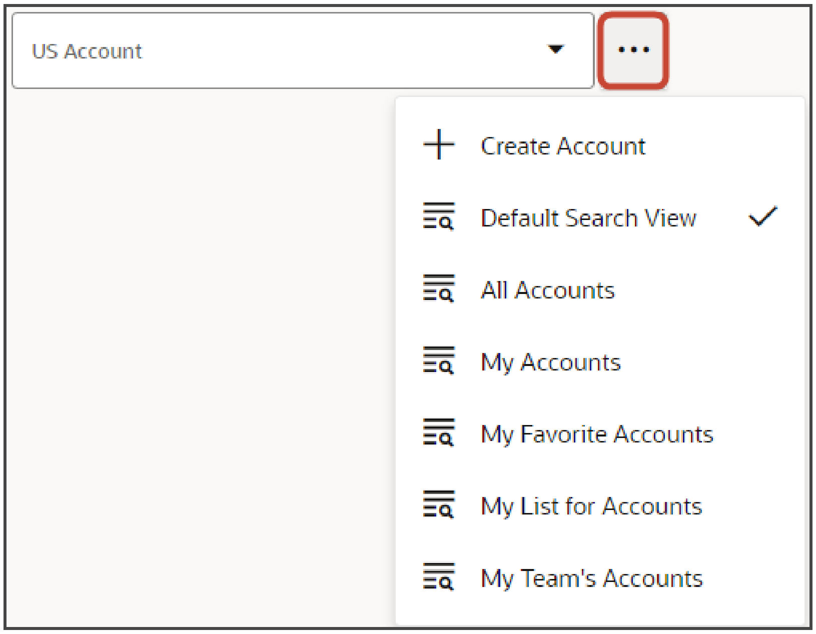


In addition, pickers are more powerful than the standard search on a field because, depending on setup, pickers can display either a list of saved searches or a list of results most relevant to the user's context:

- **Saved searches** are search filters that are predefined for an object. You (and end users) can define new saved searches, if needed.

  If the picker is configured to display a list of saved searches, then the user can apply a saved search at runtime to filter the records available to search and thus more quickly find the right record to select.

  Here's an example of a picker with a list of saved searches:



  At runtime, the picker displays the records based on the default saved search enabled for that object or the last saved search that the user selected.
- If the picker is configured to display a **list of results most relevant to the user's context**, then when the user opens the list of values, they will see records that are related to their current context in some way.

Picker Example Scenario

This topic illustrates a picker configuration example. In this example, we'll add an Account picker to a Create Payment page so that users can search for and associate an account with a payment record.

In this example, you'll do the following:

1. In Application Composer, create a custom dynamic choice list field.
2. In Visual Builder Studio:

   a. Add the dynamic choice list field to a page layout.
   b. Associate the field with a field template that uses the picker fragment.
   c. Configure the picker layout.

Prerequisites

Before creating the custom Account field, you must:

1. Complete the Adaptive Search setup, if working with a custom object.
   If you're configuring a picker for a field on a custom object, then make sure that you've enabled the custom object for Adaptive Search. The operation of a picker depends on what's already set up in Adaptive Search.
   A picker searches against all Adaptive Search fields that are enabled for keyword search. To enable additional attributes for search, see the topic *Make Additional Fields Searchable*. .
2. Create your own workspace in Visual Builder Studio if you don't yet have one.
   If you're configuring a picker for a custom dynamic choice list field that's not yet published, then make sure your workspace is associated with your Cloud Applications sandbox.
3. Add the **Common Application Components** dependency to your workspace.
   To add a dependency, click the Dependencies side tab in Visual Builder Studio.
   Use the search field to find the **Common Application Components** dependency and then click **Add**.
4. This example assumes that you've got a custom Payment object with pages already configured in Visual Builder Studio.
   You can use the CX Extension Generator to set this up quickly. See *Create a New Application Using the CX Extension Generator*.

1. Create the Custom Dynamic Choice List Field

To get started, create a custom dynamic choice list field on a custom object, Payment, in Application Composer. This dynamic choice list field displays account records.

> **Note:** Creating a custom field is a data model change. Create all data model changes in Application Composer before creating application extensions in Visual Builder Studio. You don't have to publish your sandbox before working in Visual Builder Studio, however, since your workspace is associated with your current sandbox.

To create the custom dynamic choice list field:

1. Ensure you're in an active sandbox.
2. In Application Composer, navigate to the Payment object > Fields node.
3. Create a custom dynamic choice list field with these values:

Field	Value
Display Label	**Account**
Name	**Account**
Related Object	**Account**
List Selection Display Value	**Organization Name**

Field	Value

Tip: In Application Composer, you can optionally add a filter to the dynamic choice list field to constrain the values that users see in the picker. For example, you might want the picker to display only accounts that are based in a specific country or city. Adding a filter means that the picker won't display the list of saved searches to users. Also, if you add a filter to the field in Application Composer, then you must publish your sandbox so that the filter works correctly at runtime.

When you create a dynamic choice list field in Application Composer, two fields are created.

- One field is for use with standard, non-Redwood Oracle applications. The naming convention for this standard field is **customfield_c**.

  In this example, the **Account_c** field is automatically created. You can see and modify this field in Application Composer and Visual Builder Studio.
- One field is for use with Redwood Sales. The naming convention for this standard field is **customfield_Id_c**.

  In this example, the **Account_Id_c** field is automatically created and displays in Visual Builder Studio only. This is the field that you add to Redwood Sales page layouts.

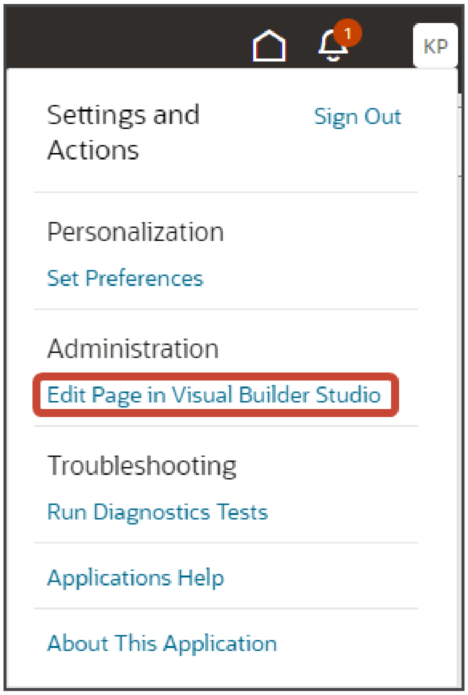
You can now add the **Account_Id_c** field to a page layout in Visual Builder Studio. We'll do that in the next section.

2. Add the Field to a Page Layout

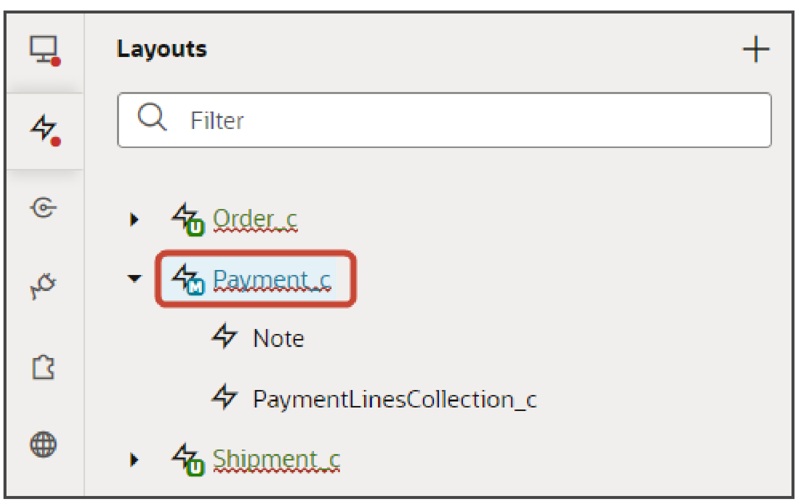
Let's add your custom field to a page layout. In this example, we'll add the field to a create page. Typically, you would also add the field to an edit page.

1. In the Redwood user experience of Sales, navigate to the page that displays the area you want to extend. In this example, navigate to the Payments list page.

2. Under the Settings and Actions menu, select **Edit Page in Visual Builder Studio**.



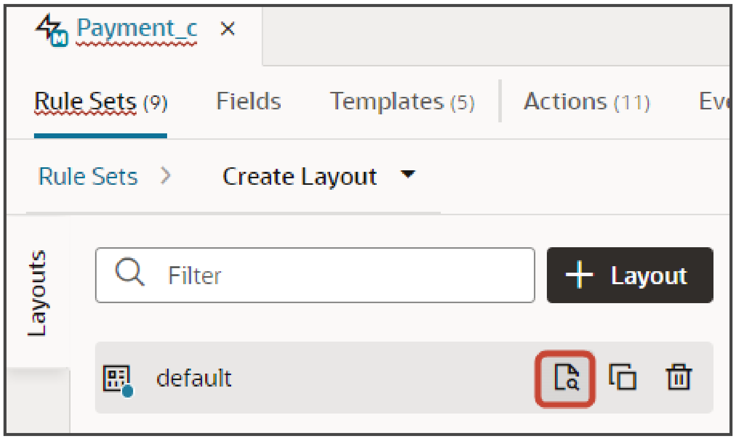
3. Select the project that's already set up for you. If only one project exists, then you will automatically land in that project.
4. Visual Builder Studio automatically opens your workspace. If more than one workspace exists, however, then you must first pick your workspace.
5. When you enter into your workspace in Visual Builder Studio, click the Layouts side tab.
6. On the Layouts side tab, click the Payment_c node.



7. On the Payment_c tab, Rule Sets tab, click the **Create Layout** rule set.

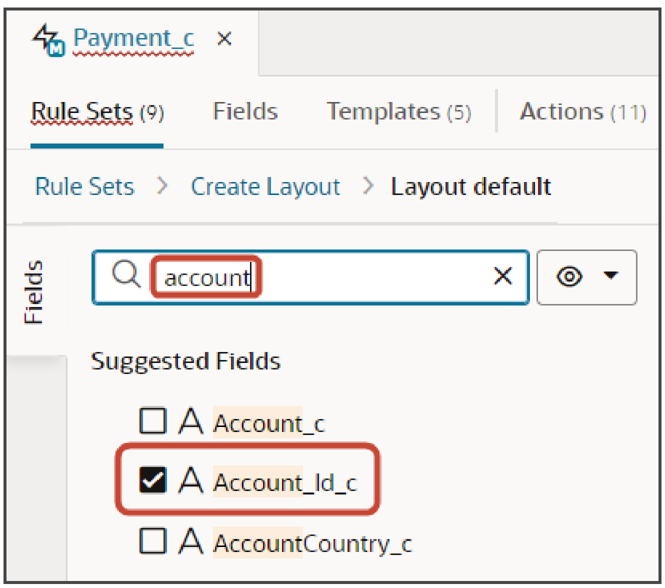
> **Note:** Optionally repeat these same steps for the **Edit Layout** rule set.

8. Click the Open icon to edit the default layout.



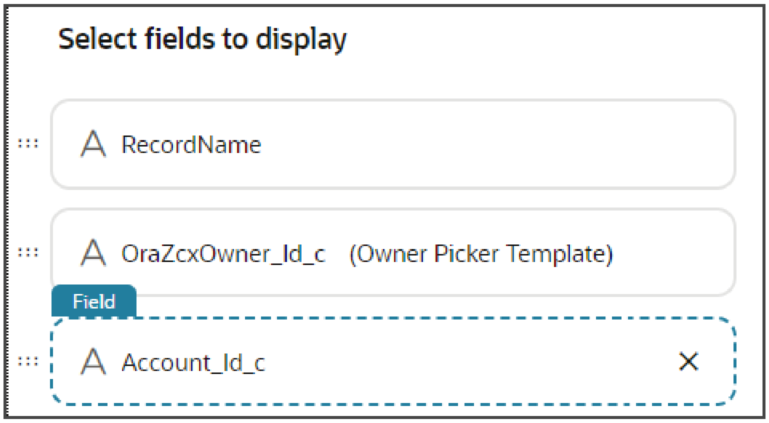
9. Scroll through the list of fields to find your custom dynamic choice list field. Visual Builder Studio shows the internal API name, not the display name.

Tip: To find your field more quickly, use the Filter field. For example, enter `account` into the Filter field.

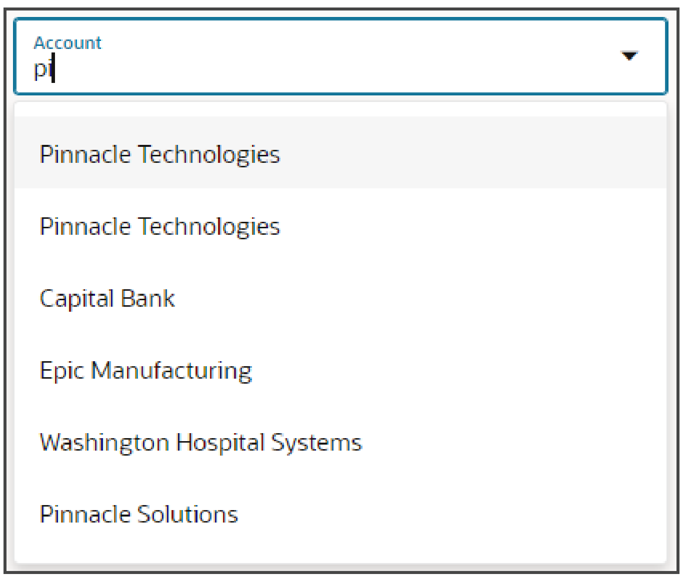


10. Select the field, **Account_Id_c**, from the field list.

    When you select a field, it displays in the list of fields to the right, at the bottom of the list. You can optionally use the field's handle to drag the field to the desired location.

    If you were to preview the create page at this point, then the Account field that displays is a simple list with only basic search filtering.

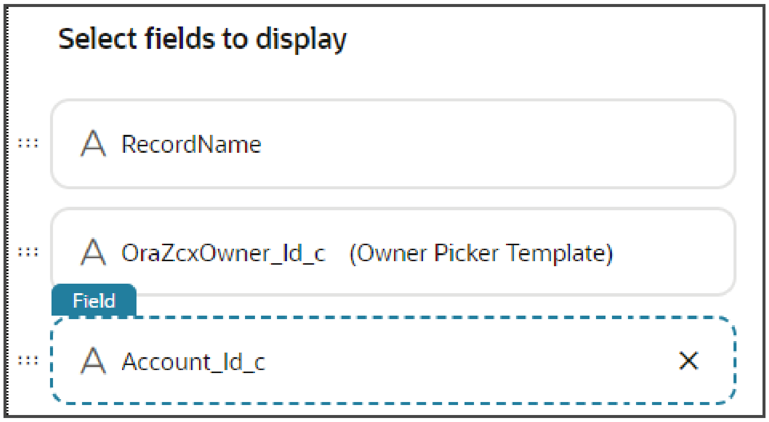
    To add a picker to the field, you must associate the field with a field template that uses the picker fragment. Let's do that next.

3. Associate the Field with a Field Template

Let's add a picker to your custom dynamic choice list field to give users enhanced searching functionality. To do this, you associate the field with a field template that uses the picker fragment.

> **Note:** The following steps illustrate the required picker parameters, but you can set other parameters, as well. See *Parameters for the cx-picker Fragment*.

1. Make sure that you're still on the Rule Sets tab, viewing the default layout.
2. Click the Account_Id_c field.

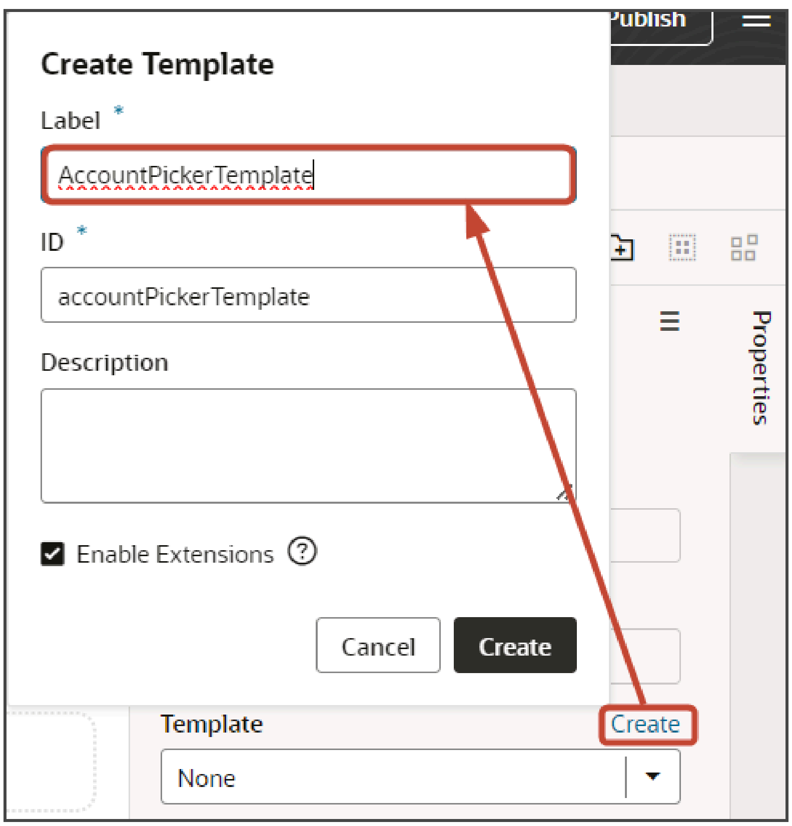


3. On the Properties pane, above the Template field, click **Create**.

> **Note:** If you're doing these steps a second time for the **Edit Layout** rule set's layout, then in the Template field, you don't need to create a field template. Instead, you can select the template that you're about to create in the next step.

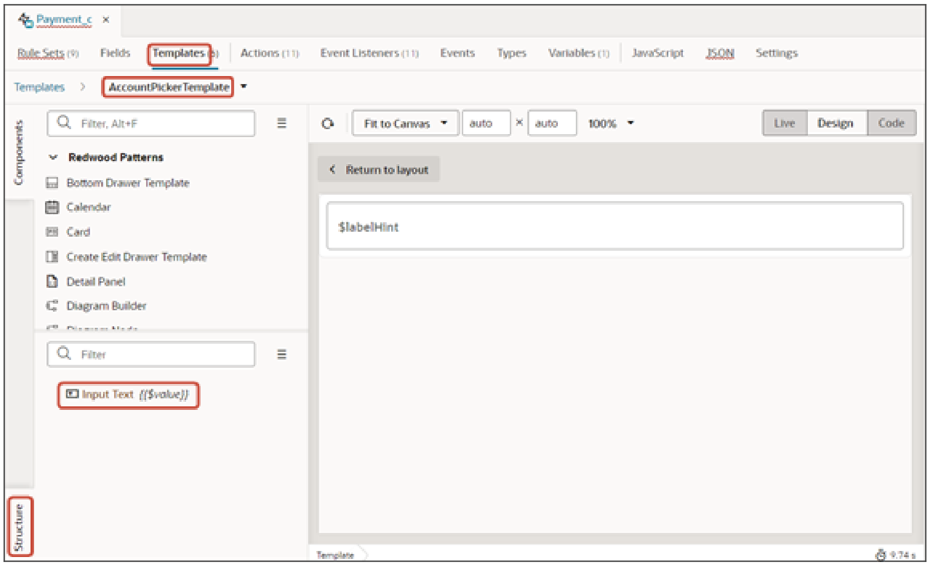
4. In the Create Template dialog, in the Label field, enter a label for the template.

   In this example, enter `AccountPickerTemplate`.



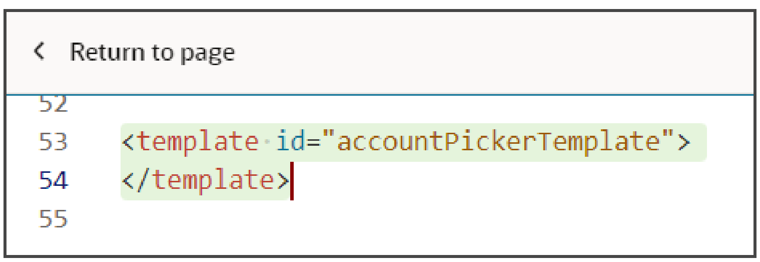
5. Click **Create**.

Visual Builder Studio creates a placeholder template with a basic structure, including an Input Text node which you can see on the Structure pane.

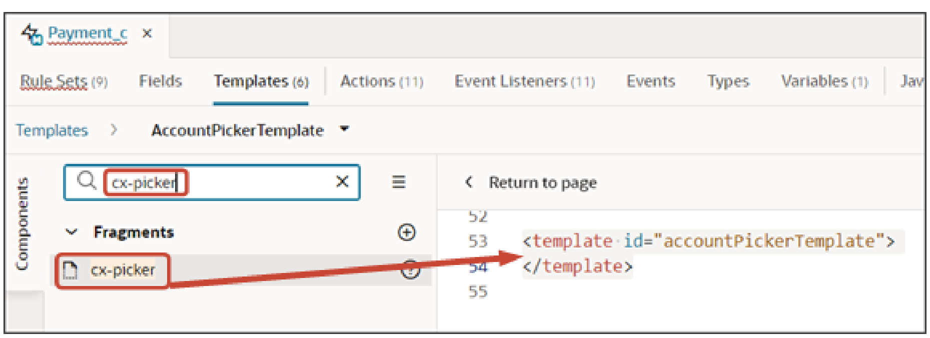


6. Delete the default Input Text node from the Structure pane by right clicking the node and clicking **Delete**.
7. Click the Code button.

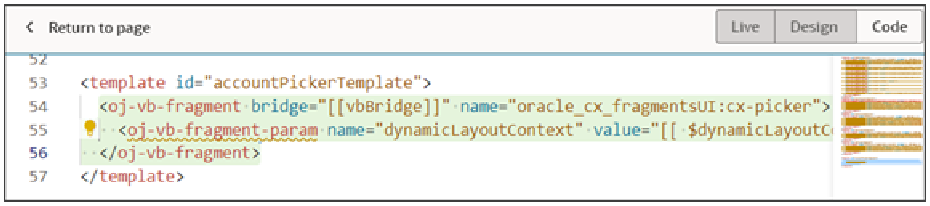
8. In the template editor, select the **accountPickerTemplate** template tags.



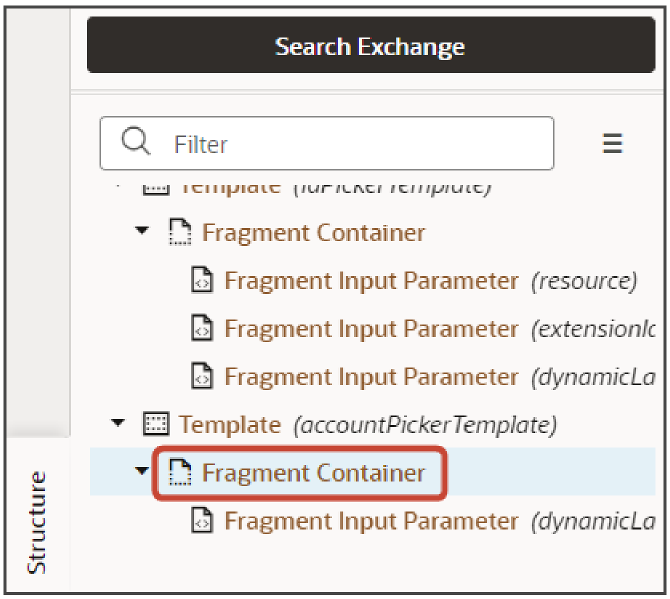
9. On the Components palette, in the Filter field, enter `cx-picker`.
10. Drag and drop the **cx-picker** fragment to the template editor, between the template tags.



11. Make sure the fragment code is selected, as illustrated in this screenshot.



Tip: On the Structure pane, selecting the Fragment Container node for the picker template accomplishes the same thing.



12. On the Properties pane for the cx-picker fragment, in the Input Parameters section, set values for the required picker parameters.

Required Parameters for the cx-picker Fragment

Parameter	Sample Value	Description
dynamicLayoutContext	`[[ $dynamicLayoutContext ]]`	This parameter is set by default and you don't have to change it, provided the field displays on a create or edit page. If this field doesn't display on a create or edit page, then refer to the **dynamicLayoutContext** description in *Parameters for the cx-picker Fragment* for more details.

Parameter	Sample Value	Description
pickerLayoutId	`PickerLayout`	This parameter points to the rule set whose layout controls how the picker looks at runtime. The default value, which you don't have to change, is `PickerLayout`, the ID of the **Picker Layout** rule set that's predefined for each object including custom objects. If you need to create a custom rule set, then create the rule set as a dynamic table and ensure that the values for the Label and ID fields are identical. Then add the ID to this parameter.
resource	`[[ {"name": "accounts","displayField":"Organiz custom","primaryKey":"PartyId" } ]`	Use this parameter to pass the target object name and end point: ◦ `name` - target object name ◦ `endpoint` - target object end point Optionally pass these additional attributes: ◦ `displayField` - the field value that's displayed in the picker field after the user makes a selection. If not provided here, then the picker displays the first field in the picker layout, by default. ◦ `primaryKey` - the field to derive the primaryKey from. If not provided, then the object's primaryKey field will be used. Be sure to update the **resource** parameter with the appropriate values for your use case. For example, replace `accounts` with the REST API name of the object that the dynamic choice list field is based on.

For additional parameters that you can set for the **cx-picker** fragment, see *Parameters for the cx-picker Fragment*.

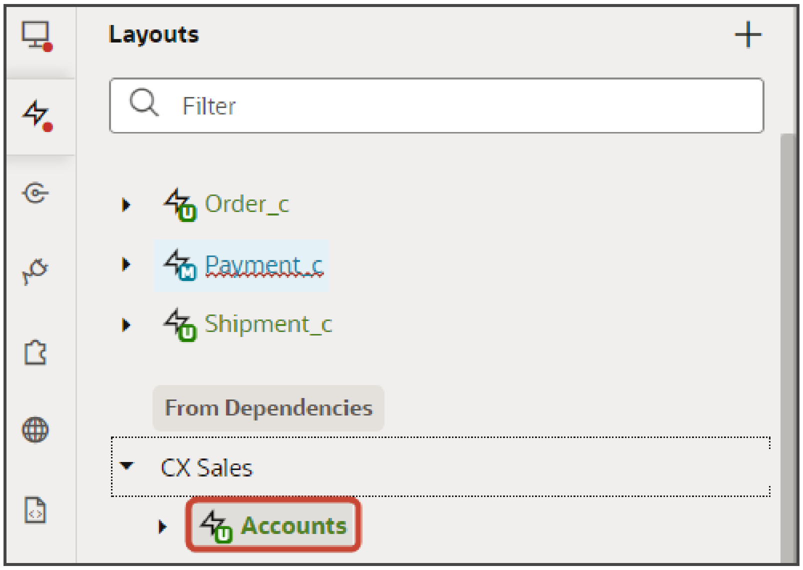
4. Configure the Picker Layout

Finally, select which fields display in the picker by modifying the **Picker Layout** rule set. This rule set's layouts control how the picker looks at runtime.

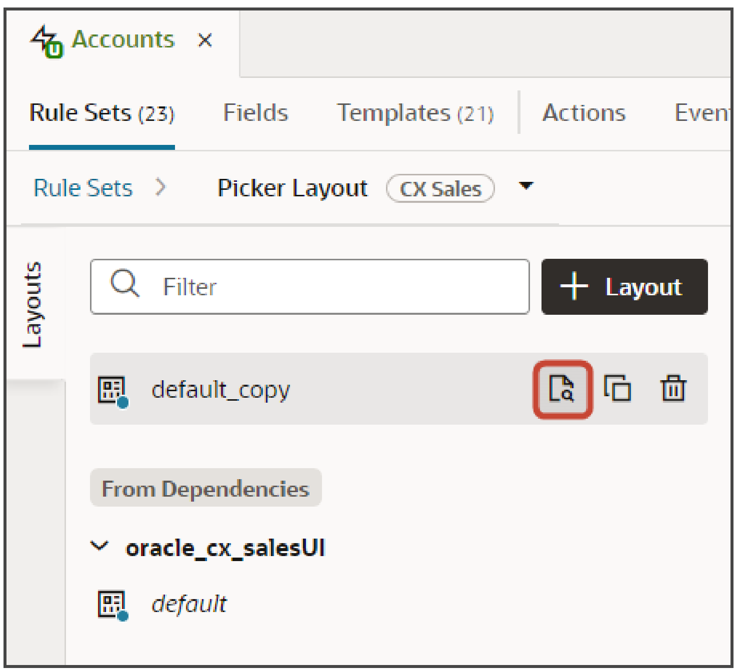
The **Picker Layout** rule set and default layout are predefined for each object, including custom objects.

In this example, we're adding an Account picker which means we must modify the **Picker Layout** rule set for the Account object.

1. On the Layouts side tab, click the **CX Sales** > **Accounts** node.



2. On the Accounts tab, Rule Sets tab, click the **Picker Layout** rule set.
3. Duplicate the **default** layout and then click the Open icon to edit the **default_copy** layout.

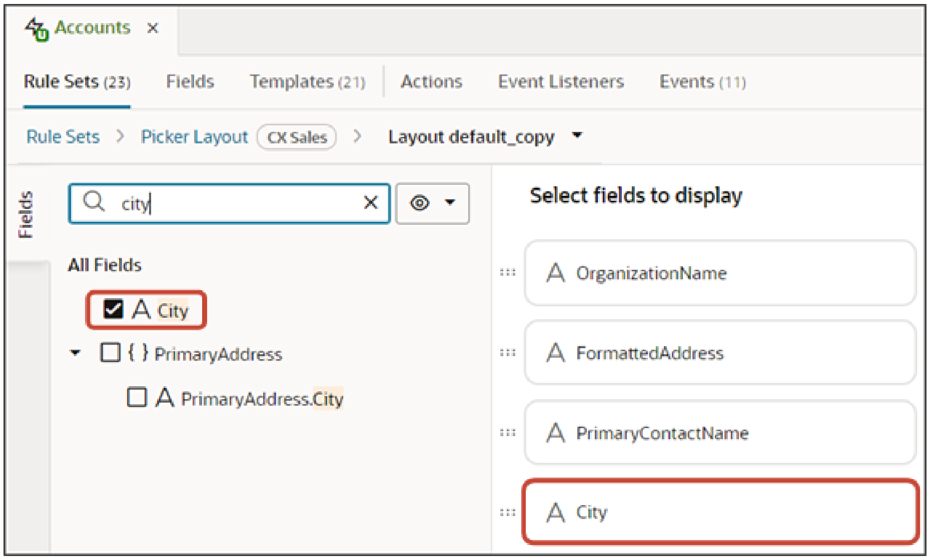


4. Scroll through the list of fields to add any desired fields to the picker layout.

> **Tip:** To find your field more quickly, use the Filter field. For example, enter `city` into the Filter field.

