# Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Preface</td>
<td>i</td>
</tr>
<tr>
<td><strong>1 Introduction to Configurator Modeling</strong></td>
<td>1</td>
</tr>
<tr>
<td>The Configurator Models Work Area</td>
<td>1</td>
</tr>
<tr>
<td>How You Create and Maintain Configurator Models</td>
<td>1</td>
</tr>
<tr>
<td><strong>2 Snapshots and Models</strong></td>
<td>7</td>
</tr>
<tr>
<td>Overview of Snapshots and Models</td>
<td>7</td>
</tr>
<tr>
<td>Snapshots</td>
<td>7</td>
</tr>
<tr>
<td>How You Import Items</td>
<td>10</td>
</tr>
<tr>
<td>Configurator Models</td>
<td>13</td>
</tr>
<tr>
<td>How You Refresh Snapshots</td>
<td>16</td>
</tr>
<tr>
<td>FAQ for Snapshots and Models</td>
<td>18</td>
</tr>
<tr>
<td><strong>3 Workspaces</strong></td>
<td>21</td>
</tr>
<tr>
<td>Overview of Workspaces</td>
<td>21</td>
</tr>
<tr>
<td>How Workspaces, Snapshots, and Models Work Together</td>
<td>21</td>
</tr>
<tr>
<td>Workspaces</td>
<td>23</td>
</tr>
<tr>
<td>Versions</td>
<td>28</td>
</tr>
<tr>
<td>How You Release Workspaces</td>
<td>30</td>
</tr>
<tr>
<td>FAQ for Workspaces</td>
<td>34</td>
</tr>
</tbody>
</table>
## 4 Model Structure

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Overview of Model Structure</td>
<td>37</td>
</tr>
<tr>
<td>Item-Based Model Structure</td>
<td>37</td>
</tr>
<tr>
<td>Supplemental Model Structure</td>
<td>39</td>
</tr>
<tr>
<td>Option Features and Options</td>
<td>40</td>
</tr>
<tr>
<td>Integer and Decimal Features</td>
<td>41</td>
</tr>
<tr>
<td>Boolean Features</td>
<td>42</td>
</tr>
<tr>
<td>Text Features</td>
<td>43</td>
</tr>
<tr>
<td>Supplemental Attributes</td>
<td>44</td>
</tr>
<tr>
<td>How You Manage Supplemental Structure Using ADFdi Integration</td>
<td>46</td>
</tr>
<tr>
<td>Configurator Model Node Properties</td>
<td>50</td>
</tr>
<tr>
<td>FAQ for Model Structure</td>
<td>52</td>
</tr>
</tbody>
</table>

## 5 Model Rules

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Overview of Model Rules</td>
<td>55</td>
</tr>
<tr>
<td>Rule Principles</td>
<td>55</td>
</tr>
<tr>
<td>Create Statement Rules</td>
<td>66</td>
</tr>
<tr>
<td>CDL Reference</td>
<td>73</td>
</tr>
<tr>
<td>Extension Rules</td>
<td>94</td>
</tr>
<tr>
<td>FAQ for Model Rules</td>
<td>107</td>
</tr>
</tbody>
</table>

## 6 Model User Interfaces

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Overview of Model User Interfaces</td>
<td>109</td>
</tr>
<tr>
<td>Configurator Model User Interfaces</td>
<td>109</td>
</tr>
<tr>
<td>The Default User Interface</td>
<td>110</td>
</tr>
<tr>
<td>Generated User Interfaces</td>
<td>111</td>
</tr>
<tr>
<td>How You Synchronize UIs with Structure</td>
<td>112</td>
</tr>
<tr>
<td>Multichannel User Interfaces</td>
<td>113</td>
</tr>
<tr>
<td>Templates, Pages, and Navigation Styles</td>
<td>114</td>
</tr>
<tr>
<td>How You Modify User Interfaces</td>
<td>117</td>
</tr>
<tr>
<td>How You Modify UI Elements</td>
<td>121</td>
</tr>
<tr>
<td>The UI Expression Language</td>
<td>124</td>
</tr>
<tr>
<td>How You Use Images for Selections</td>
<td>126</td>
</tr>
<tr>
<td>How You Visualize Configurations</td>
<td>127</td>
</tr>
<tr>
<td>FAQ for Model User Interfaces</td>
<td>130</td>
</tr>
<tr>
<td>Chapter</td>
<td>Section</td>
</tr>
<tr>
<td>---------</td>
<td>---------</td>
</tr>
<tr>
<td>7</td>
<td>Connectors</td>
</tr>
<tr>
<td></td>
<td>Overview of Connectors</td>
</tr>
<tr>
<td></td>
<td>When To Use Connectors</td>
</tr>
<tr>
<td></td>
<td>How You Create Connectors</td>
</tr>
<tr>
<td></td>
<td>How You Use Connectors</td>
</tr>
<tr>
<td></td>
<td>Connectors in Action</td>
</tr>
<tr>
<td></td>
<td>How You Integrate Connectors</td>
</tr>
<tr>
<td></td>
<td>What's the difference between a connector and a model reference?</td>
</tr>
<tr>
<td>8</td>
<td>Model Testing</td>
</tr>
<tr>
<td></td>
<td>Overview of Model Testing</td>
</tr>
<tr>
<td></td>
<td>Test Models Interactively</td>
</tr>
<tr>
<td></td>
<td>Recommendations for Efficiency</td>
</tr>
<tr>
<td></td>
<td>The Configurator Runtime Model Test Service</td>
</tr>
<tr>
<td></td>
<td>How You Run the Model Test Service</td>
</tr>
<tr>
<td>9</td>
<td>Integration</td>
</tr>
<tr>
<td></td>
<td>How You Integrate with External Applications</td>
</tr>
<tr>
<td></td>
<td>Pricing with Configurator</td>
</tr>
</tbody>
</table>
Preface

This preface introduces information sources that can help you use the application.

Using Oracle Applications

Help

Use help icons ? to access help in the application. 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. Not all pages have help icons. You can also access the Oracle Help Center to find guides and videos.

Watch: This video tutorial shows you how to find and use help.

You can also read about it instead.

Additional Resources

- **Community**: Use Oracle Cloud Customer Connect to get information from experts at Oracle, the partner community, and other users.
- **Training**: Take courses on Oracle Cloud from Oracle University.

Conventions

The following table explains the text conventions used in this guide.

<table>
<thead>
<tr>
<th>Convention</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>boldface</strong></td>
</tr>
<tr>
<td><strong>monospace</strong></td>
</tr>
<tr>
<td>&gt;</td>
</tr>
</tbody>
</table>

Meaning

- Boldface type indicates user interface elements, navigation paths, or values you enter or select.
- Monospace type indicates file, folder, and directory names, code examples, commands, and URLs.
- Greater than symbol separates elements in a navigation path.
Documentation Accessibility

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

Contacting Oracle

Access to Oracle Support

Oracle customers that have purchased support have access to electronic support through My Oracle Support. For information, visit My Oracle Support or visit Accessible Oracle Support if you are hearing impaired.

Comments and Suggestions

Please give us feedback about Oracle Applications Help and guides! You can send an e-mail to: oracle_fusion_applications_help_ww_grp@oracle.com.
1 Introduction to Configurator Modeling

The Configurator Models Work Area

The Configurator Models work area is the starting point for working with Oracle Fusion Configurator.

- Oracle Fusion Configurator functionality is based on configurator models.
- The Configurator Models work area is where you work with models based on snapshots, in workspaces.
- When you log in to Configurator Models, you always start on the Overview page.
- Since all changes to models are managed through workspaces that are in development status, the Overview page provides a filtered list of them, as a starting point.
- If your privileges allow it, you can create workspaces directly from the Overview page.
- If you don’t want to start from the list of workspaces in development status, then use the tasks in the task list to manage objects directly.
- Dynamic tabs provide concurrent access to all major tasks.

Related Topics

- Overview of Snapshots and Models
- Overview of Workspaces
- Overview of Model Structure
- Overview of Model Rules
- Overview of Model User Interfaces

How You Create and Maintain Configurator Models

Host applications can use configurator models to ensure that orders contain valid configurations of complex products and services.

To create and maintain configurator models, you perform all of these tasks, in the Configurator Models work area:

<table>
<thead>
<tr>
<th>What You Do</th>
<th>What You Work With</th>
<th>Why You Do This</th>
<th>Who You Are</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bring product data into the Configurator Models work area</td>
<td>Snapshots • Models</td>
<td>To make product data available for configuration by creating the configurator model.</td>
<td>Product expert</td>
</tr>
<tr>
<td>Make the configurator model ready for modification</td>
<td>Workspaces • Model drafts</td>
<td>To set up a safe workspace separate from production where you can try out draft changes to the model.</td>
<td>Model designer</td>
</tr>
</tbody>
</table>
## Introduction to Configurator Modeling

<table>
<thead>
<tr>
<th>What You Do</th>
<th>What You Work With</th>
<th>Why You Do This</th>
<th>Who You Are</th>
</tr>
</thead>
</table>
| Modify the configurator model draft to add configuration functionality | • Supplemental structure  
• Configurator rules  
• User interfaces | • You add supplemental structure nodes to enable richer runtime interaction with the model  
• You add rules to control how the end user can configure the model  
• You add user interfaces to present the model in the most effective way | • Model designer  
• Rules programmer  
• UI designer |
| Make the configurator model available in production | • Workspaces  
• Release process | To put the latest version of the model into production, for end users to configure the product with. | • Product release strategist |
| Maintain the configurator model                  | • Versions                     | To keep up with or get ahead of changes in your product line as it evolves.       | • Product life cycle strategist  |

**Caution:** The procedures here are summaries. They don't include all details. Be sure to consult the other chapters of this guide before working on your production data.

This flow chart summarizes the model creation and maintenance flow. You import a product model item, as a snapshot that becomes a configurator model. You modify and test the model in a workspace, then release the workspace to produce a new version of the model.

The figure shows what happens after you import a model item from the Product Information Management work area, as a snapshot. That snapshot becomes a configurator model that you can modify in a workspace. The workspace includes a draft of the model, to which you add supplemental structure, configurator rules, and user interfaces. You can release the workspace to create a new numbered version of the model, in the production environment for a host...
application. This released configurator model includes all the rules, structure and UIs that you added. End users of the host application interact with the model to produce valid configurations of the item.

**Bring Product Data into the Configurator Models Work Area**

To create a configurator model, the first step is to import a product model item from the Product Information Management work area. Here’s a summary of that process:

1. On the Manage Snapshots page of the Configurator Models work area, select **Actions > Import Model Item** to start the import process. After selecting a product model to import, submit the import, which creates a scheduled process.
2. If the scheduled import process is successful, it creates a snapshot of the product model's data. You can search for the new snapshot on the Manage Snapshots page.
3. The import process automatically creates a configurator model from the snapshot. You can search for the new model on the Manage Models page. The new model is the baseline for all further draft modifications and versions of the model.

Make the Configurator Model Ready for Modification

Before you can modify a configurator model to add configuration functionality, you have to add the model to a workspace. Here's a summary of how to do that:

1. On the Manage Workspaces page, create a workspace. The workspace contains the additions and changes that you intend to make to one or more configurator models. You can add related models to the workspace, and edit them.
   
   The workspace contains the additions and changes that you intend to make to one or more configurator models. You can also add related models to the workspace and edit them.
   
   When you add an object to a workspace, it's called a participant of that workspace.

2. Set an effective start date for the workspace. The effective start date is the date on which the changes to any objects in the workspace take effect. You can change the effective start date until you release the workspace.
   
   While you're editing a workspace, its Status is In development.

3. Open the new workspace. On the Workspace Attributes and Participants page, add the configurator model that you want to modify to the workspace by selecting Actions > Select and Add > Models. Then search for and select a model item that exists in the Product Information Management work area.
   
   In the context of this workspace, the imported model item becomes a draft of the next version of the production model.

Modify the Configurator Model Draft to Add Configuration Functionality

While a model is a draft, you can add supplemental structure, model rules, and user interfaces that enhance the ability of end users to configure the product. You perform any modifications to a model draft by editing a draft of the model inside a workspace.

- On the Workspace page, select and edit a participating model.
- On the Structure tab of the Edit Configurator Model page, you can create model structure that supplements the imported configurable structure of a model item.
- On the Rules tab, you can create configurator rules that govern how the product can be configured.
- On the User Interfaces tab, you can create user interfaces that supplement the default user interface provided by Configurator.
- You can test the behavior of the model draft at any time while you're editing it, by clicking Test Model. This launches a test session in a new Test Model tab, using test parameters that you specify.
- As a simple example of the kinds of configuration functionality you can add, consider a model that represents a car:

<table>
<thead>
<tr>
<th>Model Element</th>
<th>Added definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Structure</td>
<td>A set of options that model the guided selling question: “What kind of driving do you plan to do?”</td>
</tr>
<tr>
<td></td>
<td>- Local trips</td>
</tr>
</tbody>
</table>
### Make the Configurator Model Available in Production

To put your workspace modifications to the model draft into production, you must release the workspace that contains your draft. The draft modifications take effect in production upon the effective start date of the workspace.

1. After editing and testing your model, return to the Workspace Attributes and Participants page.
2. Ensure that the effective start date of the workspace is correct. You can change the date until you release the workspace.
3. Click **Release** to release the workspace, which creates a scheduled process for the release.

**Caution:** Always test your model draft before releasing a workspace, since your modifications might have far-ranging effects, and you can’t undo a release.

4. If the scheduled import process is successful, it creates a new version of the model. The new version appears on the Manage Models page, with an incremented version number.
5. The newly released version of the model becomes the effective version of the model in production. The new version also becomes the baseline for further draft modifications of the model.

### Maintain the Configurator Model

After a configurator model is released, you can update it to further modify its configuration functionality, or to reflect changes in the underlying product model item. Here are the most typical ways to do that:

- To add new configuration functionality for a model, create a new workspace, on the Manage Workspaces page. Then add the model to it, as a new draft.

  In the new workspace, add new supplemental structure, configurator rules, or user interfaces. Then test the model and release the workspace, producing a new version of the model.

- To reflect changes in the underlying product model item in the Product Information Management work area, you need to add the corresponding configurator model to a workspace. So, on the Workspace Attributes and Participants page, select the model and select **Actions > Add Updated Item Snapshots for Models.** If there are changes in the underlying product model item, then the snapshot is added to the workspace, causing the configurator model structure to reflect any changes to the product item. The changes in the model produce a new configurator model version when you release the workspace.
Security for Working in the Configurator Models Work Area

To perform any of the actions described here, you must have the Product Configurator Manager job role.

For more information on the Product Configurator Manager job role, see the Oracle SCM Cloud Security Reference for Product Management.

Related Topics

- How Workspaces, Snapshots, and Models Work Together
- Configurator Models
- Item-Based Model Structure
- How You Create Statement Rules
- Configurator Model User Interfaces
2 Snapshots and Models

Overview of Snapshots and Models

This chapter covers snapshots and models. Snapshots are imports of product data from which configurator models are created. The snapshots capture the product data as of the time of the import.

Snapshots are read-only definitions of approved product data imported from the Product Information Management work area into the Configurator Models work area.

You access snapshots and models in the following ways.

- **Configurator Models work area > tasks panel tab > Manage Snapshots**
  The Manage Snapshots page allows you to search for snapshots, by name, description, status, snapshot type, or structure item type. You can import snapshots of new product data or refresh the product data for selected snapshots.

- **Configurator Models work area > tasks panel tab > Manage Models**
  The Manage Models page allows you to search for models, by the name or description of an original product item or of the model created from that item. You can open any of the drafts or released versions of a selected model.

Snapshots

Snapshots are imports, into the Configurator Models work area, of product data maintained in Oracle Fusion Product Hub, which is accessed through the Product Information Management work area. Snapshots copy production data at a point in time. Each snapshot of a model item corresponds to a configurator model that’s created from it. Snapshots in Released status are the source of the product data for configurator models released into production. When a model item is imported as a snapshot, an item-based configurator model is automatically created. After import, you can refresh snapshots to maintain the synchronization of an item-based model with changes to its original source item in Product Hub.

Aspects of snapshots include:

- Creating Snapshots
- Purpose of Snapshots
- Updating Snapshots
- Sharing Snapshots
- Capturing Data in Snapshots
- Snapshots and Versions

Creating Snapshots
A new discrete snapshot is created for each product item by the Import operation, resulting in a snapshot status of **Released**. Snapshots can be updated by the Refresh operation, resulting in a snapshot status of **Modified**.

Snapshots of item classes and value sets associated with item components in a model's structure are imported when the model is imported for the first time. The important data associated with an item class is the set of transactional item attributes defined on it. The value sets that are imported are those that are associated with the TIAs in the imported item classes. When item classes or value sets are updated in the Product Information Management work area, you must explicitly refresh their snapshots in the Configurator Models work area to obtain the updates. These snapshots aren't automatically updated when the associated model is refreshed.

**Caution:** To keep transactional item attribute values current in your configurator models, you must refresh the item class and value set snapshots associated with the model.

### Updating Snapshots

When the source product data item in Product Hub changes, you can refresh the snapshot corresponding to that item, if a business decision requires doing so. Configurator then sets the status of the snapshot to Modified, indicating a possible need to update the configurator models built on that snapshot data. To update model drafts using the modified product data in the snapshot, you must add the modified snapshot to the workspaces containing the model drafts. When the model updates are complete and tested, and you release the workspace into production, then the snapshot status changes from Modified to Released. Remember that item class and value set snapshots must be explicitly refreshed.

### Updating Snapshots After Item Type Changes

Once you have imported an item, you can't change the type of the item in the Product Information Management work area and then refresh its snapshot. Such a refresh will either fail outright or lead to unexpected behavior when viewing or testing the model in the Configurator Models work area, and possible issues if the model is released to production without testing and troubleshooting. For example: Assume that you previously imported model M1 containing an item I1 which has a Structure Item Type of Standard Item. You change the type of item I1 to Option Class, and add some Standard Items as its options. If you refresh the snapshot of I1, the refresh fails.

To remedy the refresh failure, you need to restore the original item and then create a new item:

1. Undo the effects of the type change on the original item by changing the item back to its original item type. Reversing the type might also require you to remove the underlying structure, if any.
2. After restoring the original item type, refresh its snapshot.
3. Test the configurator model containing the original item to ensure that it's working as expected.
4. End-date the original item having the undesired item type, or delete it from the structure. You should only consider deletion if this model was never configured to create orders.
5. Create a new item having the item type that you need, and add it in the appropriate place in the item structure.
6. When you have the intended structure, refresh the corresponding model or snapshot that brings these changes into the Configurator Models work area.

### Sharing Snapshots

Configurator shares the same single snapshot among all models that reference that product item.

The data changes produced by refreshing are applied to any other configurator model drafts that use the snapshot, after the updated snapshot is added to a workspace and the workspace is successfully released.
**Note:** When you refresh a snapshot, any updated data potentially affects every configurator model that's based on that snapshot. You should test every model affected by snapshot refreshes before releasing it into production.

**Capturing Data in Snapshots**
Snapshots capture a selected subset of the data in a product item.

The following table describes the elements of data included in a snapshot in Configurator. These elements are available for display as columns for the snapshots listed on the Manage Snapshots page.

| Name               | Data Type | Source of Data                | Description                                                                 |
|--------------------|-----------|-------------------------------|                                                                            |
| Name               | Text      | Imported from Product Hub     | Item name or number                                                      |
| Description        | Text      | Imported from Product Hub     | Item description                                                         |
| Organization       | Text      | Imported from Product Hub     | Item organization                                                        |
| Status             | Code      | Created by Import operation   | Values are:                                                                |
|                    |           |                               | • **Released**: indicates a Snapshot being used by run time Configurator models |
|                    |           |                               | • **Modified**: indicates a successful refresh with changes detected. If refresh finds no changes, status reverts to Released. If refresh fails, status reverts to Released. |
| Snapshot Type      | Code      | Created by Import operation   | Values are:                                                                |
| Structure Item Type| Code      | Imported from Product Hub     | The type of the imported model item. Values are:                           |
| Last Refreshed By  | Text      | Created by Refresh operation  | Last user who refreshed the snapshot.                                      |
Snapshots and Versions

Although a snapshot is associated with a configurator model, a snapshot doesn’t have versions, as a model does, because a snapshot can’t be modified. A snapshot is a reflection of the state of a product item at the moment it’s imported into the Configurator Models work area. Changes made to item versions or revisions in the Product Information Management work area only affect snapshots if they change the attributes that are imported in a snapshot.

- When a model item is first imported from Product Information Management into the Configurator Models work area, a baseline version of the snapshot of that model item is released.
- The baseline version is also referred to as version zero, because its version number is 0.0.
- The baseline versions of all the constituents of the imported model item (option classes, standard items, reference models, item classes, and value sets) are also released.
- If the model is configured from a transactional application before a completely developed configurator model goes into production by a workspace release, then the baseline version of the model is used instead.

Related Topics

- How Workspaces, Snapshots, and Models Work Together

How You Import Items

You import items from Oracle Fusion Product Hub that have a Structure Item Type of Model to create snapshots from which item-based configurator models are created. When you import an item, Oracle Fusion Configurator copies selected elements of this item data and stores them in a snapshot of the selected item at the time of the import.

Aspects of importing items include:

- Importing Items
- Selecting Items to Import
- Results After Import
- Making User-Defined Attributes Available for Import
Importing Items
You import items on the Manage Snapshots page.

On the Manage Snapshots page, choose the Import Model Item command. In the Search and Select: Product Model Item dialog, search for the item that you want to import.

**Caution:** You can't import the same item more than once. The list of available items excludes items that have already been imported as snapshots. To obtain the latest updates to an item you must refresh its snapshot.

When the search results include the desired item, select that item and click the Submit button. A scheduled process is submitted, and a confirmation message provides you with the process request ID that you can use to check on the status of the Import Product Model Item process on the Scheduled Processes page.

After the process completes successfully, search on the Manage Snapshots page for the item that you imported. The item now appears as a new snapshot in the search results, along with new snapshots for any required child items. If the children are shared among other models that have previously been imported, the child items aren't re-imported. However, if the child items had been previously imported and have had subsequent updates, those changes will be refreshed as part of the parent model import. The status of those updated child snapshots will be Modified.

Selecting Items to Import
You can refine your search for items to import, to help you find the exact item desired.

Any ATO or PTO model that has optional structure can be imported. Any model item for which Primary structure has been defined can also be imported. The suitability of a product model item isn't dependent on the presence or value of a particular attribute.

When you select items to import, in the Search and Select: Product Model Item dialog, you can use the advanced set of search fields, which enable you to include item catalogs and categories in your search. Click the Advanced button to display the advanced search fields.

You can refine the search results by adding columns that display more of the large number of attributes associated with items. Select Add Columns from the View menu, then search for and select available attributes in the Add Columns dialog. You can query by example in the list of available attributes, then select attributes and click the Add button. The selected attributes are displayed in the dialog, and are included in the search results for items to import.

**Note:** Adding attributes to the search results doesn't add them to the set of attributes that are imported.

Results After Import
Importing an item produces a snapshot of the item.

In addition to a snapshot of the item itself, import produces snapshots of:

- The item's item class, if a snapshot of that item class doesn't already exist
- Any parent item classes of the item's item class, if they don't already exist
- The value sets associated with the transactional item attributes defined on the item's item class
The imported item's name or number becomes the Name of the snapshot, and the imported item's description becomes the Description of the snapshot. The following attributes are imported:

- **Item attributes:**
  - Description
  - Item Type
  - Primary UOM Code
  - Indivisible Flag
  - Organization
  - Serial Generation
  - Track in Installed Base

- **Component attributes:**
  - Minimum Quantity
  - Maximum Quantity
  - Default Quantity
  - Optional Children Are Mutually Exclusive
  - Required When Parent Is Selected
  - Start Date
  - End date
  - Sequence Number
  - Instantiation Type
  - Show in Sales

By default, snapshots include only those user-defined attributes that:

- Have been associated to the Configurator Functional Area through an item page on the item class tied to the item, in Product Hub
- Currently have values
- Are defined at the item level

When a model item snapshot is created by import, its snapshot status is automatically set to **Released**, and a corresponding configurator model is created and released. The new model version has the version number 0.0 (zero), and is referred to as the zeroth version. This version is the **baseline** for all further draft modifications and versions of the model. The zeroth model is automatically added to a workspace that can't be edited, which is named with the form `+<snapshot name> Model Creation Workspace`.

**Making User-Defined Attributes Available for Import**

Importing an item produces a snapshot of the item.

By default, the snapshot definition of an item will include only those user-defined attributes that have been associated to the Configurator Functional Area through an item page on the item class tied to the item in Oracle Fusion Product Hub. You may associate attribute groups in various item classes to the item page associated to the Configurator Functional Area to expose different user defined attributes on different items.
Importing Transactional Item Attributes

You make transactional item attributes for an item available to your configurator model by importing, and later refreshing, snapshots of item classes and value sets.