5. Select the field, **City**, from the field list.

   When you select a field, it displays in the list of fields to the right, at the bottom of the list. You can optionally use the field's handle to drag the field to the desired location.



Test the Picker Flow

You can now test the picker that you added to the list of values field.

1. From the payment_c-list page, click the Preview button to see your changes in your runtime test environment.

2. The resulting preview link will be:

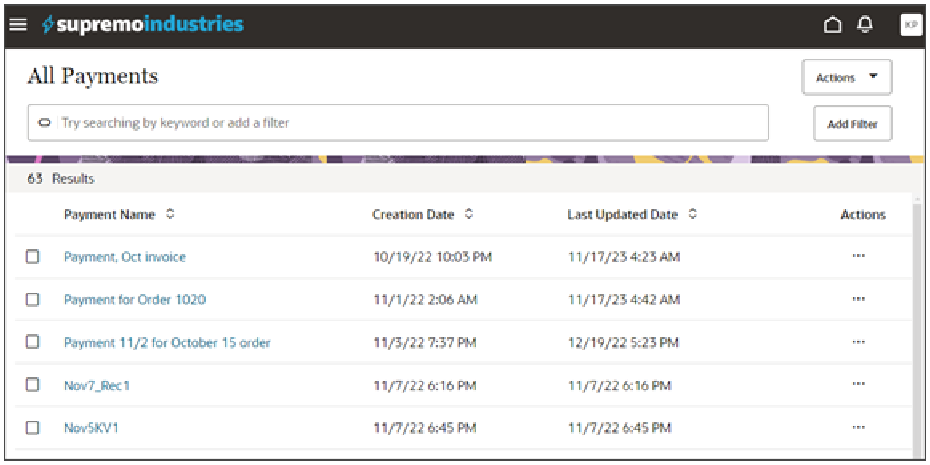
```
https://<servername>/fscmUI/redwood/cx-custom/payment_c/payment_c-list
```

3. Change the preview link as follows:

   `https://<servername>/fscmUI/redwood/cx-custom/application/container/payment_c/payment_c-list`

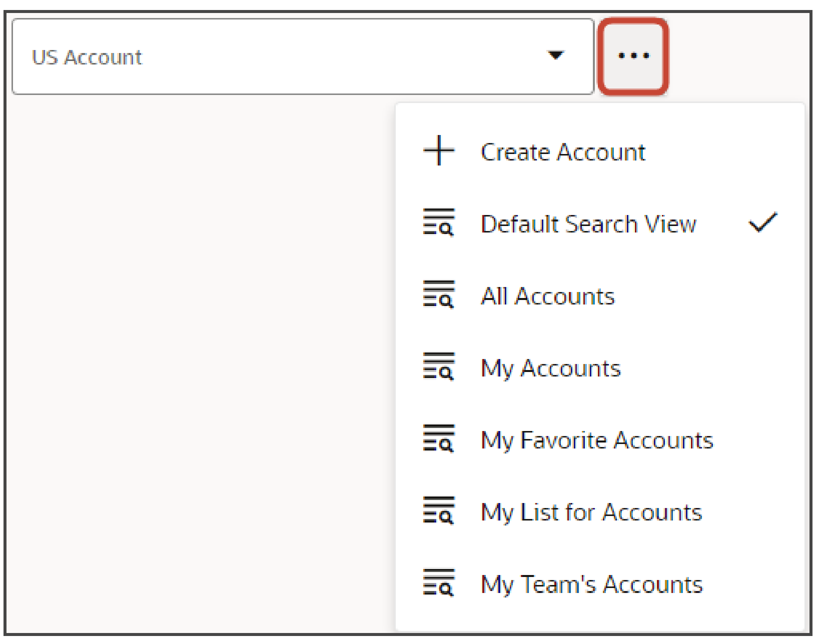
   > **Note:** You must add `/application/container` to the preview link.

   The screenshot below illustrates what the list page looks like with data.

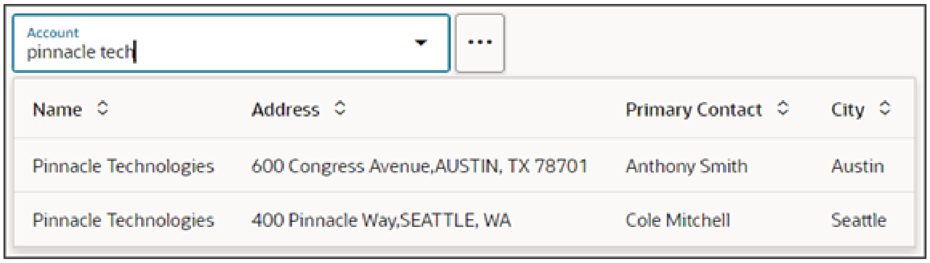


4. In the Action Bar, enter `Create Payment`.
5. Click **Create Payment**.

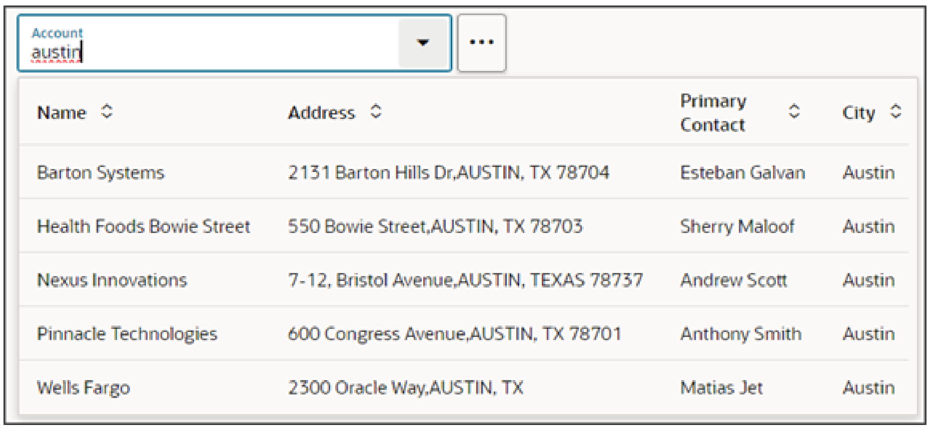
   The Create Payment drawer displays.
6. In the Create Payment drawer, click the three dots next to the Account field to view the list of saved searches.



7. If you enter some text into the field, the picker leverages Adaptive Search to return matched results. In the example below, we've entered `pinnacle tech`.



In the picker, try searching on a city, for example, `austin`, so you can see how you can search on other record attributes.



To understand how the search works, see *Search Within Fields* in the Search and Lists chapter in the Oracle Fusion Cloud Sales Automation Using Digital Sales (Redwood Sales) guide.

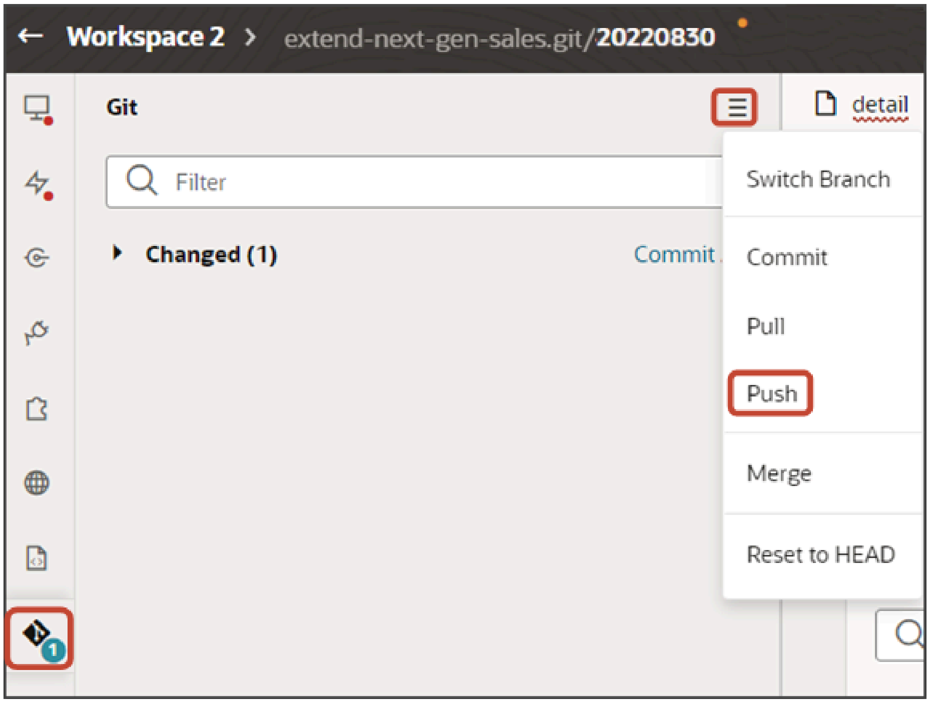
8. Once you're happy with how the picker looks, repeat these same steps for the edit layout.

   When configuring a second layout, you don't have to create a new field template and configure the picker fragment again; you can select the field template that you created in this procedure.

   You also don't need to configure the picker layout a second time.

9. Save your work by using the Push Git command.

   Navigate to the Git tab, review your changes, and do a Git push (which does both a commit and a push to the Git repository).



Parameters for the cx-picker Fragment

This topic lists the parameters that you can set for the **cx-picker** fragment. To learn how to set parameters for this fragment, see *Configure the Picker*.

Required Parameters for the cx-picker Fragment

Parameter	Sample Value	Description
dynamicLayoutContext	`[[ $dynamicLayoutContext ]]`	This parameter is set by default and you don't have to change it if the field displays on a create or edit page. If this field doesn't display on a create or edit page, but rather on a mass update page for example, then you can remove the default value in this parameter.
pickerLayoutId	`PickerLayout`	This parameter points to the rule set whose layout controls how the picker looks at runtime. The default value, which you don't have to change, is `PickerLayout`, the ID of the **Picker Layout** rule set that's predefined for each object including custom objects.

Parameter	Sample Value	Description
		If you need to create a custom rule set, then create the rule set as a dynamic table and ensure that the values for the Label and ID fields are identical. Then add the rule set's ID to this parameter.
resource	`[[ {"name": "accounts","displayField":"Organizat custom","primaryKey":"PartyId" } ]]`	Use this parameter to pass the target object name and end point: • `name` - target object name • `endpoint` - target object end point Optionally pass these additional attributes: • `displayField` - the field value that's displayed in the picker field after the user makes a selection. If not provided here, then the picker displays the first field in the picker layout, by default. • `primaryKey` - the field to derive the primaryKey from. If not provided, then the object's primaryKey field will be used. Be sure to update the **resource** parameter with the appropriate values for your use case. For example, replace `accounts` with the REST API name of the object that the dynamic choice list field is based on.

Optional Parameters for the cx-picker Fragment

Parameter	Sample Value	Description

context	To display one field vs. another, set the **context** parameter to `[[ {'variableName':true} ]]`.
	
	For example:
	`[[ {'showContact':false} ]]`
	
	To allow users to enter text values into this field, add this property to the **context** parameter:
	`[[ {'enablePlainText':true} ]]`
		You can use the context parameter to control picker behavior. For example:
		• Display different fields in a picker at runtime.
		For example, based on the value of this parameter, the picker could display either Field A or Field B.
		To set this up, see *Display Different Fields in a Picker*.
		• Allow the user to enter a text value into the field at runtime, instead of selecting an existing value.

createConfig	`[[ {'enabled':true} ]]`	Indicates if the user can create a record directly from the picker. Specify `true` or `false`. If no value is set, then the "create a record" option is automatically displayed in the picker by default.

ORACLE

<table>
<tr><th>Parameter</th><th>Sample Value</th><th>Description</th></tr>
<tr><td></td><td></td><td></td></tr>
<tr><td>isDefaultSavedSearchEnabled</td><td>This parameter is a checkbox.</td><td>Use this parameter to indicate the default saved search that's automatically applied to the list of values in the picker field.

The checkbox is not selected, by default.

• When not selected, the Default Search View option is automatically selected from the list of saved searches on the picker field.

For accounts and contacts, the default search view is All Accounts and All Contacts.

• When selected, the default selection is always the default saved search enabled for the object, regardless of the last saved search selected.

Note: This parameter is applicable only for objects that are supported by Adaptive Search.</td></tr>
<tr><td>label</td><td><code>US Account</code></td><td>Use this parameter to override the field name on the UI.

This parameter is particularly useful if a filter is defined on the dynamic choice list field in Application Composer or if you pass filter criteria using the query parameter (described below).

For example, if you add a simple filter to the dynamic choice list field to display only accounts in the US, then you can use the label parameter to change the field name. In this example, you might want to change the Account field name to a new field name, <code>US Account</code>.</td></tr>
<tr><td>pickerNameField</td><td><code>{{ $value.pickerNameField }}</code></td><td>Use this parameter in conjunction with the context parameter to support user-entered text in the picker field. Bind this parameter to a virtual field that you create, such as <code>pickerNamefield</code>.

If this field has a simple filter (from Application Composer) or a value in the query parameter, then any value in the pickerNameField parameter is ignored.</td></tr>
<tr><td>query</td><td><code>[{"op":"$co", "attribute":"OrganizationName", "value":"Pinnacle"}]</code></td><td>Use this parameter to filter the records that are available in the picker at runtime.</td></tr>
</table>

Parameter	Sample Value	Description
		For example, you could filter the records to display only accounts with a specific keyword in the name, such as `Pinnacle`. The value for this parameter should be an array[object]. If you have both a simple filter on the dynamic choice list field (from Application Composer) and also a query in this parameter, then this parameter takes precedence. If you pass a value in this parameter, then the list of saved searches won't display in the picker at runtime.
readonly	This parameter is a checkbox.	Use this parameter to make the field read only. For example, you might want to make the field editable only when users first create a record. During edit, however, you might want this field to be read only.
required	This parameter is a checkbox.	This parameter is applicable only when there is no value in the **dynamicLayoutContext** parameter. Use this parameter to make the picker field required. If the **dynamicLayoutContext** parameter does contain a value, however, then you can make the picker field required by selecting the **Required** checkbox on the Fields tab. For example: **Payment_c tab** > **Fields subtab** > **select the Account_Id_c field** > **on the Properties pane, click the Required checkbox**.
selectedRow	`{{ $page.variables.selectedRow }}`	Use this parameter in conjunction with the **value** parameter to make use of other field values from the selected row in the picker. For example, if the user picks a contact, then the contactId is stored in the **value** parameter that's bound to the picker. But, the user can also retrieve the contact's email address using this parameter.
sortCriteria	For example, `[[ [{'attribute': 'PrimaryContact.PartyName', 'direction': 'descending'}] ]]`	Use this parameter to override the default sort criteria in the picker layout. If the user selects a saved search, then the sort criteria comes from this parameter, if provided. If not provided, then the picker honors the sort criteria defined in the saved search. If not defined, then the picker sorts values based on the first column in the picker layout.

Parameter	Sample Value	Description
		If there is a simple filter on the field or if a filter is defined in the **query** parameter, or if the user selects **Related Records**, then the sort criteria comes from this parameter, if provided. If not provided, then the picker sorts values based on the first column in the picker layout.
value	`{{ $page.variables.pickerValue }}`	Use this parameter to store the value selected in the picker field, when there is no value in the **dynamicLayoutContext** parameter. For example, let's say a contact picker is on a page doesn't have a dynamic layout. If the user picks a contact, then the contactId is stored in the **value** parameter that's bound to the picker.

Display Different Fields in a Picker

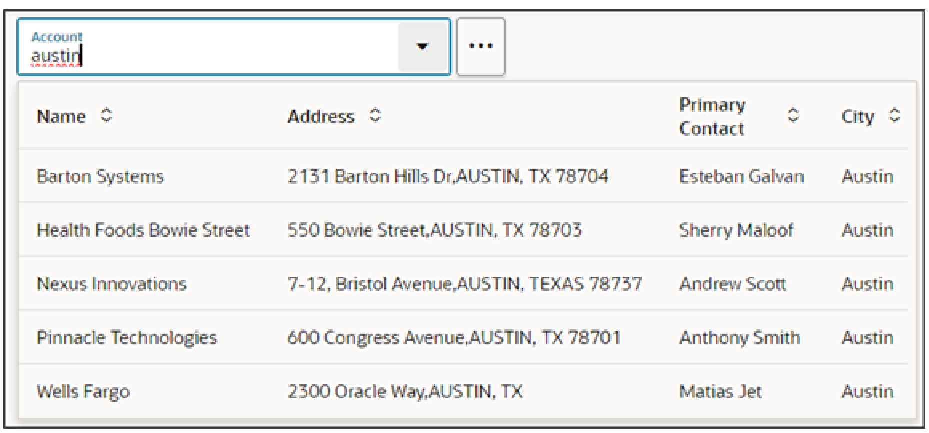
Depending on how you use the **context** parameter in the cx-picker fragment, you can display different fields in the picker at runtime. For example, the picker could display either Field A or Field B.

To display different fields in a picker, use the **context** parameter. You'll also have to do the following:

- create a custom variable
- update the picker layout display properties in the picker object's JSON

Let's look at an example using the Account picker documented in *Configure the Picker*.

In this example, the picker you already created includes both the Address and Primary Contact fields.



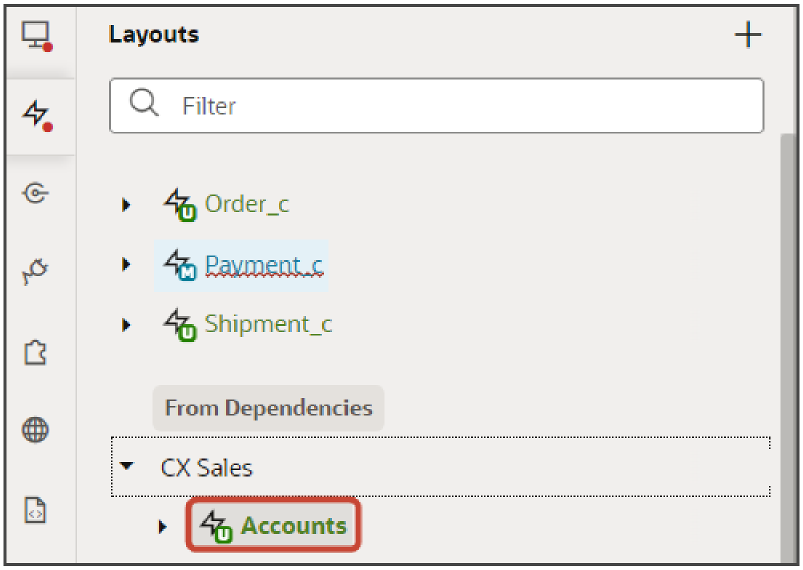
Using the **context** parameter, you can instead show either the Address or Primary Contact field, depending on the value of a custom variable.

Here's how to set this up:

1. Create the Custom Variable

First, create the custom variable on the picker's object:

1. On the Layouts side tab, click the **CX Sales** > **Accounts** node.



2. On the Accounts tab, click the Variables subtab.
3. Click **+ Variable**.
4. In the Create Variable dialog, make sure the Variable option is selected and, in the ID field, enter `showContact`.
5. In the Type field, select **Boolean**.
6. Click **Create**.

2. Add the Condition to the Picker Display Properties

Next, add the condition (which field to show depending on the value of the variable) to the picker layout display properties in the Account object's JSON.

1. On the Accounts tab, click the JSON subtab.

2. Find the picker layout display properties.

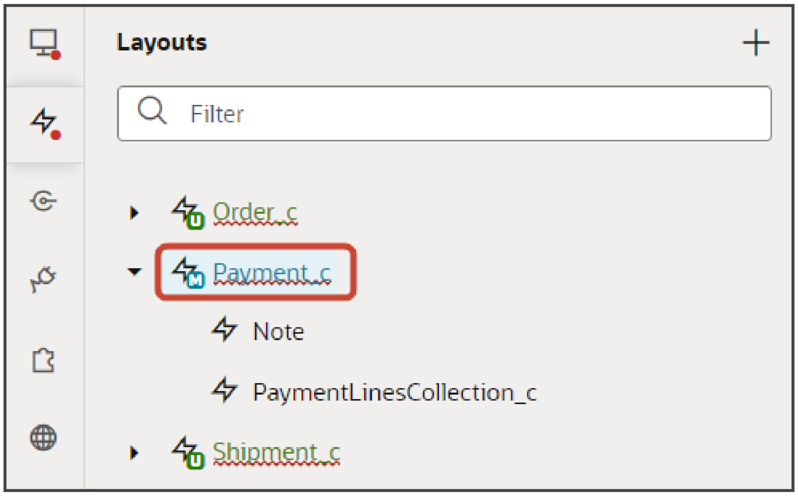
3. In the display properties section, replace the two lines for the `"FormattedAddress"` and `"PrimaryContactName"` fields with a single line:

```
"{{ $componentContext.showContact ? 'PrimaryContactName' : 'FormattedAddress' }}",
```

3. Set the Value of the Variable in the Picker Fragment

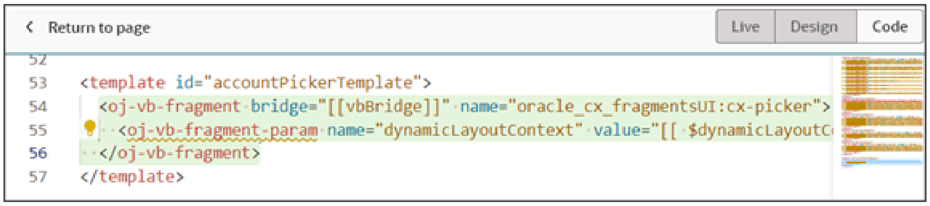
Finally, define the value of the variable (true or false) in the picker fragment itself.

1. On the Layouts side tab, click the Payment_c node.

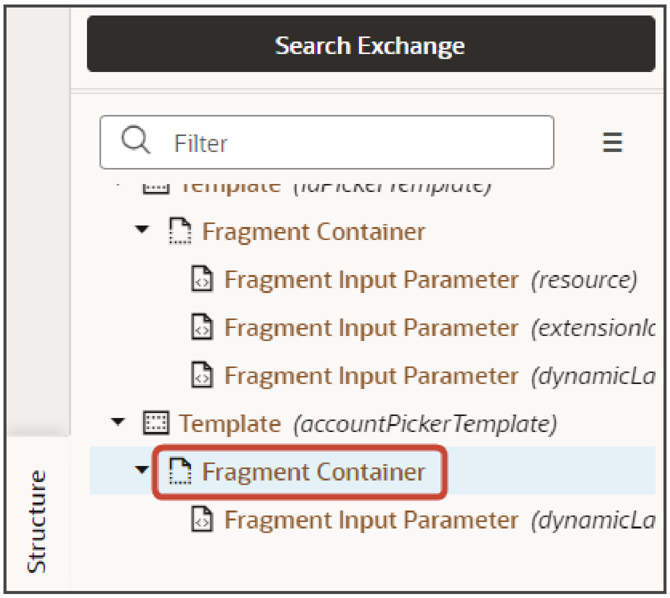


2. On the Payment_c tab, Templates tab, click the **AccountPickerTemplate** template.

3. Make sure the fragment code is selected, as illustrated in this screenshot.



Tip: On the Structure pane, selecting the Fragment Container node for the picker template accomplishes the same thing.



4. On the Properties pane for the cx-picker fragment, in the Input Parameters section, set the value for the **context** parameter. You can set the showContact variable to **true** or **false**:

```
[[ {'showContact':true} ]]
```

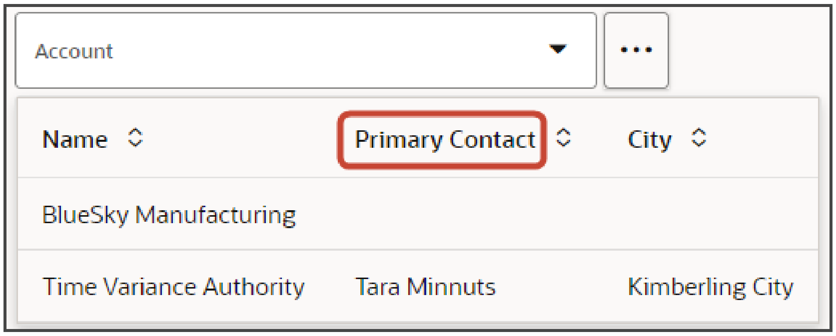
or

```
[[ {'showContact':false} ]]
```

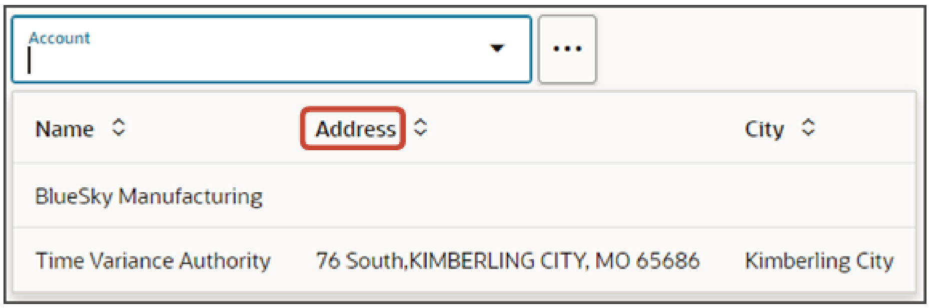
4. Test Your Setup

You can now test each variable setting.

1. Preview your extension and test the picker with the **showContact** variable as **true**:



2. Next, preview your extension and test the picker with the **showContact** variable as **false**:



Add a Mashup to a Page

For any Redwood Sales object, standard or custom, you can configure its detail page to include a mashup that references a publicly available URL. You create the mashup in Oracle Visual Builder Studio.

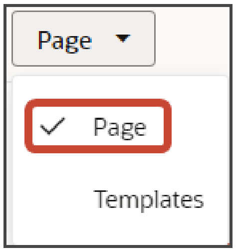
For example, you can add a Wikipedia page to a payment's detail page. At runtime, when the user views a payment, the user can enter **Show Wikipedia** into the Action Bar. The **Show Wikipedia** action lets the user view a related Wikipedia page without having to leave the payment record.

Add a Mashup to a Detail Page

Let's walk through an example of adding a mashup. In this example, we'll add a mashup to a payment's detail page.

1. In Visual Builder Studio, click the App UIs tab.
2. Expand cx-custom > payment_c, then click the payment_c-detail node.
3. On the payment_c-detail tab, click the Page Designer subtab.
4. Click the Code button.

5. Confirm that you are viewing the page in Page Designer.



6. Remove the comment tags for the dynamic container components that contain the panels and subviews.

```
<oj-vb-fragment bridge="[[vbBridge]]" name="oracle_cx_fragmentsUI:cx-detail" class="oj-flex-iten
  <oj-vb-fragment-param name="resources"
    value="[[ {'Payment_c' : {'puid': $variables.puid, 'id': $variables.id, 'endpoint': $applica
  </oj-vb-fragment-param>
  <oj-vb-fragment-param name="header"
    value="[[ {'resource': $flow.constants.objectName, 'extensionId': $application.constants.ext
  </oj-vb-fragment-param>
  <oj-vb-fragment-param name="actionBar"
    value='[[ { "applicationId": "ORACLE-ISS-APP", "resource": {"name": $flow.constants.objectNa
  </oj-vb-fragment-param>
  <oj-vb-fragment-param name="panels"
    value='[[ { "panelsMetadata": $metadata.dynamicContainerMetadata, "view": $page.variables.vi
  </oj-vb-fragment-param>
  <oj-vb-fragment-param name="context" value="[[ {'flowContext': $flow.variables.context} ]]"></
</oj-vb-fragment>
<!--
<oj-dynamic-container layout="PanelsContainerLayout" layout-provider="[[ $metadata.dynamicContai
    class="oj-flex-item oj-sm-12 oj-md-1"></oj-dynamic-container>
<oj-dynamic-container layout="SubviewContainerLayout" layout-provider="[[ $metadata.dynamicConta
  </oj-dynamic-container>
-->
```

7. Highlight the `<oj-dynamic-container>` tags for the subviews.

```
<div class="oj-flex">
  <oj-dynamic-container layout="SubviewContainerLayout" layout-provider="[[ $metadata.dynamicCon
    class="oj-flex-item oj-sm-12 oj-md-12"></oj-dynamic-container>
</div>
```

8. On the Properties pane, in the Case 1 region, click the **Add Section** icon, and then click **New Section**.
9. In the Title field, enter a title for the section, such as `Wikipedia`.
10. In the ID field, change the value to `Wikipedia`.
11. Click **OK**.

12. Manually update the template's JSON with the correct subview name.

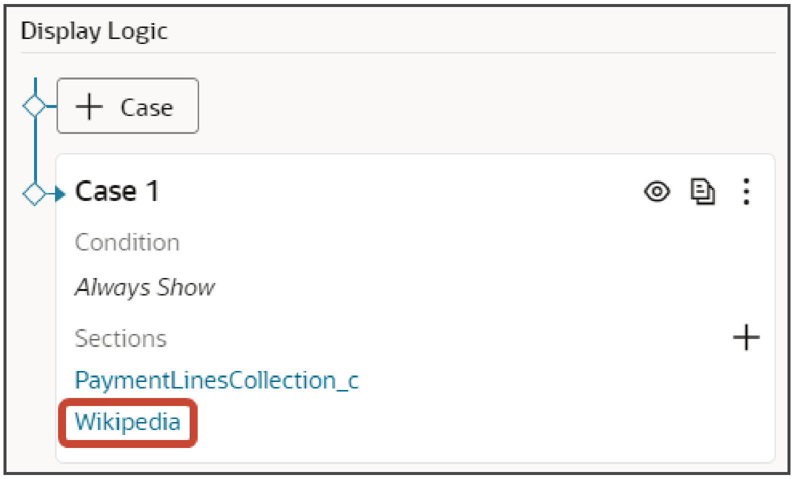
    a. On the payment_c-detail tab, click the JSON subtab.
    b. In the section for the SubviewContainerLayout's section template layout, replace the `sectionTemplateMap` and `displayProperties` values to match the subview's ID name, `Wikipedia`.

       In our example, this is what the SubviewContainerLayout's `sectionTemplateMap` and `displayProperties` should look like:

```
"layouts": {
  "SubviewContainerLayout": {
    "label": "Container Rule Set 1",
    "layoutType": "container",
    "layouts": {
      "case1": {
        "label": "Case 1",
        "layoutType": "container",
        "layout": {
          "sectionTemplateMap": {
            "PaymentLinesCollection_c": "paymentLinesCollectionC",
            "Wikipedia"  "wikipedia"
          },
          "displayProperties": [
            "PaymentLinesCollection_c",
            "Wikipedia"
          ]
        }
      }
    },
    "rules": [
      "containerLayout1-rule2"
    ]
  },
```

13. Still on the payment_c-detail tab, click the Page Designer tab.

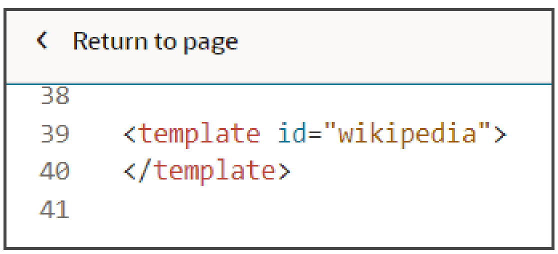
14. On the Properties pane, click the Wikipedia section that you just added.



Page Designer navigates you to the template editor, still on the payment_c-detail tab, where you can design the mashup template.

15. Click the Code button.

16. In the template editor, find the mashup template tags.



17. Add the following parameters to the fragment code so that the code looks like the below sample. Be sure to update the values for the `title` and `url` parameters as needed.

```
<template id="wikipedia">
 <oj-vb-fragment bridge="[[vbBridge]]" name="oracle_cx_fragmentsUI:cx-url">
 <oj-vb-fragment-param name="dynamicLayoutContext" value="{}"></oj-vb-fragment-param>
 <oj-vb-fragment-param name="mode" value="embedded"></oj-vb-fragment-param>
 <oj-vb-fragment-param name="title" value="Wikipedia"></oj-vb-fragment-param>
 <oj-vb-fragment-param name="url" value="https://en.wikipedia.org/wiki/"></oj-vb-fragment-param>
 </oj-vb-fragment>
</template>
```

This table describes the parameters that you can provide for a mashup:

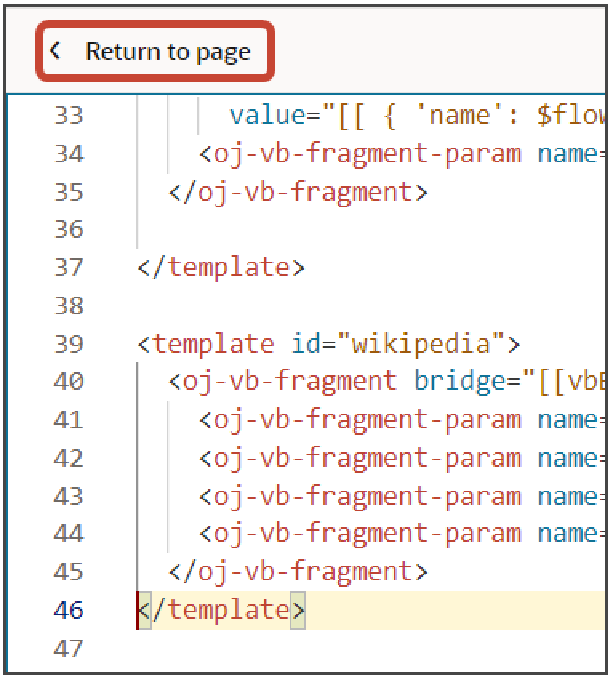
Parameters for Mashup

Parameter Name	Description
title	Enter the title of the mashup, which displays in the subview UI.

Parameter Name	Description
url	Enter the mashup's URL.

18. Comment out the dynamic container component from the payment_c-detail page.

    a. Click **< Return to page**.



    b. Click the Code button.

    c. Comment out the dynamic container components that contain the panels and subviews.

> **Note:** To add more subviews, you must first un-comment the dynamic container component so that you can add a new section for each desired subview.

19. From the payment_c-list page, click the Preview button to see your changes in your runtime test environment.

20. The resulting preview link will be:

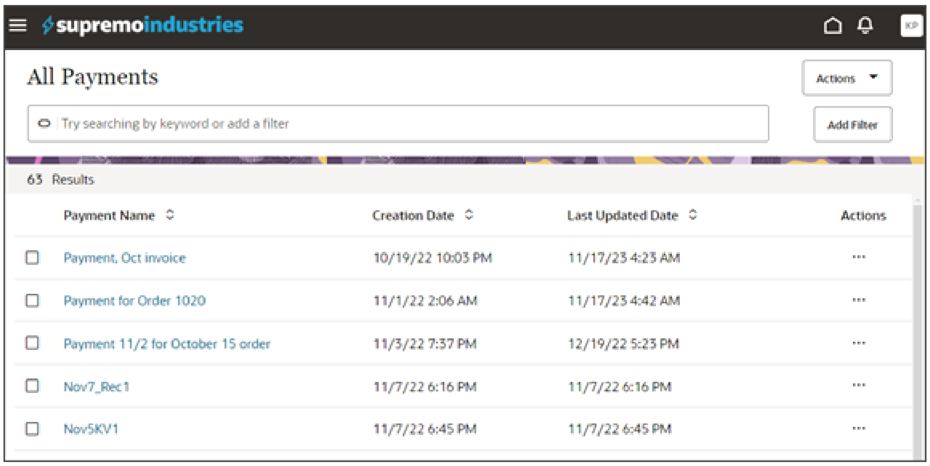
```
https://<servername>/fscmUI/redwood/cx-custom/payment_c/payment_c-list
```

21. Change the preview link as follows:

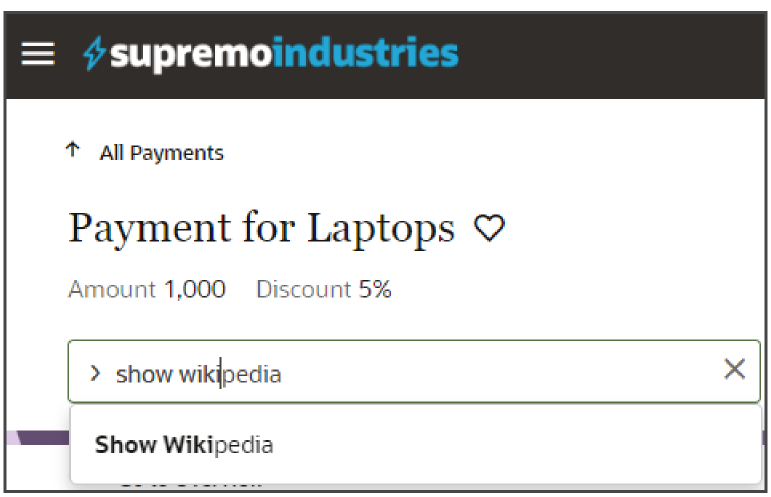
```
https://<servername>/fscmUI/redwood/cx-custom/application/container/payment_c/payment_c-list
```

> **Note:** You must add `/application/container` to the preview link.

The screenshot below illustrates what the list page looks like with data.

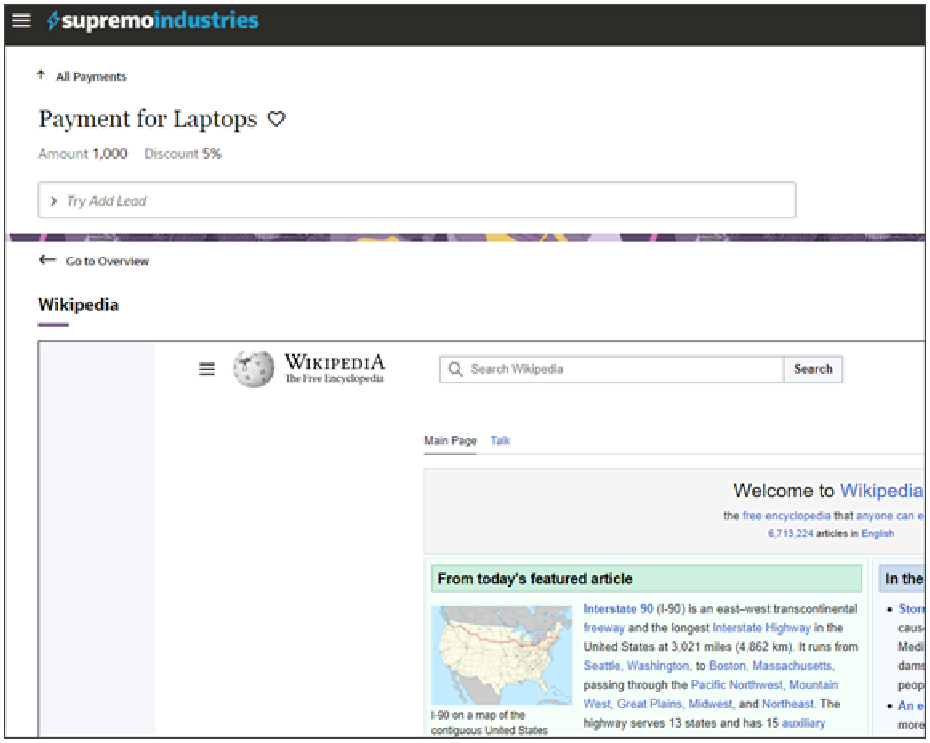


22. If data exists, you can click any record on the list page to drill down to the detail page. The detail page, including header region and panels, should display.
23. In the Action Bar, enter **`Show Wikipedia`**.



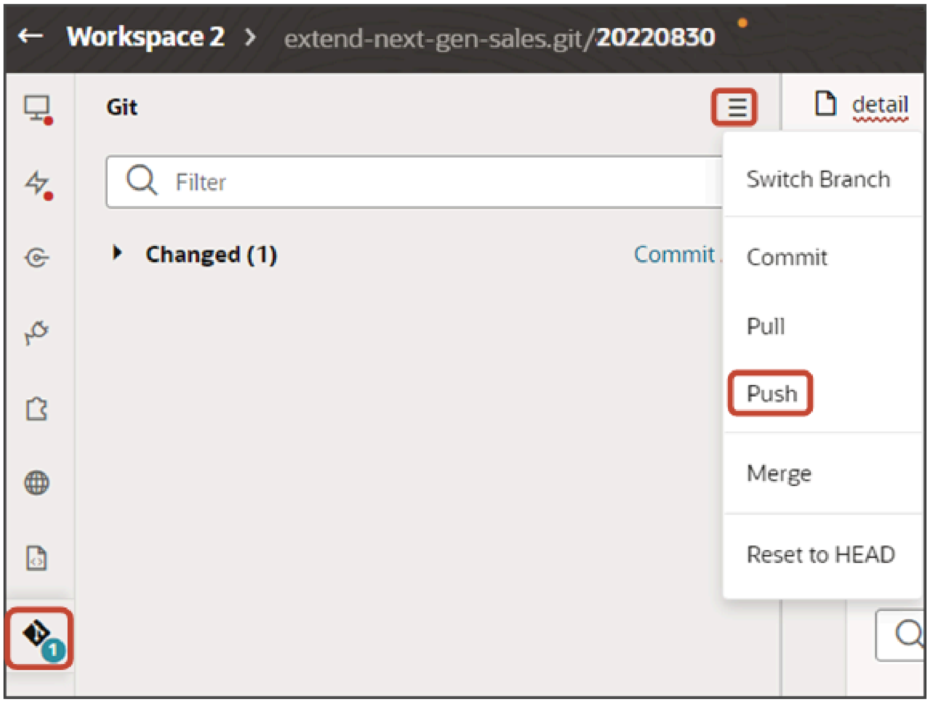
24. Click **Show Wikipedia**.

    The Wikipedia mashup displays:



25. Save your work by using the Push Git command.

    Navigate to the Git tab, review your changes, and do a Git push (which does both a commit and a push to the Git repository).

Add a Rollups Region to a Panel

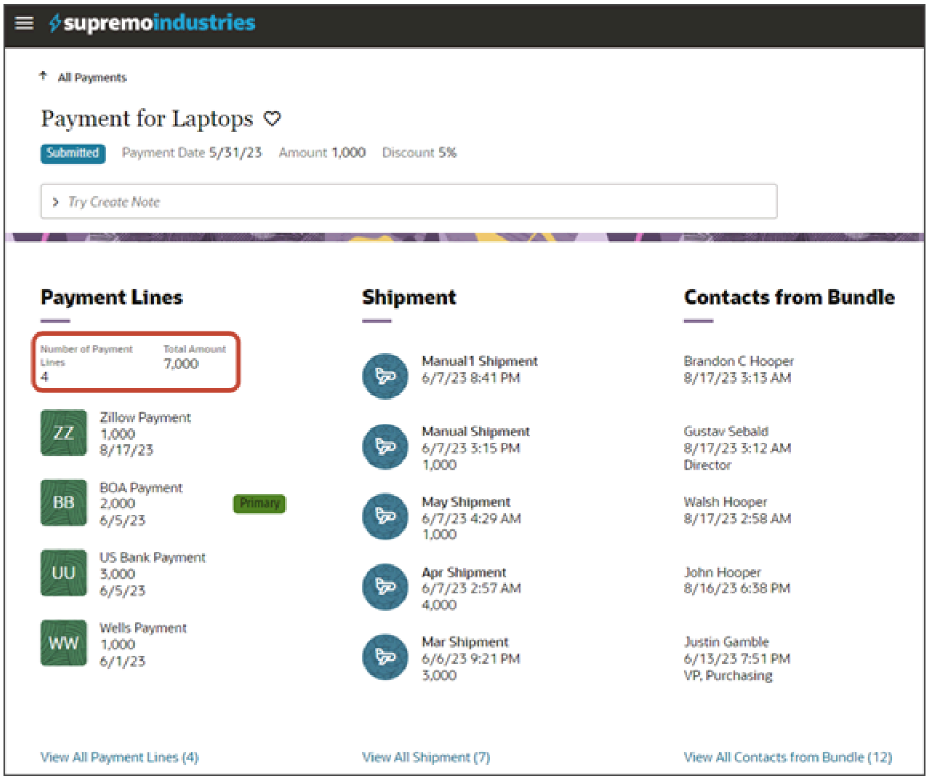
Rollups summarize data across records, for an attribute of a business object and its related objects. The summarized value of a rollup appears as a business metric inside a panel on an object's detail page. You can add new rollups to a panel using Oracle Visual Builder Studio.

Where Do Rollups Appear?

Rollups appear inside panels on an object's detail page.

You can add rollups, either predefined or custom, to panels for both custom and standard objects. Some panels for standard objects are already delivered with a rollups region.

Here's a screenshot of a rollup that displays in a panel for a payment.



Prerequisites

You can add a predefined or custom rollup to a panel.

Before adding a custom rollup, you must first create the custom rollup.

1. In Application Composer:
   - For the desired object, create a rollup object and fields.
   - Then, create and publish the fields as rollups.
2. In the Sales Setup and Maintenance work area, in the Configure Adaptive Search task, enable the rollup object and attributes.

See *What are the steps to set up rollups?*.

In the following example, we'll use a rollup object, called RollupObject, created for the Payment object. The RollupObject object has these fields:

- Number of Payment Lines (number field)
- Total Amount (currency field)

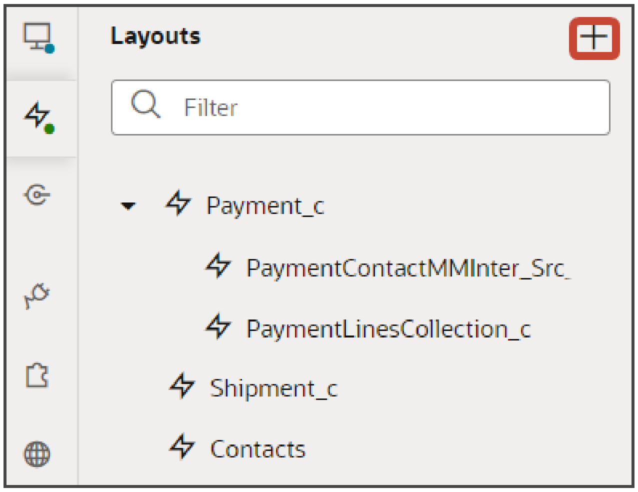
Create the Rollup Layout and Rule Set

To add a rollup to a panel, you must first create a layout for the rollup. You can then add the rollup layout to the panel.

Let's look at an example of adding a rollup to the Payment Lines panel on a payment's detail page.

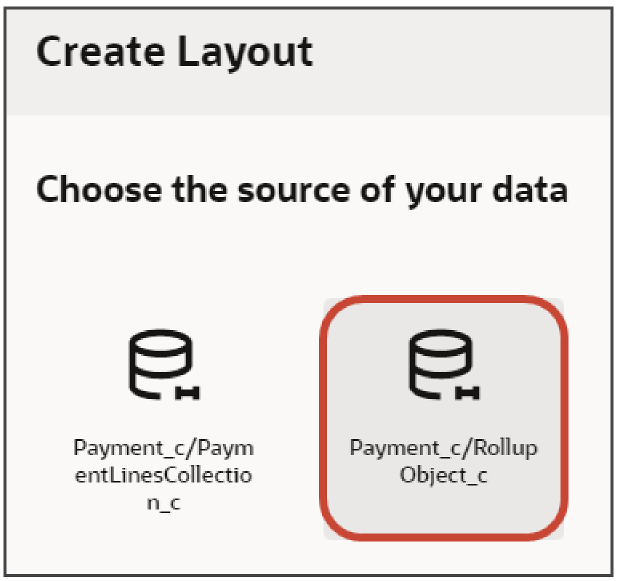
First, create the rollup layout:

1. In Visual Builder Studio, click the Layouts tab, then click the Create Layout icon.



2. In the Create Layout dialog, click the REST resource for your child object.

   In our example, the rollup object is called **RollupObject**. So, expand cx-custom and click **Payment_c/RollupObject_c**.

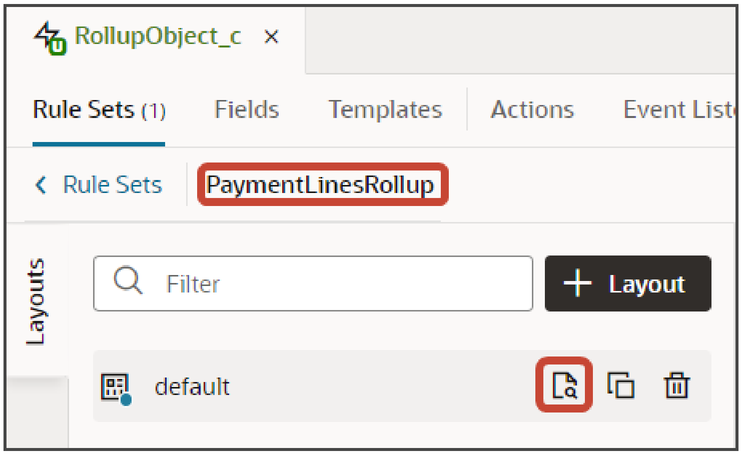


3. Click **Create**.

Next, create the associated rule set.

1. On the RollupObject_c layout tab, click **+ Rule Set** to create a new rule set for the layout.
   a. In the Create Rule Set dialog, in the Component field, select **Dynamic Form**.
   b. In the Label field, enter **PaymentLinesRollup**.
   c. In the ID field, change the value to **PaymentLinesRollup**.
   d. Click **Create**.
2. Add the rollup fields to the layout.

a. Click the Open icon next to the **default** layout.

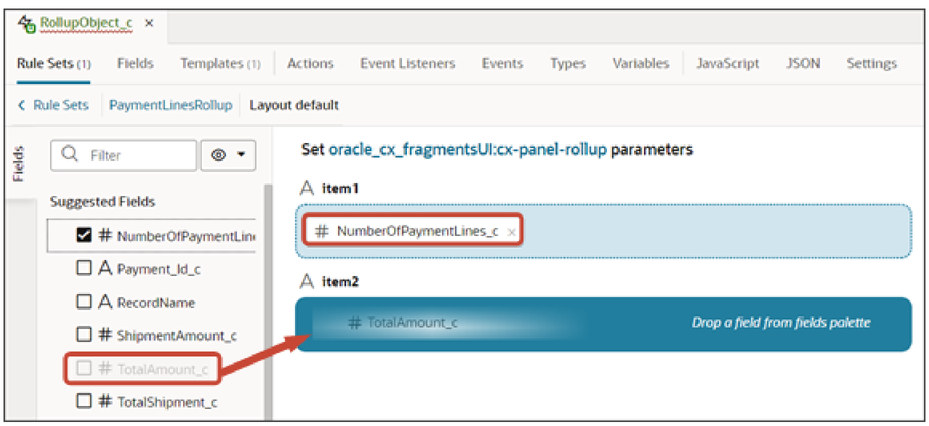


b. Click the **cx-panel-rollup** fragment.

This fragment provides the format for the rollup region.

c. The rollup layout includes two slots. From the list of fields, drag a rollup field to the desired slot.

For example, drag the TotalAmount_c field to the item2 slot.



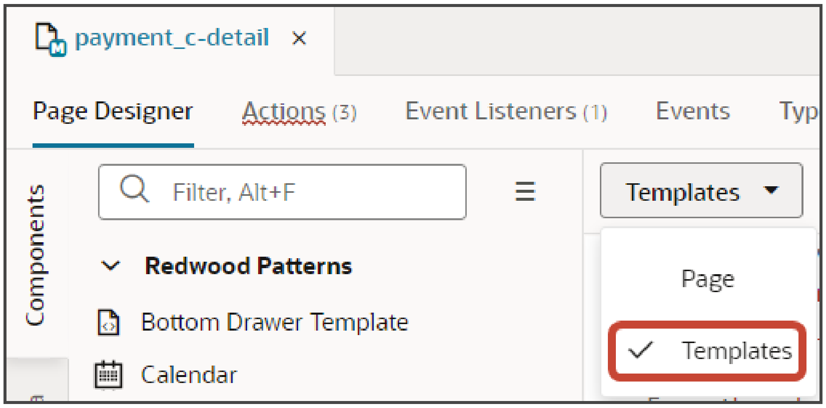
Add the Rollups Region to the Panel

In the previous section, you configured the rollups region using a layout and rule set.

Next, add the rollups region to a panel by adding a parameter to the panel's page and template. Here's how:

1. In Visual Builder Studio, click the App UIs tab.
2. Expand cx-custom > payment_c, then click the payment_c-detail node.
3. Click the payment_c-detail tab, then click the Page Designer subtab.
4. Click the Code button.

5. Select **Templates** from the drop-down list.



6. Add the following parameter to the fragment code.

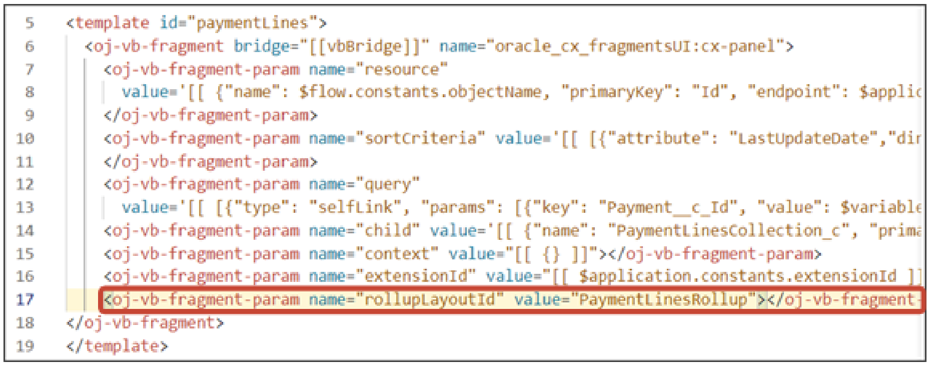
```
<oj-vb-fragment-param name="rollupLayoutId" value="PaymentLinesRollup"></oj-vb-fragment-param>
```

Be sure to replace the `rollupLayoutId` parameter's value with the appropriate value.

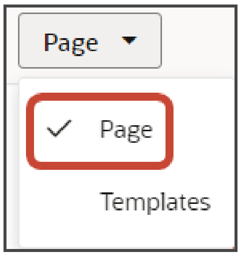
The resulting template code should look something like this:

```
<template id="paymentLines">
 <oj-vb-fragment bridge="[[vbBridge]]" name="oracle_cx_fragmentsUI:cx-panel">
 <oj-vb-fragment-param name="resource"
 value='[[ {"name": $flow.constants.objectName, "primaryKey": "Id", "endpoint":
 $application.constants.serviceConnection } ]]'>
 </oj-vb-fragment-param>
 <oj-vb-fragment-param name="sortCriteria" value='[[ [{"attribute": "LastUpdateDate","direction":
 "desc" }] ]]'>
 </oj-vb-fragment-param>
 <oj-vb-fragment-param name="query"
 value='[[ [{"type": "selfLink", "params": [{"key": "Payment__c_Id", "value": $variables.id }]}] ]]'></
oj-vb-fragment-param>
 <oj-vb-fragment-param name="child" value='[[ {"name": "PaymentLinesCollection_c", "primaryKey": "Id",
 "relationship": "Child"} ]]'></oj-vb-fragment-param>
 <oj-vb-fragment-param name="context" value="[[ {} ]]"></oj-vb-fragment-param>
 <oj-vb-fragment-param name="extensionId" value="{{ $application.constants.extensionId }}"></oj-vb-
fragment-param>
 <oj-vb-fragment-param name="rollupLayoutId" value="PaymentLinesRollup"></oj-vb-fragment-param>
</oj-vb-fragment>
```

```
</template>
```



7. Select **Page** from the drop-down list.



8. Replace the existing `resource` parameter with the following code:

```
<oj-vb-fragment-param name="resource"
 value="[[ {'name':'Payment_c', 'puid': $variables.puid, 'id': $variables.id, 'endpoint':
 $application.constants.serviceConnection ,'extensionId': $application.constants.extensionId, 'rollup':
 'RollupObject_c'} ]]">
```

Be sure to replace all attribute values with the appropriate values for your scenario.

The resulting code should look something like this:

```
<oj-vb-fragment bridge="[[vbBridge]]" name="oracle_cx_fragmentsUI:cx-detail" class="oj-flex-item oj-
sm-12 oj-md-11"
 on-view-change-event="[[$listeners.fragmentViewChangeEvent]]">
 <oj-vb-fragment-param name="resource"
 value="[[ {'name':'Payment_c', 'puid': $variables.puid, 'id': $variables.id, 'endpoint':
 $application.constants.serviceConnection ,'extensionId': $application.constants.extensionId, 'rollup':
 'RollupObject_c'} ]]">
 <oj-vb-fragment-param name="header" value="[[ {'resource': $flow.constants.objectName, 'extensionId':
 $application.constants.extensionId } ]]"></oj-vb-fragment-param>
 <oj-vb-fragment-param name="actionBar" value='[[ { "applicationId": "ORACLE-ISS-APP",
 "subviewLabel": {"PaymentContactMMInter_Src_Payment_cToPaymentContactMMInter_c_Tgt":
 $translations.CustomBundle.Contacts()}, "resource": {"name": $flow.constants.objectName, "primaryKey":
 "Id", "puid": "Id", "value": $variables.puid }} ]]'></oj-vb-fragment-param>
 <oj-vb-fragment-param name="panels" value='[[ { "panelsMetadata": $metadata.dynamicContainerMetadata,
 "view": $page.variables.view } ]]'></oj-vb-fragment-param>
 <oj-vb-fragment-param name="context" value="[[ {'flowContext': $flow.variables.context} ]]"></oj-vb-
fragment-param>
 <oj-vb-fragment-param name="row" value="{{ $variables.row }}"></oj-vb-fragment-param>
</oj-vb-fragment>
```

Test Your Panel

Test the rollups by previewing your application extension from the payment_c-list page.