The definition of a TIA for an item is part of the item class from which the item was created in the Product Information Management work area. You provide this definition to your configurator model by importing a snapshot of the item class, and adding that snapshot to the workspace in which you work on your draft modifications to the model. You keep the TIA definition current relative to its original in the Product Information Management work area by refreshing the item class snapshot.

The values for the TIA that are presented to an end user at run time are defined in a value set in the Product Information Management work area. As with item classes, you provide this set of values to your configurator model by importing a snapshot of the value set, and adding that snapshot to the workspace in which you work on your draft modifications to the model. As with item class snapshots, you keep the values current relative to the Product Information Management work area by refreshing the value snapshot. At run time, the end user provides TIA values by selecting an edit control on the item, then selecting or entering the value in a dialog box for the TIA.

There are some restrictions on using TIAs with Configurator:

- TIAs based on Date, Date/Time, Table and Dependent value sets aren’t supported.
- In the definition of a TIA in the item class hierarchy in the Product Information Management work area, modifications associated with child item classes may further constrain the definition of the TIA, but may not relax an inherited definition.

Related Topics
- How Workspaces, Snapshots, and Models Work Together

Configurator Models

Configurator models are the means by which end users configure products and services. Models incorporate the structure of the item to be configured, the configurator rules that you create to define configuration behavior, and optional user interfaces that enhance the end user’s configuration experience.

Aspects of configurator models include:

- Model structure
- Referenced models
- Configurator rules
- User interfaces
- Model versions
The following figure shows the main elements of a configurator model, which are model structure, configurator rules, and user interfaces. The figure also shows how the structure of a configurator model is based on the structure of a model item in the Product Information Management work area.

Model Structure
You can add supplemental structure to the imported model structure that's the foundation for the configurator model.

Configurator models are imported from items maintained in Oracle Fusion Product Hub (which is accessed by the Product Information Management work area) that have a Structure Type of Model. Only the optional structure and
attributes of the product item are imported, because the required elements of the product item can’t be configured by an end user. The option classes and optional standard items for each model item are also imported.

When an end user interacts with a configurator model during a configuration session, he or she selects from or provides values for certain fields and attributes.

The imported model structure is represented in the Configurator Models work area as an expandable hierarchical tree, in the Structure Hierarchy pane of the Structure tab of the Configurator Model page. Each model, option class, or standard item is a node in the tree.

You can add supplemental structure to the imported model structure. You can use the added supplemental structure to represent guided selling questions that solicit inputs from the end user, or to store inputs and selections from the end user that are used in configurator rules. Configurator rules can use inputs to selectively display the most relevant options of a very large model. By adding the available types of supplemental structure feature nodes to the structure tree, you can add lists of options, integer and decimal inputs, and Boolean choices.

You create supplemental structure on the Structure tab of the Edit Configurator Model page. Before you can add supplemental structure, you must add the model to a workspace, as a draft.

Referenced Models
Model items in the Product Information Management work area can have child models, which can in turn have child models. Such child models are imported as Configurator nodes called model references.

In the Product Information Management work area, before import, you can define the number of instances that can be created of a model reference during a run time configuration session. The attribute that defines this setting is called instantiability.

Referenced models have their own supplemental structure, configurator rules, and user interfaces, independent of those belonging to their parent models.

Configurator Rules
You can add configurator rules, which define how separate parts of a model are related to one another and ensure that users will make valid selections when configuring the model. You can write rules between nodes of a model, including nodes in child referenced models.

You define configurator rules by creating and editing statement rules, which are written in the Constraint Definition Language (CDL). You create configurator rules on the Rules tab of the Edit Configurator Model page. Before you can create rules, you must add the model to a workspace, as a draft.

The structure imported from the Product Information Management work area to the Configurator Models work area incorporates some implicit functionality that's similar to rules. For example, some option classes are defined to be required when their parent is selected, or have child options that are mutually exclusive.

User Interfaces
You can add user interfaces (UIs), to enhance the end user’s configuration session experience.

When an end user interacts with a configurator model during a configuration session, the model is presented using one of a possible set of UIs. A default UI is provided for every model, displaying all of the nodes of the model on a single page. You can create alternative UIs that suit the particular needs of your intended end users.
You define UIs on the User Interfaces tab of the Edit Configurator Model page. Before you can create UIs, you must add the model to a workspace, as a draft. When you create a UI, it's generated automatically by applying a template that you select to the model structure.

**Note:** Each configurator model has its own supplemental structure, configurator rules, and user interfaces. These elements of the model aren't inherited by or shared with other models.

### Model Versions

Configurator models are versioned objects. A model can go through multiple drafts before being released in a series of versions.

Every modification to a model that goes into production must be part of a workspace that's released. After the workspace is released, the version number of the model is incremented to the next available number for that model. When the effective start date of the workspace arrives, the model changes become effective.

There can be multiple drafts of the same model being modified at the same time, but they must be in different workspaces. A message notifies you if you add a model to a workspace when another draft exists in another workspace. There can only be a single released version of a model with the same effective start date.

You can view all of the current drafts and release versions of a model by searching for the model on the Manage Models pages and selecting it in the search results.

### Related Topics

- How Workspaces, Snapshots, and Models Work Together
- How You Create and Maintain Configurator Models
- Item-Based Model Structure
- Configurator Rules
- Configurator Model User Interfaces

### How You Refresh Snapshots

You refresh snapshots in order to keep your configurator models synchronized with changes to production items.

### Refreshing Snapshots

To refresh a snapshot:

1. Navigate to the Manage Snapshots page.
2. Search for and select the snapshot to be refreshed. You can refresh snapshots for:
   - Items (models, option classes, and standard items)
   - Item classes
   - Value sets
3. From the Actions menu, select the appropriate refresh action.
   - Refresh
   - Refresh Including Descendant Structure
The refresh actions are described in the information about refresh depth.

4. The selected Refresh action submits a scheduled process, notifying you of the request ID number. You can search for the process and view its log in the Scheduled Processes work area. The process name is Refresh Snapshot.

5. Updates to existing items that are brought into the Configurator Models work area by the refresh process are shown with the status **Modified**. After the workspace is released, the snapshot status is changed from Modified to Released. New items that are brought in by the refresh process are immediately released to production, and have the status **Released**.

## Refresh Depth

You can selectively refresh snapshots, choosing an action that determines how deeply into the item structure any product updates will be reflected.

The actions for snapshot refresh depth are described in the following table.

<table>
<thead>
<tr>
<th>Refresh Depth</th>
<th>Effects</th>
</tr>
</thead>
<tbody>
<tr>
<td>Refresh</td>
<td>Brings all approved changes to the selected item into the snapshot in the Configurator Models work area, including the additional items that are added to the item's structure. Changes to the model item, are restricted to:</td>
</tr>
<tr>
<td></td>
<td>• Operational attributes of the item itself.</td>
</tr>
<tr>
<td></td>
<td>• Immediate item structure, meaning the immediate child items of a model item, or options of an option class.</td>
</tr>
<tr>
<td></td>
<td>• Operational attributes of the child items or options.</td>
</tr>
<tr>
<td>Refresh Including Descendant Structure</td>
<td>Same as for Refresh, but also extends through any descendant structure, such as an option class under an option class.</td>
</tr>
<tr>
<td>Refresh Including Descendant Structure and Referenced Structure</td>
<td>Same as for Refresh, but also extends through any descendant referenced structure, such referenced model items.</td>
</tr>
</tbody>
</table>

## Example of Refresh

The purpose and method of refreshing snapshots is illustrated by the following typical scenario.

1. A model item M1, maintained in Oracle Fusion Product Hub, in the Product Information Management work area, has previously been imported into the Configurator Models work area, creating a configurator model M1 that's subsequently released as a model version.

2. In the Product Information Management work area, a product manager adds a new standard item SI4 to an option class OC2 in the production model M1.

3. The snapshot of the model item M1 in Configurator is now out of date. The new item SI4 won't appear if the model is used in a configuration session.

4. You refresh the snapshot of M1 on the Manage Snapshots page of the Configurator Models work area, using the **Refresh Including Descendant Structure** action.

5. After refresh, the Status of the snapshot of the option class OC2 is now **Modified**, and it now includes the new standard item SI4 for the option class OC2.
An item snapshot definition is similar to a item definition. The snapshot includes the item and its direct components. Components deeper in the structure belong to the item snapshots of their parent items.

6. You add the snapshot to a new workspace containing a draft of the configurator model. The snapshot imparts the latest product data to the model, including the addition of item SI4.

7. You possibly make changes to the model, using the product changes derived from the snapshot.

8. You test the configurator model, to determine whether your changes to the model draft produce the desired configuration behavior.

9. If testing is successful, you release the workspace, which makes the model draft into a new production version, and changes the status of the snapshot to **Released**

### Caution:
When you refresh a snapshot, any updated data potentially affects every configurator model that's based on that snapshot. You should test every model affected by snapshot refreshes before releasing it into production.

10. After the snapshot status becomes Released, its product data updates are reflected in all other models that use that snapshot.

**Related Topics**

- How Workspaces, Snapshots, and Models Work Together
- How You Create and Maintain Configurator Models
- Item-Based Model Structure
- Configurator Rules
- Configurator Model User Interfaces

## FAQ for Snapshots and Models

### What's a snapshot?

A snapshot is an imported copy of the latest configurable data in a product model item that is created and refreshed in the Configurator Models work area. Configurator models are based on the item definition imported as a snapshot.

**Related Topics**

- How Workspaces, Snapshots, and Models Work Together
- How You Create and Maintain Configurator Models

### How can I create a configurator model?

When you import a model item, a configurator model is created automatically, along with a snapshot.

**Related Topics**

- How Workspaces, Snapshots, and Models Work Together
- How You Create and Maintain Configurator Models
Why don't I see any standard items in the model or its option classes, after importing a model?

To be configurable, child items must be marked as optional in the Product Information Management work area. Otherwise, those items will not be imported, and will be absent from the imported model.

Related Topics
- How Workspaces, Snapshots, and Models Work Together
- How You Create and Maintain Configurator Models

Why don't I see any recent changes to my original item reflected in my configurator model?

The corresponding item snapshot needs to be refreshed, and added to the workspace where the configurator model is a participant.

Related Topics
- How Workspaces, Snapshots, and Models Work Together

What's the difference between a product model item and a configurator model?

A product model item includes a full set of structure and attributes that describe the item and enable related processes in which it is used.

A configurator model has only a subset of the structure and attributes of a model item, namely the optional structure that can be configured. A configurator model can also contain supplemental structure, rules, and user interfaces modified to suit your requirements.

Related Topics
- How Workspaces, Snapshots, and Models Work Together
- How You Create and Maintain Configurator Models

Why did a model open read-only?

You can only modify model versions that are drafts, and that you have locked for editing. If a model opens read-only, it may be a released version, or a draft locked by another user.

Related Topics
- Versions
- How Workspaces, Snapshots, and Models Work Together
3 Workspaces

Overview of Workspaces

This chapter covers workspaces. Workspaces are containers in which you edit and test drafts of changes that you make to configurator models. After you release a workspace, changes to drafts become effective on the effective start date that you define for the workspace.

- Workspaces enable you to modify and test one or more draft models prior to release into production.
- Modified snapshots can also be added to workspaces along with models.
- Multiple workspaces allow concurrent draft development.
- After testing, workspaces can be released so that their contents are available in production. All changes made to the models are effective as of the effective start date of the workspace.

You access workspaces in the following ways.

- **Configurator Models work area > Overview page**
  
  The Workspaces in Development table lists workspaces whose status is In Development. You can filter the view to show all workspaces or only those belonging to the current user.

- **Configurator Models work area > tasks panel tab > Manage Workspaces**

  The Manage Workspaces page allows you to search for workspaces, by name, description, status, or creator. You can access released workspaces as well as those in development.

How Workspaces, Snapshots, and Models Work Together

Configurator models are created from snapshots of product item data. Snapshots added to a workspace keep model drafts in that workspace current with item updates when the snapshots are refreshed.
The following diagram shows that item data is imported as a snapshot, is used to create a configurator model, and can be refreshed so that product updates are reflected in a workspace.

**Role of Snapshots**

Each item-based configurator model is created by importing a subset of the product data for that item. Only the item data that’s used by Oracle Fusion Configurator is imported, including user-defined attributes. The imported data is called a snapshot, because it captures the item data at the time of the import.

Configurator maintains the relationship between the snapshot and the original item. If the original item is updated in Oracle Fusion Product Hub, the updates are reflected in the snapshot when you select the snapshot on the Manage Snapshots page and choose the **Refresh** command.
Role of Models

When a product item is imported as a snapshot, an initial version of the item-based configurator model is automatically created from the imported data. Modifications aren't allowed to this initial model version. Modifications to a configurator model must be made within a workspace, where you create a draft of the model based on the initial model version or the most recently released version by adding the model as a participant in the workspace. After you release the workspace, the draft becomes an effective version of the model upon the effective start date that you specified for the workspace.

Role of Workspaces

Workspaces contain drafts of new versions of configurator models. The effective start date for these versions is an attribute of the workspace, which you define when you create the workspace. You add a model to a workspace as a participant. You can add a model by itself, or with its referenced child models.

Workspaces can also contain any refreshed snapshots that correspond to the models in the workspace. When such a supporting snapshot is added to the workspace, any updates to the snapshot are reflected in the model that was created from that snapshot. To add a snapshot to a workspace, select the corresponding model on the Workspace page and select the Add Updated Item Snapshots for Models command.

- You can add an updated snapshot when you add a model to a workspace, by selecting the Include updated item snapshots for models check box on the Select and Add: Models dialog.
- You can't add a snapshot unless it has first been updated by the Refresh operation.

Related Topics

- How You Create and Maintain Configurator Models
- Snapshots
- Configurator Models

Workspaces

A workspace is a container in the Configurator Models work area in which you edit and test drafts of changes that you make to configurator models. After you release a workspace, changes to drafts become effective on the effective start date that you define for the workspace. A workspace is the place where you create all modifications to a configurator model. You modify a model by adding configurator rules, supplemental structure, or user interfaces. A configurator model with a set of modifications made in a specific workspace is called a draft of that model.

Aspects of workspaces include:

- Creating Workspaces
- Adding Participants to Workspaces
- Locking and Unlocking Participants
- Status of Workspaces
- Cautions about Concurrent Development of Participants
- Releasing Workspaces
Creating Workspaces

You create workspaces on the Manage Workspaces page, or on the Overview page, in the list of workspaces in development. When you create a workspace you must specify a unique name and an effective start date, which is the date on which your modifications to model drafts go into effect. You can edit the workspace attributes, including its name and description.

The following figure shows a model item in the Product Information Management work area, and a new workspace with no draft models in the Configurator Models work area. An effective start date must be specified whenever you create a workspace.

![Diagram of Workspaces](image)

**Note:** You can't delete workspaces.

Adding Participants to Workspaces

Before you can modify a model, you must add it to a workspace, thus creating a draft of that model. In the workspace, you can then make changes to the model draft. When an object is added to a workspace it's called a *participant* of that workspace.

- You can't modify a model except as a draft in a workspace.
- If you remove a draft from a workspace, all the changes made to the draft are permanently lost.

You can add the following objects to workspaces as participants:

- Models: You can select and add configurator models from a list of model items that have already been imported as snapshots.
- Referenced models: You can add model's child referenced models when you add the model, or later.
- Snapshots: You can include a model's updated snapshots when you add it, or later.
The following figure shows a model item added to a workspace, without its child referenced models.

The following figure shows a model item added to a workspace, with its child referenced models.
The following figure shows a model item added to a workspace, with its updated snapshot. You would add a snapshot if it had been updated by the Refresh operation, so that the updates would be applied to the corresponding model as part of the changes in the workspace.
Locking and Unlocking Participants

To prevent making changes that conflict with another user's changes to the same object, in the same workspace, you must lock a participant before you edit it. You lock participants on the Workspace page, by selecting them in the list of workspace participants and clicking the Lock button. When you add a participant, it's automatically locked for you.

- While participants in a workspace can be locked, workspaces aren't lockable. Any number of users can edit participants concurrently.
- Locking prevents other users from changing a draft in the same workspace, but doesn't prevent changes to a draft of the same object in a different workspace.

Status of Workspaces

The status of a workspace is displayed in the Status field, among the workspace attributes on the Workspace page. When you create a workspace, its status is set to In development. When you release a workspace, its status is set to Released.

If your workspace release results in a status of In Development - release failed, then contact your help desk with the process ID of the corresponding release process. You can also examine the process log files for the Release Workspace process, in the Scheduled Processes work area, for details that might help you diagnose the issue.

You can't change the status of a workspace, except by releasing it, and you can't undo a release.

Cautions about Concurrent Development of Participants

If a participant in your new workspace already has a draft in another workspace, it's possible that you might make changes that conflict with the changes being made to the same object in that other workspace. To warn you of this possibility, a message informs you of such other drafts when you're adding the object as a participant to the workspace. If the other workspace has the same effective start date as your workspace, then you can't add the same object to both workspaces, because there would be a conflict when both drafts became effective. You must change the effective start date of one of the workspaces to resolve this potential conflict.

Releasing Workspaces

When you have completed testing and are satisfied with all the modifications you have made to the drafts in your workspace, you release the workspace, by selecting the Release button on the Workspace page. Once the workspace is released, all the modifications in the workspace become effective as of its effective start date.

- A draft of an object becomes a new version of that object immediately upon its successful release, and is assigned a new version number.
- You can't modify the effective start date of a workspace after you release it, so it's good practice to release changes just before they're needed in production.
- You can't selectively release participants. All participants in the workspace are included in the release of that workspace.
- When a workspace is released, any participants to which no changes were made in the workspace are removed from the workspace.
- After release, the participants can't be changed.

Related Topics

- Configurator Models
Versions

A version of a configurator model is its definition during a specified time period. A new version of a model is created when a draft of the model is released into production as part of the release of a workspace.

The definition of a model version is made up of the item structure and item attributes defined in the Product Information Management work area, and the supplemental structure, configurator rules, and user interfaces defined in the Configurator Models work area. The version is set when a workspace with its participants is released, and the version number of each participating model is incremented.

The only workspace participants that are versioned are models. Supplemental structure, rules, and user interfaces are part of a model. Versions of these constituents of the model are maintained internally with the model, but these versions aren't accessible to you, and knowledge of them isn't required for releasing new versions of the model.

A version isn't the same as a copy. In a version, only the changes (also called deltas) to the model are stored. The deltas are those changes that you make to a model draft relative to the most recently released version of the model.

The delta changes that you make to a model draft are only effective within the scope of your workspace until you release the workspace and the model draft becomes a released version, in your production environment. Keep in mind that a model draft doesn't become a model version until it's released.
The following figure shows the progress of versions for a model.

Baseline Versions
When you create a workspace and add a model that has already been released in another workspace to the new workspace, you’re working on a draft modification of that released version, which is considered the baseline version for your model draft.

The version of a model with the latest effective start date is referred to as the tip version, since it’s at the tip of the version history. The tip version becomes the baseline version for any drafts of that model in workspaces that are unreleased, and that also have effective start dates later than that of the tip version. If your model draft has an effective start date before the effective start date of the tip version, then the draft won’t have the tip version as a baseline. To use the tip version as a baseline, you must change the effective start date of your draft’s workspace to be later than the effective start date of the tip version.

Versions are numbered. The version number is incremented on each release of the model, and is unique for that model. If there is more than one model in a workspace that’s being released, each model is versioned independently, and is assigned its own next version number upon release of the workspace. You can view and access all the versions of a model, and all its current drafts, by searching for and selecting the model on the Manage Models page.
When a model item snapshot is first imported from the Product Information Management work area, a model is automatically created from that snapshot and released, and its first version is created.

- During the time that a model is a participant of a workspace having status of In Development, the version of the model is displayed as Draft.
- The version number of a model automatically created by importing a snapshot is zero, and that version is referred to as version zero. The import operation adds version zero of the model to a new workspace named with the format + <model name> Model Creation Workspace

Versioning of Referenced Models

The fact that multiple models in a workspace are versioned independently upon release of the workspace applies to referenced models as well.

Releasing a referenced model or an updated snapshot can impact the existing models that are being used in production. The release process automatically performs an analysis of possible impacts and writes messages about them to the log of the scheduled process for the release operation.

Related Topics
- Snapshots
- Configurator Models

How You Release Workspaces

The main purpose of releasing a workspace is to make your changes to the model drafts in that workspace effective in production at a specified date. An important secondary purpose of releasing a workspace is to release modified snapshots.

After you successfully test your model drafts, you release the workspace so that the changes in the workspace are made available for use in transactional applications such as Oracle Fusion Order Management.

While it's possible to set the effective start date of a workspace to any date in the future, for testing, you should always release a workspace into production as close to the effective start date as possible, not well in advance of it. Since the release of a workspace can't be undone, and since further changes can't be released until the effective start date of a workspace has passed, there is a possibility that you could release changes that become effective far in the future, which can't be reversed until that future date has passed. To minimize the risk of unintended restrictions on your models, by default you can't release a workspace more than one day in advance of its effective start date. You can adjust this release threshold by modifying the value of a setup parameter, according to the instructions in the following section.

Modifying the Release Threshold

You can adjust the release threshold for all workspaces by modifying the value of the setup parameter that governs the threshold.

1. Sign in with the Application Implementation Consultant role, navigate to the Setup and Maintenance work area, and select the Order Management setup offering. Select the Pricing functional area, then select, but don't open, the Manage Pricing Parameters task.
2. For the Manage Pricing Parameters task, select Actions > Export to CSV File > Create New. On the Export Setup Data to CSV File page, click Submit. Acknowledge the confirmation message and click OK to continue.
3. To review the progress of the export, select Actions > Export to CSV File > Exporting setup data. Click Refresh to update the view of the export.

4. Once the export process is complete, click Download File, to download the setup parameters in the form of a compressed file.

5. Extract the compressed file and open the ORA_QP_PRICING_PARAMETER_VALUE.csv file.

6. Update the value in the ParameterValueCode column for the parameter QP_WS_RELEASE_THRESHOLD. The default value for the parameter, in days, is 1 (1 day). You can update the value to any decimal value greater than zero. As a best practice, remove all the other rows from the spreadsheet before saving the file.

7. Save the CSV file, and add it back to the compressed file that you first downloaded, replacing the original ORA_QP_PRICING_PARAMETER_VALUE.csv file.

8. To import the modified workspace release threshold parameter, select Actions > Import from CSV File > Create New.

9. On the Import Setup Data from CSV File page, click Browse, then select the modified compressed file to upload from your computer.

10. Once the compressed file has been uploaded and successfully parsed, click Submit to import the modified workspace release threshold value. Acknowledge the confirmation message and click OK to continue.

11. To review the progress of the import process, for the Manage Pricing Parameters task on the Setup: Order Management page, select Actions > Import from CSV File > Importing setup data. Click View Progress in the dialog box.

12. On the Import Setup Data from CSV File Results page, click Refresh periodically until the import process is complete. The workspace release threshold value has now been updated.

13. To test the threshold change, open a workspace with an effective start date that's beyond the new threshold (which means beyond that threshold number of days out from the current date). Click Release.

Upon release, you should receive the following error message, with a date that corresponds to the modified threshold value.

The workspace can't be released because its effective start date is later than the latest allowed release date of 11/3/2019 03:35:51 PM.

For more information on modifying the release threshold, see Updating the Workspace Release Threshold for Oracle Configurator Cloud (Doc ID 2471288.1) on My Oracle Support.

Releases and Snapshots

When product items, item classes and value sets are updated in the Product Information Management work area, you can refresh their corresponding snapshots in the Configurator Models work area to reflect the updates. Such refreshed snapshots have a status of Modified. You can include these modified snapshots in a workspace with their associated model drafts, to impart the product updates to the model, within the scope of the workspace. You test the model drafts until you're satisfied with the effect of the product updates provided by the modified snapshots. Then you release the workspace, putting the model draft into production as a new version, and changing the status of the Modified snapshots to Released. Be aware that the effectivity of snapshots isn't based on the effective start date of the workspace, but on their effective dates defined in the Product Information Management work area. If an included snapshot's earliest change date is before the effective start date of the workspace, then you must change the workspace date to reflect that. If an included snapshot's earliest change date is before the current date, then the workspace participants will be made effective immediately upon its release.

Releases and Versions

The release of a workspace produces new versions of the models participating in that workspace.
When you create a workspace and add a model that has already been released in another workspace to the new workspace, you're working on a draft of that model that's based on the released version, which is considered the baseline version for your model draft. Any changes that you make in that workspace are now tracked as changes on top of the baseline version. None of the changes to a draft appear in production until the workspace is released.

### Releasing a Workspace

To release a workspace:

1. Navigate to the Workspace page for the workspace. (This is the page that contains the effective start date, and the list of participants.)
2. Before releasing a workspace, you can select **Generate Prerelease Report** to create a report that simulates the impacts of the release, without actually performing the release. This option is described elsewhere.
3. Click the **Release** button.
4. The Release action submits a scheduled process, and displays the Process ID number. In the Scheduled Processes work area, you can search for the process using its ID, and view the report associated with its log. The process name is Release Workspace.

   The Process ID of the most recent release request remains displayed on the Workspace page, in the **Release Process ID** field, so that you can refer to it later.
5. While the Release process is running, you can click the **Refresh** button on the Workspace page to monitor the progress of the process.

   The release process may not succeed. Factors that might prevent a successful release can include:

   - The end dating of a participating item, of a transactional item attribute in an item class, or of a value in a value set.
   - The invalid status of configurator rules in the model.

6. Upon release:

   - Any unmodified draft participants are removed from the workspace.
   - No further changes can be made to the participants or attributes of the workspace, including its effective start date.
   - All impacted rules are recompiled.
   - Impact analysis details are written to the log for the scheduled process.
   - The status of the workspace changes from **In development** to **Released**.

If the changes in the workspace being released cause any configurator rules to become invalid, the result depends on whether the impacted model is released in production, or still a draft:

- If the impacted model is a production model version, then the release is prevented. You must first add the impacted model to the workspace and fix the invalid rules.
- If the impacted model is a draft, then the release proceeds, and the invalid rules are identified in the log file for the Release Workspace scheduled process.

7. The following things are released into production:

   - New versions of participating models that have changed, including their supplemental structure, rules, and UIs
   - Updated snapshots of participating items, item classes, and value sets.
Caution: The release of updated snapshots can potentially have an impact on other models that aren't part of the released workspace. The release of models can potentially impact referring models that aren't part of the workspace. The release of new model versions causes a change to the baselines of other draft models in unreleased workspaces having effective start dates later than the effective start date of the newly released model.

As shown in the following figure, drafts of a model go into production as versions upon release of the workspace that contains the draft. Note that a workspace can be released long before its effective start date, but models don’t become production versions until that effective start date arrives. Workspaces containing the same models can be released on the same date, as long as the effective start dates aren’t the same, and the workspace with the later effective start date is released after the workspace with the earlier effective start date.

The following figure shows the progress of versions of a model as drafts of it go into production upon release of the workspace that contains it.

Evaluating the Impact of Releasing a Workspace

You can generate the Prerelease Report to see the effect of releasing a workspace before actually releasing it, so that you can determine whether further changes to your models are required, and make a decision about whether to actually release the workspace.

You can generate a Prerelease Report at any time, independently of releasing a workspace. This capability enables you to iteratively analyze impacts as you work on the draft model structure, rules and user interfaces for the model participants in the workspace.

To generate the Prerelease Report:

1. Navigate to the Workspace page for the workspace.
2. Click the Release split button, then select the option Generate Prerelease Report.
3. Make note of the process ID for the scheduled process, which is named Workspace Prerelease Report. The ID for the most recently generated report remains in the Prerelease Report Process ID field on the Workspace page, for reference. While the report process is running the status of the workspace is displayed as Generating prerelease report. After the process concludes, the status returns to In development.
4. Navigate to the Scheduled Processes work area.
5. Search for the process for your report, using the process name or ID.
6. Open or save the text file attached to the report log.
7. Examine the impacts listed in the report, address any warnings and resolve any errors reported.

Related Topics
- Configurator Models

FAQ for Workspaces

What's a workspace?

A workspace is a container in the Configurator Models work area that organizes modifications to configurator model drafts and snapshots. The modifications become effective in production when the workspace is released.

Related Topics
- Configurator Models

What objects have versions?

Only models have versions. Snapshots and workspaces don't have versions. Snapshots have a Released status (used in production, by models) and an unreleased status called Modified (modifications imparted by a snapshot refresh that haven't been released yet). Workspaces have a Released status, and an unreleased status called In development.

Related Topics
- Configurator Models

What happens if a snapshot becomes effective before a workspace does?

If you add a snapshot that is already effective to a workspace, then the snapshot's effectivity forces the workspace to become effective as soon as it is released, regardless of the effective start date you assigned to the workspace.

The following table summarizes the relationship between the effectivity of a snapshot and the effectivity of a workspace containing that snapshot.

<table>
<thead>
<tr>
<th>If a snapshot is effective</th>
<th>Then the workspace is effective</th>
</tr>
</thead>
<tbody>
<tr>
<td>Already (before the current date and time)</td>
<td>Immediately upon release of the workspace.</td>
</tr>
<tr>
<td>Before the workspace, but not yet</td>
<td>Upon the effective start date of the workspace, which you must change to be the same or earlier than the snapshot's effective change date.</td>
</tr>
</tbody>
</table>
What happens when I release a workspace?

The workspace status changes to **Released**. Other results are as follows.

- Changes to participants made in the workspace become effective upon the effective start date of the workspace.
- Snapshot changes are released to production.
- Unmodified participants are removed from the workspace. Participating models are changed from drafts to new versions, with sequential version numbers.
- You can no longer edit drafts of the participants in this workspace. You can test released models from within a released workspace.
- All rules that are impacted by the release are recompiled, including rules in models outside of the released workspace. If any rules in released models are made invalid by the release, the release is prevented from proceeding.
- Impact analysis details are written to the log for the scheduled process that performs the release. This log information records what was impacted and may suggest subsequent action to be taken.

What happens if I change the effective start date of a workspace?

Changing the effective start date of a workspace alters when model modifications become active, if they are in the future. If you make the effective start date later, and a new tip version of the model has been released before that new date, then your model draft will have a different baseline than it did when you added it to the workspace.

What happens if I remove a draft from a workspace?

If you have made changes on the draft, those changes are lost permanently, and are never applied to the participant.

Can I undo workspace release?

No. Once a workspace has been released, any workspace changes to its participants go into production, creating new versions that become the baselines for future drafts.
Overview of Model Structure

This chapter covers model structure. Configurable model structure is imported, by snapshots, from data in the Product Information Management work area. You can add supplemental structure to enhance the configuration experience.

- Imported item-based model structure enables you to accurately model configuration behavior for product items.
- Imported model structure includes:
  - Model items
  - Option classes
  - Standard items
  - Transactional item attributes

- Supplemental structure enables you to provide a framework for guided selling questions
- Supplemental structure includes:
  - Option features and options
  - Integer features
  - Decimal features
  - Boolean features
  - Text features

You access model structure in the following ways.

- **Configurator Models work area > tasks panel tab > Manage Models page > Search > Versions > click a Draft or Version > Configurator Model or Edit Configurator Model page > Structure tab**

  The Edit Configurator Model page enables you to create and edit the structure, rules, and user interfaces of a configurator model. If the model is locked by another user, then you can only view the model.

- **Configurator Models work area > tasks panel tab > Manage Workspaces page > Workspace page > click Name of participating model > Configurator Model or Edit Configurator Model page > Structure tab**

Item-Based Model Structure

Item-based Configurator models are based on item structures defined within the Product Information Management work area and imported through snapshots.
Purpose of Item-Based Model Structure

Item-based structure for a configurator model enables you to accurately model the product data that's maintained in the Product Information Management work area, and enables end users to make valid choices in configuring products. The structure of an item-based configurator model is determined in the Product Information Management work area, by the Primary item structure of a model item that's created and maintained there, then imported as a snapshot into the Configurator Models work area.

Item structure consists of model items, option classes, and standard items. The item structure is replicated in the configurator model. A model item is a single item that's the root of an item structure. When imported, model items become child reference models, which can't be directly modified when you edit the model.

If a transactional item attribute (TIA) is defined on the item class of an item, then the TIA is displayed on the model tree, below the node corresponding to the imported item. The details pane for the TIA displays the item class that provides the TIA's definition, the value set that provides the TIA's run time values, and the TIA's format details. Both the item class and value set are provided to the model by snapshots.

The item details data imported for each node in a configurator model is displayed in the Overview region of the Structure tab when you're viewing or editing the model. The following details are generated by Configurator, not imported:

- **Snapshot Status** indicates the status of the snapshot for the node, relative to the snapshot's source in the Product Information Management work area.
  - If the snapshot status is **Released**, then the snapshot is current with its source. (This status is also shown on the Manage Snapshots page.)
  - If a snapshot has been refreshed, to reflect updates to the source in the Product Information Management work area, but not yet released as a participant in a workspace, then the snapshot's status is shown as **Modified** on the Manage Snapshots page. If that refreshed snapshot is included in the same workspace with the configurator model, then the model's Snapshot Status is **Updates in workspace**, and those updates are reflected in the model. If the refreshed snapshot isn't included in the workspace, then the model's Snapshot Status is **Updates not in workspace**. To ensure that updates to the source are reflected in the configurator model, you must first add the updated snapshots to the model's workspace. You can use the Add Updated Item Snapshots for Models command on the Workspace page to add the updated snapshots to the model's workspace.

- **Creation Date** is the date when the node's snapshot was first imported.

If there are user-defined attributes defined on the item in the Product Information Management work area, they're displayed on the Item Attributes tab of the model's details. This display is read-only.

Working with Item-Based Model Structure

Model structure is the framework on which you build a configurator model. Item-based model structure imposes certain restrictions.

- Item details of the imported items in model structure are read-only in the Configurator Models work area. For example, you can't modify operational attributes (such as name or description), component attributes (such as minimum, maximum, or quantity), or any user-defined attributes of imported items.

- Only items that can be configured are imported in a snapshot. Consequently, items that are designated as being required in an item structure in the Product Information Management work area aren't imported. Because of this import restriction, some attributes and child items may appear to be missing in a configurator model, when it's compared to the originals.
You can't add or remove item-based nodes from a configurator model. Changes to the item structure must be made in the Product Information Management work area and imported or refreshed through snapshots. However, you can add supplemental structure nodes to an item-based model.

Related Topics
- How You Import Items

Supplemental Model Structure

You add supplemental model structure to an existing item-based model.

Purpose of Supplemental Model Structure

Supplemental model structure enables you to expand an item-based model to sufficiently express the configuration choices that you want your end users to have.

- Supplemental structure allows configuration of types of product features that aren't part of the item-based model based on existing product item data.
- Supplemental structure provides the framework for adding guided selling questions, which guide the end user into a simpler configuration experience by asking questions about what the user wants from a product.

Tip: You can use supplemental structure questions to gather inputs that drive configurator rules that can streamline the number of options displayed to the runtime end user.

- Supplemental structure that you create for a model is part of the definition of that model, like configurator rules and user interfaces. Supplemental structure can't be shared with other models.

The following table summarizes the types of supplemental model structure:

<table>
<thead>
<tr>
<th>This Supplemental Feature Type</th>
<th>Models This</th>
</tr>
</thead>
<tbody>
<tr>
<td>Option Features and Options</td>
<td>Lists of options</td>
</tr>
<tr>
<td>Integer Features</td>
<td>Integer values</td>
</tr>
<tr>
<td>Decimal Features</td>
<td>Decimal values</td>
</tr>
<tr>
<td>Boolean Features</td>
<td>True or False values</td>
</tr>
<tr>
<td>Text Features</td>
<td>Plain text values</td>
</tr>
</tbody>
</table>

Working with Supplemental Model Structure

You add supplemental structure to existing item-based models.
To add supplemental structure:

1. Add an item-based model to a workspace, and open it for editing. The model will be automatically locked by your login ID. You must lock a model to edit it.
2. On the Edit Configurator Model page, select the root node of the model, choose Create from the Actions menu.

   Note: You can only create supplemental structure on the root node of an item-based model.

3. Select the type of supplemental structure feature to create.
4. In the dialog box for the feature type, enter the required attributes for the feature.
5. In the Details region, you can modify the attributes of the feature.
6. Repeat these steps to add other features to the model.
7. Save your changes to the model. If you don’t save, you will be prompted to save when you leave the edit page.
8. Optionally, test the effects of your changes by clicking the Test Model button.

Option Features and Options

Option features are a type of supplemental structure that you can add to a configurator model.

Purpose of Option Features and Options

Option features and options allow you to model inputs that require the end user to select an option from a list during a configuration session.

Option features contain child nodes called options, which act like the choices in a list.

- Option features represent configurable items that contain a list of predefined options.
- Option features enable you to model a list of related choices that you can present to the end user. For example, an option feature named Color might have the options Red, Yellow, and Blue.
- The selection of specific options can be set by configurator rules.
- You can allow more than one option to be selected at a time.
- You can use option features to model multiple-choice questions.
- You can control how many choices are allowed per question, and whether the choices have an associated quantity.
- Users can't enter values for option features, because their value is the set of options chosen by the user.

Working with Option Features and Options

You add option features and options to existing item-based models.

When creating option features:

- The values you enter for Minimum Selections and Maximum Selections control how many options of a feature the end user can select at one time. Set both fields to 1 to model a list of mutually-exclusive options.
- Select Enable Option Quantities to allow end users to enter a quantity for each option of an option feature at run time. If you enable option quantities, then specify the Maximum Quantity per Option.
Sorting Options
By default, the options of an option feature appear in the runtime UI in the same order that you created them. You can specify a different order to be displayed at runtime.

To specify the runtime sort order:

1. Select the option feature.
2. Open the UI Presentation tab in the Details region.
3. Select a value for Option Sort Order. The default choice is As displayed in the structure hierarchy, which leaves the runtime options in the order they were created in.

You can sort the options in ascending or descending order, on either the option's name, or on its description (if there is one). The sorting is always alphabetic. The sorting is applied at runtime. Specifying the sort order doesn't change the order of the options in the structure.

Integer and Decimal Features

Integer features and decimal features are types of supplemental structure that you can add to a configurator model. They are collectively referred to as numeric features.

Purpose of Numeric Features
Numeric features allow you to model inputs that require the end user to enter an integer or a decimal number during a configuration session.

The special characteristics of numeric features are:

- You can use numeric features as inputs to and outputs from calculations.
- You can set the value of a numeric feature with a configurator rule, which can refer to the value of another numeric feature.
- You can specify a minimum and maximum for the value that can be entered in a numeric feature, thus defining the range of a domain of allowable values for the feature at run time.
- At run time, the domain is displayed as a range of valid values, which can be narrowed by constraints that you define with configurator rules.
- You can set the domain ordering for a numeric feature, which controls how Conﬁgurator's autocompletion functionality determines a value for numeric features in the model that don't yet have a value in the conﬁguration session.

Working with Numeric Features
You add numeric features to the root node of existing item-based models.

When defining numeric features:

- You must enter values for the fields Minimum and Maximum, to define the domain of allowable values at run time.

You can enter integer values in a decimal feature. If you attempt to enter decimal values in an integer feature, they are rounded to the nearest integer, and you are notified of the error.
• You can select a value for **Domain Ordering**, which controls how Configurator’s autocompletion functionality narrows the range in the value domain.

### Domain Ordering for Numeric Features

Setting the domain ordering for a feature controls how Configurator’s autocompletion functionality narrows the range in the domain of values for that feature at run time. Autocompletion determines a value for that feature if one hasn’t yet been provided by in a configuration session.

You set domain ordering for a numeric feature in the Details region of the Edit Configurator Model page. Select a value for the **Domain Ordering** control.

The following table describes the effect of setting domain ordering for numeric features.

<table>
<thead>
<tr>
<th>Domain Ordering Setting</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>System default</td>
<td>Accept this setting if you want autocompletion to use its own default method for determining a value. This method provides optimal run time performance and is recommended if you don’t have a preference for the node’s value or selection state at run time.</td>
</tr>
<tr>
<td>Linear search, minimum to maximum (only for integer features)</td>
<td>Select this setting if you want autocompletion to apply values within the specified domain in increasing order, beginning with the node’s <strong>Minimum</strong> value.</td>
</tr>
<tr>
<td>Linear search, maximum to minimum (only for integer features)</td>
<td>Select this setting if you want autocompletion to apply values within the specified domain in decreasing order, beginning with the node’s <strong>Maximum</strong> value.</td>
</tr>
<tr>
<td>Binary search, increasing minimum</td>
<td>Select this setting if you want autocompletion to successively split the domain in a binary search until either a single value is bound, or no solution is found. When you select this setting, autocompletion checks the upper half of the domain first.</td>
</tr>
<tr>
<td>Binary search, decreasing maximum</td>
<td>Same as <strong>Binary search, increasing minimum</strong>, except that autocompletion checks the lower half of the domain first.</td>
</tr>
</tbody>
</table>

### Boolean Features

Boolean features are a type of supplemental structure that you can add to a configurator model.

#### Purpose of Boolean Features

Boolean features allow you to model inputs that require the end user to select a single true or false value during a configuration session.

The special characteristics of Boolean features are:

• You can model yes-or-no choices in a configurator model. For example, in a guided selling scenario, you can ask the end user whether or not a certain type of product characteristic is desired.
• The value of Boolean features can be set by configurator rules.
• Boolean features in a configuration can be true, false, or unbound. You can also set the domain ordering for an Boolean feature, which controls how Configurator’s autocompletion functionality determines a value for Boolean features in the model that don’t yet have a value in the configuration session.
• At run time, a Boolean feature value is displayed by default as a check box. The possible values are: selected (true) declined (false), or blank (unbound).

Domain Ordering for Boolean Features
Setting the domain ordering for a feature controls how Configurator’s autocompletion functionality narrows the range in the domain of values for that feature at run time. Autocompletion determines a value for that feature if one hasn’t yet been provided by in a configuration session.
You set domain ordering for a Boolean feature in the Details region of the Edit Configurator Model page. Select a value for the Domain Ordering control.

The following table describes the effect of setting domain ordering for Boolean features.

<table>
<thead>
<tr>
<th>Domain Ordering Setting</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Prefer false</td>
<td>Select this setting if you want autocompletion to set the Boolean feature to False (declined) first and then, if it can’t be set to False, set it to True (selected).</td>
</tr>
<tr>
<td>Prefer true</td>
<td>Select this setting if you want autocompletion to set the Boolean feature to True (selected) first, and then, if it can’t be set to True, set it to False (declined).</td>
</tr>
</tbody>
</table>

Text Features
Text features are a type of supplemental structure that you can add to a configurator model.

Purpose of Text Features
Text features allow the end user to enter plain text values at run time for use within a configuration session. Configurator rules can get and set values of text features.

Uses of text features include:
• Entry of names, addresses, or other free-form information.
• Setting the value of text features with configurator rules.
• In large and more complex configurations where the configuration flow spans multiple pages, end users often need to see contextual configuration information across the entire process. You can add text features to supplemental model structure that can be used to capture this contextual information. Text features can then be used in rules to populate basic UI text elements, which are UI elements independent of model structure that allow you to conditionally display text in the configurator runtime user interface. For example, a text element could be used to display details of the product options selected on each UI page.
Working with Text Features

Consider the following when working with text features:

- The values of text features can be used in the following type of configurator rules:
  - Default Rules: Text features can be used in default rules, to produce a default value for a text element. Default rules are applied at runtime when an end user manually selects options and enters values during a configuration session, or on the initialization of the configuration.
  - Constraint Rules: Text features can be assigned values by configurator rules, and can also be used in equality constraint rules. These rules are applied when an end user manually selects options and enters values during a configuration session.
  - Extension Rules: Text features can be used in configurator extension rules.

- Use the system attribute `Value()` to access the textual value of a text feature.

The following statement rule example compares the values entered for two text features and selects an option of an option feature if the values are equal.

```
Equals( 'UserLogin'.'Email Address'.Value() , 'UserLogin'.'Confirm Email Address'.Value() ) // TRUE if text values are equal.
REQUIRES
'UserLogin'.'Do addresses match?'.'Success. Addresses match.'State() // If equal, select option for success.
```

The following example sets the value of a text feature if a specified option is selected.

```
'UserLogin'.'Do addresses match?'.'Success. Addresses match.'State() // Success option is selected.
REQUIRES
'UserLogin'.'Address Check Result'.Value() = "Your address is accepted." // Set value of text feature.
```

The following example sets the default value of a text feature to a text string. You must set the rule class of the rule to Default.

```
'UserLogin'.'Last Name'.Value() = "Enter your last name here."
```

Restrictions on Text Features

Restrictions on text features include:

- You specify the number of characters that an end user can enter as the value of a text feature, in the Maximum Length field of the Details region of the Structure tab. The maximum possible length is 32,000 (not 32K or 32,768).

- When using rules to compare the dynamically set values of text features, you can only perform comparisons for equality or non-equality, with the `Equals()` and `NotEquals()` functions, and the `=` and `<>` operators. Comparisons are case-sensitive.

- Any alphanumeric value is valid, but values aren't parsed for any special processing.

- Text features can't be bound by autocompletion. Because text values are free-form, and have no domain or range, autocompletion can't assign values to text features. Consequently, if a value for a text feature has been defined as required for a configuration, you can't complete a configuration without entering a value for that text feature.
Supplemental Attributes

You can define supplemental attributes on supplemental structure nodes of a model, to use in the ways that you use user-defined item attributes or transactional item attributes on item-based nodes of a model.

About Supplemental Attributes

In the Configurator Models work area, you can add supplemental attributes on supplemental structure nodes. Supplemental attributes are only available within the Configurator Models work area. An important use of supplemental attributes is to enable the definition of compatibility rules between item-based nodes and supplemental nodes. For compatibility rules, you can define supplemental attributes on supplemental nodes that correspond to user-defined attributes or transactional item attributes on item-based nodes.

You can update values of a supplemental attribute using ADF Desktop Integration with Microsoft Excel (ADFdi).

The values of supplemental attributes depend on value sets. When creating a new supplemental attribute or adding an existing supplemental attribute to a model node, the value set for the values of the supplemental attribute must already exist. You create and maintain value sets using the Manage Value Sets task in the Setup and Maintenance work area.

You can use supplemental attributes in statement rules written in the Constraint Definition Language and in extension rules written in the Groovy language.

Adding Existing Supplemental Attributes to Nodes

If the supplemental attribute that you want to use already exists, you can add it to the nodes in your model.

To add an existing supplemental attribute to a node:

1. Open the model draft to which the supplemental attribute will be added.
2. Select the supplemental structure node, then select the Supplemental Attributes subtab.
3. Click the Add button.
4. Open the list for the name of the supplemental attribute to add. Search for and select the desired supplemental attribute. When you select the supplemental attribute, the default values for its description, value set, and data type are added, and the default value, if any, of the supplemental attribute is selected.
5. You can specify a different value, but you can’t change the other elements of an existing supplemental attribute. You can’t add the same supplemental attribute to a node more than once.
6. The supplemental attributes that you added are saved when you save the model draft.

Creating Supplemental Attributes Associated with Nodes

If the supplemental attribute that you want to use doesn’t already exist, you can create it and associate it to a node in your model.

To create a new supplemental attribute and associate it to a node:

1. Identify the value set to be used for the supplemental attribute that you want to create. If necessary, create the value set using the Manage Value Sets task in the Setup and Maintenance work area.
2. Open the model draft to which the new supplemental attribute will be added.
3. Select the supplemental structure node, then select the Supplemental Attributes subtab.
4. Click the Add button.
5. Open the list in the Name field and click the Create link.
6. In the Create Supplemental Attribute dialog box, enter a unique name for the new supplemental attribute.
7. Select the value set to apply to the new supplemental attribute. The value set must belong to the module Product Model.
8. Select a default value from the value set, if it has one. The value will be applied to all future associations of the supplemental attribute to model nodes. If no value is specified, the supplemental attribute will be created without a value and future associations won’t have a default value.
9. Save the model draft.

Using Supplemental Attributes in Rules

You can use supplemental attributes in configurator rules.

For example, you can define an attribute-based compatibility rule between a user-defined attribute of an item-based model node and a supplemental attribute of a supplemental model node, to determine compatible options.

The syntax for referring to a supplemental attribute in CDL is:

```
<nодеPath>.suppAttrs["<attrName>"]
```

Example:

```
COMPATIBLE
&color1 OF 'CAR4DRSDN'.'Packages', // supplemental node
&color2 OF 'CAR4DRSDN'.'Interior Options'.'Trim Colors' // item-based model node
WHERE
&color1.suppAttrs["Color"] = &color2.userAttrs["Trim_AG.Color"];
```

You can use supplemental attributes in extension rules, by calling one of these methods:

```
IRuntimeNode.getConfigurationAttributeByName(java.lang.String name)
IRuntimeNode.getConfigurationAttributeByName(java.lang.String name, IConfigurationAttribute.AttributeType type)
IRuntimeNode.getConfigurationAttributes()
IRuntimeNode.getConfigurationAttributes(IConfigurationAttribute.AttributeType type)
```

Supplemental Attributes in User Interfaces

You can use supplemental attributes in UI expressions, to override the default caption of certain UI elements.

Use the token `suppAttrs['supprAttrName']` in a UI expression such as `#{amn.suppAttrs['color']}`, to display the value of the supplemental attribute "color" in a UI item.