1. From the payment_c-list page, click the Preview button to see your changes in your runtime test environment.

2. The resulting preview link will be:

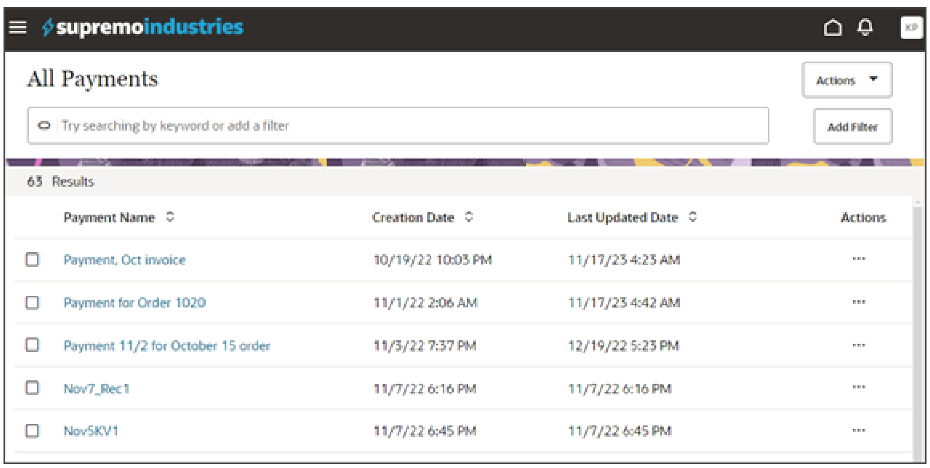
   `https://<servername>/fscmUI/redwood/cx-custom/payment_c/payment_c-list`

3. Change the preview link as follows:

   `https://<servername>/fscmUI/redwood/cx-custom/application/container/payment_c/payment_c-list`

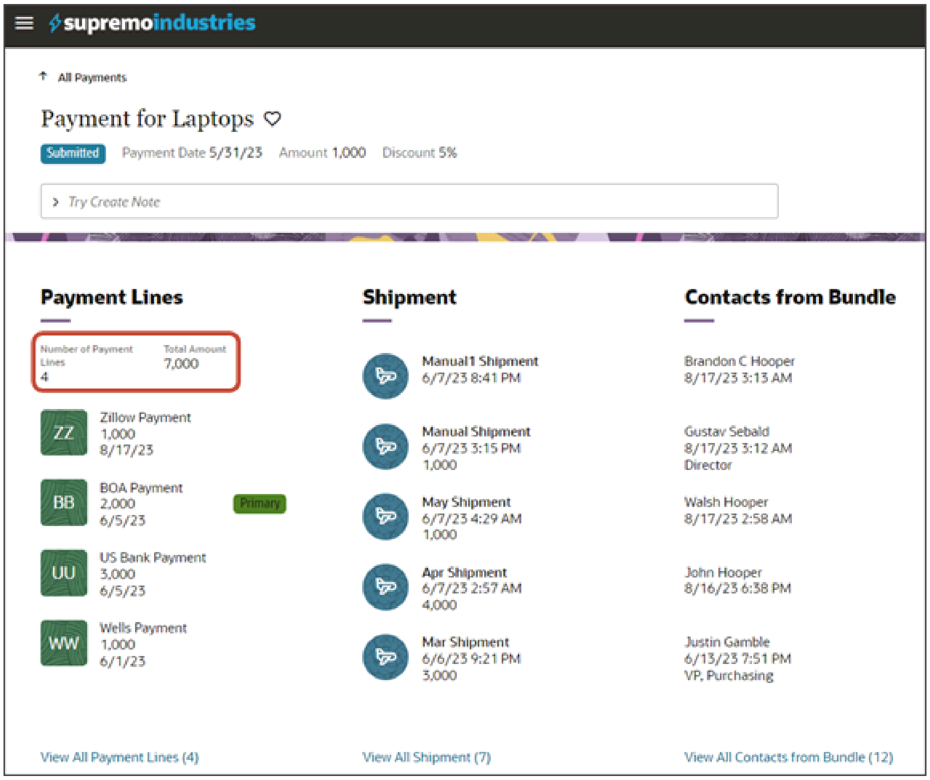
   > **Note:** You must add `/application/container` to the preview link.

   The screenshot below illustrates what the list page looks like with data.

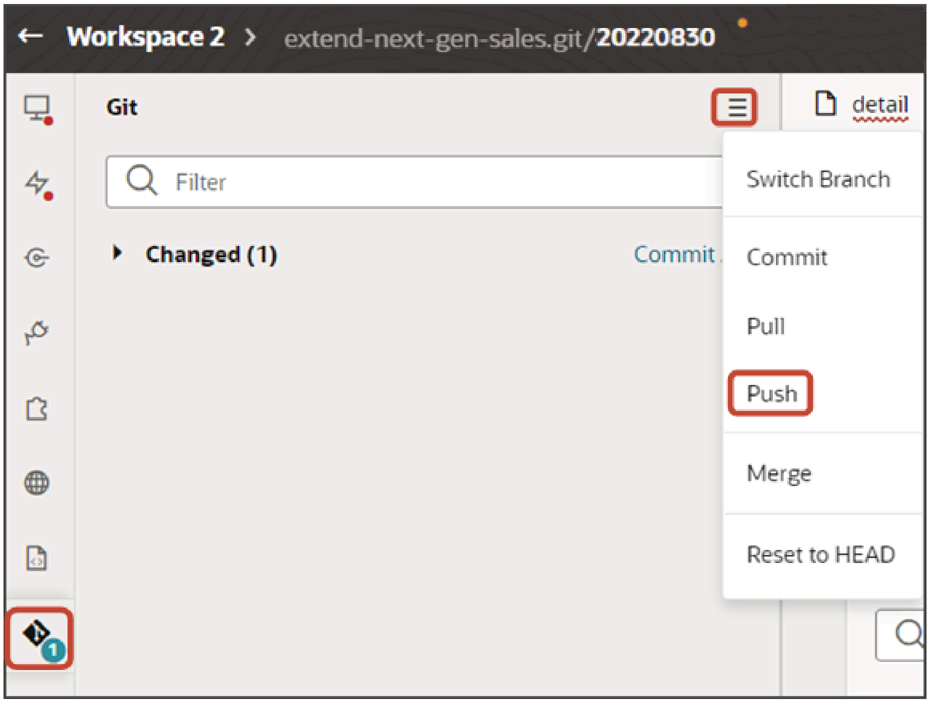
4. If data exists, you can click any record on the list page to drill down to the detail page. The detail page, including header region and panels, should display.

   You should now see the Payment Lines panel on the detail page, with a region for rollups.



5. Save your work by using the Push Git command.

   Navigate to the Git tab, review your changes, and do a Git push (which does both a commit and a push to the Git repository).



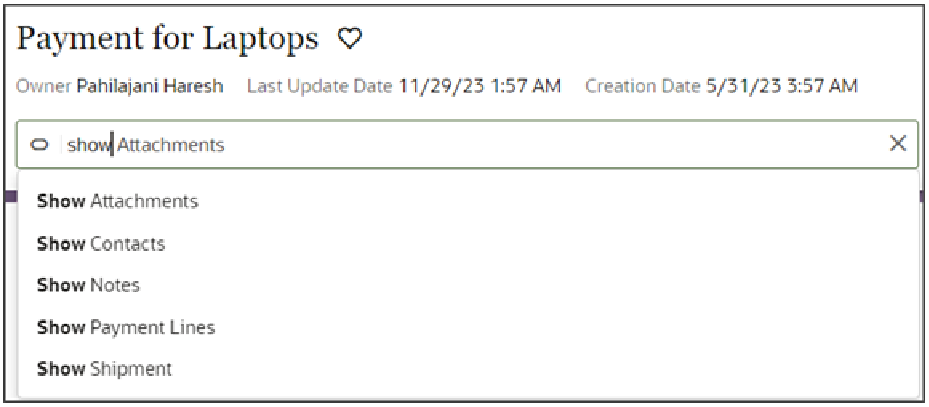
Understanding "Show" Actions

Whenever you add a subview, a Show action is automatically created. The Show action displays in the Action Bar so that users can display the related subview. Show actions are not smart actions and you don't need to manually create them. The only change you might want to make for Show actions is the label. Each Show action string is hard-coded but you can change it to a string that can be translated.

What's a Show Action?

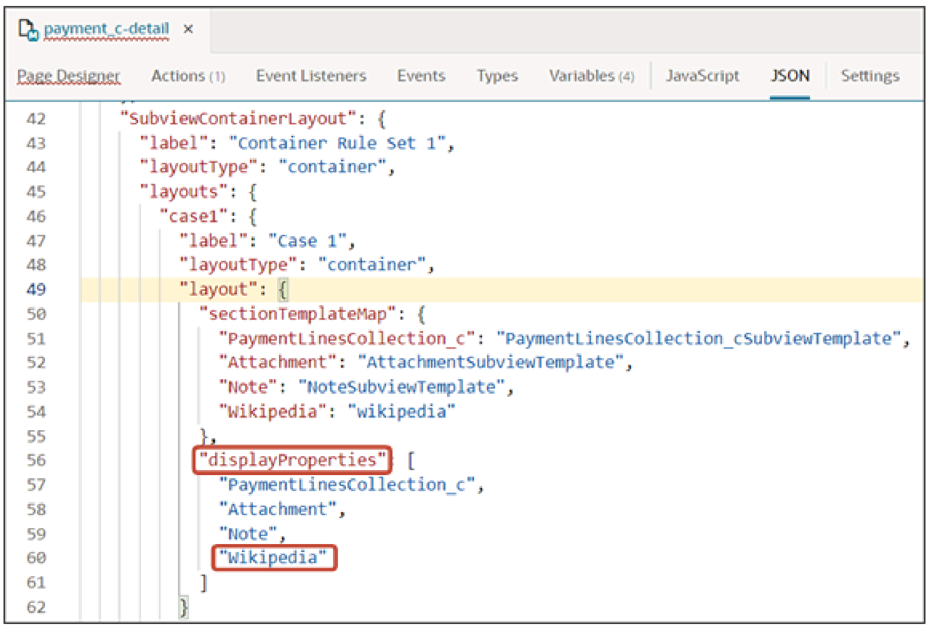
Show actions are similar to smart actions because they are both available from the Action Bar. However, Show actions are **not** smart actions. Instead, Show actions are actions that are automatically displayed specifically so that users can navigate to subviews for various objects.

For example, these Show actions were automatically created when you added subviews for the below objects:



Show Action Labels

The labels for Show actions are derived from each subview's display property. The display property was specified when the section was initially added for the subview. You can view subview display properties on the detail page's JSON.



Since Show action labels are automatically derived from the display property strings, the labels are hard-coded and not translatable. If needed, you can make them translatable.

Create a Translatable String

Let's look at an example. If you add a subview for a mashup that displays Wikipedia, then the **Show Wikipedia** action is automatically created without any action required on your part.

But, maybe you have users who need to see the Show Wikipedia string in Korean. In that case, you can change the hard-coded string to a string that's translatable.

To create translatable Show actions:

1. Add the translatable string to your custom translation bundle.

   See *Create a Translation Bundle, If You Don't Have One Already*.
2. Create a constant that refers to the string in your translation bundle.

   a. On the payment_c-detail tab, click the Variables subtab.
   b. Click **+ Variable**.
   c. In the Create Variable dialog, make sure the Constant option is selected and, in the ID field, enter `subviewLabel`.
   d. In the Type field, select **Object**.
   e. Click **Create**.
   f. On the Properties pane for the new **subviewLabel** constant, in the **Default Value** field, enter the reference to the string that you added to the translation bundle.

      Use the following format, where the first `Wikipedia` instance is the subview's display property and the second `wikipedia` instance is the string key that you added to the translation bundle:

      ```
      {"Wikipedia":"[[ $translations.CustomBundle.wikipedia() ]]"}
      ```

      Be sure to replace the translation bundle name and string with your own information, as needed.

      If you have multiple subviews and you need translatable Show actions for each one, then you can add that to the default value for the **subviewLabel** constant. For example:

      ```
      {
       "PaymentContactMMInter_Src_Payment_cToPaymentContactMMInter_c_Tgt":"[[ $translations.CustomBundle.Contacts
       "Wikipedia":"[[ $translations.CustomBundle.wikipedia() ]]"
      }
      ```

3. Add the new **subviewLabel** constant to the payment_c-detail page's cx-detail fragment code.
   a. On the payment_c-detail tab, click the Page Designer subtab.
   b. Click the Code button.

   c. Add the **subviewLabel** constant to the "actionBar" parameter in the fragment code, as follows:

```
"subviewLabelExtension": $page.constants.subviewLabel
```

The fragment code should look like the below sample. Be sure to replace `Payment_c` with your custom object's REST API name.

```
<oj-vb-fragment bridge="[[vbBridge]]" name="oracle_cx_fragmentsUI:cx-detail" class="oj-flex-item
 oj-sm-12 oj-md-12">
 <oj-vb-fragment-param name="resources" value="[[ {'Payment_c' : {'puid': $variables.id, 'id':
 $variables.id, 'endpoint': $application.constants.serviceConnection }} ]]"></oj-vb-fragment-
param>
 <oj-vb-fragment-param name="header" value="[[ {'resource': $flow.constants.objectName,
 'extensionId': $application.constants.extensionId } ]]"></oj-vb-fragment-param>
 <oj-vb-fragment-param name="actionBar"
 value='[[ { "applicationId": "ORACLE-ISS-APP", "resource": {"name": $flow.constants.objectName,
 "primaryKey": "Id", "puid": "Id", "value": $variables.puid }, "subviewLabelExtension":
 $page.constants.subviewLabel } ]]'>
 </oj-vb-fragment-param>
 <oj-vb-fragment-param name="panels" value='[[ { "panelsMetadata":
 $metadata.dynamicContainerMetadata, "view": $page.variables.view } ]]'></oj-vb-fragment-param>
 <oj-vb-fragment-param name="context" value="[[ {'flowContext': $flow.variables.context} ]]"></oj-
vb-fragment-param>
 <oj-vb-fragment-param name="row" value="{{ $page.variables.row }}"></oj-vb-fragment-param>
</oj-vb-fragment>
```

Add the CreatedBy and LastUpdatedBy Fields to Notes Panels and Subviews

Users can add notes to a record, and those notes will display on a Notes panel as well as on a Notes subview page. As an administrator, you can optionally display the CreatedBy and LastUpdatedBy fields for each note. If you add either of these fields to a Note layout, then you must use a specific field template so that the user names display correctly at runtime. This topic illustrates how to add the correct field template.

In this example, we'll add the LastUpdatedBy field to the Notes panel and subview that display on a Payment record. You can follow the same set of steps if you want to display the CreatedBy field, as well.

Update the Field Templates File

In the field templates file, add a new field template.

1. In Visual Builder Studio, click the Source side tab.
2. On the Source side tab, navigate to **extension1** > **sources** > **dynamicLayouts** > **self** > **field-templates-overlay.html**.

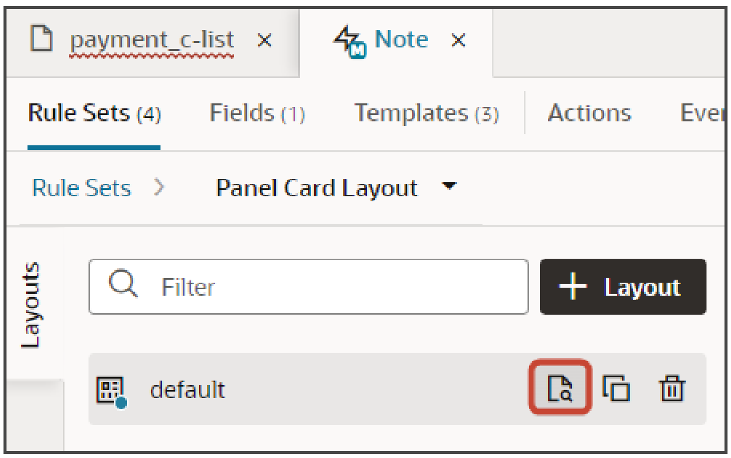
3. In the field-templates-overlay.html file, add this field template:

```
<template id="userNameTemplate">
<oj-vb-fragment name="oracle_cx_fragmentsUI:cx-profile" bridge="[[ vbBridge ]]">
<oj-vb-fragment-param name="user" value="[[ { 'userName' : $value() } ]]"></oj-vb-fragment-param>
</oj-vb-fragment>
</template>
```

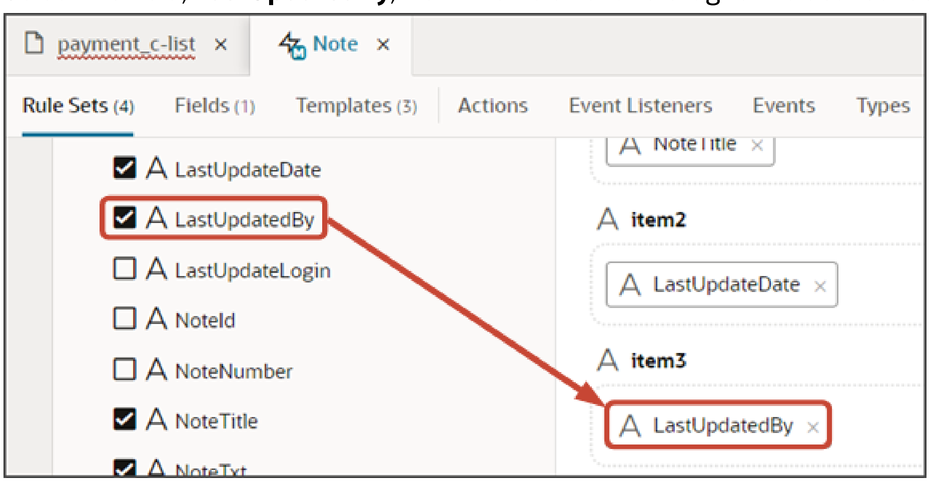
Add the LastUpdatedBy Field to the Panel

Add the LastUpdatedBy field to the Notes panel that displays on a Payment record.

1. In Visual Builder Studio, click the Layouts side tab.
2. On the Layouts side tab, click the **Payment_c** > **Note** node.
3. Click the **Rule Sets** subtab.
4. Click the **Panel Card Layout** rule set.
5. Click the Open icon to edit the default layout.



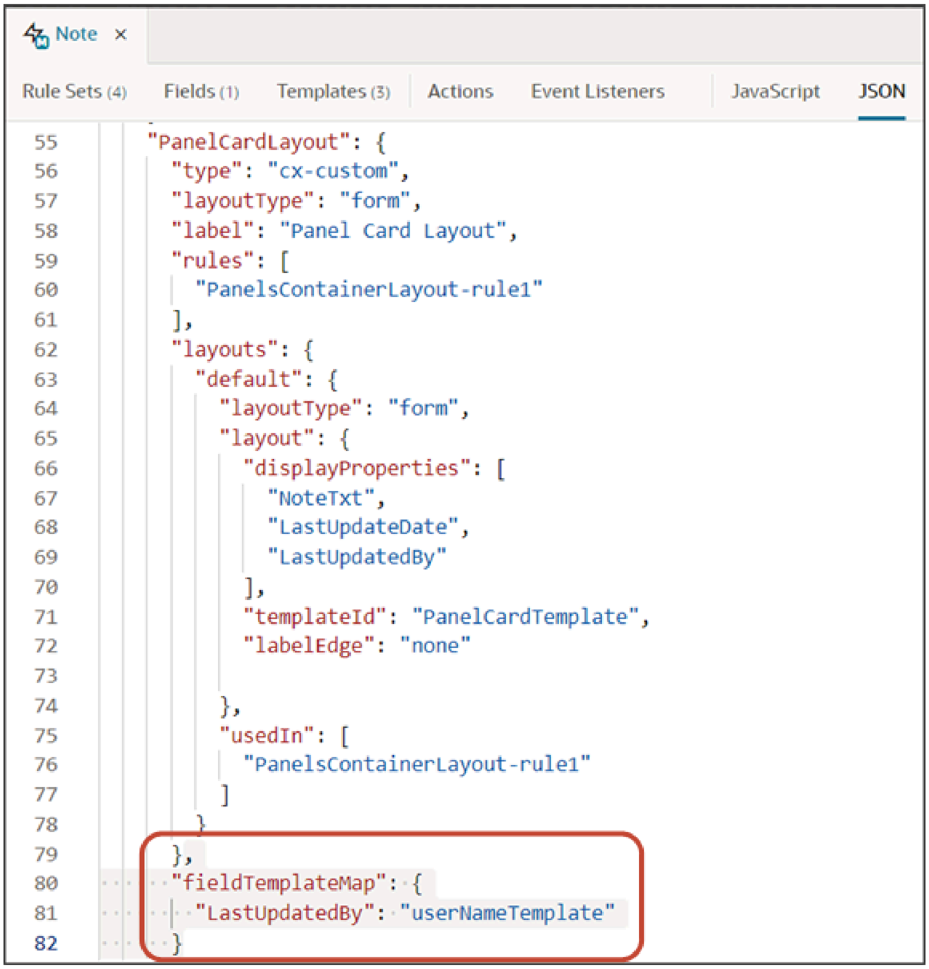
6. Select the field, **LastUpdatedBy**, from the field list and drag to the desired location on the panel layout.



7. Associate the LastedUpdatedBy field with the userNameTemplate field template:
   a. On the Note tab, click the JSON subtab.
   b. In the "PanelCardLayout" section, add a "fieldTemplateMap" section with a row for the LastedUpdatedBy field:

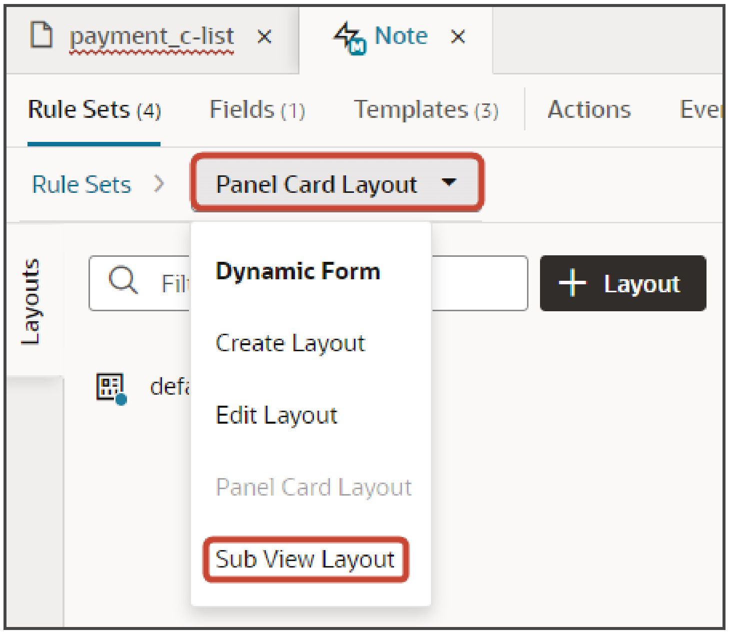
```
,
"fieldTemplateMap": {
"LastUpdatedBy": "userNameTemplate"
}
```

The resulting JSON will look like this:



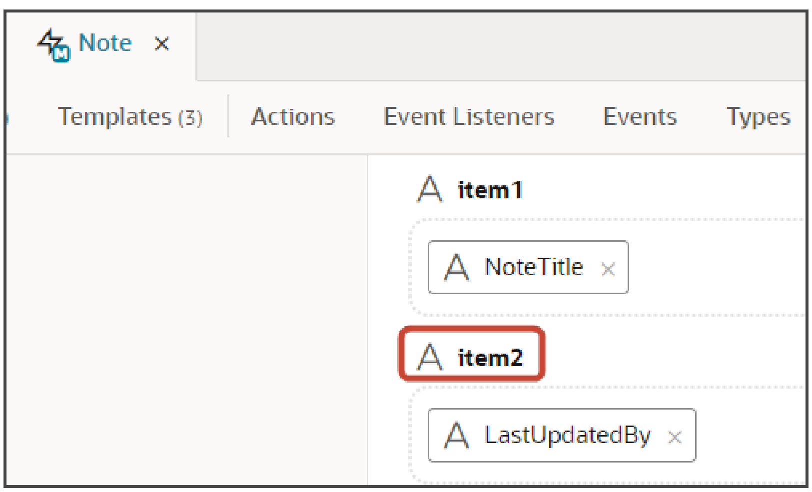
Add the LastUpdatedBy Field to the Subview

1. Switch to the **Sub View Layout** rule set.



2. Click the Open icon to edit the default layout.
3. Select the field, **LastUpdatedBy**, from the field list, and drag to the desired location on the subview layout.

   For example, drag the field to the item2 slot.



4. Associate the LastedUpdatedBy field with the userNameTemplate field template:
   a. On the Note tab, click the JSON subtab.
   b. In the "SubViewLayout" section, update the existing "fieldTemplateMap" section with a row for the LastedUpdatedBy field:

```
,
"fieldTemplateMap": {
```

```
"NoteTxt": "noteTemplate",
"LastUpdatedBy": "userNameTemplate"
}
```

Test the Flow

You can now test the Notes panel and subview to confirm that they both display the name of the person who last updated the note.

1. From the payment_c-list page, click the Preview button to see your changes in your runtime test environment.

2. The resulting preview link will be:

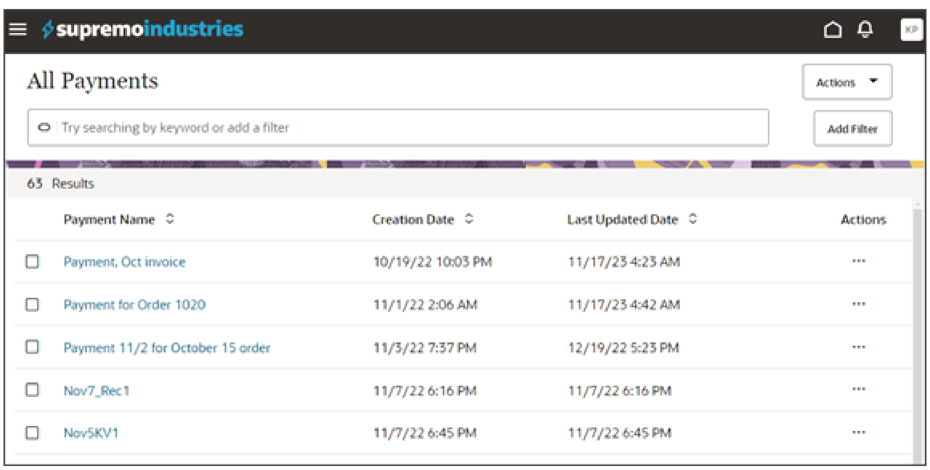
```
https://<servername>/fscmUI/redwood/cx-custom/payment_c/payment_c-list
```

3. Change the preview link as follows:

```
https://<servername>/fscmUI/redwood/cx-custom/application/container/payment_c/payment_c-list
```

> **Note:** You must add `/application/container` to the preview link.

The screenshot below illustrates what the list page looks like with data.

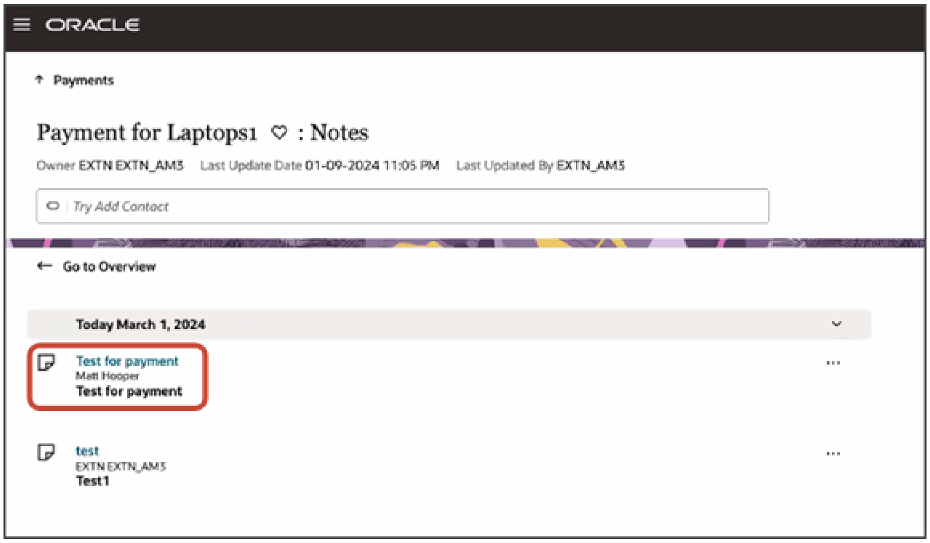


4. Click any existing payment to view its detail page.
5. In the Action Bar, enter `create Note`.
6. Click **Create Note**.

   The Create Note drawer displays.
7. Create a note and then click **Create**.
8. On the Notes panel, you should see the newly created note, along with the full user name, not the ID, of the person who last updated the note.

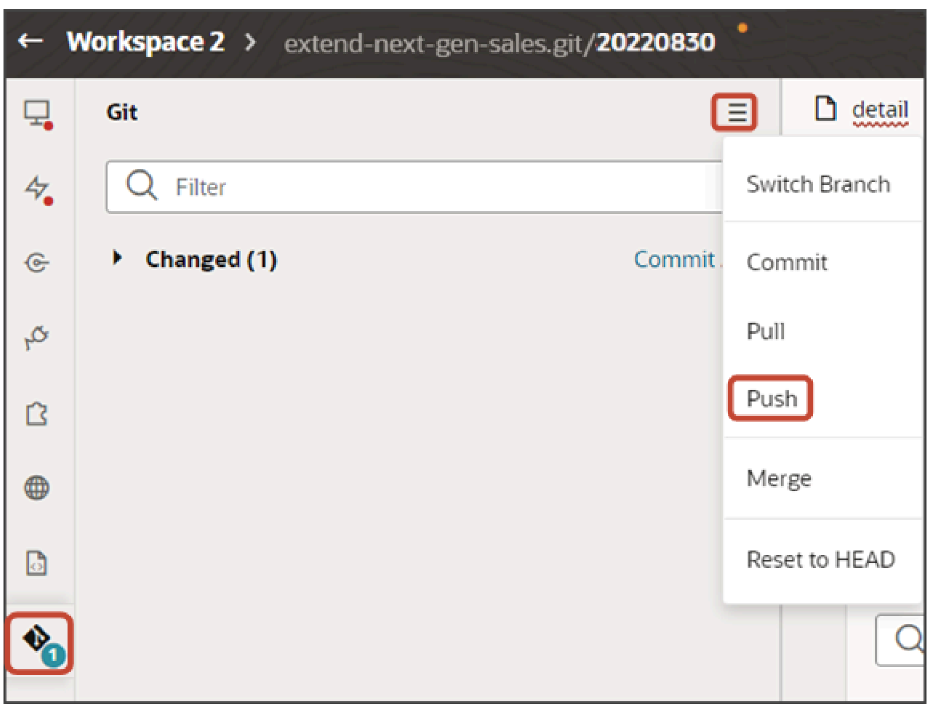
9. Click **View All Notes** to view the Notes subview.

   The subview should also display the full user name, not the ID, of the person who last updated the note.

10. Save your work by using the Push Git command.

    Navigate to the Git tab, review your changes, and do a Git push (which does both a commit and a push to the Git repository).

Link to a Smart Action Using a URL

You can construct a URL that calls a smart action in the Redwood version of Sales. Construct this URL whenever needed and then use it as a deep link. Depending on the smart action added to the URL, clicking the link will either execute a smart action without involving a UI (to delete a record, for example) or navigate directly to an open drawer on a Sales page (to create a record, for example).

To construct the URL, append the smart action ID as a parameter to the detail page URL.

1. Obtain the smart action ID.

   You can retrieve the smart action ID from Application Composer.
2. Obtain the URL of the detail page.