How You Manage Supplemental Structure Using ADFdi Integration

You can use ADF Desktop Integration with Microsoft Excel (ADFdi) to maintain a common supplemental structure across multiple models in a single operation, by editing structure data in a spreadsheet connected to your development environment.
Management Preliminaries
Before using this feature, perform the following preliminary actions.

- Download and install the latest version of ADF Desktop Integration with Microsoft Excel.
  To obtain the ADFdi installation, select **Download Desktop Integration** from the Navigator menu.
- Download the template spreadsheets that you use to manage supplemental structure data.
  To download the template spreadsheets from the Configurator Models work area:
    - On the Manage Models page, select **Download Spreadsheet for Managing Model Structure** from the Action menu.
    - On the Manage Models page, select **Download Spreadsheet for Managing Cross-Model Structure** from the Action menu.
    - On the Manage Snapshots page, select **Download Spreadsheet for Managing Item Snapshots** from the Action menu.
  You can also download populated versions of the spreadsheets by selecting a task under File Based Data Management in the task panel of the Configurator Models work area.

General Management Procedure
The following generalized procedure applies to a variety of update options, depending on the type of supplemental structure management you're performing.

The models containing the structure to be managed must be:

- In Draft status
- Participants in an unreleased workspace (one with status of **In Development**)
- Not locked by another user

1. From the File-Based Data Management section of the task drawer panel, click one of these tasks:
   - Manage Model Structure
   - Manage Cross-Model Structure
   - Manage Item Snapshot Attributes
2. When prompted, open the Excel spreadsheet.
3. When prompted, sign in with the credentials that you use for the Configurator Models work area. You may need to sign in each time you make the ADFdi connection.
   The spreadsheet that corresponds with the task you selected opens automatically in Microsoft Excel.
   Each spreadsheet includes controls for working with ADFdi functions, on a worksheet ribbon tab named for the management task you selected.
4. Click the **Search** button on the ADFdi tab. Enter selection criteria for the models or snapshots that you want to update. You can restrict the search by model or snapshot name, or workspace name, or both.
   When your search completes, select the model to manage and click **OK**. The spreadsheet becomes populated with a row for each node in the model or item in the snapshot. Cells representing item-based nodes (which you can't modify) are displayed with shading.
   Optionally, you can log out of your ADFdi connection, to work offline, by clicking a button on the ADFdi tab.
5. Insert the new rows or cell values that accomplish your update. Don't replicate data in an existing row by copying and pasting the entire row, because the generated read-only value in the Key column must be unique. Only replicate cells that are editable, which are displayed without shading. A marker appears in the Changed column of the spreadsheet in each row that you modify.

Tip: You can maintain a separate offline spreadsheet containing the data to be used for updates, then copy and paste that data into the ADFdi-connected spreadsheets that you download.

6. Click the ADFdi button that submits the update, such as Create or Update.

The upload process finds all the selected model drafts that contain the model elements pertaining to your update and updates their definitions based on the changes you made in the downloaded ADFdi spreadsheet.

7. Examine the status messages section of the spreadsheet for information on the update, or possible errors. If the Status cell of a row contains Insert failed, click No when you're prompted whether to discard the pending changes from your update, and examine the Status Viewer for details about the failure.

Available Management Actions
You can use ADFdi to perform the following actions on a selected set of one or more models in a single operation.

- For option features and their options:
  - Create an option feature and its options
  - Add new options to an existing option feature
  - Delete options from an existing option feature
  - Update option feature attributes in an existing option feature and option descriptions in existing options of that option feature
  - Create, update, or delete multiple option features and their individual options, for multiple models

- For supplemental attributes:
  - Create supplemental attributes and their values and associate them with the options of an option feature
  - Update existing supplemental attribute values and associate them with the options of an option feature
  - Associate existing supplemental attributes and their values to an existing option feature
  - Remove supplemental attributes and their values associated with the options of an option feature

- For item snapshots:
  - Manage the date effectivity of attribute values for item snapshots

The following supplemental structure node types are available for ADFdi management:

- Option features and their options
- Boolean features
- Decimal features
- Integer features
- Text features
Spreadsheet Details

Each ADFdi spreadsheet associated with a management task has its own set of elements.

The following table describes the spreadsheet elements for managing kinds of supplemental structure using ADFdi integration.

<table>
<thead>
<tr>
<th>Spreadsheet elements</th>
<th>For managing supplemental model structure</th>
<th>For managing cross-model supplemental structure</th>
<th>For managing supplemental attributes and values of snapshots</th>
</tr>
</thead>
<tbody>
<tr>
<td>Command to download empty spreadsheet from the Actions menu</td>
<td>On the Manage Models page, select <strong>Download Spreadsheet for Managing Model Structure</strong></td>
<td>On the Manage Models page, select <strong>Download Spreadsheet for Managing Cross-Model Structure</strong></td>
<td>On the Manage Snapshots page, select <strong>Download Spreadsheet for Managing Item Snapshots</strong></td>
</tr>
<tr>
<td>Task to download spreadsheet based on search. From the File-Based Data Management list in the task panel, click:</td>
<td><strong>Manage Model Structure</strong></td>
<td><strong>Manage Cross-Model Structure</strong></td>
<td><strong>Manage Item Snapshot Attributes</strong></td>
</tr>
<tr>
<td>Default spreadsheet name</td>
<td>ManageModelNodes.xlsx</td>
<td>ManageCrossModelOptionFeature.xlsx</td>
<td>ManageItemSnapshotSupplAttrValAssoc.xlsx</td>
</tr>
<tr>
<td>Spreadsheet tab with ADFdi buttons on spreadsheet ribbon</td>
<td>Manage Supplemental Structure</td>
<td>Manage Cross-Model Option Feature</td>
<td>Supplemental Attributes and Values to Item Snapshots</td>
</tr>
<tr>
<td>Spreadsheet columns for entering data</td>
<td><strong>Parent Name</strong></td>
<td><strong>Parent Name</strong></td>
<td><strong>Structure Item Type (read-only)</strong></td>
</tr>
<tr>
<td>* indicates required input</td>
<td>*Name</td>
<td>*Name</td>
<td><strong>Item Class (read-only)</strong></td>
</tr>
<tr>
<td>**indicates conditionally required input</td>
<td>Description</td>
<td>Description</td>
<td><strong>Organization (read-only)</strong></td>
</tr>
<tr>
<td></td>
<td>Node Type (valid values can be selected)</td>
<td>Node Type (valid values are OPTION_FEATURE and OPTION)</td>
<td><strong>Name (read-only)</strong></td>
</tr>
<tr>
<td><strong>Domain Ordering</strong></td>
<td><strong>Minimum</strong></td>
<td>Minimum Value</td>
<td><strong>Description (read-only)</strong></td>
</tr>
<tr>
<td><strong>Minimum</strong></td>
<td><strong>Maximum</strong></td>
<td>Maximum Value</td>
<td><strong>Snapshot Status (read-only)</strong></td>
</tr>
<tr>
<td><strong>Maximum</strong></td>
<td></td>
<td>Enable option quantities</td>
<td><strong>Supplemental Attribute</strong></td>
</tr>
<tr>
<td></td>
<td>The following columns are available in a separate range of the spreadsheet. Enter data in them to create associations of existing supplemental attributes and values with the supplemental structure nodes that you create.</td>
<td><strong>Maximum Quantity per Option</strong></td>
<td><strong>Supplemental Attribute Value</strong></td>
</tr>
<tr>
<td></td>
<td><strong>Parent Name</strong></td>
<td></td>
<td><strong>Effective From</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td><strong>Effective To</strong></td>
</tr>
</tbody>
</table>
### Confi gurator Model Node Properties

During a runtime confi guration session, model nodes have values for a number of properties. You can use these values to affect confi guration choices or results.

The values of node properties can be used for purposes such as the following:

- In confi gurator rules that include system attributes
- In display conditions for user interfaces

The following table lists all the available node properties, and indicates whether a property is available for each of the confi gurator model node types.

<table>
<thead>
<tr>
<th>Property</th>
<th>Option Class</th>
<th>Option Feature</th>
<th>Option</th>
<th>Standard Item</th>
<th>Boolean Feature</th>
<th>Decimal Feature</th>
<th>Integer Feature</th>
<th>Model (Single Instance)</th>
<th>Model (Multiple Instances)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ChangedByAutocomplete</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>DefinitionMaxQuantity</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>DefinitionMaxSelected</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>DefinitionMaxValue</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>DefinitionMinQuantity</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>DefinitionMinSelected</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>DefinitionMinValue</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Property</td>
<td>Option Class</td>
<td>Option Feature</td>
<td>Option</td>
<td>Standard Item</td>
<td>Boolean Feature</td>
<td>Decimal Feature</td>
<td>Integer Feature</td>
<td>Model (Single Instance)</td>
<td>Model (Multiple Instances)</td>
</tr>
<tr>
<td>---------------------</td>
<td>--------------</td>
<td>----------------</td>
<td>--------</td>
<td>---------------</td>
<td>-----------------</td>
<td>-----------------</td>
<td>-------------------</td>
<td>-------------------------</td>
<td>-----------------------------</td>
</tr>
<tr>
<td>DefinitionMinValue</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Description</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>DetailedSelection</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>DisplayName</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>DisplayNamePath</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Excluded</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>HasChildren</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>HasTransAtt</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>InErrorMode</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>InputRequired</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>InputRequiredInSubtree</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>IsBound</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>IsBoundQuantity</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>IsMinSelectionSatis</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>LogicState</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>MaxQuantity</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>MaxSelected</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>MaxValue</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>
### Property | Option Class | Option Feature | Option | Standard Item | Boolean Feature | Decimal Feature | Integer Feature | Model (Single Instance) | Model (Multiple Instances)
--- | --- | --- | --- | --- | --- | --- | --- | --- | ---
MinQuantity | Yes | No | Yes | Yes | No | No | No | Yes | Yes
MinSelected | No | Yes | No | No | No | No | No | No | No
MinValue | No | No | No | No | No | Yes | Yes | No | No
Name | Yes | Yes | Yes | Yes | Yes | Yes | Yes | Yes | Yes
Proposed | Yes | Yes | Yes | Yes | Yes | Yes | Yes | Yes | Yes
Quantity | Yes | No | Yes | Yes | No | No | No | Yes | Yes
Selected | Yes | Yes | Yes | Yes | Yes | No | No | Yes | Yes
SelectedCount | Yes | Yes | No | No | No | No | No | No | No
SelectionState | Yes | Yes | Yes | Yes | Yes | No | No | Yes | Yes
Valid | Yes | Yes | Yes | Yes | Yes | Yes | Yes | Yes | Yes
Value | No | No | No | No | No | Yes | Yes | No | No

**Note:** The values of some node properties are only available at run time. For example, the value of the property Quantity is only available after an item has been selected during a configuration session.

### FAQ for Model Structure

**How can I tell if a model's snapshot data is up to date?**

For nodes that were created from snapshots, you can check the **Snapshot Status** field for the node on the View or Edit Configurator Model page. This status indicates whether the node's snapshot is current (status of **Released**) or has pending updates that are not yet reflected in the model, and, if so, whether or not that snapshot is in the model's workspace (status of **Updates in workspace** or **Updates not in workspace**). If those pending updates are not yet in the workspace, you can use the **Add Updated Item Snapshots for Models** command on the Workspace page to add them to the workspace.
How should I include very large option classes in a model?

In item-based models, some option classes might contain hundreds or even thousands of options. By default, a configurator user interface creates a UI control for the option class that contains all the options. Such a UI control can affect the responsiveness and usability of the model.

To implement a more streamlined UI, consider modeling the option class through supplemental structure. For example, for an option class **Seat Cushion** that contains thousands of options for different cushions:

1. Create a supplemental option feature such as **Cushion Width**, whose options are the available widths: **14 inch**, **16 inch**, **17 inch**, and so on.
2. Create additional option features (or other types of supplemental structure nodes) as needed, to describe other dimensions or characteristics of the cushion.
3. Modify the default UI that you generated for the model, to add UI controls that display the new supplemental option features, instead of the original option class.
4. Create statement or extension rules that gather the end user’s runtime inputs to the option features, and process those defining characteristics to select the desired cushion. For example, one statement rule might be something like:

   ```
   'Task Chair'.'Cushion Width' > 30 AND 'Task Chair'.'Cushion Width' <= 50
   REQUIRES
   'Task Chair'.'Seat Cushion OC'.'SEAT_CUSHION_12345';
   ```

5. Modify the model UI to display the desired cushion that was selected. You can add a running summary to the UI, to track runtime changes to the cushion options.

In a runtime configuration session, the end user responds to the supplemental structure questions, thus providing inputs to the statement or extension rules, which then present the most relevant options to the user, instead of the entire set of options contained by the option class.

**Related Topics**

- How You Create Statement Rules
- How You Create Extension Rules
- How You Modify User Interfaces
5 Model Rules

Overview of Model Rules

This chapter covers model rules. Rules define the behavior of the model during configuration.

- Rules enable you to make the configuration process simpler for end users by guiding them through complex choices.
- Rules ensure that only valid choices are made, preventing delays and disruptions in downstream processing.
- You create rules by entering statements in the Constraint Definition Language (CDL) in the rules editor.
- The class of a rule indicates how the rule is to be enforced during the configuration process.
- The type of a rule is classified by the operator or operation employed in the rule.

You access model rules in the following ways.

- Configure Models work area > tasks panel tab > Manage Models page > Search > Versions > click a Draft or Version > Configurator Model or Edit Configurator Model page > Rules tab
  The Edit Configurator Model page enables you to create and edit the structure, rules, and user interfaces of a configurator model. If the model is locked by another user, then you can only view the model.
- Configure Models work area > tasks panel tab > Manage Workspaces page > Workspace page > click Name of participating model > Configurator Model or Edit Configurator Model page > Rules tab

Rule Principles

Configurator Rules

The Oracle Fusion Configurator constraint engine acts on a configurator model, according to configurator rules, to produce valid configurations.

Typical uses for configurator rules are to:

- Set default choices or values for model nodes
- Automatically select options based on another choice by end user
- Prevent users from selecting invalid combinations
- Control how many instances of a model can be created at run time
- Calculate and set values for numeric options

A valid configuration is like a solution to a problem consisting of variables in the configurator model. Rules constrain how the problem is solved.

- Variables have a defined domain and a run time range of values.
- Variables must be bound to a value.
- Variables can require input.
• Configurator can automatically try to solve the problem, using a feature called autocompletion.

Rules are defined in the the Configurator Models work area as statement rules, written in the Constraint Definition Language (CDL).

Rules can be disabled. Invalid rules in participating models should be disabled or deleted before releasing a workspace. When testing a model, invalid rules are ignored.

Implicit rules are imported from the Product Information Management work area. An example of an implicit rule is an option class defined with options that are mutually exclusive. Similar rule behavior is also implicit in supplemental structure, such as minimum and maximum selections in an option feature.

Configurator rules that you create for a model are part of the definition of that model, like supplemental structure and user interfaces. Configurator rules can’t be shared with other models.

Each rule has a class, which indicates how the rule is to be enforced during the configuration process. The rule classes are:

• Constraint
• Default
• Search Decision

Each rule has a type, which is a classification based on the operator or operation employed in the rule. The rule types are:

• Logic
• Numeric comparison
• Numeric accumulator
• Attribute-based compatibility

Rules, which produce valid model configurations, are different from run time user interface conditions, which can display or enable elements in a user interface, depending on the value of an attribute. User interface conditions don't affect the selections in a configuration.

Rule Classes

A rule's class determines when and how it's applied at run time, not its run time behavior.

The rule classes available in configurator rules are described in the following table.

<table>
<thead>
<tr>
<th>Rule Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Constraint</td>
<td>A rule with a class of Constraint must always be true. See the fuller description in the information about Constraints.</td>
</tr>
<tr>
<td>Default</td>
<td>A rule with a class of Default is like a less strict constraint. The constraint engine will try to make the rule true but if that's not possible the rule will be ignored. A Default rule gives you some control over the sequence of rule evaluation</td>
</tr>
</tbody>
</table>
### Rule Class

<table>
<thead>
<tr>
<th>Rule Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default</td>
<td>Some operators can't be used in Default rules.</td>
</tr>
<tr>
<td></td>
<td>See the fuller description in the information about <strong>Defaults</strong>.</td>
</tr>
<tr>
<td>Search Decision</td>
<td>Like Default, but but applied during the process of finishing a configuration (the autocompletion process).</td>
</tr>
<tr>
<td></td>
<td>See the fuller description in the information about <strong>Search Decisions</strong>.</td>
</tr>
</tbody>
</table>

### Choosing Rule Classes

When defining a rule in the Configurator Models work area, you must assign the rule to a Rule Class. Oracle Fusion Configurator refers to a rule's Rule Class when an end user is manually configuring a product and when the autocompletion process is running.

A rule's Rule Class determines the following at run time:

- The rule's general behavior
- Whether the rule is mandatory (meaning that it must always be True in the configuration)
- At what point in the configuration session the rule is applied (Defaults and Search Decisions only)

Caution: New Defaults and Search Decisions appear at the end of their respective sequence, and changing an existing rule's Rule Class from Default or Search Decision to Constraint may adversely affect a well-defined sequence. Therefore, be sure to review and unit test the sequence of Defaults and Search Decisions after modifying a rule's Rule Class.

### Constraints

Constraints are applied at run time while an end user manually selects options and enters values during a configuration session.

Constraints must always be true in the context of a configuration. For example, when the end user makes a selection that violates a Constraint at run time, Configurator displays a contradiction message informing the user that the previous action can't be applied.

A Constraint may be expressed as a logical expression, a numeric comparison, or a compatibility expression.

Unlike Defaults and Search Decisions, you can't specify the order in which you want Configurator to consider Constraints at run time.

Relational operators can be the primary operator within a Constraint. Consider the following examples that use the equals ( = ) and greater-than ( > ) relational operators:

\[
x = y + (q*z) \\
a > b
\]

In these simplified expressions of rules, the left-hand side of the expression can propagate (or "push") numeric information to the right-hand side. In the constraint engine, the right-hand side of the expression can also propagate ("push back") to the left-hand side. The ability to define such Constraints allows rules to be bidirectional, meaning that they can propagate in both directions.
Defaults

Like Constraints, Defaults are also applied at run time while an end user manually selects options and enters values during a configuration session. However, unlike Constraints, Defaults are:

- Flexible, and they don’t lead to a contradiction at run time when, for example, the end user deselects an option that was selected by the rule.
- Applied at run time, on the initialization of the configuration, before the user makes any selections, in the order specified in the rule definition. Defaults act like initial selections that can be overridden as the user proceeds through a configuration.

A Default can fail due to a conflict with one of the end user's inputs or the propagation of a Constraint. You can specify the order in which Defaults propagate.

You can use Defaults to guide end users toward a preferred solution by defining several contradictory rules that will be processed in the order you specify at run time.

For example, a manufacturer of laptop computers prefers that their customers purchase the lightweight version of a laptop instead of the heavier model, and the Deluxe carrying case rather than the Basic version. To guide buyers toward purchasing the lightweight laptop with the Deluxe case, without preventing them from selecting alternative options, the manufacturer defines the following rules and sequence:

1. Laptop 900 Implies 900-LTW
2. Laptop 900 Implies Deluxe Case
3. Laptop 900-HVW Implies Deluxe Case

With these rules in place, the lightweight version of the laptop (900-LTW) and the Deluxe Case will be selected by default when the end user selects the Laptop 900 model. If the end user then selects the heavier model (the 900-HVW), the Deluxe Case will still be selected.

Many constraints can be defined as a Default. For example, your Model contains a Numeric Feature called Weight which has a range (domain) of 1000 - 5000. You prefer a solution in which the value of this item is less than or equal to 3000, so you defined the following Statement Rule and assign a Rule Class of Default:

```
Weight <= 3000
```

When this rule propagates at run time, the range for the Weight item is reduced and appears as follows in the run time UI:

```
Range: 1000 to 3000
```

If the user makes some selections that result in a Weight outside this range, the default is retracted (overridden) without consequence.

Rules classified as Defaults may be expressed as assignment functions.

Search Decisions

Rules that you classify as Search Decisions are applied:

- During the autocompletion process
- After all user selections, Constraints, and Defaults have been applied and propagated
- Before the application of the constraint engine's inherent Search Decisions
- In the order that you specify in the rule definition

Rules classified as Search Decisions may be expressed as assignment functions, logical expressions, or numeric comparisons.
Specifying a Sequence for Defaults and Search Decisions

At run time, the autocompletion process applies rules classified as Defaults and Search Decisions according to the sequence that you specify in the rule definition. When you define a rule and classify it as either a Default or Search Decision, Configurator assigns a default sequence number which places the rule at the end of its respective list. For example, you have five existing rules with the "Defaults" Rule Class. When you create a new rule and specify a Rule Class of Defaults, the new rule appears at the end of the list of Defaults rules, and its sequence number is 6.

If you want a rule to appear earlier in its respective sequence, click either Reorder Defaults or Reorder Search Decisions in the Rules area of the Edit Model page.

Note: At run time, any rules that have cross-model or cross-instance participants won't necessarily be applied in their specified sequence relative to the other rules defined in the Model. The order of a set of such rules that apply to the same scope (combination of Models or instances) will remain as defined relative to one another, but as a group they will be applied after instantiation of the scope (all the rule's participants) and application of all the rules of that class defined within the various instances of the scope.

To modify the order in which Defaults or Search Decisions are applied at run time:

2. Click one of the arrow controls to move the selected rule up or down in the sequence.
3. Review the updated sequence.
4. If you're satisfied with the changes, click Save.

Logic Rules

Logic rules are a rule type that enforces a relationship. Logic rules enable you to express constraints among elements of your model in terms of logical relationships. For example, selecting Option A may require that Options B and C be included in the configuration.

When defining a Logic Rule, you specify the rule's behavior by using the CONSTRAIN keyword with one of the following logic relations:

- Requires
- Negates
- Implies
- Excludes

The following sections describe each type of relation and present tables illustrating their behavior. In each table, the arrows point to the logic state the option has after an end user selects it. The arrows indicate the direction in which the rule propagates.

Notice that a rule can propagate from Operand A to Operand B of the relation, or from Operand B to Operand A. Notice also that for some values and some logic relations the rule doesn't propagate; therefore logic state of the option on the other side of the rule doesn't change.

Note: The terms "true" and "false" are used here to indicate only whether an option is included or excluded from the configuration.
Requires
Logic rules that use the Requires relation "push both ways," which means that selecting an option on one side of the rule has the same effect on the option on the other side of the rule. See the following examples for details.

The following figure shows the effect of the Requires relation.

```
A   Requires   B
---+-------------+---
True| True
False| False
True| True
False| False
```

- If the end user selects an option on one side of the rule, the option on the other side of the rule is also selected. The same is true when the end user deselects an option. In other words, both options must be either included in the configuration, or excluded from the configuration.

Negates
The Negates relation is similar to the Requires relation, in that it also "pushes both ways." However, the Negates relation prevents an option from being selected when an option on the other side of the rule is selected. In other words, selecting one option prevents the other option from being included in the configuration.

The following figure shows the effect of the Negates relation.

```
A   Negates   B
---+-------------+---
True| False
False| True
False| True
True| False
```

- If the end user selects Option A, it becomes true and Option B is set to false.
• If the end user then deselects Option A, it becomes false and Option B becomes true. In other words, Option B is selected
• If the end user selects Option B first, it becomes true and Option A becomes false.
• If the end user then deselects Option B, Option A becomes true.

Implies

The following figure shows the effect of the Implies relation.

<table>
<thead>
<tr>
<th></th>
<th>IMPLIES</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="true" alt="User" /></td>
<td><strong>True</strong></td>
<td><img src="true" alt="User" /></td>
</tr>
<tr>
<td><img src="false" alt="User" /></td>
<td><strong>Unknown</strong></td>
<td><img src="false" alt="User" /></td>
</tr>
<tr>
<td><strong>Unknown</strong></td>
<td><strong>True</strong></td>
<td><img src="false" alt="User" /></td>
</tr>
<tr>
<td><img src="false" alt="User" /></td>
<td><strong>False</strong></td>
<td><img src="false" alt="User" /></td>
</tr>
</tbody>
</table>

• If the end user selects Option A it becomes true and Option B is also selected. In other words, Option B’s logic state becomes true.
• Deselecting Option A causes Option A to become false and the state of Option B is unknown. In other words, Option B is available for selection.
• If the end user selects Option B first, it becomes true and Option A is unknown.
• If the end user deselects Option B, both Option B and A become false.

Excludes
The following figure shows the effect of the Excludes relation.

<table>
<thead>
<tr>
<th>A</th>
<th>EXCLUDES</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>True</td>
<td></td>
<td>False</td>
</tr>
<tr>
<td>False</td>
<td></td>
<td>Unknown</td>
</tr>
<tr>
<td>False</td>
<td></td>
<td>True</td>
</tr>
<tr>
<td>Unknown</td>
<td></td>
<td>False</td>
</tr>
</tbody>
</table>

- If the end user selects Option A, it becomes true and Option B becomes false. In other words, Option B is excluded from the configuration. If the end user tries to select Option B, Configurator displays a contradiction message.
- If the end user deselects Option A, Option A becomes false and Option B becomes Unknown. In other words, Option B is available for selection.
- If the end user selects Option B first, Option A becomes false.
- If the end user deselects Option B, then Option A becomes Unknown.

Logic States and Rule Variables

During a configuration session, the configurator engine attempts to narrow the possible domain of valid values for nodes of a model. You define the outer bounds of the domain by defining nodes to have a minimum and maximum value.

A variable that has neither been selected nor excluded from a configuration at run time has a logic state of Unbound. A variable is unbound when its domain is open, which means that either a value hasn't been assigned or the set of its members hasn't been finalized. Variables may be unbound because the end user hasn't yet made a selection, entered a value, or run Finish and Review (autocompletion).

At run time, the value of the SelectionState and DetailedSelectionState system attributes is Selectable for options that are neither selected nor excluded from a configuration.

Options may also be excluded from a configuration by Defaults or Search Decisions, or by autocompletion.

How You Use Attributes in Model Rules

When defining a rule, you select model nodes that will participate in the rule. You can also use attributes associated with model nodes in your rule definition.

Use the syntax provided in the following table to refer to types of node attributes and obtain their values. In the Syntax column, `<nodePath>` represents the node path to a node, and `<attrName>` represents the name of an attribute of that node.
<table>
<thead>
<tr>
<th>Attribute Type</th>
<th>Syntax</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>User-defined attributes</td>
<td><code>&lt;nodePath&gt;.userAttrs[&quot;&lt;attrGroupName&gt;.&lt;attrName&gt;&quot;]</code></td>
<td>If a model node has user-defined attributes, you can insert them in rule definition text from the Item tab of the Attributes pane of the rule editor. Select an attribute and click the <strong>Insert into Rule Text</strong> button. The values of user-defined attributes are static at run time, since the values are part of the model definition.</td>
</tr>
<tr>
<td>Supplemental attributes</td>
<td><code>&lt;nodePath&gt;.suppAttrs[&quot;&lt;attrName&gt;&quot;]</code></td>
<td>If a model node has supplemental attributes, you can insert them in rule definition text from the Supplemental tab of the Attributes pane of the rule editor. Select an attribute and click the <strong>Insert into Rule Text</strong> button. The values of supplemental attributes are static at run time, since the values are part of the model definition.</td>
</tr>
<tr>
<td>Transactional item attributes (TIA)</td>
<td><code>&lt;nodePath&gt;.transAttrs[&quot;&lt;attrName&gt;&quot;]</code></td>
<td>If a model node has transactional item attributes, you can insert them in rule definition text from the Structure pane of the rule editor. Select a transactional item attribute from the structure tree and click the <strong>Insert into Rule Text</strong> button. The values of transactional item attributes are dynamic at run time, since the values are determined during a configuration session by user action or model rules. The example divides the linear length of a window by 5 (5 feet in length for material) and adds it to the number of standard tracks needed.</td>
</tr>
<tr>
<td>Configurator system attributes, such as:</td>
<td><code>&lt;nodePath&gt;.&lt;attrName&gt;()</code></td>
<td>You can insert system attributes in rule definition text from the System tab of the Structure pane of the rule editor. Select an attribute and click the <strong>Insert into Rule Text</strong> button.</td>
</tr>
<tr>
<td>• Name</td>
<td></td>
<td></td>
</tr>
<tr>
<td>• Value</td>
<td></td>
<td></td>
</tr>
<tr>
<td>• Quantity</td>
<td></td>
<td></td>
</tr>
<tr>
<td>• State</td>
<td></td>
<td></td>
</tr>
<tr>
<td>• Options</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

'Home Theater System'. 'Speaker System'. userAttrs("PhysicalAttributes.Color")

'Home Theater System'. 'Speaker System'. suppAttrs("color")

ADD 'Custom Window'. 'Frame'. transAttrs("Linear Length")/5 TO 'Custom Window'. 'Frame'. 'Track'. Quantity()
When using transactional item attributes in rule expressions, observe the following:

- To refer to a specific attribute value, you can't reference it with a path-style notation, as if it were a child of the attribute. You must reference it as a literal in a conditional expression, such as:

  \[(x) \text{ EXCLUDES } (y.\text{transAttrs["BaseWeight"] < 10})\]

- You can't use the Configurator system attribute \text{Selection()} on a TIA, even if it has an enumerated value set that's displayed at run time. Option features do support \text{Selection()}.

- You can map a constraint over all occurrences of a particular TIA within a set of nodes. Use expression syntax such as the following example, which constrains against the selection or entry of the value 1 for the TIA \text{Weight} when the node \(x\) has a value of 1.

  \[(x = 1) \text{ EXCLUDES } \text{OC.Selection().transAttrs["Weight"] = 1}\]

### Accumulator Rules

Accumulator Rules are a rule type that enables end-user selections to add to or subtract from a value from a variable at run time. For example, when the end user selects the 512 MB RAM option you want to add 512 to a Total called Total RAM Selected.

### Background

Accumulator Rules can only have a Rule Class of Constraint. They can't be classified as Defaults or Search Decisions.

Numeric rules express constraints between parts of a Model in terms of numeric relationships. Use Numeric rules to enable end-user selections to contribute to or consume from a numeric feature or option quantity.

If you create a statement rule with a Rule Class of either Defaults or Search Decision, and the rule's text defines an Accumulator rule (meaning that it uses either the Add or Subtract operators), then Configurator displays a message similar to the following when you validate the rule: The rules of the rule class <RULE_CLASS> may not contain the accumulator operators <ADDTO> or <SUBTRACTFROM>.

It is important to understand that Accumulator Rules don't simply add or subtract a quantity from a variable. All rules of this type defined against the same target node can be considered terms in a constraint against that node. This is because all addition and subtract expressions in a Model become a single constraint on the target node. In other words, the target node equals the sum of all addition expressions defined against it in the Model minus the sum of the SubtractFrom expressions.

Additionally, if the target node is involved in any other constraints, the equality constraint generated by its addition and subtraction expressions must be satisfied along with all the others. As with all other constraints, the equality constraint is bidirectional, so it can "push back" on the values of the participants on the left-hand-side of the rule.

Keep the following in mind when using Accumulator Rules:

- If the Model contains multiple Accumulator Rules that add to or subtract from the same target node, and that node exists in a referenced Model, generating logic creates a single constraint that equates the target to a sum of all the terms expressed in the individual rules in that model.
If addition or subtraction rules are defined against a given target within multiple parent models in a reference model hierarchy, each of the generated equality constraints must be satisfied individually. In other words, the addition and subtraction terms are not accumulated across multiple referencing models.

- When a numeric rule compares two decimal values, they may be interpreted as not being equal even when they're, due to their representation by the Java double data type. The decimal tolerance value (also referred to as epsilon value) is set to 0.000000001 (scale of 9), which will then be used by the configurator engine to convert the decimal comparison to a mathematical equation for evaluation. For example, the expression $a=b$ could be converted to $|a-b| \leq \epsilon$. This would give an accurate result as long as the scale is less than or equal to the number of decimal digits specified in the epsilon (the decimal tolerance value, which is 0.000000001).

**Procedure**

To create an Accumulator Rule:

1. Create a Rule of type Statement Rule.
2. Enter the text of the Statement Rule, using the addition or subtraction operator according to the following syntax:
   
   - **ADD** \text{n} **TO** \text{x}  
   - or  
   - **SUBTRACT** \text{n} **FROM** \text{x}

   Where \text{n} is a number and \text{x} is a numeric feature.

**Compatibility Rules**

Compatibility rules test the compatibility between the children of option classes.

**Background**

Compatibility rules define which items are allowed in a configuration when other items are selected. A compatibility rule tests the compatibility between the children of one or more option classes (which are typically standard items, but may also include other option classes).

Compatibility rules can only have a rule class of Constraint. They can't be classified as Defaults or Search Decisions.

Compatibility rules describe the conditions for compatibility across a set of options from the participant features. In other words, if a selection is made from each participant feature and those selections don't satisfy the compatibility criterion, there is a contradiction. A compatibility rule can't constrain selections from only a subset of the participant features.

When defining any type of compatibility rule, Configurator doesn't allow more than one of the rule's participants to have a Maximum Selections value that's greater than 1. If the Maximum Selections value of one of the rule participant's changes after the rule is created, Configurator displays an error when you compile the model.

**Example**

Use the COMPATIBLE...OF keyword to define a compatibility rule.

```
COMPATIBLE  
&color OF Frame.Color,
```
Create Statement Rules

How You Create Statement Rules

Statement rules are the type of rule you use to build configuration constraints between elements of a model.

Creating a Statement Rule

To create a statement rule, you use the rule editor in the Configurator Models work area.

The following table presents the actions required to create statement rules, and the reason for each action.

<table>
<thead>
<tr>
<th>Action</th>
<th>Reason</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. On the Rules tab of the Edit Configurator Model page, create a statement rule by selecting Create &gt; Statement Rule from the Actions menu.</td>
<td>Configurator rules are created as statement rules. The new rule appears in the Rules pane. You can create rule folders there to keep your rules organized. You can create several statement rules and work on them concurrently. A statement rule can contain multiple statements.</td>
</tr>
<tr>
<td>2. In the Create Statement Rule dialog box, type in a name, and an optional description, and select a rule class.</td>
<td>The rule class governs how the rule is used during configuration. You can change the rule class later.</td>
</tr>
<tr>
<td>3. Enter the text of the rule in the text pane below the button bar in the Definition region.</td>
<td>Statement rules must be written in the Constraint Definition Language (CDL).</td>
</tr>
<tr>
<td>3a. Use the menus to insert CDL syntactical elements into the rule text.</td>
<td>Using the menus guarantees that you will insert only valid syntactical elements. However, you must validate the statement rule to guarantee that it's syntactically valid as a whole, and test the rule in the model to determine whether it performs as expected.</td>
</tr>
<tr>
<td>3b. Use the Structure pane to reference model structure nodes.</td>
<td>Your CDL statements contain many references to nodes of the model structure. To insert a syntactically correct reference to a node, search for and select it in the Structure pane, then click the Insert into Rule Text button on the toolbar, or chose that action from the context menu on the node.</td>
</tr>
<tr>
<td>4. Use the Validate button to validate the rule text.</td>
<td>The Validate button checks the validity of the syntax of your CDL statements, and checks the references in the rule text to model structure. If the rule is invalid or has an error, that's indicated in the Status indicator for the rule on its Details region. You can leave a rule in Invalid status if necessary. Invalid rules are ignored when testing the model.</td>
</tr>
</tbody>
</table>
Testing a Configurator Rule

To verify that the behavior of a configurator rule is what you expect, test the model.

To test a configurator rule:

1. Click the Test Model button. You don’t have to be on the Rules tab of the model to test rules.
2. In the Test Model dialog box, ensure that you select a user interface that includes nodes of the model that are affected by the rule behavior that you intend to test, in the User Interface field. By default, the previously tested UI is selected.
3. Make selections among the configuration options, emulating both the likely and also the possible choices made by an end user. Navigate through the pages of the UI, and observe how your own selections affect other selections that are made by the rule you’re defining.

The Statement Rule Editor

You use the statement rule editor to create statement rules.

Features of the Statement Rule Editor

The statement rule editor provides features that assist you in efficiently creating statement rules.

The statement rule editor is part of the Rules tab of the Edit Configurator Model page.

The Rules tab also provides

- The Rules pane, which gives you access to the rules you define, displayed in a tree control. The status of rules that are modified or have errors is indicated by icons.

  You can create rule folders in the tree, to organize your rules.

  The Actions menu contains actions that let you reorder Default and Search Decision rules.

- The Structure pane, which enables you to insert syntactically correct references to a node directly into the rule text, by selecting the node and clicking the Insert into Rule Text button.

The following table summarizes the features and benefits of the rule editor.
<table>
<thead>
<tr>
<th>Feature</th>
<th>Benefit</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Enabled check box</strong></td>
<td>You can disable a rule while you work on its definition, or if you don't want its behavior affecting the model.</td>
</tr>
<tr>
<td><strong>Status</strong></td>
<td>The status values are:</td>
</tr>
<tr>
<td></td>
<td>• Modified: The rule text was edited, or some attribute of the rule was changed since the last time it was compiled.</td>
</tr>
<tr>
<td></td>
<td>• Error: An error prevented the rule from being compiled and executed, or the rule has a syntax error.</td>
</tr>
<tr>
<td></td>
<td>• Valid: The rule is valid, and can be compiled and tested.</td>
</tr>
<tr>
<td><strong>End Date</strong></td>
<td>The read-only End Date field displays the date when the rule becomes invalid because one of the participants in the rule was end-dated.</td>
</tr>
<tr>
<td><strong>Definition region</strong></td>
<td>This region of the editor page groups the controls for modifying the rule.</td>
</tr>
<tr>
<td><strong>Rule Class selection</strong></td>
<td>Enables you to change the rule class that you selected when creating the rule.</td>
</tr>
<tr>
<td></td>
<td>If the class is a Default or Search Decision, a <strong>Reorder</strong> button is enabled so that you can change the sequence in which your defaults or search decisions are executed.</td>
</tr>
<tr>
<td><strong>Validate button</strong></td>
<td>Checks the validity of the syntax of your rule text.</td>
</tr>
<tr>
<td><strong>Text pane</strong></td>
<td>You enter the CDL rule text of your statement rule here. Basic text editing is provided. The toolbar above the text pane provides insertion menus and text buttons that enable you to enter valid CDL code without using the keyboard.</td>
</tr>
<tr>
<td><strong>Insertion menus</strong></td>
<td>A set of menus enables you to insert valid CDL syntactical elements directly into the rule text.</td>
</tr>
<tr>
<td></td>
<td>The menus are:</td>
</tr>
<tr>
<td></td>
<td>• <strong>Keywords</strong>: Major CDL keywords around which statements are built.</td>
</tr>
<tr>
<td></td>
<td>• <strong>Logic Operators</strong>: The operators for defining logic rules.</td>
</tr>
<tr>
<td></td>
<td>• <strong>Functions</strong>: a set of cascading menus for entering functions that are grouped by categories:</td>
</tr>
<tr>
<td></td>
<td>• Logic</td>
</tr>
<tr>
<td></td>
<td>• Arithmetic</td>
</tr>
<tr>
<td></td>
<td>• Trigonometric</td>
</tr>
<tr>
<td></td>
<td>• Set</td>
</tr>
<tr>
<td></td>
<td>• String</td>
</tr>
<tr>
<td></td>
<td>The menus are tear-off style, so you can tear an often-used menu off the toolbar and leave it positioned over the editor page for convenient access while you work.</td>
</tr>
<tr>
<td>Feature</td>
<td>Benefit</td>
</tr>
<tr>
<td>-----------------------</td>
<td>------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Text buttons</td>
<td>A set of buttons enables you to insert valid CDL syntactical elements directly into the rule text by clicking. The groups of buttons provide:</td>
</tr>
</tbody>
</table>
|                       | • Characters for delimiting statements  
|                       | • Comparison operators  
|                       | • Boolean operators  
|                       | • Mathematical operators |

### How You Manage Rules Using ADFdi Integration

You can use ADF Desktop Integration with Microsoft Excel (ADFdi) to maintain a common set of rules across multiple models and environments, by downloading rule definition data to a spreadsheet connected to your development environment, optionally modifying the definitions, and uploading the definitions to the same or another model or environment.

#### Rule Management Summary

You can perform the following operations when managing configurator rules:

<table>
<thead>
<tr>
<th>Feature</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Search</td>
<td>To manage rules for a draft model, search for the model by its name, across workspaces, or by the name of the workspace that contains the model.</td>
</tr>
<tr>
<td>Create</td>
<td>To create a rule for the model you searched for, download its rule data to an ADFdi-enabled spreadsheet, add details for the rule, then upload the rule data.</td>
</tr>
<tr>
<td>Update</td>
<td>To update an existing rule for the model you searched for, download its rule data to an ADFdi-enabled spreadsheet, make changes to certain details for the rule, then upload the rule data.</td>
</tr>
</tbody>
</table>

#### Management Procedure

The following procedure applies to both updating or supplementing existing rules, and to replicating rules in another environment.

Before using this feature, you must have downloaded and installed the latest version of ADF Desktop Integration with Microsoft Excel, by selecting **Download Desktop Integration** from the Navigator menu.

The models containing the rules to be managed must be in Draft status and must be participants in an unreleased workspace (a workspace with the status of **In Development**).

To create or modify rules using ADFdi:

1. From the File-Based Data Management section of the task drawer panel, click the Manage Rules task.
2. When prompted, open the Excel spreadsheet.
3. When prompted, sign in with the credentials that you use for the Configurator Models work area. You may need to sign in each time you make the ADFdi connection.
The spreadsheet for managing rules opens automatically in Microsoft Excel. The spreadsheet includes controls for working with ADFdi functions, on a worksheet ribbon tab named Manage Rules.

4. Click the **Search** button on the ADFdi tab. Enter selection criteria for the models whose rules you want to manage. You can restrict the search by model or workspace name, or both.

When your search completes, select the model whose rules you want to manage and click **OK**. The spreadsheet becomes populated with a row for each rule currently existing in the model.

 Optionally, you can log out of your ADFdi connection, to work offline, by clicking a button on the ADFdi tab.

5. If you want to modify the downloaded rules, you can edit existing values for the rules, or insert new rows to create new rules or cell values that accomplish your update, using the columns of the spreadsheet as described in the following table. The columns whose values can be updated correspond to fields on the Rules tab of the Edit Configurator Model page.

Don’t replicate data in an existing row by copying and pasting the entire row, because the generated read-only value in the Key column must be unique. Only replicate cells that can be updated, which are displayed without shading.

<table>
<thead>
<tr>
<th>Column</th>
<th>Can Be Updated?</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Changed</td>
<td>No</td>
<td>Indicates that a row was changed or entered since the last download or upload.</td>
</tr>
<tr>
<td>Flagged</td>
<td>No</td>
<td>Reserved for future use.</td>
</tr>
</tbody>
</table>
| Status      | No              | Displays the results of the validation of the row data that’s performed on each upload. Values include:  
<p>|             |                 | o Update failed                                                             |
|             |                 | o Insert failed                                                              |
|             |                 | o Row inserted successfully                                                  |
| Model       | Yes             | Name of the existing model containing the rule.                             |
|             |                 | You can’t create a new model this way.                                      |
| Workspace   | Yes             | Name of the existing workspace containing the model draft that contains the rule. |
|             |                 | You can’t create a new workspace this way.                                  |
| Name        | Yes             | Name of the rule.                                                           |</p>
<table>
<thead>
<tr>
<th>Column</th>
<th>Can Be Updated?</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Entering a new name creates a new rule in the specified model, workspace, and folder. You can't change the name of an existing rule this way. You can't use a rule name that already exists in another draft of the model.</td>
</tr>
<tr>
<td>Description</td>
<td>Yes</td>
<td>Description of the rule. You can change the description as desired.</td>
</tr>
<tr>
<td>Folder Path</td>
<td>Yes</td>
<td>Name and path of the folder containing the rule. Entering a new folder name creates a new folder under the top-level Rules folder for the model. To create or use subfolders, use the following notation: subfolder/ sub-subfolder/.</td>
</tr>
<tr>
<td>Type</td>
<td>Yes</td>
<td>Type of the rule. Only Statement Rules are supported.</td>
</tr>
<tr>
<td>Rule Class</td>
<td>Yes</td>
<td>The current rule class of the rule. To change, the class, select one of: Constraint, Default, Search Decision</td>
</tr>
<tr>
<td>Rule Text</td>
<td>Yes</td>
<td>The CDL text that defines the rule. The rule text is validated when you upload it, following all the rules that apply to rules in the Configurator Models work area. It's possible to upload invalid rule text, but it will cause the status of the uploaded rule to be Error. Ensure that node names are enclosed with single quotation marks, so that nodes can be located correctly when the rule text is parsed. If a rule begins with a node name, add an additional single quotation mark before it, to accommodate the behavior of Microsoft Excel.</td>
</tr>
<tr>
<td>Enabled</td>
<td>Yes</td>
<td>Whether the rule is enabled or not. Select Yes or No.</td>
</tr>
</tbody>
</table>
A marker appears in the **Changed** column of the spreadsheet in each row that you modify. This marker indicates which rows will be uploaded to the model.

**Tip:** You can maintain a separate offline spreadsheet containing the data to be used for updates, then copy and paste that data into the ADFdi-connected spreadsheets that you download.

6. Click the ADFdi **Create or Update** button, which submits the update using your current ADFdi connection.

   The upload process updates the rule definitions based on the changes you made in the downloaded ADFdi spreadsheet. After every upload, each row in the spreadsheet is subjected to two-step validation:

   a. The data in the spreadsheet row is validated by ADFdi. The results are displayed in the **Status** column.
   b. The rule text is validated by Configurator. The results are displayed in the **Rule Status** column.

7. Examine the messages in the **Status** and **Error Message** columns of the spreadsheet for information on the update, or possible errors.

   If the **Status** cell of a row contains **Insert failed**, click **No** when you're prompted whether to discard the pending changes from your update, and examine click the **Status Viewer** button to examine details about the failure.

To see the effect of your rule updates, you might have to close the affected Rules page, if it's open, and reopen it to refresh the display.

**Migrating Rules**

You can migrate rules from one instance of the Configurator Models work area to another instance, in a way similar to creating or updating rules in a single instance.

For example, assume that you want to migrate rules from a Test instance to a Production instance. To migrate rules:

1. Sign in to Test, search for the model containing the rules, and download the rules from that model.
2. Copy the downloaded rule data from the spreadsheet for Test. Only copy the cells that allow updating. You can modify the rules, as long as you follow the restrictions described here about which data can be changed.  

   **Tip:** You can set aside a backup of the copied rule data by pasting it in a separate spreadsheet that's not connected by ADFdi.

3. Sign in to Production, search for the model that will receive the rules, and download the rules from that model. The workspaces and models referenced in your copied rule definitions must exist in Production.

4. Copy the rule data that came from Test, and paste it into the spreadsheet connected to Production, in the columns that are eligible for updating.

5. Click the ADFdi **Create or Update** button, to upload the rule data to the model in Production.

**Related Topics**
- How You Manage Supplemental Structure Using ADFdi Integration

### CDL Reference

#### The Constraint Definition Language

The Constraint Definition Language (CDL) enables you to define configurator rules and the constraining relationships among items in configurator models, by entering them as text.

#### Overview of the Constraint Definition Language (CDL)

The Constraint Definition Language (CDL) is a modeling language. CDL enables you to define configurator rules, the constraining relationships among items in configurator models, by entering them as text. A rule defined in CDL is an input string of characters that's stored in the CZ schema of the Oracle Applications database, validated by a parser, translated into executable code by a compiler, and interpreted at run time by Configurator.

You use CDL to define a Statement Rule in the Configurator Models work area by entering the rule's definition as text. Because you use CDL to define them, Statement Rules can express complex constraining relationships.

#### Relationships Expressed in CDL

Using CDL, you can define the following relationships:

- Logical
- Numeric
- Compatibility
- Comparison

#### Anatomy of a Configurator Rule Written in CDL

This topic provides an overview of how the syntax, semantics, and lexical structure of a rule written in CDL relate to one another.

This section contains the following topics:

- Rule Definition
Rule Definition

A configurator rule has a name, associated model, rule text, other attributes such as rule class. The rule definition is written in CDL and consists of one or more individual statements that express the intent of the rule, along with optional comments.

When creating a Statement Rule in the Configurator Models work area, you enter the name and other attributes in input or selection fields and the rule definition in the text box provided for that purpose.

Rule Statements

Statements define the rule's intent, such as to add a value of 10 to Integer Feature X when Option A is selected.

Multiple statements in a rule definition must be separated from one another with semi-colons (;). CDL supports two kinds of statements: Explicit and Iterator.

CDL statements are parsed as tokens. Everything in CDL is a token, except white space characters and comments.

Statements consist of one or more clauses. Clauses consist of keywords and one or more expressions. Keywords are predefined tokens that determine CDL syntax and embody the semantics of the language. CONSTRANIN, COMPATIBLE, REQUIRES, IMPLIES, LIKE, NOT, and others are all examples of keywords.

An expression is the part of a statement that contains an operator and the operands involved in a rule operation. An operator is a predefined keyword, function, or character that involves the operands in logical, functional, or mathematical operations. REQUIRES and the plus sign (+) are examples of operators. An operand can be an expression, a literal, or an identifier. The literal or identifier operand can be present in the rule as a singleton or as a collection. Model nodes that are referred to in a rule are called rule participants.

Literals are tokens of a specific data type, such as Numeric, Boolean (True or False), or Text. The specific values of literals are equivalent to the notion of constants.