   For example:

   ```
   https://<servername>/fscmUI/redwood/cx-sales/application/container/accounts/accounts-detail?
   id=300000008600956&puid=38005&view=foldout
   ```
3. Append the smart action ID parameter as follows:

   ```
   &actionId=<smart action ID>
   ```
4. The resulting URL can be used to link to a smart action:

   For example:

   ```
   https://<servername>/fscmUI/redwood/cx-sales/application/container/accounts/accounts-detail?
   id=300000008600956&puid=38005&view=foldout&actionId=SDA-Delete-accounts
   ```

   Note that once the action is completed, the URL changes to:

   ```
   https://<servername>/fscmUI/redwood/cx-sales/application/container/accounts/accounts-detail?
   id=300000008600956&puid=38005&view=foldout&actionId=completed
   ```

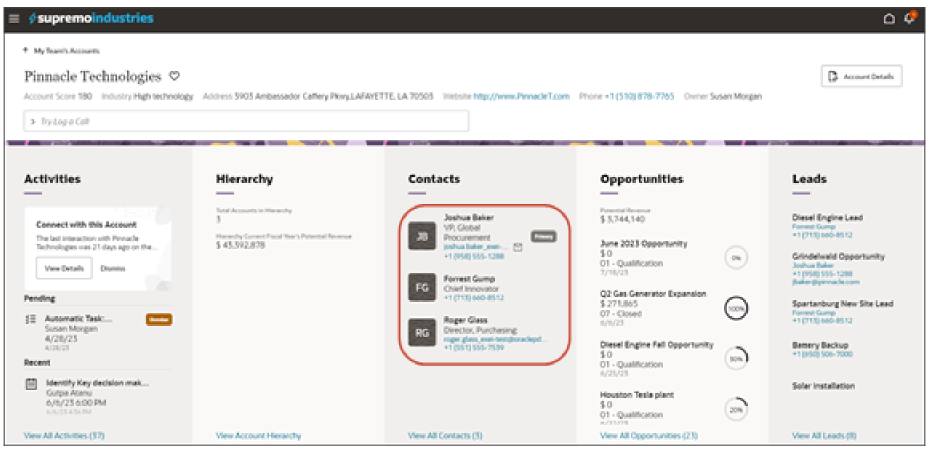
3 Additional Configuration Tasks

Configure the Contents of a Panel

An individual record's detail page includes key information displayed in a region of panels. Each panel contains information related to the record, such as related contacts and opportunities. Most panels display information in a list format. You can configure these lists using Oracle Visual Builder Studio.

What's Inside a Panel?

A panel often contains a list, which you can configure. Here's an example of a list inside a panel:



Lists can display up to 5 records, depending on screen size. If the screen size is small, then the list automatically adjusts to display fewer records. However, users can click the View All link that displays at the bottom of the panel to navigate to a second page to see all records in the list. This second page is called the subview.

What Can You Change in a List?

In Visual Builder Studio, you can modify the information that displays in each list.

You can:

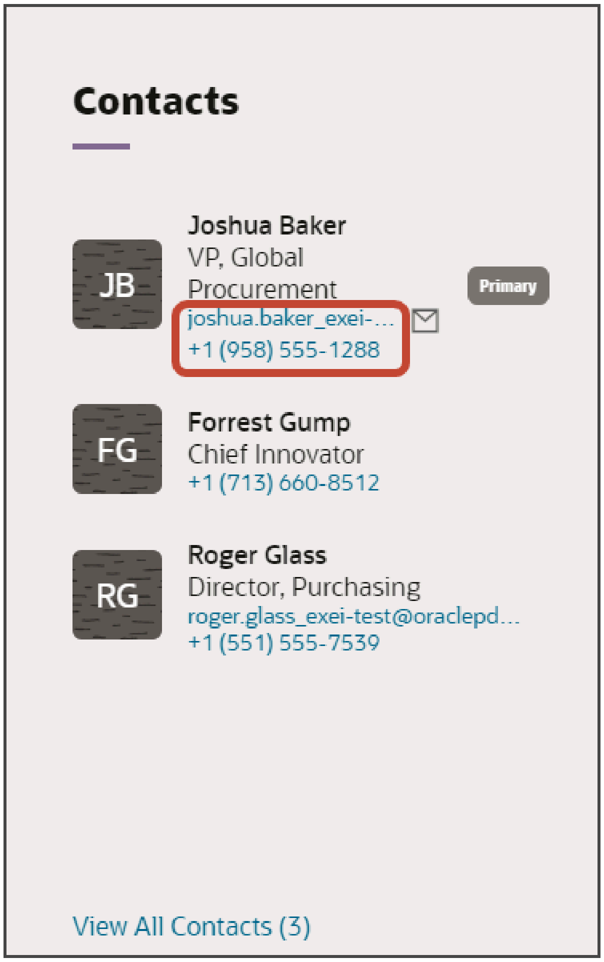
- Add and remove fields
- Change the display order of fields in the list

This topic illustrates how to change the display order of fields that display on panels on an account's detail page. We'll look at both the Contacts panel and Opportunities panel.

To configure the subview, see *Configure the Subview Layout*.

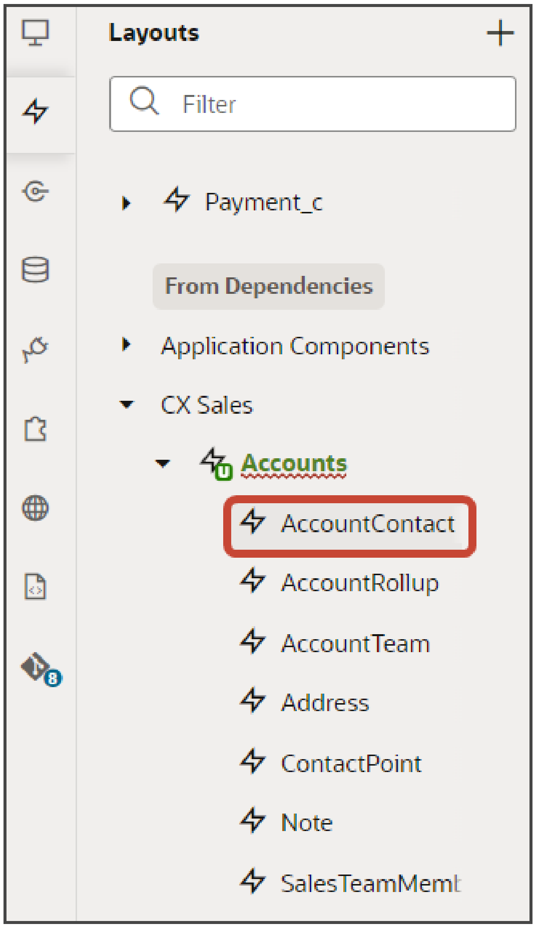
Change the Display Order of Contact Panel Fields

Let's change the display order of fields in a panel list. In this example, we'll switch the order of the email and phone number fields on the Contacts panel on the Account detail page.



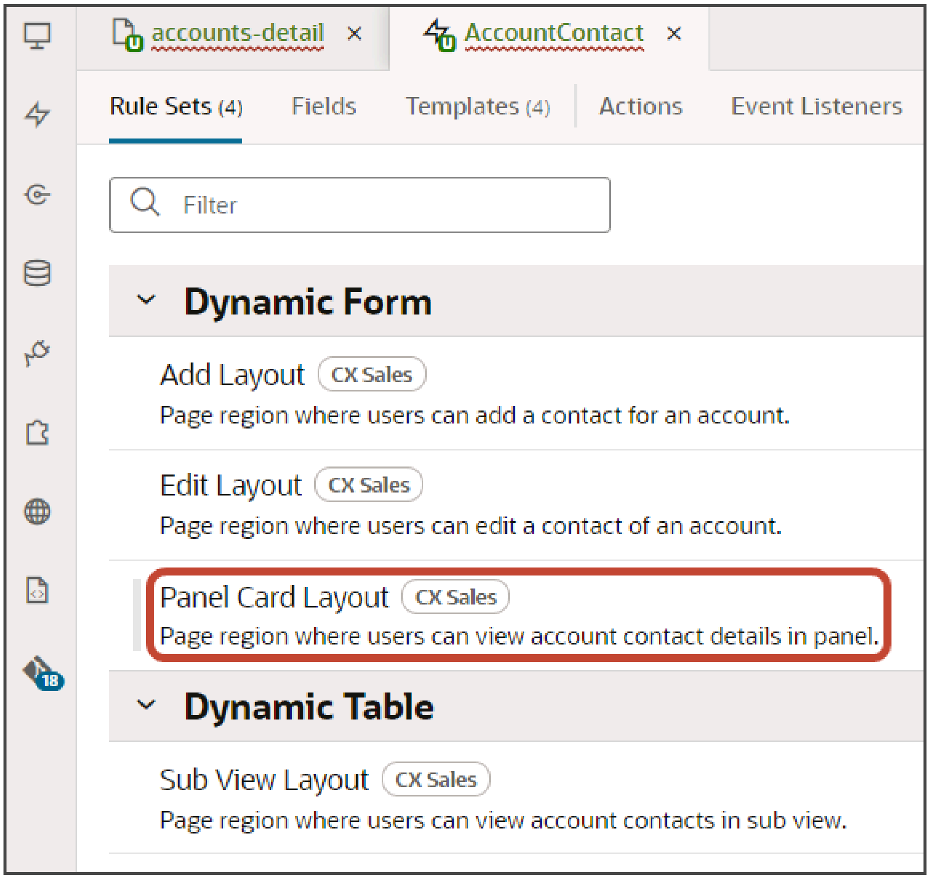
1. In Visual Builder Studio, navigate to the Layouts tab and expand the CX Sales node > Accounts > AccountContact.

   The AccountContact node contains the rule sets for the Contacts panel on the Account object.

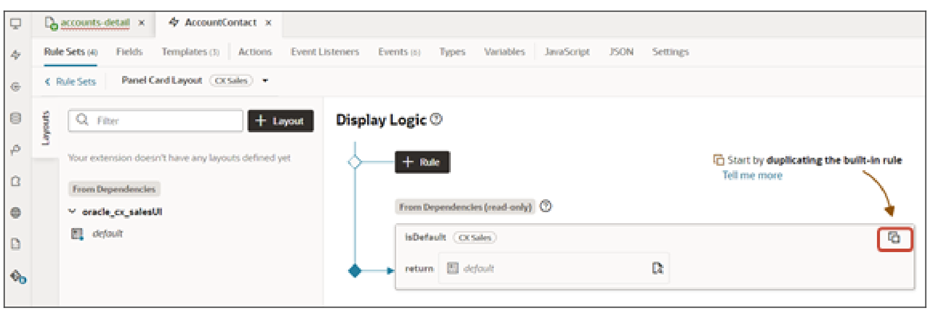


Note: When configuring the contents of a panel, consider what kind of relationship the panel's object has with the primary object. In this case, the Account object has a many-to-many relationship with Contact. This means that you'll find layouts for the Contact object on the AccountContact node, nested under the Accounts node.

2. On the AccountContact tab > Rule Sets subtab, click the Panel Card Layout.



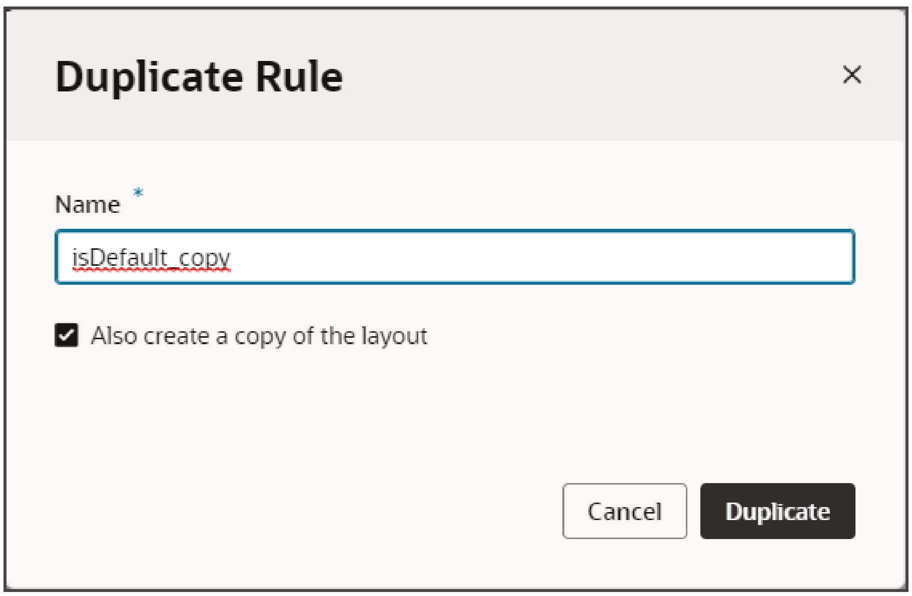
Both a default layout as well as a default rule are displayed for the Panel Card Layout.



3. Click the Duplicate Rule icon.

4. In the Duplicate Rule dialog, accept the default rule name or enter a new name. The name you enter here is both the rule name and also the layout name, so enter a layout name that makes sense for you.

   Also, make sure that the **Also create a copy of the layout** checkbox is selected.



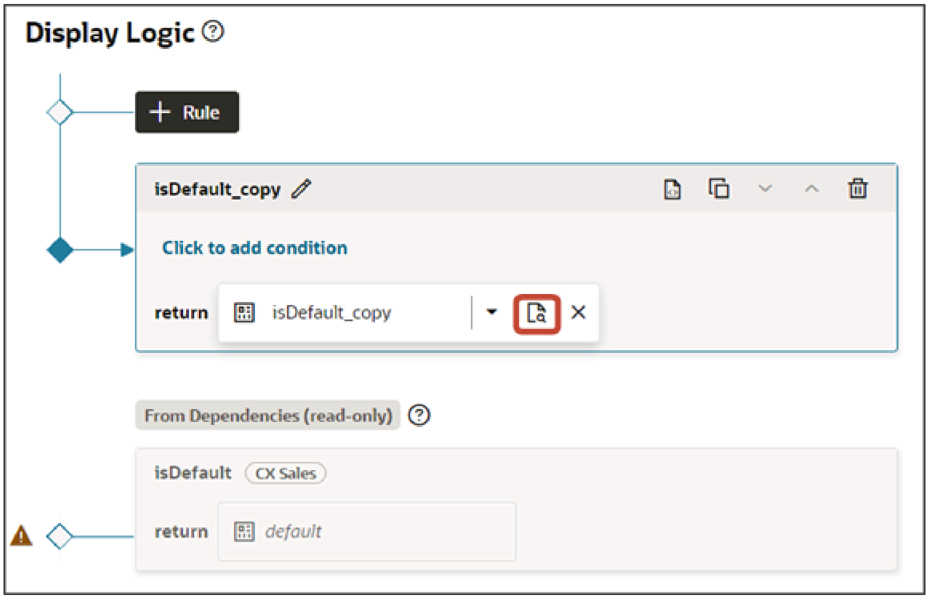
5. Click **Duplicate**.

   The new rule displays at the top of the list of existing rules, which means that this rule will be evaluated first at runtime. If the rule's conditions are met, then the associated layout is displayed to the user.

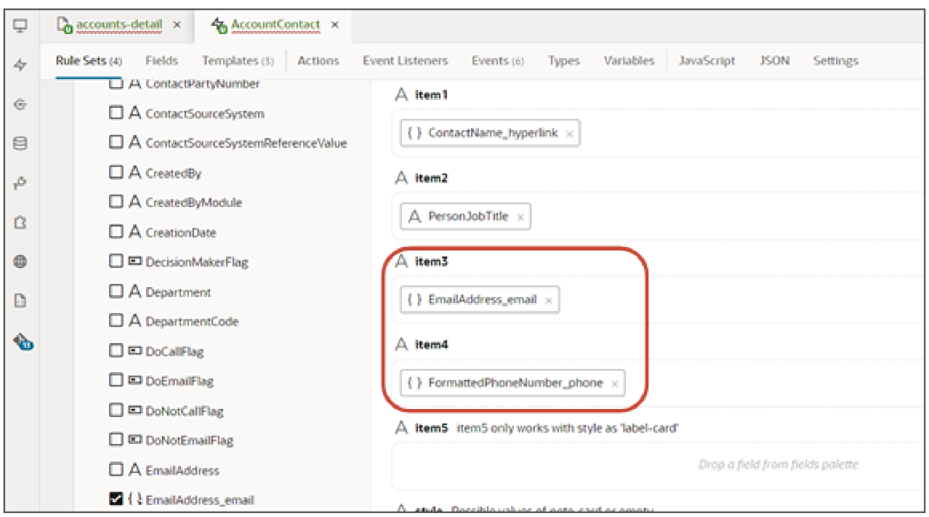
   In this example, we're not adding any conditions which means that the associated layout will always be displayed.

6. Modify the rule's copied layout.

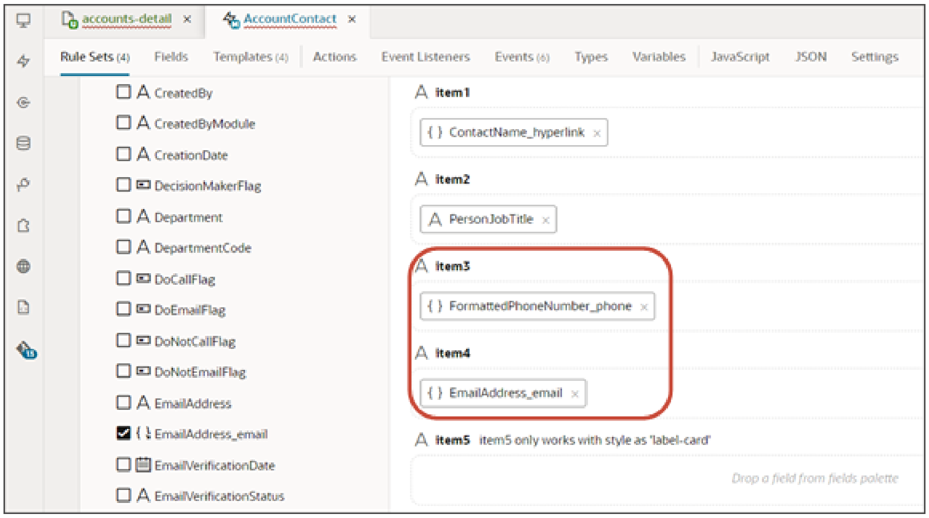
   a. Click the Open icon to edit the copied layout.



   b. Scroll down the list of fields in the layout until you locate the email and phone fields.



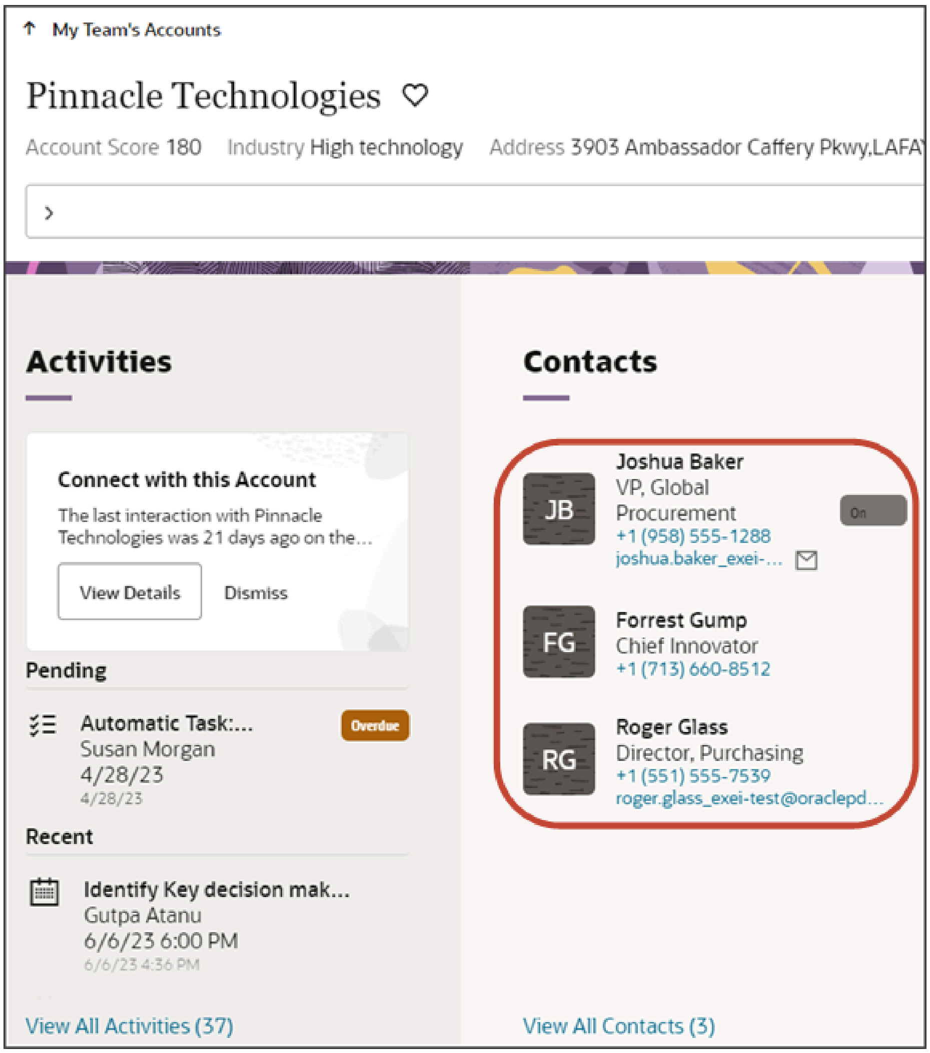
   c. Delete each field from the Item3 and Item4 slots, and then add the fields back. This time, however, switch the order so that the phone field is in the Item3 slot and the email field is in the Item4 slot.


accounts-detail
AccountContact
Rule Sets (4)
Fields
Templates (4)
Actions
Event Listeners
Events (6)
Types
Variables
JavaScript
JSON
Settings
CreatedBy
CreatedByModule
CreationDate
DecisionMakerFlag
Department
DepartmentCode
DoCallFlag
DoEmailFlag
DoNotCallFlag
DoNotEmailFlag
EmailAddress
EmailAddress_email
EmailVerificationDate
EmailVerificationStatus
item1
ContactName_hyperlink
item2
PersonJobTitle
item3
FormattedPhoneNumber_phone
item4
EmailAddress_email
item5 item5 only works with style as 'label-card'
Drop a field from fields palette

7. Click the Preview button to see your changes in your runtime test environment.

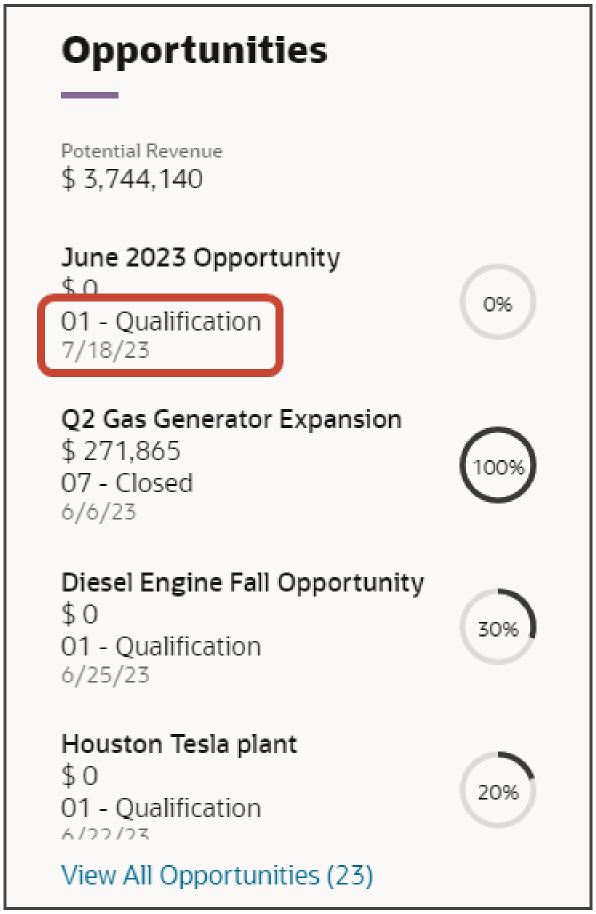
The preview link must include the `application/container` segments in the URL. If not, then change the preview link using the following example URL:

```
https://<servername>/fscmUI/redwood/cx-sales/application/container/accounts/accounts-detail?
id=300000003513233&puid=7050&view=foldout
```



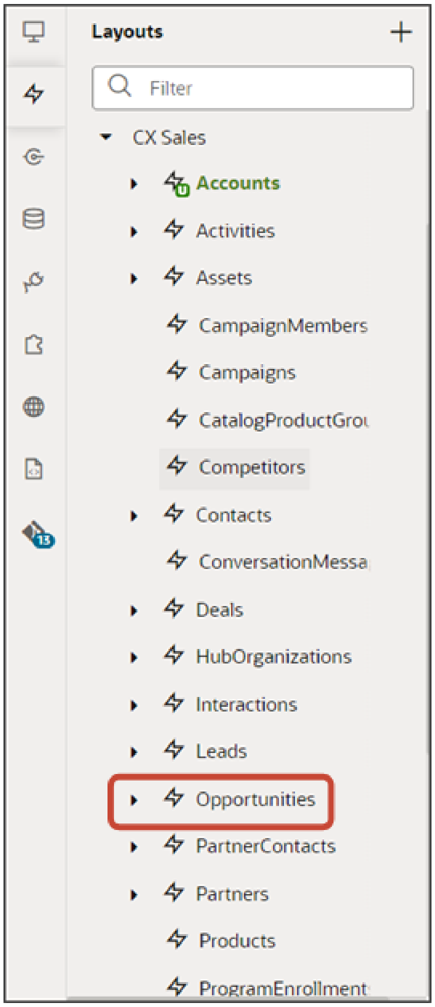
Change the Display Order of Opportunity Panel Fields

In this example, we'll switch the order of the sales stage and effective date fields on the Opportunities panel on the Account object.



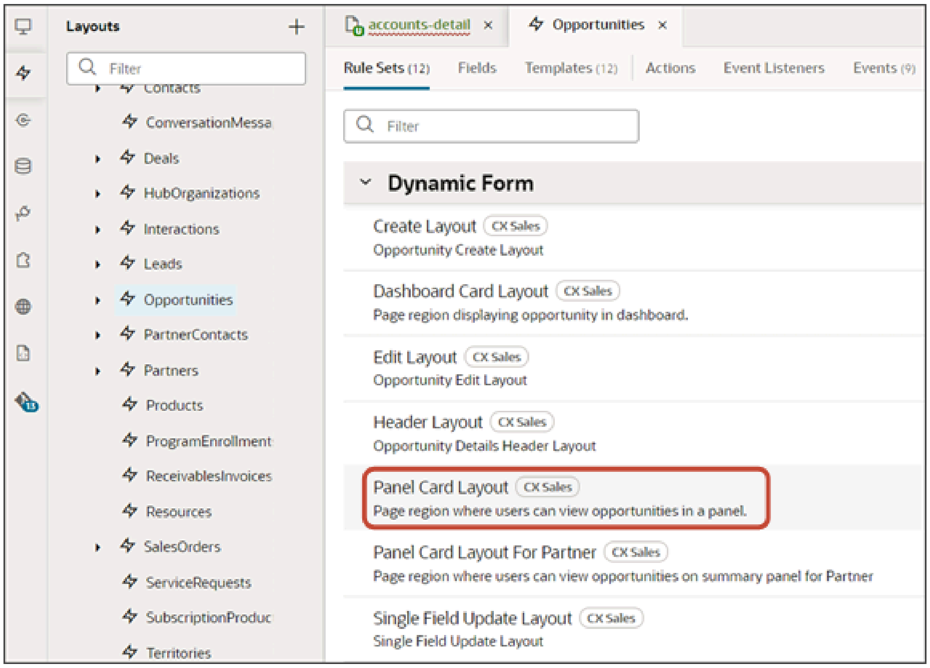
1. In Visual Builder Studio, navigate to the Layouts tab and expand the CX Sales node > Opportunities.

   The Opportunities node contains the rule sets for the Opportunities panel on the Account object.

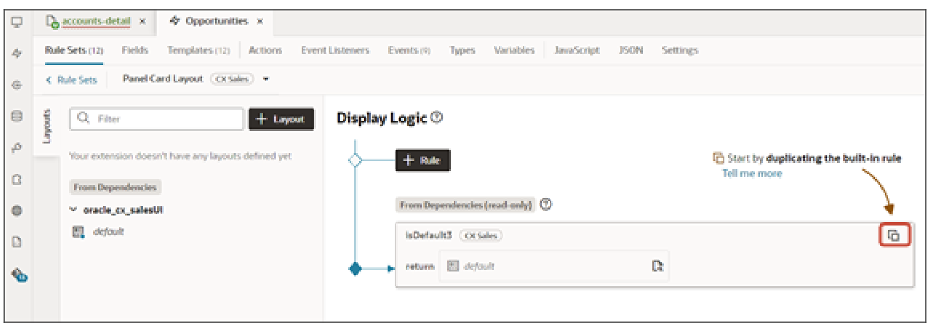


Note: When configuring the contents of a panel, consider what kind of relationship the panel's object has with the primary object. In this case, the Account object has a one-to-many relationship with Opportunity. This means that you'll find layouts for the Opportunity object on the Opportunities node.

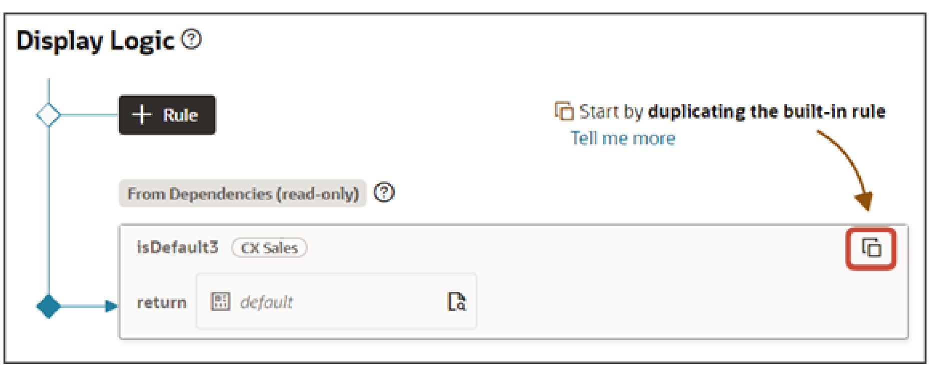
2. On the Opportunities tab > Rule Sets subtab, click the Panel Card Layout.