An identifier is a token that identifies model objects or formal parameters. When an identifier identifies a model object it, refers to a model node or attribute and the sequence of letters and digits starts with a letter. These kinds of identifiers are called references. When an identifier is a formal parameter, it identifies a local variable and is used in an iterator statement. Formal parameters are prefixed with an ampersand (&).

For greater readability and to convey meaning such as the order of operations, CDL supports separators. Separators are tokens that maintain the structure of the rule by establishing boundaries between tokens, grouping them based on some syntactic criteria. Separators are single characters such as the semi-colon between statements or the parentheses around an expression.

More information about these statements and the CDL elements they contain is described with CDL Statements.

Data Types

Following are valid data types when defining a rule in CDL:

- INTEGER
- DECIMAL
- BOOLEAN
- TEXT
- Node types
Under certain circumstances, a data type of a variable isn't compatible with the type expected as an argument. The Configurator parser doesn't support explicit conversion or casting between the data types. The parser performs implicit conversion between compatible types. See the following table for details.

If a rule definition has the wrong data types, the parser returns a type mismatch error message. Invalid Collection shows a collection whose data types can't be implicitly converted to be compatible.

The following table shows which data type each source data type implicitly converts.

<table>
<thead>
<tr>
<th>Source data type (or collection of the same type)</th>
<th>Implicitly converts to (or collection of the same type)</th>
</tr>
</thead>
<tbody>
<tr>
<td>INTEGER</td>
<td>DECIMAL</td>
</tr>
<tr>
<td>NODE of type Standard Item, Option Class, Model, Option Feature, Option, or Boolean Feature</td>
<td>BOOLEAN, INTEGER, DECIMAL, Node type</td>
</tr>
<tr>
<td>NODE of type Integer Feature</td>
<td>INTEGER, DECIMAL</td>
</tr>
<tr>
<td>NODE of type Decimal Feature</td>
<td>DECIMAL</td>
</tr>
</tbody>
</table>

Unless specified otherwise, all references to matching types assume the implicit data type conversions.

Although TEXT is included as a data type here, it can only be used in a static context. You can't use a TEXT literal, reference, or expression in the actual body of a CONSTRAIN, ADD, or SUBTRACT expression. The Configurator compiler validates this condition when you compile the model.

**CDL Statements**

A rule definition written in CDL consists of one or more statements that define the rule's intent. The two kinds of statements are:

- Explicit statements
- Iterator statements

The difference between explicit and iterator statements is in the types of participants involved.

**Explicit Statements**

Explicit statements express relationships among explicitly identified participants and restrict execution of the rule to those participants and the model containing those participants.
In an explicit statement, you must identify each node and attribute that participates in the rule by specifying its location in the model structure. An explicit statement applies to a specific model, thus all participants of an explicit statement are explicitly stated in the rule definition.

CDL supports several kinds of explicit statements, which are identified by the keywords CONSTRAIN, COMPATIBLE, ADD...TO, and SUBTRACT...FROM.

The following example shows such an explicit statement consisting of a single expression of the logical IMPLIES relation.

```plaintext
CONSTRAIN a IMPLIES b;
CONSTRAIN (a+b) * c > 10 NEGATES d;
```

**Iterator Statements**

Iterators are query-like statements that iterate, or repeat, over elements such as constants, model references, or expressions of these. Iterators express relations among participants that are model node elements of a collection, or participants that are identified by their attributes, and allow the rule to be applied to options of option features, or children of option classes, that have the same attributes. Iterators allow you to use the attributes of model nodes to specify the participants of constraints or contributions. This is especially useful for maintaining persistent sets of constraints when the model structure or its attributes change frequently. Iterators can also be used to express relationships between combinations of participants, such as with compatibility rules.

Iterator statements can use local variables that are bound to one or more iterators over collections. This is a way of expressing more than one constraint or contribution in a single implicit form. During compilation, a single iterator statement explodes into one or more constraints or contributions.

The available iterators that make a rule statement an iterator statement are:

- FOR ALL....IN
- WHERE

**Multiple Iterators in One Statement**

The syntax of the FOR ALL clause allows for multiple iterators. The statement can be exploded to a Cartesian product of two or more collections.

The example below produces a Cartesian product as the rule iterates over all the items of the Tint option class in the Glass child model and over all the items of the Color option class in the Frame child model of a Window model. Whenever the Stain user-defined attribute in an item of the Color option class equals the Stain user-defined attribute in an item of the Tint option class, the selected color pushes the corresponding stain to TRUE. So, for example, when `&color.userAttrs["Paints_AG.Stain"]` and `&tint.userAttrs["Paints_AG.Stain"]` both equal Clear, selecting the White option causes the Clear option to be selected.

**Example: Multiple Iterators in One CONSTRAIN Statement**

```plaintext
COMPATIBLE
&color OF Frame.Color,
&tint OF Glass.Tint
WHERE &color.userAttrs["Paints_AG.Stain"] = &tint.userAttrs["Paints_AG.Stain"]; 
```

The difference between this and a compatibility rule is that this code selects participants without over-constraining them, while a compatibility test deselects participants that don't pass the test.

In the following example, the numeric value of feature a contributes to feature b for all the options of a and b when the value of their user-defined attribute UDA2 is equal.

```plaintext
Oracle SCM Cloud
Modeling Configurations for SCM
Chapter 5
Model Rules

In an explicit statement, you must identify each node and attribute that participates in the rule by specifying its location in the model structure. An explicit statement applies to a specific model, thus all participants of an explicit statement are explicitly stated in the rule definition.

CDL supports several kinds of explicit statements, which are identified by the keywords CONSTRAIN, COMPATIBLE, ADD...TO, and SUBTRACT...FROM.

The following example shows such an explicit statement consisting of a single expression of the logical IMPLIES relation.

```plaintext
CONSTRAIN a IMPLIES b;
CONSTRAIN (a+b) * c > 10 NEGATES d;
```

**Iterator Statements**

Iterators are query-like statements that iterate, or repeat, over elements such as constants, model references, or expressions of these. Iterators express relations among participants that are model node elements of a collection, or participants that are identified by their attributes, and allow the rule to be applied to options of option features, or children of option classes, that have the same attributes. Iterators allow you to use the attributes of model nodes to specify the participants of constraints or contributions. This is especially useful for maintaining persistent sets of constraints when the model structure or its attributes change frequently. Iterators can also be used to express relationships between combinations of participants, such as with compatibility rules.

Iterator statements can use local variables that are bound to one or more iterators over collections. This is a way of expressing more than one constraint or contribution in a single implicit form. During compilation, a single iterator statement explodes into one or more constraints or contributions.

The available iterators that make a rule statement an iterator statement are:

- FOR ALL....IN
- WHERE

**Multiple Iterators in One Statement**

The syntax of the FOR ALL clause allows for multiple iterators. The statement can be exploded to a Cartesian product of two or more collections.

The example below produces a Cartesian product as the rule iterates over all the items of the Tint option class in the Glass child model and over all the items of the Color option class in the Frame child model of a Window model. Whenever the Stain user-defined attribute in an item of the Color option class equals the Stain user-defined attribute in an item of the Tint option class, the selected color pushes the corresponding stain to TRUE. So, for example, when `&color.userAttrs["Paints_AG.Stain"]` and `&tint.userAttrs["Paints_AG.Stain"]` both equal Clear, selecting the White option causes the Clear option to be selected.

**Example: Multiple Iterators in One CONSTRAIN Statement**

```plaintext
COMPATIBLE
&color OF Frame.Color,
&tint OF Glass.Tint
WHERE &color.userAttrs["Paints_AG.Stain"] = &tint.userAttrs["Paints_AG.Stain"]; 
```

The difference between this and a compatibility rule is that this code selects participants without over-constraining them, while a compatibility test deselects participants that don't pass the test.

In the following example, the numeric value of feature a contributes to feature b for all the options of a and b when the value of their user-defined attribute UDA2 is equal.
Multiple Iterators in One ADD...TO Statement

```
ADD &var1 TO &var2
FOR ALL &var1 IN {OptionsOf(a)}, &var2 IN {OptionsOf(b)}
WHERE &var1.userAttrs["UDA2"] = &var2.userAttrs["UDA2"];
```

CDL Syntax Details

Miscellaneous syntactical elements of CDL are described here.

**CDL Syntactical Elements**
The following table lists miscellaneous syntactical elements of CDL.

<table>
<thead>
<tr>
<th>Element</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Comments</td>
<td>Comments are included in rule definitions at your discretion to explain the rule. Comments aren't tokens and therefore ignored by the Configurator parser.</td>
</tr>
<tr>
<td>White space</td>
<td>White space, which includes spaces, line feeds, and carriage returns, format the input for better readability. The white space category of elements aren't tokens and therefore ignored by the Configurator parser.</td>
</tr>
<tr>
<td>Case Sensitivity</td>
<td>Keywords aren't case sensitive. Keyword operators aren’t case sensitive. Model object identifiers are case sensitive. Formal parameters are case sensitive and can’t be in quotes. The constants E and PI as well as the scientific E aren’t case sensitive. The keywords TRUE and FALSE aren’t case sensitive. Text literals are case sensitive. All keywords, constant literals, and so on aren’t case sensitive.</td>
</tr>
<tr>
<td>Quotation Marks</td>
<td>Model structure node names that contain white space, or text that would be interpreted by the parser as keywords or operators, must be enclosed in single quotation marks.</td>
</tr>
</tbody>
</table>

**Syntax Notation**

The following table describes the valid statement syntax notation for CDL. The table lists the available symbols and provides a description of each. This notation is used for CDL examples and in the syntax reference.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-- or //</td>
<td>A double hyphen or double slash begins a single line comment that extends to the end of the line.</td>
</tr>
<tr>
<td>/* */</td>
<td>A slash asterisk and an asterisk slash delimits a comment that spans multiple lines.</td>
</tr>
</tbody>
</table>
### Symbol Description

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&amp;lower case</td>
<td>Lower case prefixed by the ampersand sign is used for names of formal parameters and iterator local variables.</td>
</tr>
<tr>
<td>UPPER CASE</td>
<td>Upper case is used for keywords and names of predefined variables or formal parameters.</td>
</tr>
<tr>
<td>Mixed Case</td>
<td>Mixed case is used for names of user-defined Model nodes, names of user-defined rules.</td>
</tr>
<tr>
<td>;</td>
<td>A semi-colon indicates the end of one statement and the beginning of the next.</td>
</tr>
</tbody>
</table>

In the examples for CDL, an implied carriage return occurs at the end of each line, unless otherwise noted. You must press the Enter key at the end of a line of input. The following table lists the typographic and symbol conventions used in this book, such as ellipses, bold face, italics.

<table>
<thead>
<tr>
<th>Convention</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>. . . . . .</td>
<td>Vertical ellipsis points in an example mean that information not directly related to the example has been omitted.</td>
</tr>
<tr>
<td>...</td>
<td>Horizontal ellipsis points in statements or commands mean that parts of the statement or command not directly related to the example or relevant to the discussion have been omitted.</td>
</tr>
<tr>
<td><strong>boldface text</strong></td>
<td>Boldface type in text indicates a new term, a term defined in the glossary, specific keys, and labels of user interface objects. Boldface type also indicates a menu, command, or option, especially within procedures</td>
</tr>
<tr>
<td>[ ]</td>
<td>Brackets enclose optional clauses from which you can choose one or none.</td>
</tr>
<tr>
<td>&gt;</td>
<td>The left bracket alone represents the MS DOS prompt.</td>
</tr>
<tr>
<td>$</td>
<td>The dollar sign represents the DIGITAL Command Language prompt in Windows and the Bourne shell prompt in Digital UNIX.</td>
</tr>
<tr>
<td>%</td>
<td>The per cent sign alone represents the UNIX prompt.</td>
</tr>
<tr>
<td>name()</td>
<td>In text other than code examples, the names of programming language methods and functions are shown with trailing parentheses. The parentheses are always shown as empty. For the actual argument or parameter list, see the reference documentation. This convention isn't used in code examples.</td>
</tr>
</tbody>
</table>
## Terminology

The following table defines the terms used here.

<table>
<thead>
<tr>
<th>Term</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cartesian product</td>
<td>A set of tuples that's constructed from two or more given sets and comprises all permutations of single elements from each set such that the first element of the tuple is from the first set and the second is from the second set, and so on.</td>
</tr>
<tr>
<td>clause</td>
<td>A segment of a rule statement consisting of a keyword and expression.</td>
</tr>
<tr>
<td>collection</td>
<td>A set of multiple operands within parentheses and separated by commas.</td>
</tr>
<tr>
<td>compiler</td>
<td>The part of Configurator that first parses rule definitions and then generates code that's executable at run time.</td>
</tr>
<tr>
<td>explicit statement</td>
<td>Explicit statements express relations among explicitly identified participants and restrict execution of the rule to those participants and the Model containing those participants.</td>
</tr>
<tr>
<td>expression</td>
<td>A subset of the statement that contains operators and operands</td>
</tr>
<tr>
<td>formal identifier</td>
<td>A variable that's defined in the scope of an iterator statement to represent an iterating identifier.</td>
</tr>
<tr>
<td>iterator statement</td>
<td>Iterators are query-like statements that iterate, or repeat, over one or multiple relations or constraints.</td>
</tr>
<tr>
<td>non-terminal</td>
<td>The kind of symbols used in the notation for presenting CDL grammar that represent the names of grammar rules.</td>
</tr>
<tr>
<td>parser</td>
<td>A component of the Configurator compiler that analyzes the syntactic and semantic correctness of statements used in rule definitions.</td>
</tr>
<tr>
<td>relationship</td>
<td>A type of constraint expressed in a single statement or clause. A relationship can be equivalent to a simple rule. A Statement Rule expresses one or more relationship types but isn't itself a type of relationship.</td>
</tr>
<tr>
<td>signature</td>
<td>The distinct combination of a function's attributes, such as name, number of parameters, type of parameters, return type, mutability, and so on.</td>
</tr>
<tr>
<td>singleton</td>
<td>A single operand that isn't within a collection.</td>
</tr>
<tr>
<td>statement</td>
<td>The entire sentence that expresses the rule's intent. A CDL rule definition can consist of multiple statements, each consisting of clauses containing expressions, and separated by semi-colons.</td>
</tr>
</tbody>
</table>
### Term | Description
--- | ---
terminal | The kind of symbols used in the notation for presenting CDL grammar that represent the names, characters, or literal strings of tokens.
token | The result of translating characters into recognizable lexical meaning. All text strings in the input stream to the parser, except white space characters and comments, are tokens.
unicode | A 16-bit character encoding scheme allowing characters from Western European, Eastern European, Cyrillic, Greek, Arabic, Hebrew, Chinese, Japanese, Korean, Thai, Urdu, Hindi and all other major world languages, to be encoded in a single character set.

### Separators
Separators are characters that serve as syntactic filling between the keywords and the expressions. Their goal is to maintain the structure of the token stream by introducing boundaries between the tokens and by grouping the tokens through some syntactic criteria.

The following table lists the separators that are valid in CDL.

| Separator | Description |
--- | --- |
( | The open parenthesis indicates the beginning of function arguments or the beginning of an expression. |
) | The close parenthesis indicates the end of function arguments or the end of an expression. |
, | The comma separates arguments or collection elements. |
; | The semi-colon separates statements. |
. | The dot character separates identifiers in compound references. |

### CDL Expressions
An expression is usually part of (or sometimes all of) a CDL statement. It has two operands that are connected by an operator, or functions and their arguments.

### Examples
The following example shows a simple mathematical expression where the two operands are 2 and `frame.border`, and the operator is * (multiplication).
The following example shows a simple mathematical expression used as the second operand in another expression, where the first operand is \( \text{window.frame.width} \) and the operator is \(-\) (subtraction).

\[
\text{window.frame.width} - 2 \times \text{frame.border}
\]

For an example of CDL rules using these expressions, consider a Window Model. If you want to calculate the size of the glass to be put into a window frame where the glass is inserted in the frame 1/2 inch at each side, and the frame border is 1 inch, you might write the two accumulator rules in the following example.

\[
\begin{align*}
\text{ADD window.frame.width} - 2 \times \text{frame.border} + 2 \times 0.5 & \text{ TO glass.width; } \\
\text{ADD window.frame.height} - 2 \times \text{frame.border} + 2 \times 0.5 & \text{ TO glass.height; }
\end{align*}
\]

Following are some additional examples of expressions.

The following expressions result in a BOOLEAN value:

\[
\begin{align*}
\text{a} > \text{b} \\
\text{a AND b} \\
(\text{a + b}) \times \text{c} > 10 \\
\text{a.prop LIKE "%abc%"}
\end{align*}
\]

The following expressions result in a INTEGER or DECIMAL value:

\[
\begin{align*}
\text{a + b} \\
((\text{a + b}) \times \text{c})^{10}
\end{align*}
\]

**CDL Functions**

In addition to operators, expressions can contain functions, which may take arguments and return results that can be used in the rest of the expression. All standard mathematical functions are implemented in CDL.

The result of each function can participate as an operand of another operator or function as long as the return type of the former matches with the argument type of the latter.

Functions perform operations on their arguments and return values which are used in evaluating the entire statement. Functions must have their arguments enclosed in parentheses and separated by commas if there is more than one argument. Function arguments can be expressions.

For example, both of the following operations have the correct syntax for the Round function, provided that Feature-1 and Feature-2 are numeric Features:

\[
\begin{align*}
\text{Round (13.4)} \\
\text{Round (Feature-1 / Feature-2)}
\end{align*}
\]

CDL supports the following functions:

- Arithmetic
- Trigonometric
- Logical
- Set
- Text
- Hierarchy or Compound
# Arithmetic Functions

The following table lists the arithmetic functions that are available in CDL. The term infinity is defined as a number without bounds. It can be either positive or negative.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abs(x)</td>
<td>Takes a single number as an argument and returns the positive value (0 to +infinity). The domain range is -infinity to +infinity. Returns the positive value of x. Abs(-12345.6) results in 12345.6</td>
</tr>
<tr>
<td>AggregateSum(x)</td>
<td>Can be used in a Constraint, Default, or Search Decision, but only as a sub-expression.</td>
</tr>
<tr>
<td>Round(x)</td>
<td>Takes a single decimal number as an argument and returns the nearest integer. If the A side of a numeric rule is a decimal number, contributing to an imported bill of materials that accepts decimal quantities, then the Round(x) function is unavailable. The reason that the Round(x) function is unavailable is that the contributed value doesn't need to be rounded as the B side accepts decimal quantities. This function is available when the bill of materials item accepts only integer values.</td>
</tr>
<tr>
<td>RoundDownToNearest(x, y)</td>
<td>This is a binary function. x is a number between -infinity and +infinity, y is a number greater than 0 and less than +infinity. A number is returned between -infinity and +infinity. The first argument is rounded to the nearest smaller multiple of the second argument. For example, RoundDownToNearest(433, 75) returns 375.</td>
</tr>
<tr>
<td>RoundToNearest(x, y)</td>
<td>This is a binary function. x is a number between -infinity and +infinity, y is a number greater than 0 and less than +infinity. A number is returned between -infinity and +infinity. RoundToNearest(433, 10) returns 430.</td>
</tr>
<tr>
<td>RoundUpToNearest(x, y)</td>
<td>This is a binary function. The number x is between -infinity and +infinity, and the number y is greater than 0 and less than +infinity. A number is returned between -infinity and +infinity. The first argument is rounded up to the nearest multiple of the second argument. For example, RoundUpToNearest(34.10.125) returns 34.125.</td>
</tr>
<tr>
<td>Ceiling(x)</td>
<td>Takes a single decimal number as an argument and returns the next higher integer. For example, ceiling(4.3) returns 5, and ceiling(-4.3) returns -4.</td>
</tr>
<tr>
<td>Floor(x)</td>
<td>Takes a single decimal number as an argument and returns the next lower integer. For example, floor(4.3) returns 4, and floor(-4.3) returns -5.</td>
</tr>
<tr>
<td>Log(x)</td>
<td>Takes a single number greater than 0 and less than +infinity and returns a number between -infinity and +infinity. Returns the logarithmic value of x. An error occurs if x=0.</td>
</tr>
<tr>
<td>Log10(x)</td>
<td>Takes a single number greater than 0 and less than +infinity and returns a number between -infinity and +infinity. Returns the base 10 logarithm of x. An error occurs if x=0.</td>
</tr>
<tr>
<td>Min(x,y,z...)</td>
<td>Returns the smallest of its numeric arguments.</td>
</tr>
<tr>
<td>Function</td>
<td>Description</td>
</tr>
<tr>
<td>--------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Max(x,y,z...)</td>
<td>Returns the largest of its numeric arguments.</td>
</tr>
<tr>
<td>Mod(x,y)</td>
<td>This is a binary function. Returns the remainder of x/y where x and y are numbers between -infinity and +infinity. If y is 0, then division by 0 is treated as an error. If x=y, then the result is 0. For example, Mod(7,5) returns 2.</td>
</tr>
<tr>
<td>Exp(x)</td>
<td>Returns e raised to the x power. Takes a single number between -infinity and +infinity and returns a value between 0 and +infinity.</td>
</tr>
<tr>
<td>Pow(x,y)</td>
<td>This is a binary function. Returns the result of x raised to the power of y. The number x is between -infinity and +infinity. The integer y is between -infinity and +infinity and the returned result is between -infinity and +infinity. If y=0, then the result is 1. For example, Pow(6,2) returns 36.</td>
</tr>
<tr>
<td>Sqrt(x)</td>
<td>Sqrt(x) returns the square root of x. Takes a single number between 0 and +infinity and returns a value between 0 and +infinity. An input of -x results in an error.</td>
</tr>
<tr>
<td>Truncate(x)</td>
<td>Truncate(x) takes a single decimal number x and truncates it, removing all digits after the decimal point. For example, truncate(4.15678) returns 4.</td>
</tr>
</tbody>
</table>

**Trigonometric Functions**

The following table lists the trigonometric functions that are available in CDL.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sin(x)</td>
<td>Takes a single number x between -infinity and +infinity and returns a value between -1 and +1.</td>
</tr>
<tr>
<td>ASin(x)</td>
<td>Takes a single number between -1 and +1 and returns a value between -pi/2 and +pi/2. ASin(x) returns the arc sine of x. An input outside the range between -1 and +1 results in an error.</td>
</tr>
<tr>
<td>Sinh(x)</td>
<td>Returns the hyperbolic sine of x in radians. Takes a single number between -infinity and +infinity and returns a value between -1 and +infinity. An error is returned when the result exceeds the double. For example, sinh(-99) is valid but sinh(999) results in an error.</td>
</tr>
<tr>
<td>Cos(x)</td>
<td>Takes a single number between -infinity and +infinity and returns a value between -1 and +1. Returns the cosine of x.</td>
</tr>
<tr>
<td>ACos(x)</td>
<td>Takes a single number between -1 and +1 and returns a value between 0 and pi. ACos(x) returns the arc cosine of x. An input outside the range between -1 and +1 results in an error.</td>
</tr>
<tr>
<td>Cosh(x)</td>
<td>Takes a single number between -infinity and +infinity and returns a value between -infinity and +infinity. Returns the hyperbolic cosine of x in radians. An error is returned if x exceeds the maximum value of a double: cosh(-200) is valid whereas cosh(-2000) results in an error.</td>
</tr>
</tbody>
</table>
Function | Description
---|---
Tan(x) | Takes a single number \( x \) between \(-\infty\) and \(+\infty\) and returns a value between \(-\infty\) and \(+\infty\).
ATan(x) | Takes a single number between \(-\infty\) and \(+\infty\) and returns a value between \(-\pi/2\) and \(+\pi/2\). \( ATan(x) \) returns the arc tangent of \( x \).
Tanh(x) | Returns the hyperbolic tangent of \( x \). Takes a single number \( x \) between \(-\infty\) and \(+\infty\) and returns a value between \(-1\) and \(+1\).

**Logical Functions**
The following table lists the logical functions that are available in CDL.

Function | Description
---|---
AllTrue | A logical AND expression. Accepts one or more logical values or expressions. Returns true if all of the arguments are true, or false if any argument is false. Otherwise, the value of AllTrue is unknown.
AnyTrue | A logical OR expression. Accepts one or more logical values or expressions. Returns true if any of the arguments are true, or false if all arguments are false. Otherwise, the value of AnyTrue is unknown.
Not | Accepts a single logical value or expression. Returns True if the argument is False or unknown. If the argument is True, the value is unknown.

**Text Functions**
The following table lists the text functions that are available in CDL.

Note: As with any TEXT data type, don’t use a text function in the body of a CONSTRAIN or accumulator statement unless it evaluates to a constant string. The compiler validates this condition.

Text functions can only be used in static context; for example the WHERE clause of iterators.

Function | Description
---|---
Contains | Compares two operands of text literals and returns true if the first contains the second.
Matches | Compares two operands of text literals and returns true if they match.
### Function Description

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BeginsWith</td>
<td>Compares two operands of text literals and returns true if the first begins with the characters of the second.</td>
</tr>
<tr>
<td>EndsWith</td>
<td>Compares two operands of text literals and returns true if the first ends with the character(s) of the second.</td>
</tr>
<tr>
<td>Equals</td>
<td>Compares two operands of text literals and returns true if the first equals the second.</td>
</tr>
<tr>
<td>NotEquals</td>
<td>Compares two operands of text literals and returns true if the first doesn’t equal the second</td>
</tr>
</tbody>
</table>

### Hierarchy or Compound Functions

The following table lists the compound function that’s available in CDL.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>OptionsOf</td>
<td>Takes Option Class or Feature as an argument and returns its Options.</td>
</tr>
</tbody>
</table>

### CDL Operators

Operators are predefined tokens consisting of Unicode characters to be used as the expression operators among the expression operands. An operator specifies the operation to be performed at run time between the operands.

This section includes the following topics:

- Operators Supported By CDL
- Operator Results
- Operator Precedence
- LIKE and NOT LIKE Operators
- Text Concatenation Operator
- COLLECT Operator

#### Operators Supported By CDL

The following table lists the predefined operators supported by CDL.

<table>
<thead>
<tr>
<th>Operator Type</th>
<th>Operators</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Logical</td>
<td>AND</td>
<td>AND requires two operands and returns true if both are true.</td>
</tr>
<tr>
<td>Operator Type</td>
<td>Operators</td>
<td>Description</td>
</tr>
<tr>
<td>--------------------</td>
<td>-----------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Logical</td>
<td>OR</td>
<td>OR requires two operands and returns true if either is true.</td>
</tr>
<tr>
<td>Logical</td>
<td>NOT</td>
<td>NOT requires one operand and returns its opposite value: false if the operand is true, true if the operand is false.</td>
</tr>
<tr>
<td>Logical</td>
<td>REQUIRES</td>
<td>REQUIRES requires two operands.</td>
</tr>
<tr>
<td>Logical</td>
<td>IMPLIES</td>
<td>IMPLIES requires two operands.</td>
</tr>
<tr>
<td>Logical</td>
<td>EXCLUDES</td>
<td>EXCLUDES requires two operands.</td>
</tr>
<tr>
<td>Logical</td>
<td>NEGATES</td>
<td>NEGATES requires two operands.</td>
</tr>
<tr>
<td>Logical and Comparison</td>
<td>LIKE</td>
<td>LIKE requires two text literal operands and returns true if they match. See LIKE and NOT LIKE Operators for restrictions.</td>
</tr>
<tr>
<td>Logical and Comparison</td>
<td>NOT LIKE</td>
<td>LIKE requires two text literal operands and returns true if they match. See LIKE and NOT LIKE Operators for restrictions.</td>
</tr>
<tr>
<td>Logical, Arithmetic, and Comparison</td>
<td>=</td>
<td>Equals requires two operands and returns true if both are the same.</td>
</tr>
<tr>
<td>Logical, Arithmetic, and Comparison</td>
<td>&gt;</td>
<td>Greater than requires two operands and returns true if the first is greater than the second.</td>
</tr>
<tr>
<td>Logical, Arithmetic, and Comparison</td>
<td>&lt;</td>
<td>Less than requires two operands and returns true if the first is less than the second.</td>
</tr>
<tr>
<td>Logical, Arithmetic, and Comparison</td>
<td>&lt;&gt;</td>
<td>Not equal requires two operands and returns true if they're different.</td>
</tr>
<tr>
<td>Logical, Arithmetic, and Comparison</td>
<td>&lt;=</td>
<td>Less than or equal to requires two operands and returns true if the first operand is less than or equal to the second.</td>
</tr>
<tr>
<td>Logical, Arithmetic, and Comparison</td>
<td>&gt;=</td>
<td>Greater than or equal requires two operands and returns “true” if the first operand is greater than or equal to the second.</td>
</tr>
<tr>
<td>Operator Type</td>
<td>Operators</td>
<td>Description</td>
</tr>
<tr>
<td>--------------</td>
<td>-----------</td>
<td>-------------</td>
</tr>
<tr>
<td>Arithmetic</td>
<td>*</td>
<td>Performs arithmetic multiplication on numeric operands.</td>
</tr>
<tr>
<td>Arithmetic</td>
<td>/</td>
<td>Performs arithmetic division on numeric operands.</td>
</tr>
<tr>
<td>Arithmetic</td>
<td>-</td>
<td>Performs arithmetic subtraction on numeric operands.</td>
</tr>
<tr>
<td>Arithmetic</td>
<td>+</td>
<td>Performs arithmetic addition on numeric operands.</td>
</tr>
<tr>
<td>Arithmetic</td>
<td>^</td>
<td>Performs arithmetic exponential on numeric operands.</td>
</tr>
<tr>
<td>Arithmetic</td>
<td>%</td>
<td>Performs arithmetic modulo on numeric operands.</td>
</tr>
<tr>
<td>Text</td>
<td>+</td>
<td>Performs a concatenation of text strings. See Text Concatenation Operator for restrictions.</td>
</tr>
<tr>
<td>Other</td>
<td>Assign(node)</td>
<td>Used only in Defaults and Search Decisions to force a node to be bound at a particular point in the specified sequence. If the Domain Ordering setting is specified in the node's details page, binding occurs according to this setting. Otherwise, the constraint engine's implicit binding method for this operator type is used.</td>
</tr>
<tr>
<td>Other</td>
<td>IncMin()</td>
<td>Used only in Defaults and Search Decisions. Similar to ASSIGN, but this operator overrides any explicit or implicit domain ordering method for binding the node and attempts a binding using a binary search with increasing minimum. This operator is valid for integers and decimals, including items and options with quantity. Applies to the node's default system attribute when a system attribute isn't explicitly referenced (for example, State, Quantity, or Value). When used with items, you can specify the RelativeQuantity attribute as an alternative.</td>
</tr>
</tbody>
</table>
### Operator Type

<table>
<thead>
<tr>
<th>Operators</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( )</td>
<td>Parentheses ( ) are used to group sub-expressions.</td>
</tr>
<tr>
<td>,</td>
<td>Comma (,) is used to separate function arguments.</td>
</tr>
<tr>
<td>.</td>
<td>Dot (.) is used for referencing objects in the Model tree structure.</td>
</tr>
<tr>
<td>-</td>
<td>Unary minus (-) is used to make positive values negative and negative values positive.</td>
</tr>
</tbody>
</table>

### Operator Results

The result of each expression operator can participate as an operand of another operator as long as the return type of the former matches with the argument type of the latter.

The following table lists the basic return data types of each type of operator.

#### Mapping of Operators and Data Types

<table>
<thead>
<tr>
<th>Operators</th>
<th>Data Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arithmetic</td>
<td>INTEGER</td>
</tr>
<tr>
<td></td>
<td>DECIMAL</td>
</tr>
<tr>
<td>Logical</td>
<td>BOOLEAN</td>
</tr>
<tr>
<td>Comparison</td>
<td>BOOLEAN</td>
</tr>
</tbody>
</table>

### Operator Precedence

Operators are processed in the order given in the following table. Operators with equal precedence are evaluated left to right.

The following table lists the precedence of expression operators in CDL. The columns are Operator, Precedence (direction), and Description.

#### Precedence of Operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Precedence (direction)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( )</td>
<td>1 (right)</td>
<td>Parenthesis</td>
</tr>
<tr>
<td>.</td>
<td>2 (right)</td>
<td>Navigation</td>
</tr>
</tbody>
</table>
LIKE and NOT LIKE Operators

Although LIKE and NOT LIKE are included as text relational operators, they can only be used in static context; for example, the WHERE clause of iterators. As with any TEXT data type, you can’t use LIKE and NOT LIKE with runtime participants unless it evaluates to a constant string. Configurator validates this condition when you compile the model.

LIKE Expression Resulting in a BOOLEAN Value

\[ \text{a.attr.Value()} \text{ LIKE } "%eig%" \]

A TRUE result is returned if the text of a.attr contains the characters 'eig', such as a.attr = 'weight' or 'eight'. FALSE is returned if the text of a.attr = 'rein'.

In the following example, selecting option A and B implies that options within C are selected when the value of their associated user-defined attribute is "A1B1".

\[
\text{Constrain Alltrue('A','B') implies } &C \\
\quad \text{for all } &C \text{ in } \{\text{optionsof('C')}\} \\
\quad \text{where } &C\.userAttrs["Selections.AB Compatibility"] \text{ like } "A1B1"
\]

In the example below, selecting option A and B implies that options within C are selected when the value of their associated user-defined attribute is something other than "A1B1".

\[
\text{Constrain Alltrue('A','B') implies } &C \\
\quad \text{for all } &C \text{ in } \{\text{optionsof('C')}\} \\
\quad \text{where not } (&C\.userAttrs["Selections.AB Compatibility"] \text{ like } "A1B1")
\]

For a list of comparison operators, see Operators Supported By CDL.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Precedence (direction)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>^</td>
<td>3 (right)</td>
<td>Arithmetic power</td>
</tr>
<tr>
<td>Unary +, - NOT</td>
<td>4</td>
<td>Unary plus and minus, Not</td>
</tr>
<tr>
<td>*, /, %</td>
<td>5 (left)</td>
<td>Arithmetic multiplication and division</td>
</tr>
<tr>
<td>Binary +, -</td>
<td>6 (left)</td>
<td>Arithmetic plus and minus, text</td>
</tr>
<tr>
<td></td>
<td></td>
<td>concatenation</td>
</tr>
<tr>
<td>&lt;, &gt;, =, &lt;=, &gt;=, &lt;&gt; LIKE, NOT LIKE</td>
<td>7 (left)</td>
<td>Comparison operators</td>
</tr>
<tr>
<td>AND</td>
<td>8 (left)</td>
<td>Logical AND</td>
</tr>
<tr>
<td>OR</td>
<td>9 (left)</td>
<td>Logical OR</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Logic operators</td>
</tr>
<tr>
<td>DEFAULTS, EXCLUDES, NEGATES, IMPLIES, REQUIRES</td>
<td>10 (left)</td>
<td></td>
</tr>
</tbody>
</table>
Text Concatenation Operator

Although + is included as a text concatenation operator, it can only be used in static context; for example, the WHERE clause of iterators. As with any TEXT data type, you can’t use text concatenation in the actual body of a constrain or contributor statement unless it evaluates to a constant string. Configurator validates this condition when you compile the model.

COLLECT Operator

The COLLECT keyword is used to build a collection and provide it to an operator that takes a collection as an argument. For example, instead of AnyTrue(x, y, z) you can write AnyTrue(COLLECT &c FOR ALL &c IN {x,y,z}).

CDL Attribute Compatibility Rules

The COMPATIBLE...OF keyword supports compatibility statements.

COMPATIBLE...OF Keyword

The COMPATIBLE keyword is used at the beginning of a compatibility statement that defines compatibility based on user-defined attribute values common to standard items of different option classes.

Example

In the following example, the rule iterates over all the items of the Tint option class in the Glass child model and over all the items of the Color option class in the Frame child model of a Window model. A color and tint are compatible whenever the Stain user-defined attribute in an item of the Color option class equals the Stain user-defined attribute in an item of the Tint option class.

```cdl
COMPATIBLE
&color OF Frame.Color,
&tint OF Glass.Tint
WHERE &color.userAttrs["Paints_AG.Stain"] = &tint.userAttrs["Paints_AG.Stain"];
```

CDL Iterator Statements and the FOR ALL...IN and WHERE Keywords

The FOR ALL, IN, and WHERE keywords support iterator statements.

FOR ALL and IN Keywords

The FOR ALL and IN keywords begin the two clauses of an iterator statement. The IN keyword specifies the source of iteration.

Note: The IN clause can contain only literal collections or collections of model nodes, such as OptionsOf. There’s no specification of instances, so all instances of a given Model use the same iteration.

WHERE Keyword

The WHERE keyword begins a clause of an iterator statement that acts as a filter to eliminate iterations that don't match with the WHERE criteria.
In the example FOR ALL IN and WHERE Clause using Node Attributes, the result is only as many contributions to option a as there are children in the criteria specified in the WHERE clause.

Note: The conditional expression in the WHERE clause must be static. When using the COLLECT operation in a WHERE and an IN clause, the operands must be static.

Note: Configurator evaluates compatibility rules from the top down, and gives no priority or precedence to an expression based on its use of the AND or OR operator. In other words, the first relation you enter is evaluated, followed by the second, and so on.

Examples
In the following example, the result is 3 contributions to option a.

```
ADD &var TO d
FOR ALL &var IN {a, b, c};
```

In the following example, the result is as many contributions to numeric feature d as there are children in option class a, whose user-defined attribute UDA3 is less than 5. This example also shows a collection enclosed in braces (see Collection Literals).

Example: FOR ALL IN and WHERE Clause using Node Attributes

```
ADD &var.userAttrs["AG_name.NumAttr"] + 10 TO d
FOR ALL &var IN {OptionsOf(a)}
WHERE &var.userAttrs["AG_name.UDA3"] < 5;
```

In both examples, a single statement explodes into one or more constraints or contributions without explicitly repeating each one. In both examples, the iterator variable can also participate in the left hand side of the accumulator statement.

CDL Iterator Statements and the COLLECT Operator

The COLLECT operator supports iterator statements.

This section includes the following topics:

- Syntax for the operator COLLECT
- How an iterator can use the COLLECT operator to specify the domain of the collection that’s passed to an aggregation function.
- Using the DISTINCT keyword to collect distinct values from an attribute

COLLECT Operator

Aggregation functions such as Min(...), Max(...), Sum(...), and AnyTrue(...) accept a collection of values as an operand. An iterator can use the COLLECT operator to specify the domain of the collection that’s passed to the aggregation function. In many cases FOR ALL serves that purpose. The following example shows a single contribution of the maximum value of the collection of children of option feature a using a COLLECT operator and a FOR ALL iterator.

COLLECT Operator, Single Contribution

```
ADD Max({COLLECT &var FOR ALL &var IN {OptionsOf(a)}}) TO d;
```

The previous example has the same result as the following example:

```
ADD Max &var TO d
FOR ALL &var IN {OptionsOf(a)} ;
```
The COLLECT operator is necessary when limiting an aggregate. The following example shows a rule where the iteration of the FOR ALL and WHERE clauses result in an error for every element of the collection \{Option11, Option32, OptionsOf(Feature1)} that doesn’t contain the user-defined attribute UDA1.

```cdl
CONSTRAIN &varA IMPLIES model.optionClass.item
FOR ALL &varA IN {Option11, Option32, OptionsOf(OptionFeature1)}
WHERE &varA.userAttrs["UDA1"] = 5;
```

The following example uses COLLECT, which prevents the error.

**COLLECT Operator Contributions**

```cdl
CONSTRAIN &varA IMPLIES model.optionClass.item
FOR ALL &varA IN {Option11, Option32, {COLLECT &varB
FOR ALL &varB IN OptionsOf(OptionFeature2)
WHERE &varB.userAttrs["UDA1"] = 5}};
```

COLLECT can be used in any context that expects a collection. The COLLECT operator can be used along with a complex expression and a WHERE clause for filtering out elements of the source domain of the collection.

Since COLLECT is an operator that returns a collection, it can also be used inside of a collection literal, as long as the collection literal has a valid inferred data type. The Configurator compiler flattens the collection literal during logic generation, which allows collections to be concatenated. See Collection Literals for details.

The COLLECT operator can have only one iterator, because the return type is a collection of singletons. CDL doesn’t support using a Cartesian product with the COLLECT operator.

The COLLECT operator can’t put dynamic variables in the IN and WHERE clauses, as this may result in a collection that’s unknown at compile time.

The COLLECT operator can use the DISTINCT keyword to collect distinct values from a user-defined attribute, as shown in the following example, which prevents the selection of options having different values for the user-defined attribute Shape from the option class optionClass3. optionClass3 has zero Minimum Quantity and no limit on Maximum Quantity.

**COLLECT Operator with DISTINCT**

```cdl
AnyTrue({COLLECT &opt1
FOR ALL &opt1 IN {'optionClass3'.Options()}
WHERE &opt1.userAttrs["Physical.Shape"] = &shape})
EXCLUDES
AnyTrue({COLLECT &opt2
FOR ALL &opt2 IN {'optionClass3'.Options()}
WHERE &opt2.userAttrs["Physical.Shape"] <> &shape})
FOR ALL &shape IN
{COLLECT DISTINCT &node.userAttrs["Physical.Shape"]
FOR ALL &node IN 'optionClass3'.Options()}
```

**CDL Constraint Statements and the CONSTRRAIN Keyword**

The CONSTRRAIN keyword is used at the beginning of a constraint statement.

A constraint statement uses an expression to express constraining relationships. You can omit the CONSTRRAIN keyword from a constraint statement.

Each constraint statement must contain one and only one of the following keyword operators:

- **REQUIRES**
- **NEGATES**
• IMPLIES
• EXCLUDES

For a description of these constraints, see the topic on Logic Rules.

Examples

The following examples show constraint statements with and without the CONSTRAIN keyword.

Constraint statements with the CONSTRAIN keyword

```
CONSTRAIN a IMPLIES b;
CONSTRAIN (a+b) * c > 10 NEGATES d;
```

Constraint statements without the CONSTRAIN keyword

```
a IMPLIES b;
(a + b) * c > 10 NEGATES d;
```

The following example expresses that if one Option of Feature F1 is selected, then by default select all the rest of the Options.

Constraint Statement with the FOR ALL...IN Iterator

```
CONSTRAIN F1 DEFAULTS &var1
FOR ALL &var1 IN F1.Options();
```

CDL Compatibility Statements and the COMPATIBLE ...OF Keyword

The COMPATIBLE keyword is used at the beginning of a compatibility statement that defines compatibility based on user-defined attribute values common to standard items of different option classes.

A Compatibility statement requires the keyword COMPATIBLE and two or more identifiers. The syntax of COMPATIBLE...OF is essentially the same as that of FOR ALL....IN. For each formal identifier in the COMPATIBLE clause, there must be a matching identifier in the OF clause. The conditional expression determining the set of desired combinations is in the WHERE clause.

The CDL of a compatibility rule must include at least two iterators.

Example

In the following example, the rule iterates over all the items of the Tint option class in the Glass child model and over all the items of the Color option class in the Frame child model of a Window model. A color and tint are compatible whenever the Stain user-defined attribute in an item of the Color option class equals the Stain user-defined attribute in an item of the Tint option class.

```
COMPATIBLE
&color OF Frame.Color,
&tint OF Glass.Tint
WHERE &color.userAttrs["Paints_AG.Stain"] = &tint.userAttrs["Paints_AG.Stain"];`
CDL Accumulator Statements and the ADD or SUBTRACT Keywords

Unlike constraint statements, accumulator statements contain numeric expressions. In an accumulator statement, the ADD and TO keywords are required.

Example
You use ADD ... TO in an accumulator rule.

```
ADD a TO b;
ADD (a + b) * c TO d;
```

ADD ...TO with Decimal Operands and Option Classes or Collections
Plan carefully when writing rules with decimal operands and option classes, or collections. The following table explains what action should be taken when A accumulates to B and B is either an option class with multiple options, or B is a collection. The columns are If, AND, and Then.

<table>
<thead>
<tr>
<th>If</th>
<th>AND</th>
<th>Then</th>
</tr>
</thead>
<tbody>
<tr>
<td>A resolves to a decimal</td>
<td>Option 1 and Option 2 are both integers</td>
<td>Use the Round() function on A</td>
</tr>
<tr>
<td></td>
<td>Option 1 and Option 2 are both decimals</td>
<td>No further action is needed on A</td>
</tr>
<tr>
<td></td>
<td>Option 1 is decimal and Option 2 is integer</td>
<td>Use Round() function on A to meet the most limiting restriction - Option 2 an integer.</td>
</tr>
<tr>
<td>A is an integer</td>
<td>Option 1 and Option 2 are both integers</td>
<td>No further action is needed on A</td>
</tr>
</tbody>
</table>

Extension Rules

Overview of Extension Rules
Extension rules extend the functionality of configurator rules, through code that you write yourself.

This section provides information on the following:

- The structure and elements of extension rules
- The configuration events that trigger the execution of extension rules
- The procedure for creating extension rules
- The basics of writing the your own code in extension rules
Extension Rules

Extension rules extend the functionality of a configurator model with code that you write using established interfaces, to support business requirements that may not be available through statement rules.

- An extension rule is bound to one or more predefined events that can occur during a configuration session. Example: A change in the value of a decimal feature node.
- An extension rule is associated with a base node. The base node is the model node on which the rule is listening for the events bound to the rule. Listening for events means detecting the occurrence of events. Example: The specific decimal feature node whose value changed.
- The behavior of an extension rule is defined by its rule text, which is a valid script in the Groovy scripting language. Simple example: A function defined in Groovy calculates the effect of applying a different discounts to the list price of an item, in order to arrive at the sale price.

**Caution:** This simplified example is for instructional purposes only. The pricing information displayed in Oracle Fusion Configurator at run time is normally provided by integration with Oracle Fusion Pricing.

- In order to call a specific Groovy function defined in the script, the function is bound to the occurrence of an event, in an event binding. Example: The invocation of the discounting function is bound to a change in the value of the decimal node whose changed value is the new discount to apply.
- To provide flexibility, a bound event can be listened for inside a specified event binding scope. Example: a change in node value is listened for only on the base node of the rule.
- If the bound function has arguments, then each argument must be bound to the source of a value for the argument, in an argument binding. Example: The discounting function has arguments for the list price and the discount; the list price argument is bound to the model node in which an end user enters the list price, and the discount argument is bound to the model node in which an end user enters the discount.
- If the base node can have multiple instances, then an instantiation scope must be specified, to determine, when the rule is invoked, whether a separate instance of the rule is created for each instance of the base node, or a single instance of the rule is created for the whole set of instances of the base node.

Configurator Events

An event is an identifiable action or condition that occurs in a model during a run time configuration session, such a change in the value of a node. Events have names, such as postValueChange.

Event Binding

Event binding connects a method in an extension rule to a configurator event. An extension rule must have at least one event binding.

At run time, Configurator detects and reacts to configurator events using objects called listeners, which are registered to listen for (meaning detect) the occurrence of specified events. You don't explicitly specify listeners when you use extension rules. When you create an event binding for an extension rule, Configurator registers the appropriate listener for the specified event.
If an event occurs during a run time configuration session, and there are bindings for that event with that scope in any extension rules in the model, then Configurator runs all the bound methods for that event.

The events that you can bind to a extension rule are predefined for Configurator, and are described elsewhere in this topic. When you define an extension rule, you choose one of these events as part of the binding of a method in your extension rule text.

**Event Binding Scope**

Event binding scope is the scope in which the configurator event in an event binding is evaluated. An event binding always has a binding scope.

When defining an event binding, you bind events within a certain scope. This event binding scope tells listeners that are registered for the event where in the run time model tree to listen for an occurrence of that event. The following table describes the scopes for event binding.

<table>
<thead>
<tr>
<th>Event Binding Scope</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>Base Node</td>
<td>The extension rule is executed only for the base node.</td>
</tr>
</tbody>
</table>

**Event Descriptions**

An extension rule must always be bound to one of the following predefined events.

<table>
<thead>
<tr>
<th>Event</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>postConfigInit</td>
<td>Event dispatched immediately after initializing a new or restored configuration session. You can use this event to set values in your model as soon as an end user starts configuring the model.</td>
</tr>
<tr>
<td>postConfigSave</td>
<td>Event dispatched immediately after a configuration is saved. You can use this event to set values in your model whenever an end user saves changes to the configuration of the model.</td>
</tr>
<tr>
<td>postValueChange</td>
<td>Event dispatched immediately after the value of a node is changed. You can use this event to set values in your model whenever an end user changes the value or selection of the node associated to the event explicitly or by rules.</td>
</tr>
<tr>
<td>preAutoComplete</td>
<td>Event dispatched immediately before the autocompletion process runs. The autocompletion process is triggered by clicking Finish or Finish and Review in the configurator UI, at run time. You can use this event to set values in your model before an end user reviews the configuration, or saves the configuration and finishes the configuration session.</td>
</tr>
</tbody>
</table>

**How You Create Extension Rules**

You can extend the functionality of your configurator model by creating extension rules, which use code that you write in the Groovy scripting language.
The essential tasks in creating an extension rule are:

- Creating the rule and assigning a base node
- If necessary, selecting an instantiation scope
- Entering rule text
- Adding event bindings
- If necessary, adding argument bindings
- Validating the rule and testing it

Prerequisites
The following things are required when defining an extension rule.

- The model node that's to be the base node for the extension rule must already exist.

  **Note:** If the model node is an item type that might be end dated, then the extension rule will become invalid upon the end date of the item. The end date of the extension rule is displayed in the **End Date** field of the Details region of the Rules tab of the Edit Configurator Model page.