Both a default layout as well as a default rule are displayed for the Panel Card Layout.

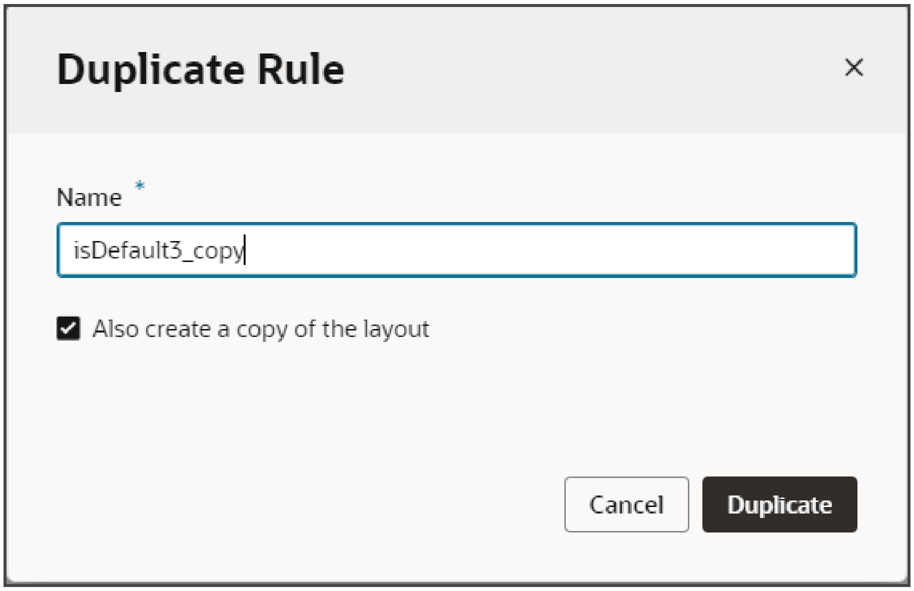


3. Click the Duplicate Rule icon.



4. In the Duplicate Rule dialog, accept the default rule name or enter a new name. The name you enter here is both the rule name and also the layout name, so enter a layout name that makes sense for you.

   Also, make sure that the **Also create a copy of the layout** checkbox is selected.



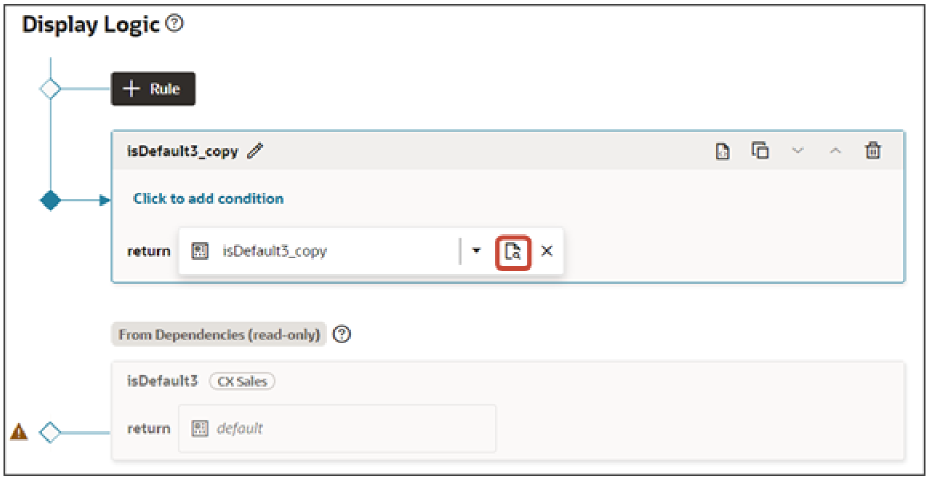
5. Click **Duplicate**.

   The new rule displays at the top of the list of existing rules, which means that this rule will be evaluated first at runtime. If the rule's conditions are met, then the associated layout is displayed to the user.

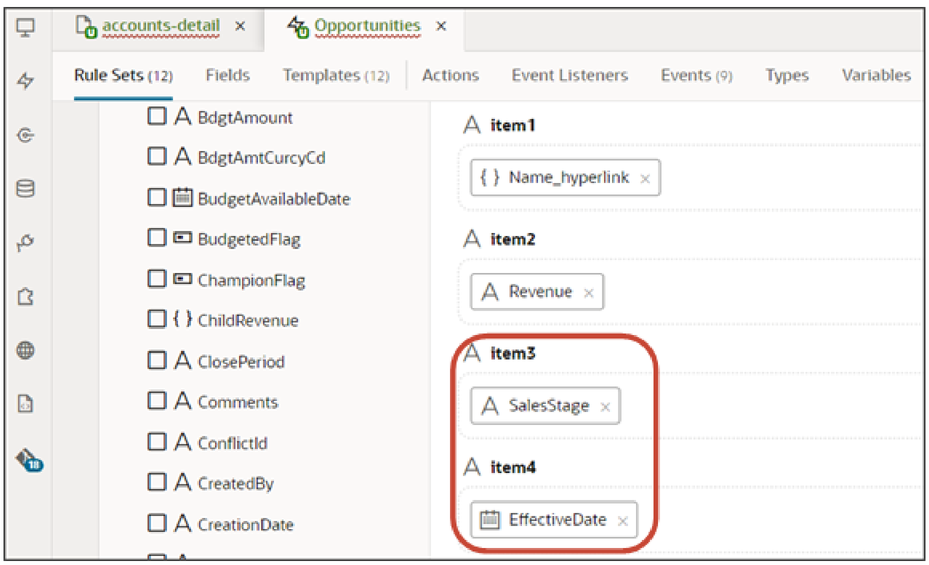
   In this example, we're not adding any conditions which means that the associated layout will always be displayed.

6. Modify the rule's copied layout.

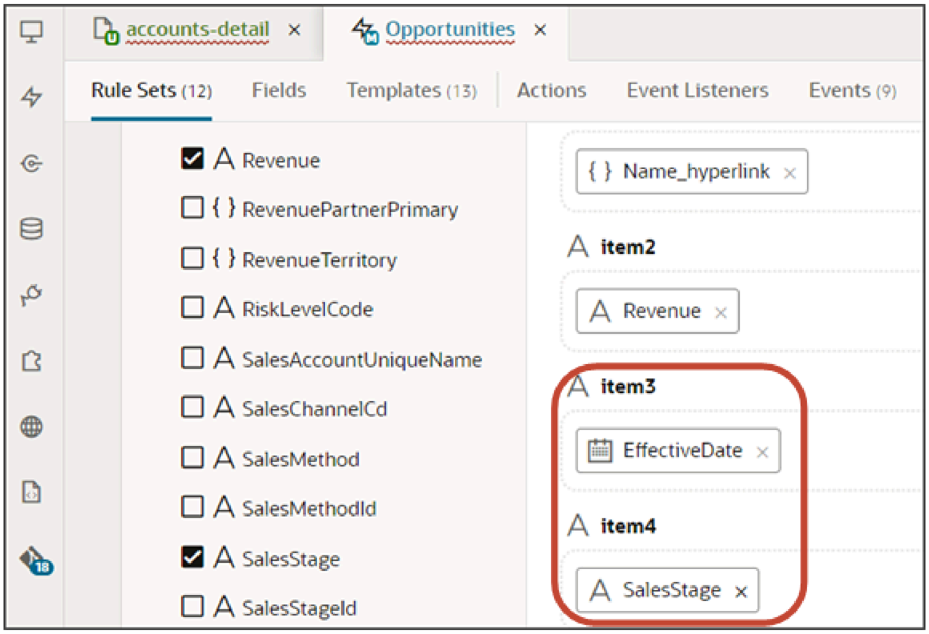
   a. Click the Open icon to edit the copied layout.



   b. Scroll down the list of fields in the layout until you locate the sales stage and effective date fields.



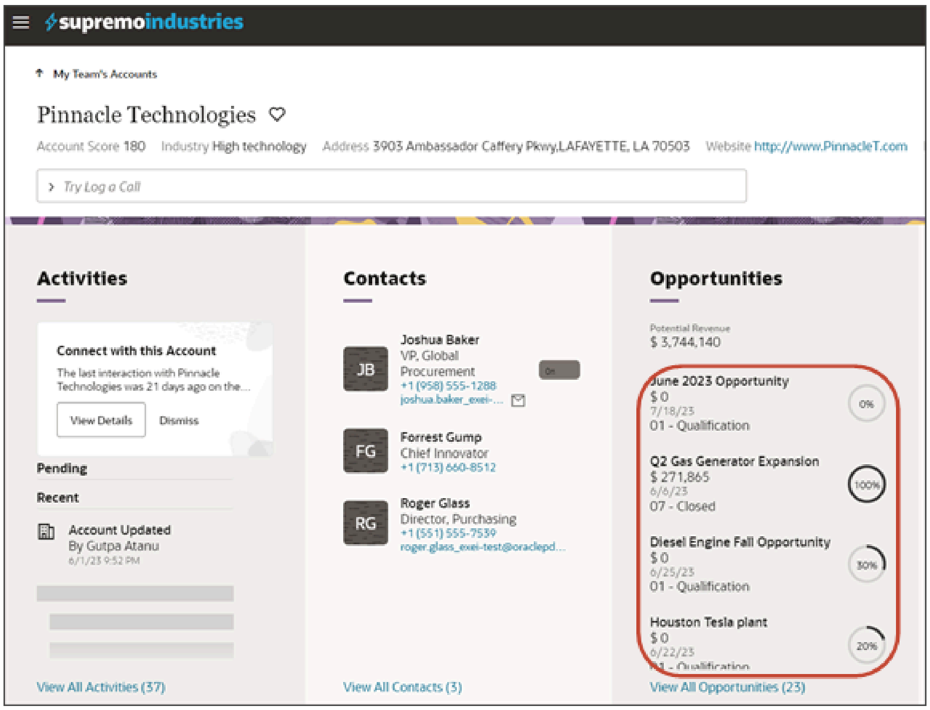
   c. Delete each field from the Item3 and Item4 slots, and then add the fields back but switch the order.


accounts-detail
Opportunities
Rule Sets (12)
Fields
Templates (13)
Actions
Event Listeners
Events (9)
Revenue
RevenuePartnerPrimary
RevenueTerritory
RiskLevelCode
SalesAccountUniqueName
SalesChannelCd
SalesMethod
SalesMethodId
SalesStage
SalesStageId
Name_hyperlink
item2
Revenue
item3
EffectiveDate
item4
SalesStage

7. Click the Preview button to see your changes in your runtime test environment.

The preview link must include the `application/container` segments in the URL. If not, then change the preview link using the following example URL:

```
https://<servername>/fscmUI/redwood/cx-sales/application/container/accounts/accounts-detail?
id=300000003513233&puid=7050&view=foldout
```



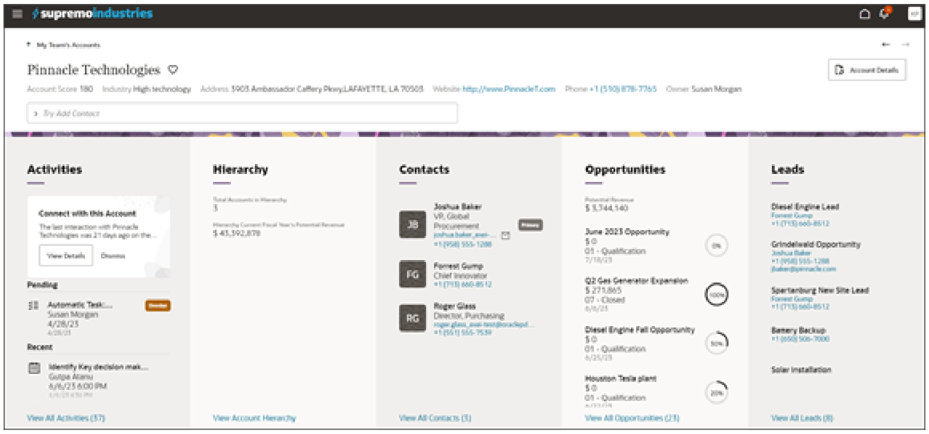
Configure the Subview Layout

An object's detail page includes a region of panels with information. Each panel, however, can display only a few records due to panel size. To see all records, users can navigate to a second page called a subview. This topic illustrates how to modify those subview pages using Oracle Visual Builder Studio.

What's Inside the Subview?

A subview contains a list of all records that the panel, due to limited real estate, can't display.

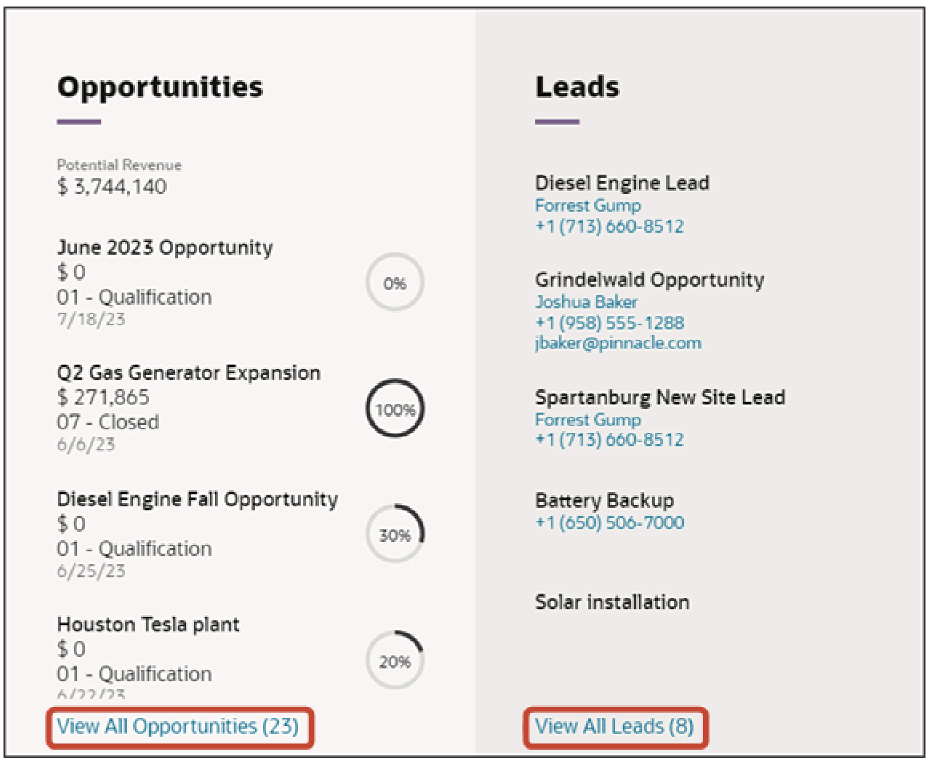
For example, here's an example of an account detail page with 5 panels:



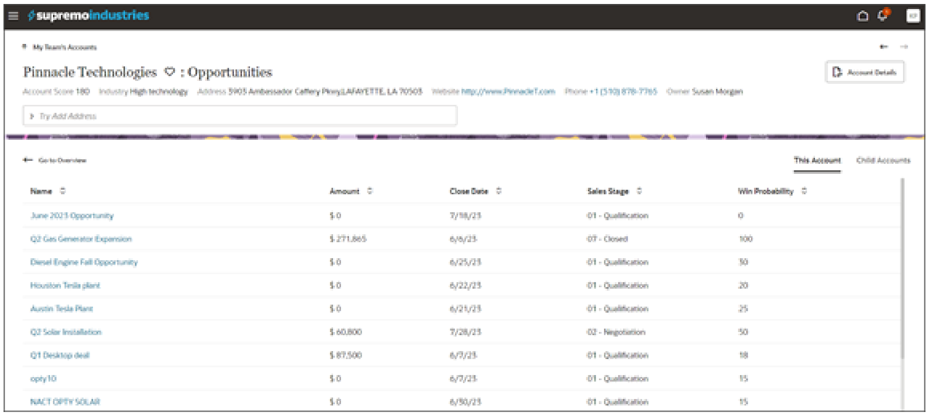
Notice how each panel displays only a few records.

To see all records, users can click the View All link that displays at the bottom of the panel.

Here's an example of some View All links. Note that after the link itself, a number indicates the number of total records listed on the subview.



The subview displays all those records in a table.



What Can You Change in a Subview Table?

In Visual Builder Studio, you can modify the information that displays in a subview table.

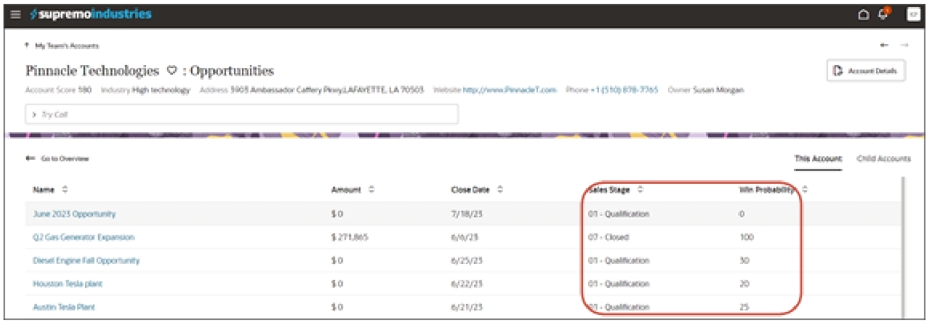
You can:

- Add and remove columns
- Change the display order of columns in the table

This topic illustrates how to change the display order of columns in a subview table. We'll look at the Opportunities subview that's available from an account detail page.

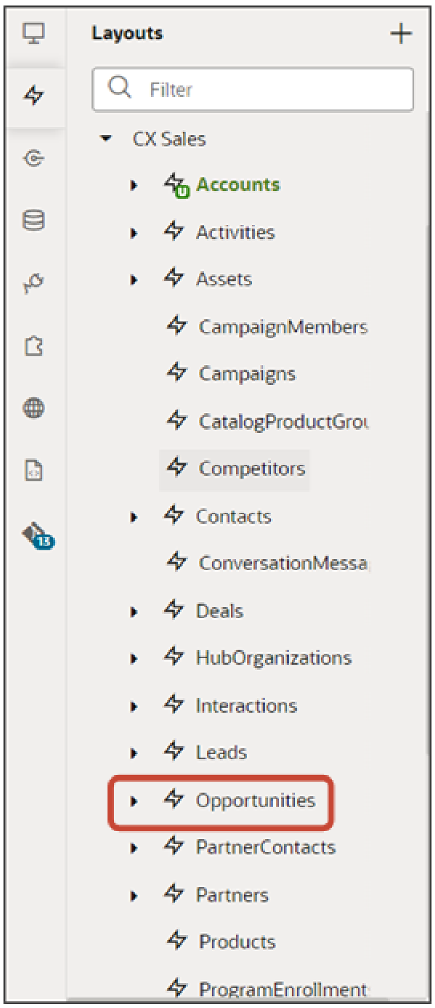
Change the Display Order of Opportunity Subview Columns

Let's change the display order of columns in a subview table. In this example, we'll switch the order of the sales stage and win probability columns on the Opportunities subview, accessed from the Opportunities panel on the Account detail page.

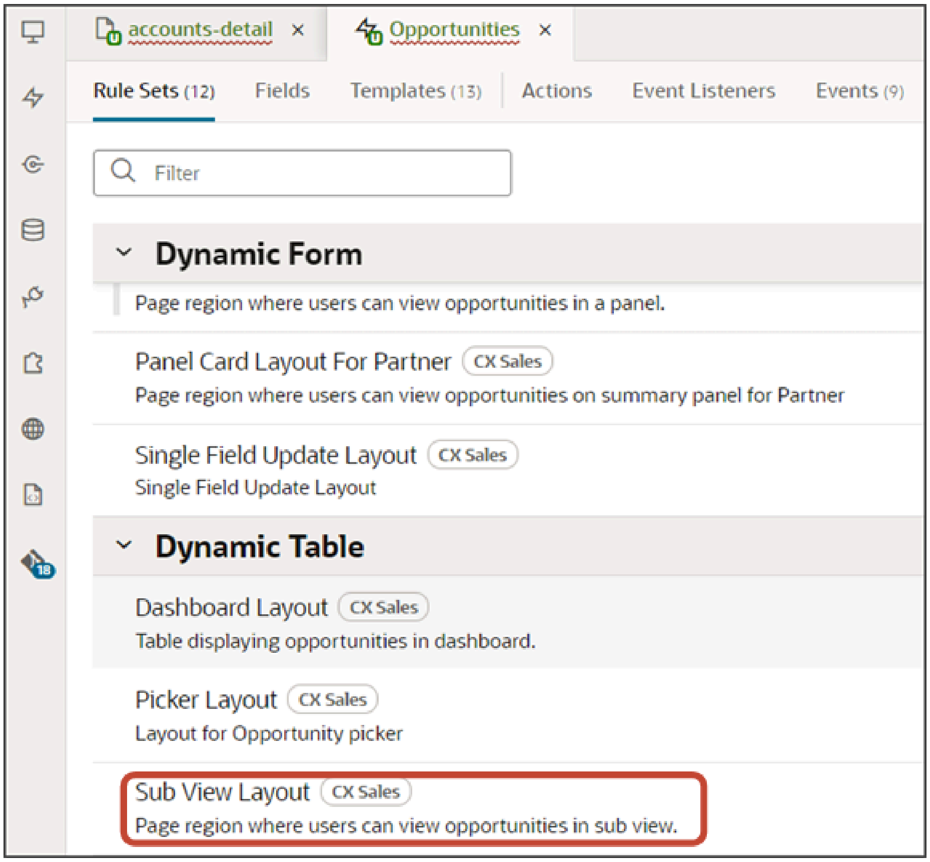


1. In Visual Builder Studio, navigate to the Layouts tab and expand the CX Sales node > Opportunities.

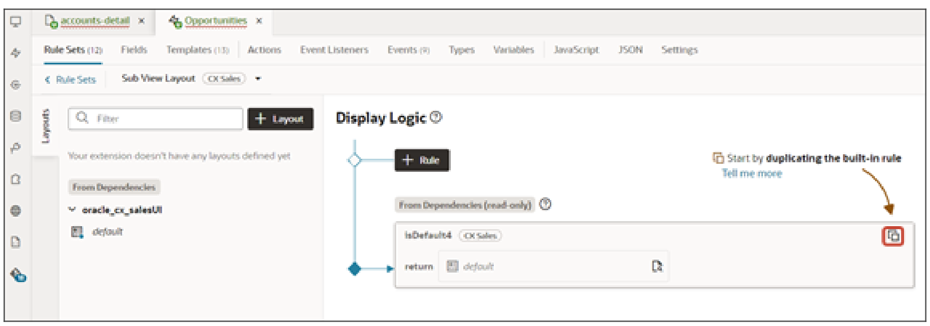
   The Opportunities node contains the rule sets for the Opportunities panel on the Account object.

2. On the Opportunities tab > Rule Sets subtab, click the Sub View Layout.



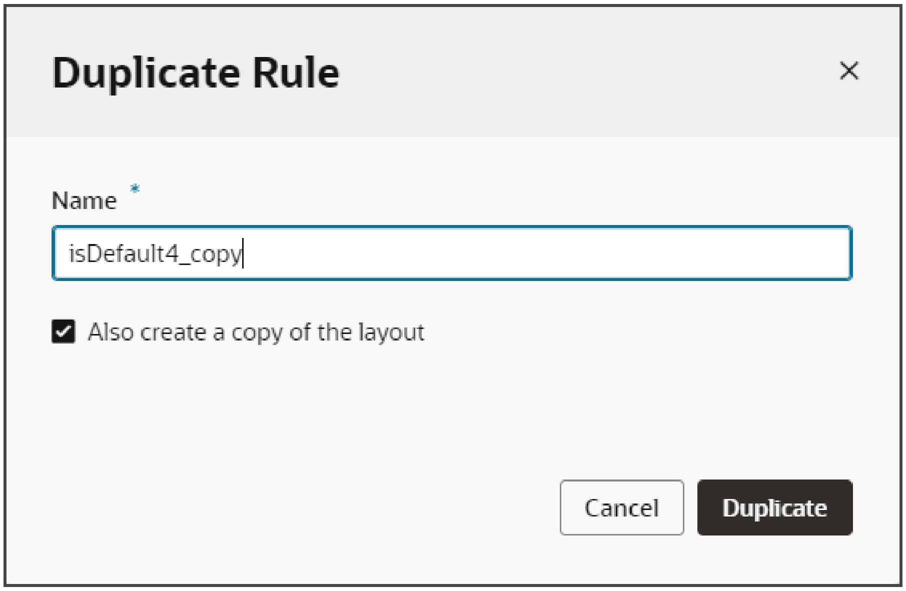
Both a default layout as well as a default rule are displayed for the Sub View Layout.



3. Click the Duplicate Rule icon.

4. In the Duplicate Rule dialog, accept the default rule name or enter a new name. The name you enter here is both the rule name and also the layout name, so enter a layout name that makes sense for you.

   Also, make sure that the **Also create a copy of the layout** checkbox is selected.

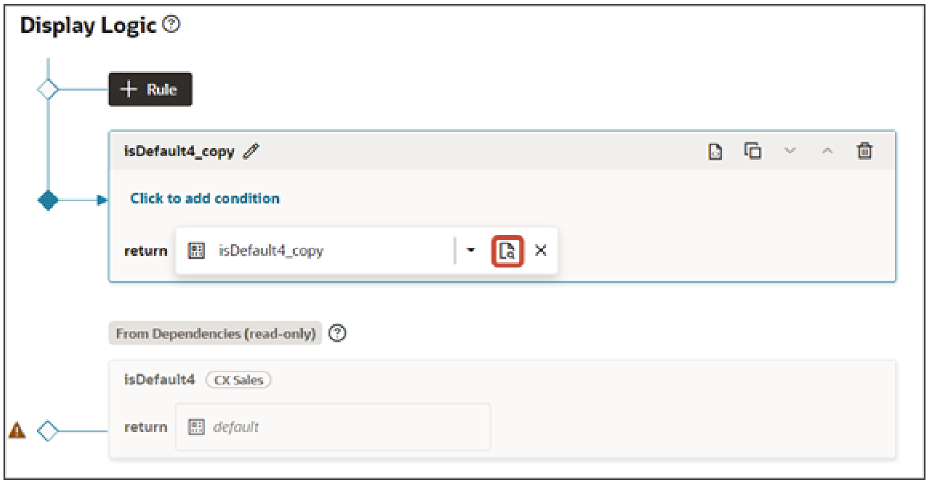
5. Click **Duplicate**.

   The new rule displays at the top of the list of existing rules, which means that this rule will be evaluated first at runtime. If the rule's conditions are met, then the associated layout is displayed to the user.

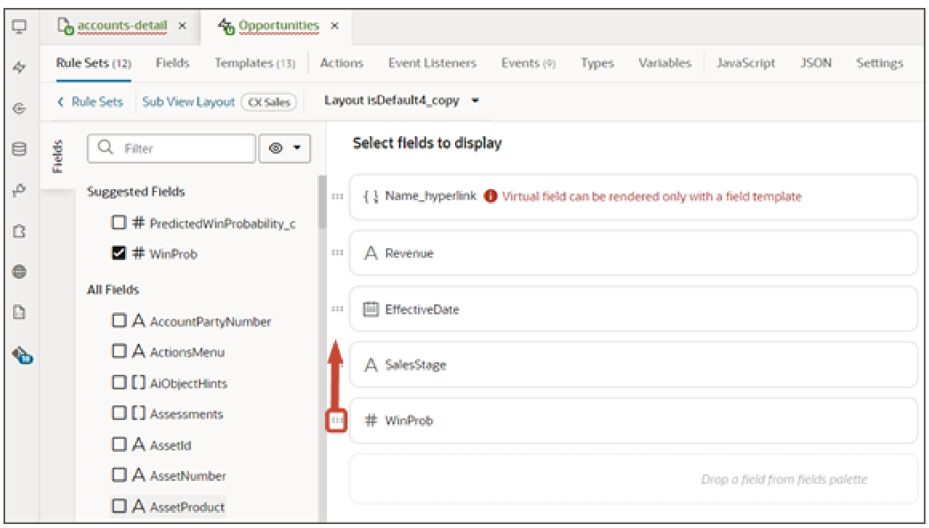
   In this example, we're not adding any conditions which means that the associated layout will always be displayed.

6. Modify the rule's copied layout.

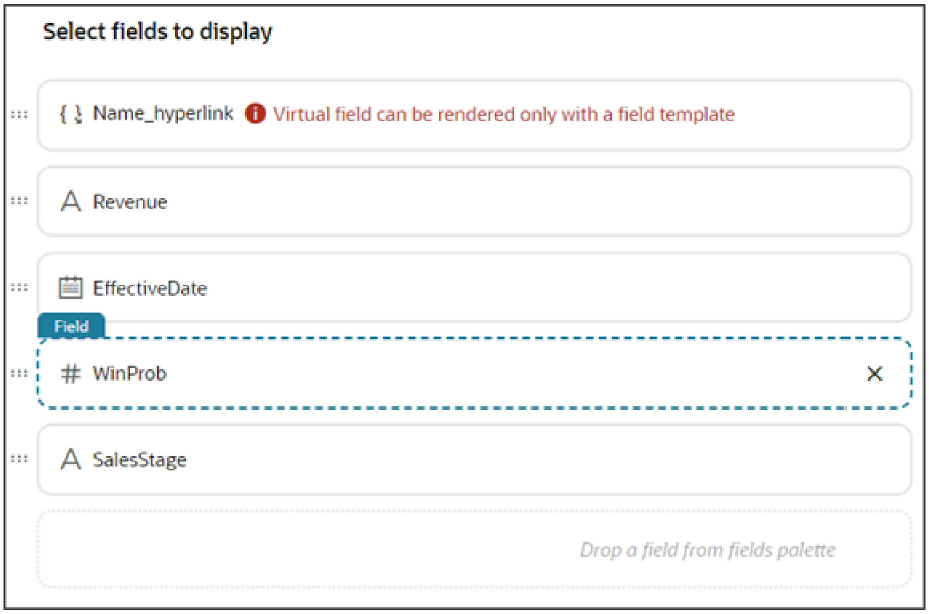
   a. Click the Open icon to edit the copied layout.



   b. In the list of fields in the layout, use the handle next to the win probability field to move it above the sales stage field.



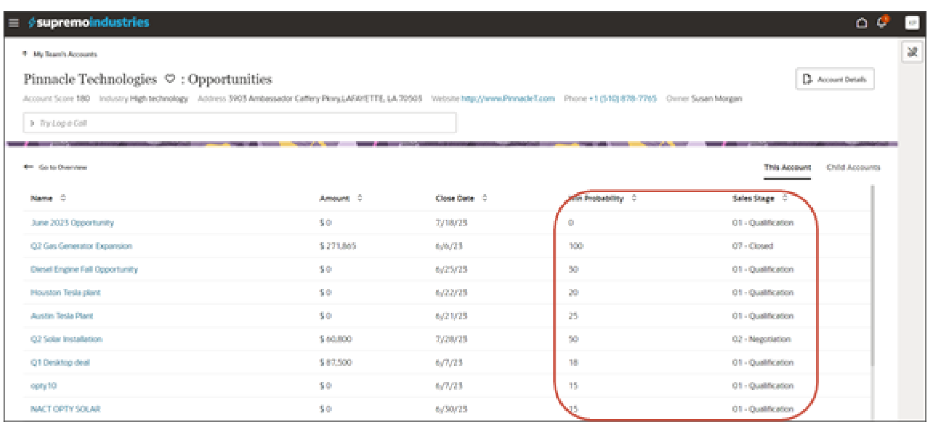
   Here's a screenshot of the final location of the win probability field.



7. Click the Preview button to see your changes in your runtime test environment.

The preview link must include the `application/container` segments in the URL. If not, then change the preview link using the following example URL:

```
https://<servername>/fscmUI/redwood/cx-sales/application/container/accounts/accounts-detail?
id=300000003513233&puid=7050&view=foldout
```



Make Values of a DCL Field Dependent on the Values of Another Field

You can create a field, such as a dynamic choice list field (DCL), that displays different values depending on the values of a different field. In this example, we'll create a DCL field for the Create Contact page that shows addresses for the account associated with the contact. Salespeople can use the field to select an address for the contact from the available account addresses.

Create the Dynamic Choice List Field

Tip: View the following video on Oracle Cloud Customer Connect for a summary of the setup in Oracle Visual Builder Studio: *Dependent DCL Field*.

1. Open Application Composer in a sandbox.
2. In the left panel, make sure that **CRM Cloud** is selected in the **Application** field.
3. Expand the **Contact** standard object.
4. Click **Fields**.
5. In the Fields page, click **Actions** > **Create**.
6. Select the **Choice List (Dynamic)** option.
7. In the Create Dynamic Choice List: Basic Information page, enter the following:

Field	Sample Entry	Explanation
Display Label	Bill-To Address	The label users see in the UI.
Display Width	40	Width of the box displaying the address elements.
Name	BillToAddress	Unique internal name.

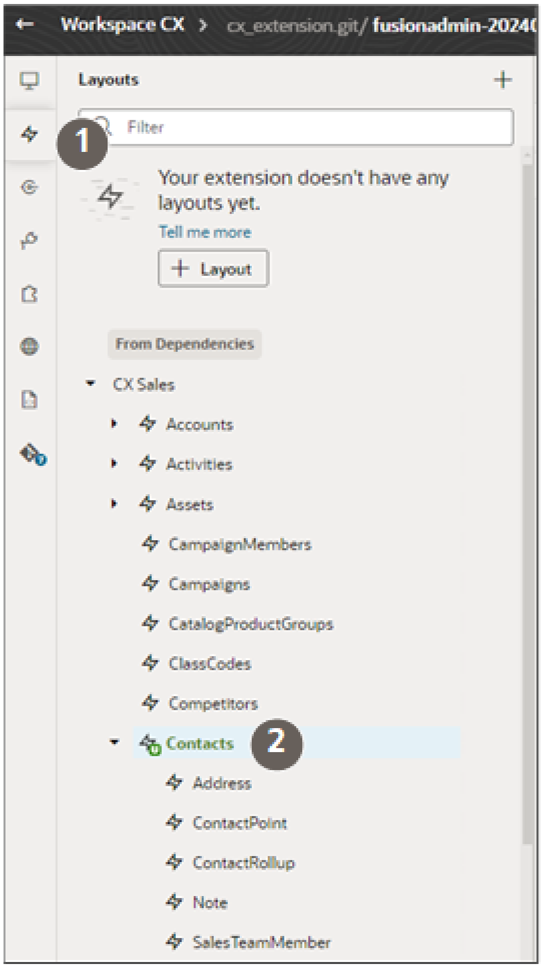
8. Leave the Constraints with the default selected values.
9. Click **Next**.
10. On the List of Values page, make these entries:

Field	Sample Entry	Explanation
Related Object	Address	The source of the values.
List Selection Display Value	Country	You can select any of the values as these aren't used for this use case.

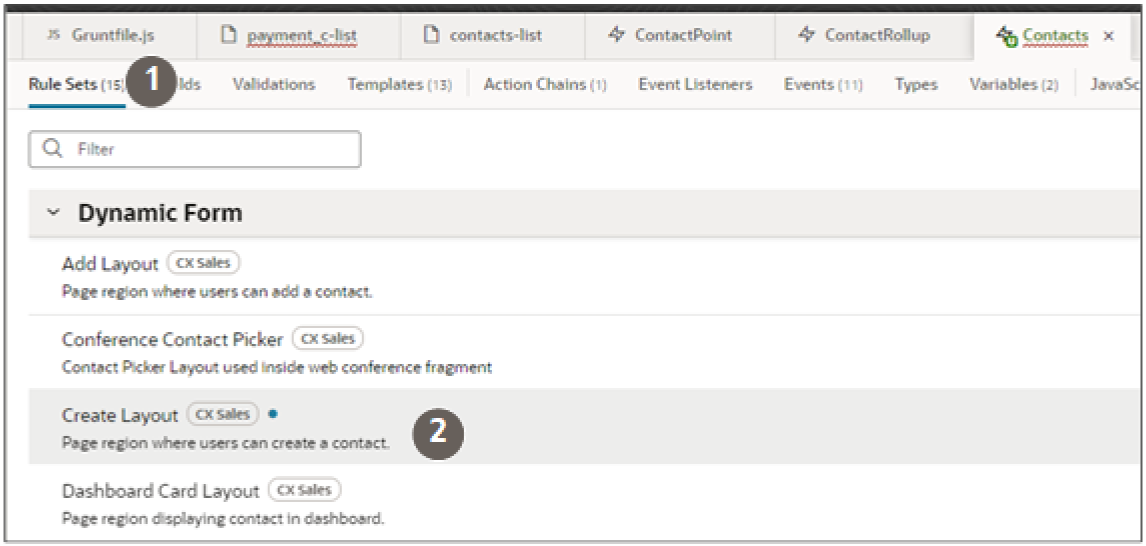
11. You can leave the other sections blank.
12. Click **Submit**.

Specify the DCL Field Behavior and Add It to the Layout

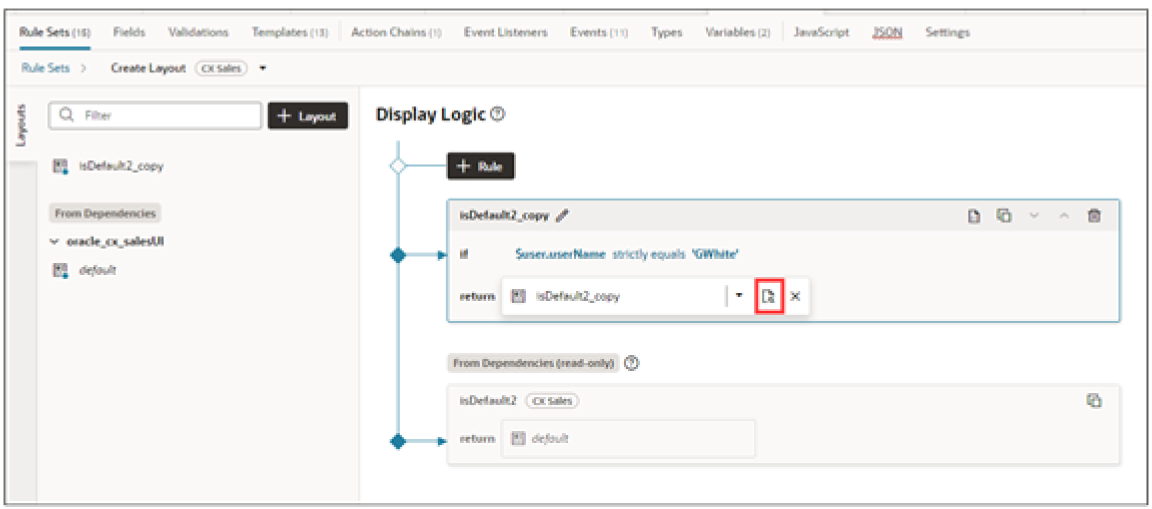
1. Open Visual Builder Studio.
2. Click the **Layouts** tab.
3. On the Layouts tab, click **CX Sales** > **Contacts**.



4. Click **Rule Sets** > **Create Layout (CX Sales)**.



5. Duplicate the default rule with the **Also create a copy of the layout** option selected.
6. Add a rule condition to the new rule.
7. Click **Open** on the new layout rule to open the layout copy.



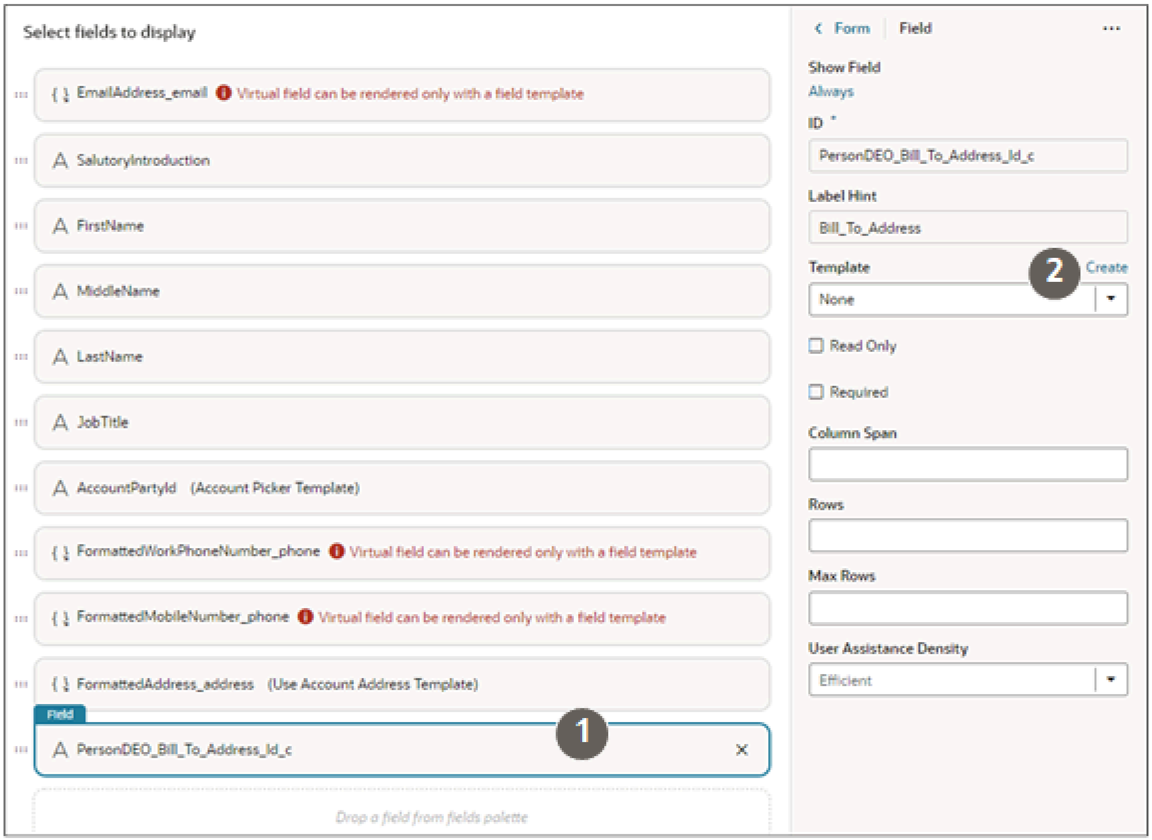
8. Find the **PersonDEO_Bill_To_Address_id_c (Bill-to Address)** field and add it to the layout (highlighted by callout 1 in the following screenshot)
9. Create a variable for the field template:

   a. Still in the Contact layout tab, click **Variables**
   b. Click Create Variables (callout 1 in the following screenshot).
   c. Enter a variable ID, such as **billToAddresses**.
   d. For **Type**, select **Any**.

10. Create a field template that you'll need for the layout:

    a. Click the **Rule Sets** tab.

    b. Click **Create** for the **Template** field (callout 2) in the Field (right-hand) pane.



    c. In the Create Template window, enter a name with no spaces, such as **billToAddressTemplate** and leave the **Enable Extension** option selected.

    d. Click **Create**.

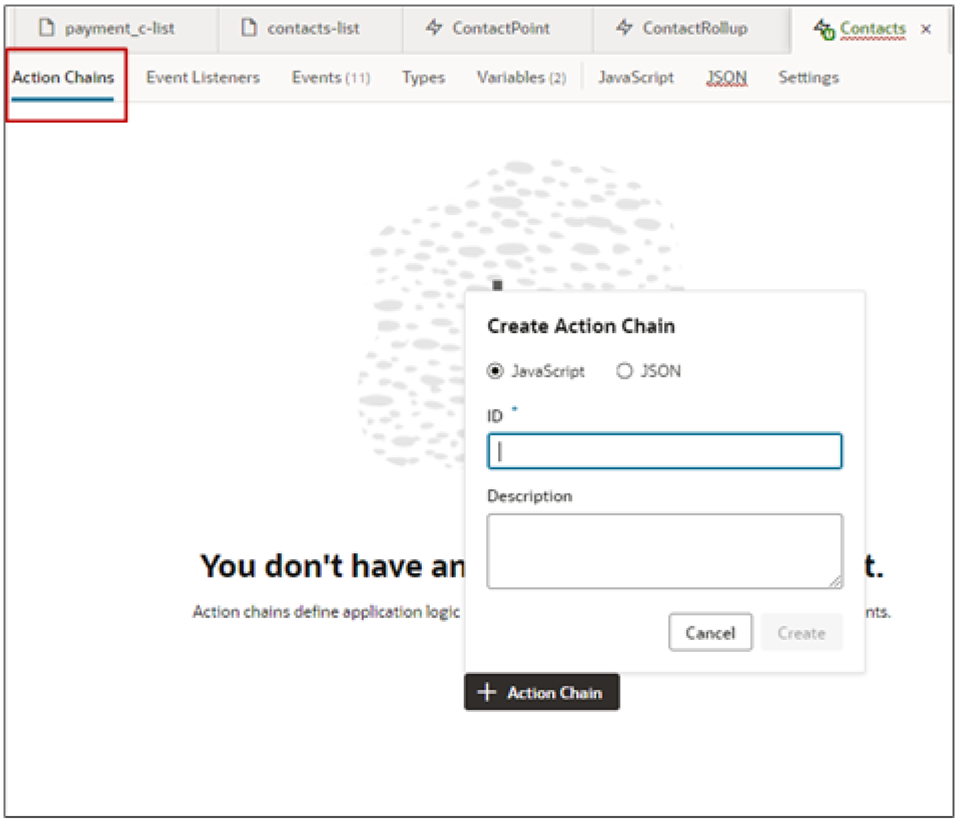
    e. Click the **Code** option

    f. Here's sample code to enter:

```
<template id="billToAddress">
 <oj-select-single label-hint="billToAddressID" data="[[$variables.billToAddresses]]"
 value="{{$value}}"></oj-select-single>
</template>
```

11. Create an action chain that does the following:
    - Check if the updated field in the record is the account Party ID.
    - If the account Party ID is updated, then store that Party ID in the constant accountPartyNumber
    - Create a REST call that returns all of the addresses for that Party ID
    - Store the returned addresses (FormattedAddresses) in an array.
    - Assign the values in the array to the variable billToAddresses which will be part of the Create Contact UI.

    a. On the Contacts tab, click **Action Chains**.

    b. Click **Create Action Chain** (+Action Chain).
    c. In the Create Action Chain window, leave the **Java Script** option selected and enter any name as an ID, in this example: GetAddresses.
    d. Click **Create**.
    e. Switch to the **Code** view and enter the code:
    f. Here's a sample:

```
define([
 'vb/action/actionChain',
 'vb/action/actions',
 'vb/action/actionUtils',
 'ojs/ojarraydataprovider',
 ], (
 ActionChain,
 Actions,
 ActionUtils,
 ArrayDataProvider
```

```
) => {
 'use strict';
 class test extends ActionChain {
 /**
 * @param {Object} context
 * @param {Object} params
 * @param {{row:object,related:object[],fieldsToShow:string[]}} params.previous
 * @param
 {{row:object,previousRow:object,modifiedField:string,pickedRowsData:object,parentRow:object,mode:string}}
 params.event
 * @return {{row:object,related:object[],fieldsToShow:string[]}}
 */
 async run(context, { previous, event }) {
 const { $layout, $extension, $responsive, $user } = context;
 if (event.modifiedField === 'AccountPartyId') {
 const accountPartyNumber = event.pickedRowsData ['accounts.AccountPartyId'];
 const addressesResponse = await Actions.callRest(context, {
 endpoint: 'oracle_cx_salesUI:cx/getall_accounts-Address',
 uriParams:{
 'accounts_Id': accountPartyNumber.PartyNumber,
 },
 });
 if (addressesResponse.ok){
 const billToAddresses = addressesResponse.body.items.map((address)=> {return
 {label:address.FormattedAddress,value:address.FormattedAddress}});
 $layout.variables.billToAddresses = new ArrayDataProvider(billToAddresses,
{keyattributes:"value"});
 }
 }
 if (event.modifiedField === 'PersonDEO_BillToAddress_Id_c'){
 debugger;
 }
 return previous;
 }
 }
 return test;
});
```

12. Create an event listener for the field template:
    a. Click the **Event Listeners** tab.
    b. Click the **Create Listener** button (+Event Listener).
    c. In the Create Event Listener page, select **ContactsOnFieldValueChangeEvent.**
    d. Click **Next**
    e. Select the action chain you just created. In this example, GetAddresses.
    f. Click **Finish**.
13. Test your field:
    a. Click the **Preview** button to test your newly-created field.

    b. On the Contacts list page, enter **Create Contact** in the **Action Bar**.
    c. Select an account that includes a number of addresses.
    d. Click in the **Bill-To-Address** field to select an address.

Change Navigation to Pages in Your Sales Application

Using the Dispatcher feature in Application Composer, you can change which page opens when a salesperson clicks on a record name link on pages in both standard and custom objects. You can redirect links on the list pages, detail pages, and the edit/create pages. The redirected link can open standard or custom pages and subviews. You can specify different destinations for different job roles.

Clicking the opportunity name link on the opportunity list page, for example, normally opens the opportunity detail page, which provides an overview of key activities, contacts, products, and other information. Getting to what a customer is interested in purchasing requires an extra click. If salespeople are more interested in what the customer is buying than in a general overview, then you can open the subview that lists the opportunity products and revenue directly, saving that extra click.

If you created a simple custom object, you can even skip the detail page altogether and open the edit page instead.

How Dispatcher Works

Using the Dispatcher, you can create a set of rules that can open different pages for different job roles. Each dispatcher rule replaces the URLs pointing to the same location. Dispatcher doesn't identify individual links on the page. If a page includes multiple links that go to the same destination, all are replaced. You can even redirect a URL in all the pages in the application to a new destination with one rule.

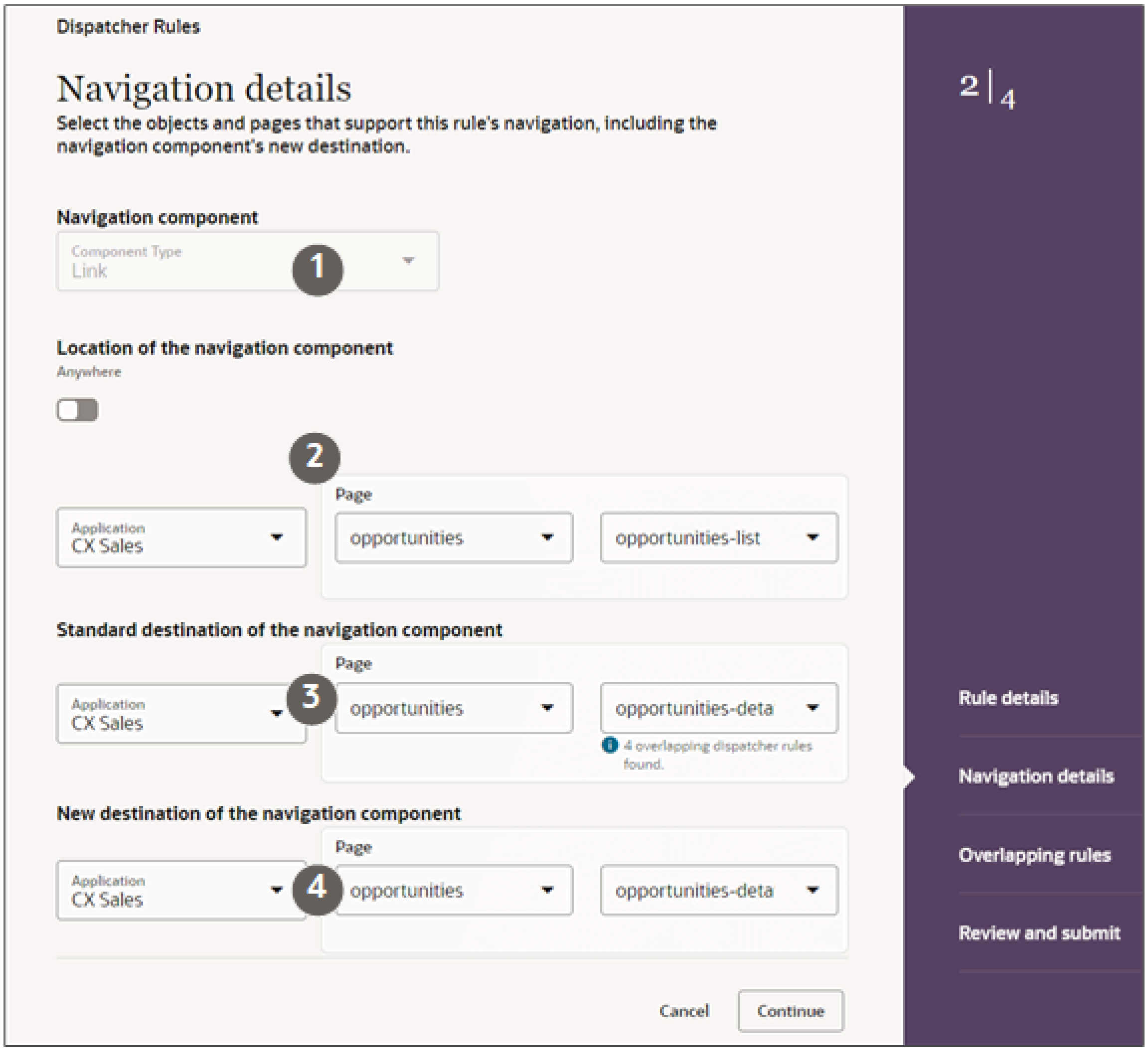
Creating a rule involves 4 steps:

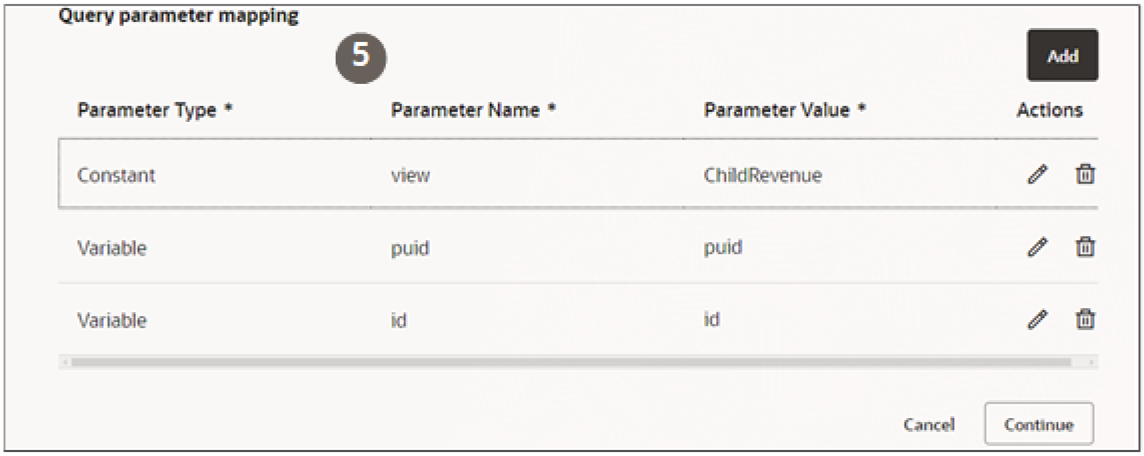
1. Rule Details, where you specify if the rule applies to everyone in the organization or to specific job roles.
2. Navigation Details, where you enter the scope of the redirection rule and both the old and the new destination.
3. Overlapping Rules, where you specify the order in which to process any overlapping rules.
4. Review and submit.

What you enter in the Navigation Details step is key, so here's an overview of the 5 sections in this step. You must scroll down to see the last section. Detailed instructions for creating rules follow.

Section	Description
Navigation Component (1)	In this release, you can redirect only links from the object name link.
Location of the Navigation Component (2)	The scope of the links you want to redirect. You can redirect the links in all the pages of the Sales application, in a specific object, or narrow the scope to a specific page.
Standard Destination of the Navigation Component (3)	The current destination for the link you're redirecting. You can redirect the links on the list page, the detail page, the edit page, and the create page. **Note:** Using Dispatcher, you can't redirect links in subviews.
New Destination of the Navigation Component (4)	The new destination page for the link. Subviews are part of the detail page. So, if you're redirecting the link to a subview, you select the detail page.
Query Parameter Mapping (5)	If you're redirecting a link to a subview, then you identify the subview by adding a constant with a value that you obtain from the subview URL.

Section	Description
	If you're redirecting to an edit page, you add the constant: **mode = edit**. The variables are standard for all standard objects and custom objects created by the CX Extension Generator.





Example Entries for Redirecting Opportunity List Page Links to the Product Revenue Page

Here's what to enter in the Navigation Details step sections to redirect the opportunity name links on the opportunity list page to the Products subview.

- **Location of Navigation Component**

  You're restricting the redirection to the links on the opportunity List page, so make these entries:

Field	Entry
Application	CX Sales
Page	opportunities
2nd Page field	opportunities-list

- **Standard Destination of the Navigation Component**

  Normally, the application opens the detail page when users click the opportunity name on the List page.

Field	Entry
Application	CX Sales
Page	opportunities
2nd Page field	opportunities-detail

- **New Destination of the Navigation Component**
  You're redirecting the navigation to a subview of the detail page, so your entries are the same as for the standard destination. Subviews are part of the detail page.

Field	Entry
Application	CX Sales
Page	opportunities
2nd Page field	opportunities-detail

- **Query Parameter Mapping**
  To redirect to the Product subview, you add a constant with the value of ChildRevenue:

Field	Entry
Parameter Type	Constant
Parameter Name	view
Parameter Value	ChildRevenue

Steps to Create and Activate Dispatcher Rules

1. Open Application Composer outside a sandbox.
2. Click **Dispatcher**.
3. On the Dispatcher page, click **Create**.
4. In the Rule Details page, enter a name for the rule.
5. In the **Rule Conditions** section, specify the audience for the rule. You have two options:
   - Make the rule apply to the all job roles in the organization by turning on **Apply Rule Globally**.
   - Apply the rule to specific job roles you enter in the **Role Filter** field.
6. Click **Continue** to move to the **Navigation Details** step.
7. In the **Location of the Navigation Component** section, specify the scope of the rule:
   - To have the link redirected on all pages, turn on **Anywhere**.
   - Narrow the scope of the redirection to an object and page:
     - In the **Application** field, select either **CX Sales** for standard pages, or **CX Custom**.
     - In the **Page** fields, make these selections:
       a. In the first **Page** field, select the object.
       b. In the 2nd Page field, specify the page type:

Available Values	Description
any	Redirects links on all pages for the object.

Available Values	Description
list	Redirects links on the list page.
edit	Redirects links on the edit and create pages.
detail	Redirects links on the detail page.

8. In the **Standard Destination of the Navigation Component** section, enter the current navigation destination. Your entries identify the URL to be replaced.

    a. In the first **Page** field, select the object.
    b. In the 2nd Page field, select the page.

Available Values	Description
detail	The detail page (called the Overview page at runtime).
edit	The edit/view page.
list	The list page.

9. In the **New Destination of the Navigation Component** section, enter the new navigation destination.

    a. In the first **Page** field, select the object.
    b. In the 2nd Page field, select the page.

Available Values	Description
detail	Redirects to the detail page or subview.
edit	Redirects to the edit or the create page. If you're redirecting to the edit page, then you must also add the constant **mode = edit** in the Query Parameter Mapping section. If you don't add a constant, the user is redirected to the Create page.
list	Select to redirect to the list page.

10. If you're redirecting the link to a subview or to the edit page, then you must add a constant in the **Query Parameter Mapping** section:

    a. Click **Add**.

    b. If you're redirecting to the edit page, then make the following entries:

Field	Entry
Parameter Type	Constant
Parameter Name	mode
Parameter Value	edit

    c. If you're redirecting to a subview, then enter the following:

Field	Entry
Parameter Type	Constant
Parameter Name	view
Parameter Value	Enter the last part of the subview URL following `view=`.

Here's an example of a URL for the Products subview on an opportunity:

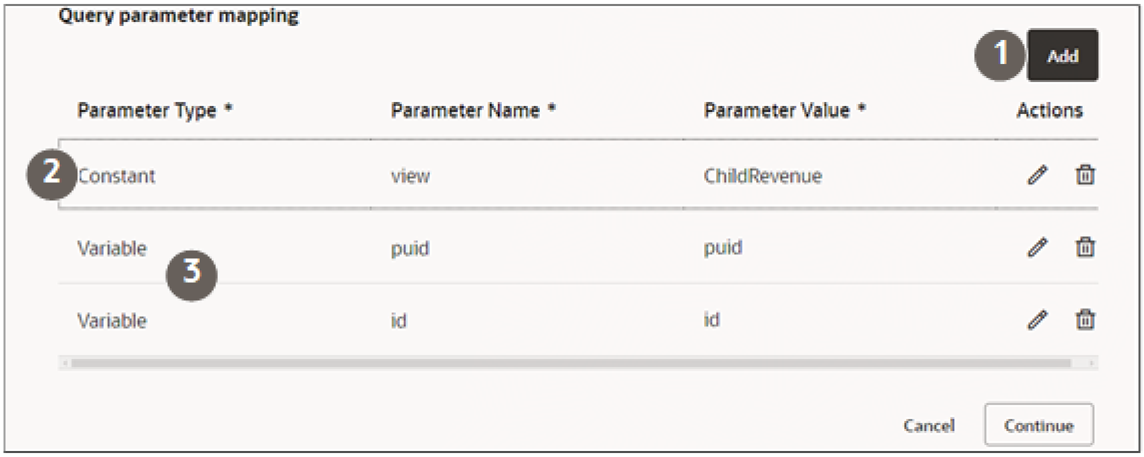
```
https://<domain>/fscmUI/redwood/cx-sales/application/container/opportunities/opportunities-detail?id=300000009863286&puid=39003&view=ChildRevenue
```

> **Note:** For standard subviews and subviews generated by the CX Extension Generator, the application automatically adds 2 parameters: the variables **puid** and **id**. These parameters are required.

Here's a screenshot of the Query Parameter Mapping section

Callout	Description
1	The **Add** button.
2	Constant entry.

Callout	Description
3	The 2 required variables are included automatically.



11. Click **Continue** to move to the **Overlapping Rules** step.
12. Review the order of any rules with overlapping functionality and specify the order of priority by dragging them into position using the handles on each row. The rule at the top gets executed first.
13. Click **Continue** to move to the **Review and Submit** step.
14. Click **Submit**.
15. On the Dispatcher list page, select **Action** > **Mark Active**.

Configure What Information Displays in the Product Catalog

Here's how to configure what information displays in the product catalog in your Sales in the Redwood User Experience application. You can configure both product groups and products and you can configure different layouts for different roles in your organization.

Before you start, make sure that the product catalog includes products, product groups, and the attributes that you want to expose. Attributes that are blank don't show up in the UI.

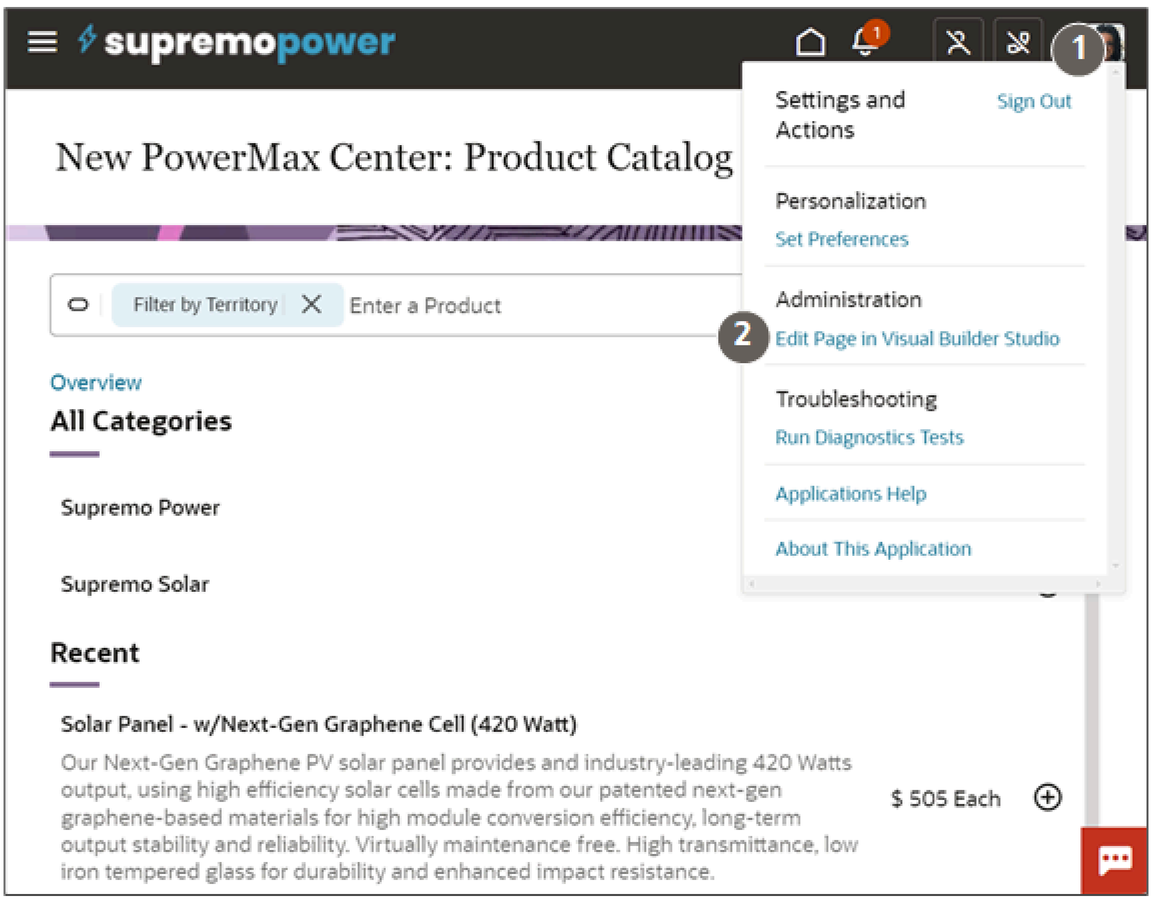
You open Visual Builder Studio from the Product Catalog page and the page must show what you're configuring: product groups and a product under the Recent heading.

Configure Product Groups

1. Open an opportunity.

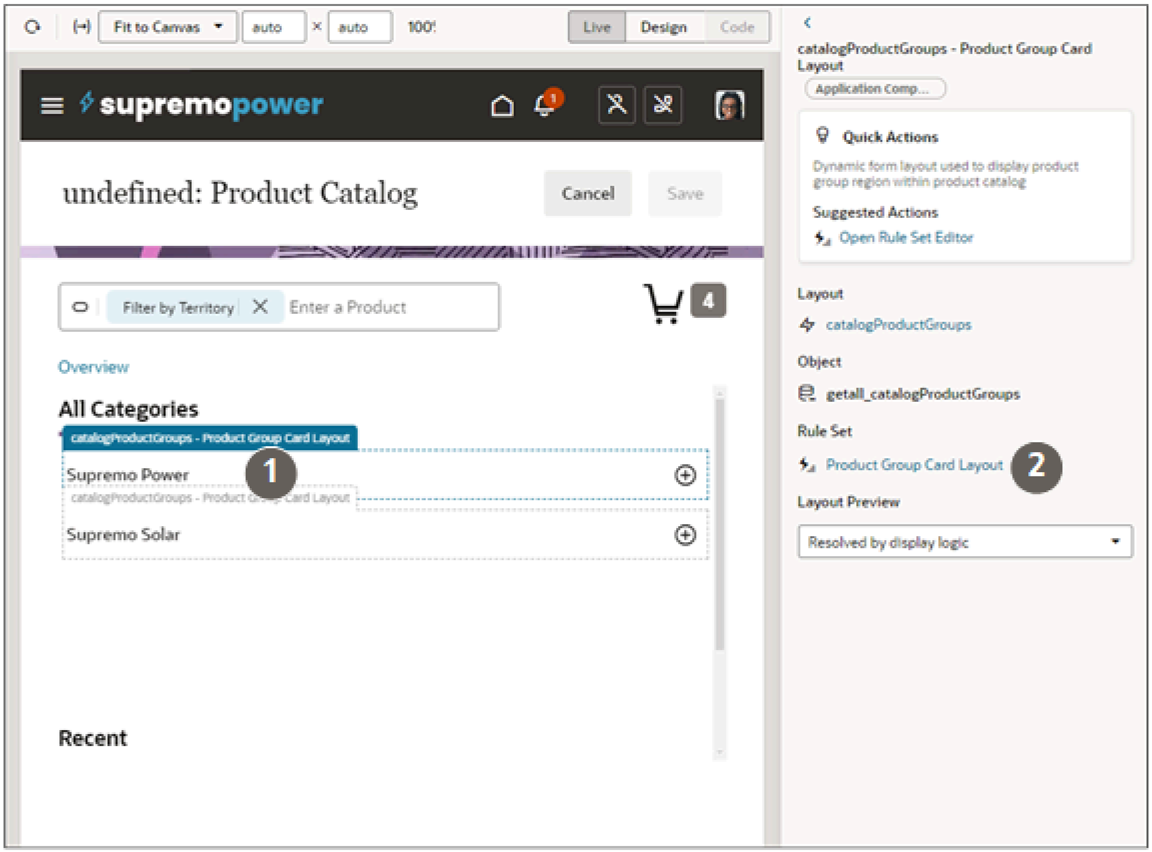
2. Enter **Catalog** in the **Action Bar** and select **Browse Catalog**.

3. In the Product Catalog page, click your profile and select **Settings and Actions** > **Edit Page in Visual Builder Studio** to open Visual Builder Studio (VBS).

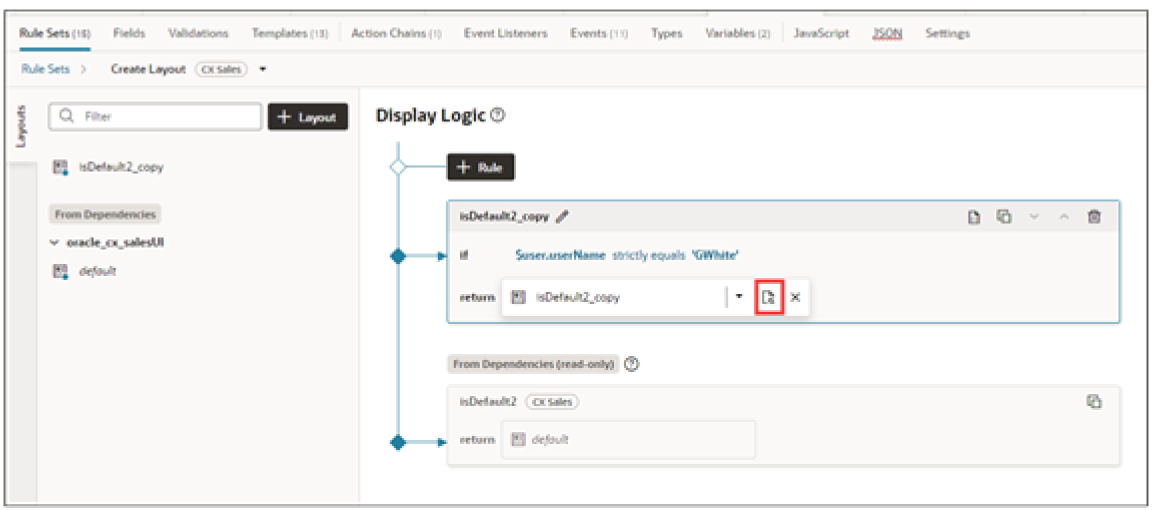


4. In the central VBS panel, click one of the product groups in the page under the **All Categories** heading to display a border for the **Product Group Card Layout** (callout 1 in the following screenshot).

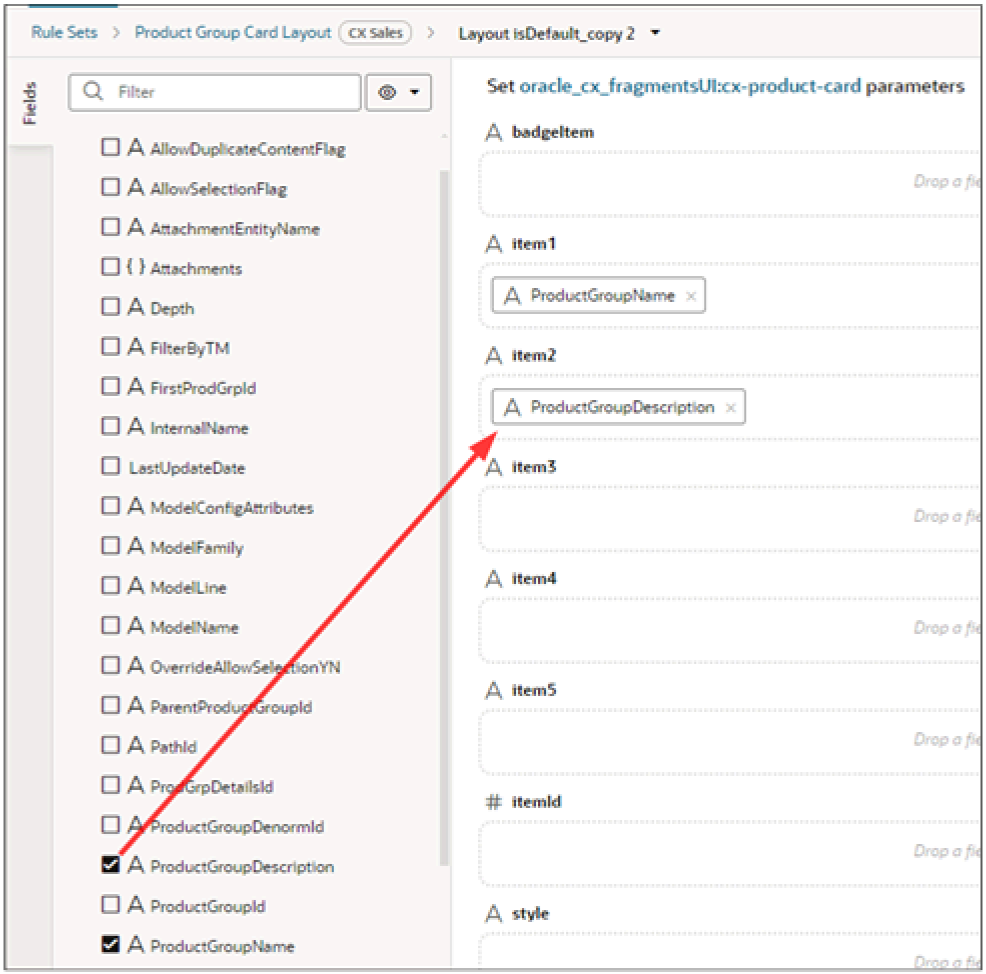
5. Under the **Rule Set** heading in the right pane, click the **Product Group Card Layout** link (callout 2).



6. Duplicate the default layout and open it by clicking the **Open** button highlighted in the following screenshot.



7. You can drag additional fields from the Fields tab. Or you can remove and reorder them.



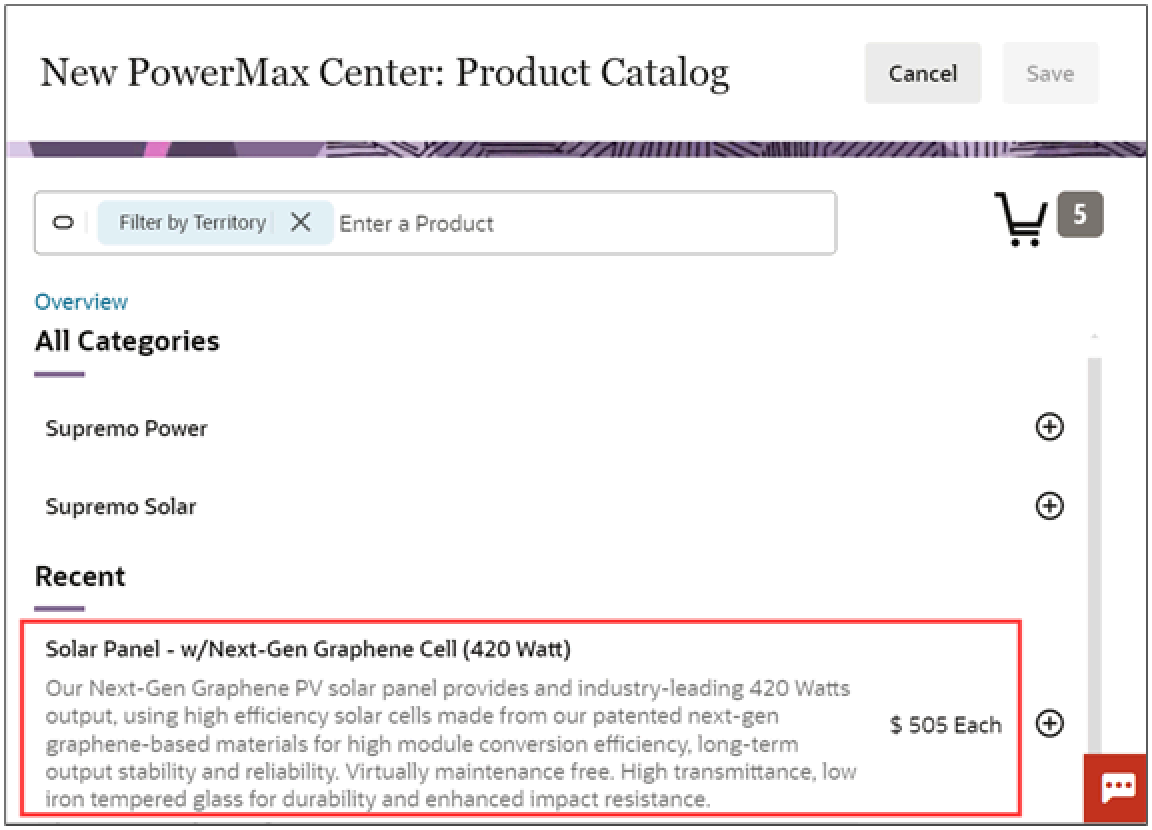
8. Preview your configuration by clicking the **Preview** button.

9. Click **Publish** to make your configuration permanent.

Configure Products

The steps to configure products is very similar to configuring product groups. The main difference: To easily identify the layout, you must display a product under the **Recent** heading of the product catalog. You can do this by adding a product to the opportunity from the catalog and then adding another.

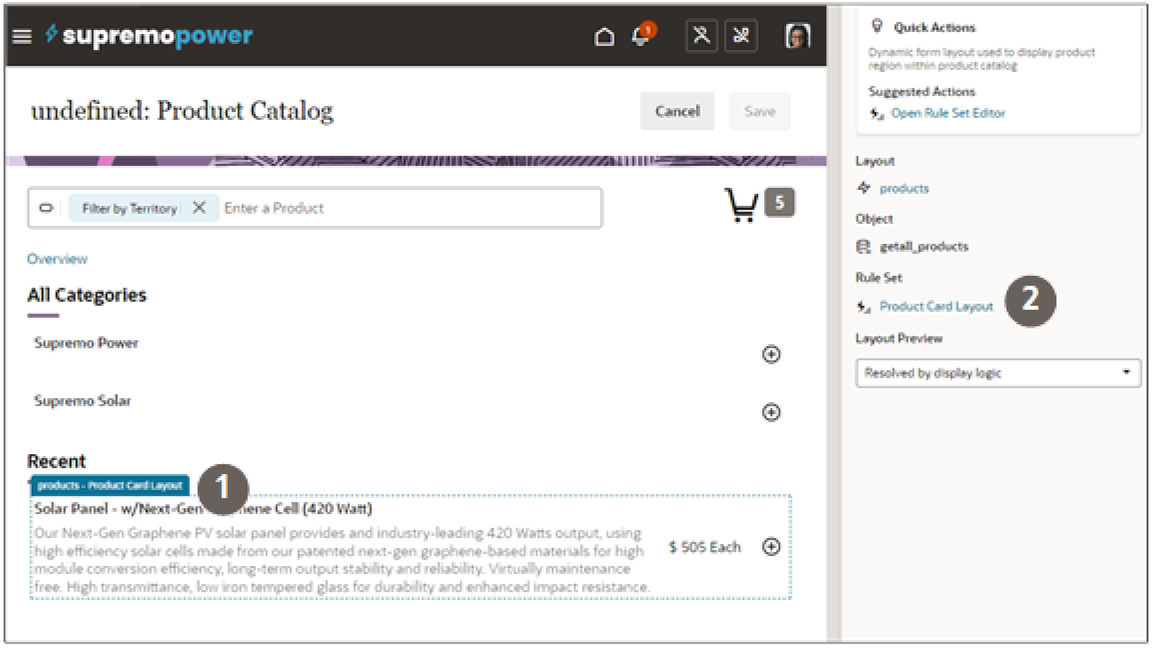
Here's a screenshot of the product catalog showing a product under the **Recent** heading.



Here's a recap of the detailed steps:

1. Open an opportunity.
2. Enter **Catalog** in the Action Bar and select **Browse Catalog**.
3. Add a product to the opportunity from the catalog and save.
4. Add a second product. The first product should appear under the Recent heading.
5. From the Product Catalog page, click **Settings and Actions** > **Edit Page in Visual Builder Studio** to open Visual Builder Studio.
6. Click the product in the page under the **Recent** heading to display a border for the **Product Card Layout** (callout 1 in the following screenshot).

7. Under the **Rule Set** heading in the right pane, click the **Product Card Layout** link (callout 2).



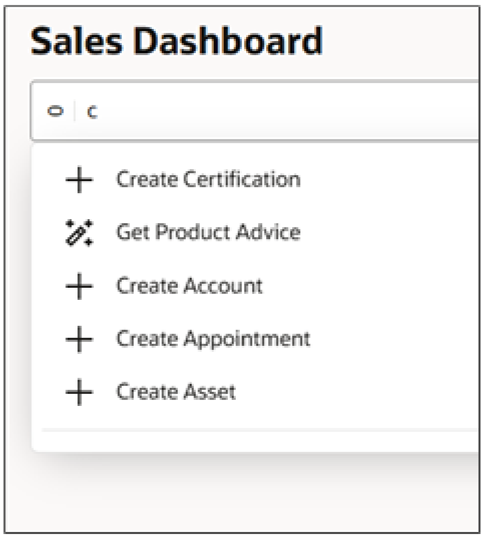
8. In the Display Logic pane, duplicate the default layout and open it.
9. Drag additional fields from the Fields tab. You can also remove and reorder fields.
10. Preview your configuration by clicking the **Preview** button.
11. Click **Publish** to make your configuration permanent.

4 Global Create Actions and AI Agent Integrations

Global Actions in the Sales Dashboard

Salespeople can use a special type of smart action, called a global action, to create records and launch AI agents directly from the Sales Dashboard search bar. Standard smart actions are always used in the context of a particular object and record. Global actions don't depend on context, and so can be used in the sales dashboard search bar.

As salespeople start making entries in the search bar, the enabled global actions are listed automatically.

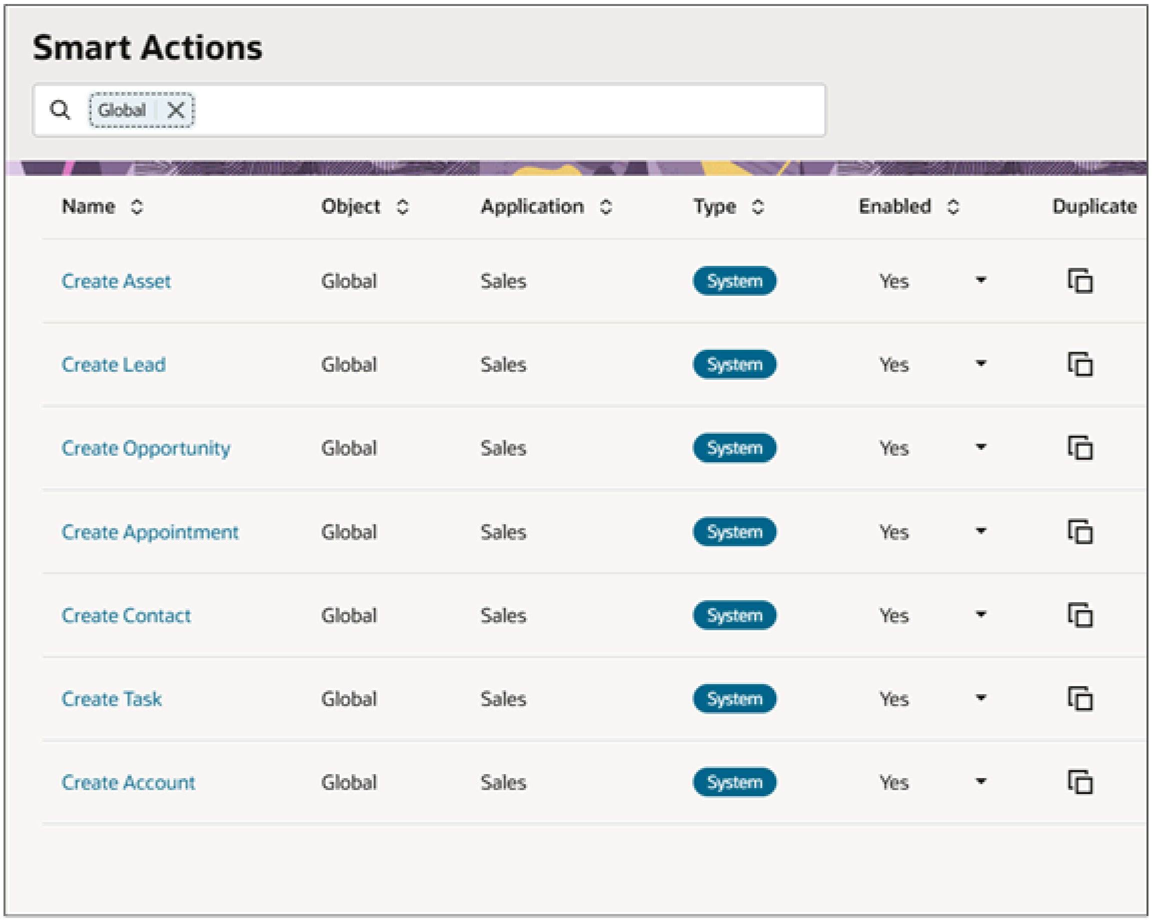


Global Actions that Create Records for Standard Sales Objects

For standard sales objects, the "create" global actions are enabled by default for all sales dashboard users.

In Application Composer, open the Smart Actions work area. Here you can:

- See the list of all global create actions, by entering **Global** as a filter.
- Disable any actions you don't want to use for all users.
- Restrict the actions to specific job roles by duplicating the actions and editing the duplicates.



Global Create Smart Actions for Custom Objects

When you use the CX Extension Generator to create the UI for your custom object, all of the standard smart actions are created for you, including the global create smart action for use in the sales dashboard.

If you used the CX Extension Generator prior to the 25B update for your custom object, or if you didn't use it at all, then generate the global create action for each object in the Extension Generator as described in the topic *Create the Global Create Actions for Custom Objects*.

On the Smart Actions page, you can take the same actions on the generated smart actions as on those available for standard sales objects. You can:

- See the list of all global create actions, by entering **Global** as a filter.
- Disable any actions you don't want to use for all users.
- Restrict the actions to specific job roles by duplicating the smart actions and editing the duplicates.

Smart Action to Open an AI Agent

You create the global action to launch an AI agent in the Smart Action work area in Application Composer. For detailed steps, see the topic *Set Up Global Actions to Launch AI Agents from the Sales Dashboard*.

Create the Global Create Actions for Custom Objects

If you used the CX Extension Generator to generate the custom object UI before the 25B update, or if you didn't use the CX Extension Generator at all, follow these steps to create the global create smart action. Starting with the 25B update, all the standard smart actions are created for you, including the global create smart action for use in the sales dashboard.

1. In a sandbox, open Application Composer.
2. Click **CX Extension Generator**.
3. Click **Create New Extension**.

   There's no need to import files from Oracle Visual Builder Studio for creating smart actions.
4. Add the custom object.
5. Click **Generate Extensions**.

   The application generates the standard smart actions, including the Create global smart action, and automatically downloads a zip file.

   > **Note:** If sales pages for this custom object already exist in Oracle Visual Builder Studio, then don't import the .zip file. You can discard it.

Set Up Global Actions to Launch AI Agents from the Sales Dashboard

Here's how to create smart actions that salespeople can use to launch AI Agents from the search bar in the sales dashboard.

> **Note:** For information on how to create and publish AI agents see the topic *Deploy Sales AI Agents using RAG tools*.

1. Enter into a sandbox that's enabled for Application Composer.
2. In Application Composer, click **Smart Actions**, available under Common Setup heading.
3. On the Smart Actions page, click **Create**.

   Application Composer displays the Create Smart Action guided process in a new browser tab.
4. On the Kind of Action page:
   a. Click the **Global action** option to indicate that you're creating an action for the Sales Dashboard.
   b. Click **Continue**.
5. On the Basic Details page:
   a. In the **Name** field, enter a display name for the action.
   b. The **Action ID** field automatically generates a value based on your entry. You can update the ID with another unique value.

      This value must be unique across all smart actions.

c. Click **Continue**.

6. On the Availability page:

    a. In the **Application** field, **Sales** is the only option and it's selected automatically. Global smart actions aren't available in Service.
    b. In the **Role Filter** field, you can select the roles that can view the action.

        If you don't select a role, then the action will be available to all roles.
    c. Click **Continue**.
7. On the Action Type page:

    a. In the **Type** field, select **AI Agent**.
    b. From the Agent Code list, select the published AI agent.
    c. Click **Continue**.
8. Leave the Confirmation Message page blank and click **Continue**.
9. On the Review and Submit page, review the action's configuration and click **Submit** when ready.