Creating Extension Rules and Assigning the Base Node

2. Enter a name and optional description for the extension rule.
3. Open the model tree in the Structure pane, and select the model node that's to be the base node of the extension rule.
   
   You can associate an extension rule with any type of node, within the model containing the rule.
4. Right-click the model node and select **Set as Base Node** from the context menu. The fully-qualified node name path of the model node is inserted in the **Base Node** field.
   
   You can also enter the node path directly as plain text, in the **Base Node** field. Any node names that contain spaces must be delimited with single quotation marks. You can copy a node path from the **Path** field on the Details region of the Structure tab for the model. If the node's name is unique within the model, then you can type the node name (but not description) directly into the Base Node field. When you save the model, the full node path will be added for you.
5. If the base node can have multiple instances, then the **Instantiation Scope** control is enabled. The default selection is **Instance**.

Entering the Rule Text
The rule text defines the behavior of the extension rule.

1. Enter the text of a valid Groovy script into the **Rule Text** field. Click **Save** to save the rule text and the rest of the rule definition.

   **Tip:** You can save the rule as you work on the rule text, even if it's not complete or correct.

2. To ensure that the extension rule, and the rule text, are valid, click **Validate**, at any point during the definition.

Adding Event Bindings
Add at least one event binding to the rule.

1. In the Event Bindings table, click **Create**
2. In the new row for the event binding, from the **Event** list, select the event to bind.
3. From the **Event Scope** list, select the event scope in which the event is to be evaluated.
4. From the **Class** list, select the class that contains the method that you’re binding to the event.

If no classes are defined in the script, select **ScriptClass**, which contains globally-defined methods.

**Tip:** Click **Validate** to refresh the **Class** and **Method** lists after any changes to the script to add classes or methods. If you haven’t yet clicked **Validate**, then the lists will be empty.

5. From the **Method** list, select the method that you’re binding to the event.

If no methods are explicitly defined in the script, select **run()**, which executes globally-defined methods.

You can create multiple event bindings for an extension rule. The sequence in which you create them isn’t significant for the execution of the rule.

### Adding Argument Bindings

When you select a method with arguments for the event binding, the Argument Bindings table is automatically populated with a row for each argument. The **Name** column for each argument contains the read-only name of the argument, copied from the method definition in the rule text.

1. Select a row in the Argument Bindings table.
2. From the **Specification** column for each argument, select the specification for how the method obtains the argument value.
3. If you select a specification of **Literal**, then enter a numeric or string value, without quotation marks.
4. If you select a specification of **Model Node**, then open the model tree in the Structure pane, and select the model node that provides the value for the argument. Then right-click the node and select **Set as Argument Value**. If the node’s name is unique within the model, then you can type the node name (but not description) directly into the Model Node field. When you save the model, the full node path will be added for you.
5. Repeat the argument binding for each argument in the Argument Bindings table.
6. Click **Validate**, to validate both the rule text and the rest of the rule.
7. Click **Save**, **Save and Compile**, or **Save and Close**.

### Validating Extension Rules

You can ensure that the entire extension rule (both rule text and bindings) is valid, by clicking **Validate**, at any time during the definition. (In contrast, the **Validate** button for statement rules only validates the rule text.)

You can validate the rule at any point while you’re defining it. Some of the requirements that are commonly reported by validation are the following:

- An extension rule must have a base node.
- The rule text can’t be empty.
- An extension rule must have at least one event binding.
- If your Groovy method bound to the event has arguments, you must bind the arguments.
- Groovy annotations aren’t allowed.
- Your Groovy script must be syntactically correct. Any Groovy syntax and programming errors are reported.
- If you change your script in a way that makes existing event bindings invalid, then the invalid events are marked with an error icon. For example, changing the name of a method invalidates any even bindings using that method. A rule with invalid bindings becomes invalid and is ignored when testing the model.
The Extension Rule Text Editor

The Rule Text section of an extension rule provides the following features for working with rule text:

- Undo and redo of text edits
- Line numbering
- The Find control with Next and Previous buttons
- The Find and Replace dialog box, opened by clicking the Find and Replace control
- The Go to Line control, activated by clicking the Jump to line control
- The rule text pane, expanded and restored by clicking the Maximize control. While the text pane is expanded, any validation errors are displayed in a region below the text pane.
- The Collapse Pane slider, to hide the rule text pane
- Colorization of keywords, literals, and comments
- Colorized matching of brackets and parentheses (green if closed, red if not closed)
- Automatic indenting of new lines

How You Write Extension Rule Text

The scripts that can be used as rule text for extension rules are written in the Groovy scripting language, and use interfaces that access configurator model objects.

The Groovy Scripting Language

The Groovy language provides a convenient means of writing scripts for extension rules.

Groovy is object-oriented and dynamically compiled. It can be used as a scripting language for the Java platform. Groovy is widely described elsewhere, at http://groovy-lang.org, and other public sources.

For writing extension rule scripts, some relevant features of Groovy are:

- Many base Java packages are automatically imported, so that you don't have to import them in your scripts.
- You can declare variables with the def keyword, without having to declare their type.
- You can define methods and variables outside a class. They're considered as global within the script. Global definitions are executed within the built-in class ScriptClass, and executed under the built-in method run().
- Some advantages to placing methods and variables in classes are:
  - You can use encapsulation and inheritance.
  - You can build complex logic, as in Java.
  - Organizing methods in classes makes it easier to select them from the Class and Method lists when adding event bindings.
- Advantages to not placing methods and variables in classes include:
  - If you don't use classes, you can write code and test your code more quickly, simply adding and running code bound to the global class ScriptClass and the global method run().
### Accessing Important Configurator Objects

Certain objects in the Configurator runtime core package provide access to the objects most commonly needed during a configuration session.

When an extension rule runs, the Configurator framework automatically initializes an object that represents the event specified by the event binding of the rule. This object is an instance of the interface `ICXEvent` named `cxEvent`.

The following table presents the objects most commonly needed during a configuration session. For each object, the table provides:

- A short description of the object
- The Java interface containing methods to access that object
- A short example of code that creates an instance of the object
- A short example of code that uses an instance of the object

<table>
<thead>
<tr>
<th>Object</th>
<th>Entity Accessed</th>
<th>Interface Used</th>
<th>Example of Creating the Object</th>
<th>Example of Using the Object</th>
</tr>
</thead>
<tbody>
<tr>
<td>Configurator event</td>
<td>The event specified by an event binding in the rule.</td>
<td>ICXEvent</td>
<td>No code needed. The object <code>cxEvent</code> is created automatically when the rule runs.</td>
<td>Get the current configuration associated with the event that triggered the rule.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td><code>IConfiguration config = cxEvent. getConfiguration();</code></td>
</tr>
<tr>
<td>Base node of rule</td>
<td>The base node associated with the rule.</td>
<td>ICXEvent</td>
<td>Get the base node of the rule triggered by a bound event.</td>
<td>Test whether the base node of the rule is selected by the end user.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td><code>def baseNode = cxEvent. getBaseNode();</code></td>
<td><code>if (baseNode.isSelected()) ..</code></td>
</tr>
<tr>
<td>Configuration</td>
<td>The active configuration during a session.</td>
<td>IConfiguration</td>
<td>Get the configuration affected by the rule triggered by a bound event.</td>
<td>Get the root node of the item-based model currently being configured.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td><code>IConfiguration config = cxEvent. getConfiguration();</code></td>
<td><code>def root = config. getRootBomModel();</code></td>
</tr>
<tr>
<td>Root node of model</td>
<td>The root node of the model being configured.</td>
<td>IBomModelInstance</td>
<td>Get the root node of the item-based model currently being configured.</td>
<td>Get a child node of the current model, using its item name to search the model tree, starting from its root node.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td><code>IBomModelInstance root = config. getRootBomModel();</code></td>
<td><code>def childItem = root. getChildByName(&quot;CM85010&quot;)</code></td>
</tr>
</tbody>
</table>
Interacting with Model Node Values

You can get, and set, the values and states of model nodes using the interfaces described here.

The following table presents the objects that represent the types of nodes in a configurator model. For each object, the table provides:

- The Java interface containing methods to access that object
- The prototype of a method for getting the current value or state of the object.
- The prototype of a method for setting a new value or state for the object.
- Note that Groovy can derive data types when they're used at run time, so it's not strictly necessary to declare return types in your script. However, it's a good practice to understand the objects and interfaces involved in your model interactions.

The interfaces of the Oracle Fusion Configurator API also provide many other interfaces and methods for other types of interactions with node objects. This table provides an introduction to methods that are useful for common operations. For details, see the Java API Reference for Oracle Fusion Configurator, available on the Oracle Help Center.

<table>
<thead>
<tr>
<th>Object</th>
<th>Interface</th>
<th>Get Value</th>
<th>Set Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integer feature</td>
<td>IIntegerFeature</td>
<td>int getValue()</td>
<td>void setValue(int value)</td>
</tr>
<tr>
<td>Decimal feature</td>
<td>IDecimalFeature</td>
<td>double getValue()</td>
<td>void setDecimalValue(double value)</td>
</tr>
<tr>
<td>Option of option feature</td>
<td>IOptionFeature</td>
<td>IOption getSelectedOption()</td>
<td>void select()</td>
</tr>
<tr>
<td>Boolean feature</td>
<td>IBooleanFeature</td>
<td>boolean isSelected()</td>
<td>void toggle()</td>
</tr>
</tbody>
</table>

Other Model Interactions

You can perform a variety of important interactions with model nodes using the interfaces described here.

- Getting and Setting Logic States
- Accessing properties
- Access to options
- Overriding contradictions
- Handling logical contradictions
- Handling exceptions

Reference Documentation for Available Classes

The package containing the classes for interacting with configurator is oracle.apps.scm.configurator.runtime.core.

The reference documentation for the interfaces in that package that you use in writing extension rule scripts is the Java API Reference for Oracle Fusion Configurator, available on the Oracle Help Center.

Some members of supported classes and interfaces aren't supported for use in extension rules, and their use will cause an error upon compilation. Unsupported members are omitted from the API reference documentation. If your script
refers to a class or member that's not available for use in extension rules, a validation message identifies the invalid reference.

Create an Extension Rule

In this example, you define an extension rule that calculates the effect of applying different discounts to the list price of an item, in order to arrive at the sale price. The formula used in this example to calculate the discount is simple, but you can use this technique to construct more complex extension rule functionality, through the script in your rule text.

The tasks in this example are:

1. Define supplemental features for list price, discount, and sale price.
2. Create an extension rule, with the discount feature node as the base node.
3. Write a Groovy script that defines a discounting function and applies it to the supplemental features.
4. Define event and argument bindings for the function.
5. Test the rule.

Caution: This simplified example is for instructional purposes only. The pricing information displayed in Oracle Fusion Configurator at run time is normally provided by integration with Oracle Fusion Pricing.

Defining Supplemental Features

The supplemental features are used for the end user inputs for list price and discount, and the calculated output of the sale price.

1. On the Overview page of the Configurator Models work area, select Create from the Actions menu, to create a new workspace. Set the Effective Start Date to tomorrow’s date.
2. Open the workspace. On the Workspace page, select Select and Add Models from the Actions menu.
3. On the Select and Add: Models page, search for a model, select it, then click OK, to add it to the workspace. Ignore any warning about drafts in other workspaces.
4. On the Workspace page, click the model's name to open it for editing.
5. On the Edit Configurator Model page, select the root node of the model, and select Create Decimal Feature from the Actions menu.
6. Create the following decimal features, as shown in these tables.

<table>
<thead>
<tr>
<th>Field</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>List Price</td>
</tr>
<tr>
<td>Minimum</td>
<td>20,000</td>
</tr>
<tr>
<td>Maximum</td>
<td>60,000</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Field</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>Discount</td>
</tr>
</tbody>
</table>
Field | Value
--- | ---
Minimum | 0
Maximum | 10

Field | Value
--- | ---
Name | Sale Price
Minimum | 0
Maximum | 100.000

7. Click **Save**.

Creating the Extension Rule

The extension rule will apply the discount to the list price, and put the result in the sale price.

2. Enter a **Name** for the extension rule, such as **Apply Discount**.
3. In the Structure palette, expand the model tree, select the node **Discount**, then right-click and select **Set as Base Node** from the context menu.
4. Click **Save**.

Writing the Rule Text

The behavior of the extension rule is defined in the script entered in the Rule Text field, which is written in the Groovy scripting language.

1. In the Rule Text field, enter the following script.

   ```groovy
   // Import the needed Configurator interfaces
   import oracle.apps.scm.configurator.runtime.core(IConfiguration
   import oracle.apps.scm.configurator.runtime.core.IBomModelInstance
   import oracle.apps.scm.configurator.runtime.core.IDecimalFeature

   // Define the discounting function.
   def applyDiscount ( p_listPrice, p_discount ) {

   // Get values of nodes from arguments.
   double listPrice = ((IDecimalFeature)p_listPrice).getValue()
   double discount = ((IDecimalFeature)p_discount).getValue()
   double salePrice = 0

   // Calculate the price.
   salePrice = listPrice - ( listPrice * ( discount / 100 ) )

   // Get the node whose value will be set.
   IConfiguration config = cxEvent.getConfiguration()
   IBomModelInstance root = config.getRootBomModel()
   IDecimalFeature salePriceNode = root.getChildByName("Sale Price")
   ```
// Set the value.
((IDecimalFeature)salePriceNode).setDecimalValue(salePrice)
}

2. Click Save.
3. Click Validate.
4. You should receive the error message The rule is invalid. The extension rule must have at least one event binding defined.

Defining the Event Binding
To make the script execute, you must bind it to a configurator event. So now you add an event binding to the rule.

1. In the Event Bindings table, click Create.
2. In the new row for the event binding, from the Event list, select postValueChange.
3. From the Event Scope list, select base node.
4. From the Class list, select ScriptClass.
5. From the Method list, select applyDiscount. The arguments p_listPrice and p_discount are displayed with the function name.
6. Click Save.

Defining the Argument Bindings
When you selected the applyDiscount method in the Event Bindings table, the Argument Bindings table should have automatically appeared, populated with a row for each argument.

1. In the Argument Bindings table, select the row for the argument named p_listPrice.
2. From the Specification column for that row, select Model node.
3. Expand the model tree in the Structure pane, and select the node List Price, then right-click the node and select Set as Argument Value.
4. Click Validate.
5. You should receive the error message The rule is invalid. The node referenced in the argument p_discount bound to the event postValueChange was not found.
6. Repeat the preceding argument binding steps for the argument p_discount and the node Discount.
7. Click Validate.
8. You should receive the information message No errors were detected.
9. Click Save and Compile.
10. You should receive the confirmation message Model compilation has completed without errors.

Testing the Model
Test the model to verify the functionality of the extension rule.

1. On the top of the Edit Configurator Model page, select Test Model.
2. In the Test Model dialog box, ensure that User Interface is set to Default, then click OK.
3. On the Test Model page, fields for the three decimal features that you added should appear, in the sequence that you created them:
   o List Price
   o Discount
   o Sale Price
4. In the List Price field, enter the value 30,000. Notice that you can only enter values between the minimum and maximum that you defined when you created the decimal feature.
5. In the Discount field, enter the value 5. Notice again that the allowable values are between the minimum and maximum that you defined.
6. When you press Enter in the Discount field, or tab out of it, notice that the value of the **Sale Price** field changes, from empty to 28500.

7. Enter a different value in the Discount field, and notice the changes in the Sale price field. This change happens because you bound the execution of the extension rule to changes in the value of Discount.

8. Enter a different value in the List Price field, and notice that there is no change in the Sale price, because there is no binding in the rule to changes in the value of List Price.

9. Click **Finish** to end the test session.

### How You Obtain External Information Using Extension Rules

You can call an external service from within an extension rule. This is useful when you need to consult an external source to assist in the proper configuration of a model. It is often more efficient to obtain data (such as product specifications or government regulations) from an external source than to attempt to maintain it in your model.

In order to call an external service from an extension rule, you must:

- Register a web service connection to be used by a service call
- Define an extension rule in which the script calls the service using that connection

#### Registering the Web Service Connection

In order for Configurator to access an external web service, you must register a connection for it.

To define a web service connection:

1. In the Setup and Maintenance work area, select **Search** from the task panel. Search for and select the task **Manage External Service Details for Extensions**.
2. In the Web Service Details table, select **Add Row** from the Actions menu.
3. From the **Target System** list, select a registered trading community partner application or Oracle Fusion application.
4. In the **Connector Name** field, enter the name that you will use to refer to the service, in your extension rule, such as **AxelWS**.
5. In the **Connector URL** field, enter the URL to the external web service, such as `http://axel04.com:7011/services/AxelServicesPort`.
6. From the **Invocation Mode** list, select **Synchronous service**, which is the mode supported.
7. Optionally enter other service information that may be required for calling the web service, such as User Name and Password.
8. Select **Save and Close**.

#### Defining the Extension Rule

In order for Configurator to call an external web service, you must define an extension rule that uses a particular API to call the service and specify the desired operation.

1. Navigate to the Configurator Models work area.
2. In a workspace, add your model and open it for editing.
3. On the Rules tab of the Edit Configurator Model page, select **Create Extension Rule** from the Actions menu.
4. In the Structure pane, select a base node for the rule, such as the root node of the model.
5. In the Rule Text pane, enter a valid Groovy script that calls the external service. A sample script is provided in this topic.
6. When defining the event binding for the rule, choose when you want the service to be called:
   - To call the service when the configuration session begins, choose `postConfigInit`. 
To call the service when the associated model node value is changed, choose postValueChange.

For the example script provided, the event binding must have a **Class** of ScriptClass and a **Method** of getCPUHw(String region).

The argument binding for the argument region must be a value that can be interpreted by the web service.

### Calling an External Service

The example script provided here demonstrates the essential points of calling an external web service in an extension rule script.

**Important points to remember:**

- It is essential to import the classes SoapServiceResponse and SoapServiceRequest. The need for other imports is determined by the desired behavior of your script.
- When constructing the payload string, this example uses the namespace http://services.axel and the service method cpuhw. Assume that these are defined in the web service that you registered.
- You must use the method invokeSoapService() from the configuration object to call the service, and pass it the connector name that you registered and the payload string that you constructed. In this example, the connector name is AxelWS.
- Some external SOAP services can only be invoked using SOAP 1.1 specs. They don't work via SOAP 1.2, possibly because WSDL is only SOAP 1.1 compatible. In order to get around this issue, add the following HTTP headers when invoking such services from an extension rule:
  - **Content-Type**: text/xml
  - **SOAPAction**: executeWebCX

Here is an example of how these HTTP headers can be set in an extension rule:

```java
SOAPMessage message = request.getSoapMessage();
message.getMimeHeaders().addHeader("Content-Type", "text/xml; charset=utf-8");
message.getMimeHeaders().addHeader("SOAPAction", "executeWebCX");
```

If these HTTP headers aren't set then a SOAP 1.1 call fails.

This example script constructs a SOAP message for a web service that tries to select some items:

```java
import oracle.apps.scm.configurator.runtime.core.IRuntimeNode;
import oracle.apps.scm.configurator.runtime.core.ServiceException;
import oracle.apps.scm.configurator.runtime.core.SoapServiceRequest;
import oracle.apps.scm.configurator.runtime.core.SoapServiceResponse;
import javax.xml.soap.SOAPBody;
import javax.xml.soap.SOAPMessage;
import javax.xml.soap.SOAPElement;
import javax.xml.namespace.QName;
import org.w3c.dom.Document;
import org.w3c.dom.Element;
public class WebCXmin implements Serializable {
    public WebCXmin() {} 
    public String callWebService(IRuntimeNode node) {
        SoapServiceRequest request = new SoapServiceRequest(node.getConfiguration());
        SOAPMessage message = request.getSoapMessage();

        // create XML payload here
        QName bodyName = new QName("ws", "executeWebCX");
```
SOAPBodyElement bodyElement = request.getSoapBody().addBodyElement(bodyName);
bodyElement.addNamespaceDeclaration("ws", "http://services.axel/");
bodyElement.setPrefix("ws");

QName className = new QName("className");
SOAPElement classes = bodyElement.addChildElement(className);
classes.addNode("axel.ce.ws.webcx.DSPrimeTestCX");

QName name = new QName("params");
SOAPElement params = bodyElement.addChildElement(name);

QName entry = new QName("entry");
SOAPElement entries = params.addChildElement(entry);

QName key = new QName("key");
SOAPElement keys = entries.addChildElement(key);
keys.addNode("json");

QName value = new QName("value");
SOAPElement values = entries.addChildElement(value);
values.addNode("{"qty": 2, "children": [], "type": "MI", "state": ["UTRU", "SELD", "USLD"]}");

message.getMimeHeaders().addHeader("Content-Type", "text/xml; charset=utf-8");
message.getMimeHeaders().addHeader("SOAPAction", "executeWebCX");

SoapServiceResponse response;
try {
    // Invoke the external web service that was registered in the Setup and Maintenance work area
    response = node.getConfiguration().invokeSoapService("AxelWS", request);
} catch (ServiceException e1) {
    throw new Exception(" msg=" + e1.getMessage());
}

def base = node;
def tf = base.getChildByName("TextFeature");
Document doc = response.getSoapBody().extractContentAsDocument();
Element root = doc.getDocumentElement();
tf.textValue = "SOAP Body node: " + root.getFirstChild().getTextContent();

FAQ for Model Rules

What happens if I change the names of supplemental structure nodes that are rule participants?

When you compile the model, any rules that reference those renamed nodes become invalid.

- A warning message identifies the affected rules.
- The warning message identifies the node names that were used when the rules were defined, but which no longer have those names.
- The Status field of the invalid rules changes from Valid to Error.
- If you update a rule to use the new node name, the rule becomes valid again.
Why can I test, but not release, a model containing invalid rules?

You can test a model containing invalid rules, because the testing phase of model development enables you to make those rules valid. But you can't release a model containing invalid rules, because invalid rules prevent users from creating valid configurations.

If you don't make an invalid rule valid, you can still release the model if you disable that rule.
6 Model User Interfaces

Overview of Model User Interfaces

This chapter covers model user interfaces. User interfaces present a configurator model to the end user for interaction. A model can have a variety of user interfaces to fit different usages.

- UIs are composed of templates that represent UI items, and template maps that connect the templates to nodes in the model. At run time they are dynamically rendered together to present a user interface that accurately represents the model structure.
- A model can have a variety of user interfaces, to fit different usages.
- If no user-defined UI has been created for a model, a default UI is presented.
- You can generate UIs for a model, based on its structure, then further modify them, using the What You See Is What You Get (WYSIWYG) page editor that shows live model data as it will appear at run time. When generating a UI, you can choose from a predefined set of navigation styles.
- Some changes to the original product model are automatically reflected in its configurator model UIs, but certain changes must be explicitly performed.
- You can control the presence of items in the UI with display conditions.
- You can set applicability parameters that allow a model to use multiple UIs, each targeted to a different sales channel.

You access user interfaces in the following ways.

- **Configurator Models work area > tasks panel tab > Manage Models page > Search > Versions > click a Draft or Version > Configurator Model or Edit Configurator Model page > User Interfaces tab**
  The Edit Configurator Model page enables you to create and edit the structure, rules, and user interfaces of a configurator model. If the model is locked by another user, then you can only view the model.

- **Configurator Models work area > tasks panel tab > Manage Workspaces page > Workspace page > click Name of participating model > Configurator Model or Edit Configurator Model page > User Interfaces tab**

Configurator Model User Interfaces

The user interface (UI) of a configurator model is what the end user sees and interacts with to configure the product represented by the model.

- UIs let users select options of the model by presenting controls based on the model structure.
- UIs can be dynamically generated at run time, or explicitly generated and saved. Explicitly generated UIs can be modified to suit your requirements.
- A configurator model can have multiple UIs, applied to suit varying styles of end user interaction.
- UIs that you create for a model are part of the definition of that model version, like supplemental structure and configurator rules. UIs can’t be shared with other models. The use of UI templates enables you to provide a consistent user experience among your models.
- Each UI uses one of a set of predefined navigation styles to enhance the end user experience.
- UIs can be integrated with the UI of a hosting application.
- UIs consist of:
  - UI metadata that represents the model structure in terms of pages, regions, and items, and inter-page navigation.
  - UI templates that contain the visual content for the UI.
  - UI template maps that map model node types to UI templates

These elements are exposed in explicitly generated and saved UIs, allowing modification. They aren’t exposed in dynamically generated UIs, including the default UI.

**UI Templates**

Configurator user interfaces consist of a set of templates, which are dynamically rendered at run time.

The following templates are the building blocks of user interfaces

- The shell template for a UI keeps together all the other regions or parts of the UI and provides the navigation and actions for the UI.
- The layout templates for a UI determine the visual layout (such as a form or stack) of the control templates or elements within a layout region. Each UI can have one or more layout templates per page.
- Control templates represent UI items and allow user interaction, such as selection or input.
- Message and utility templates provide UI elements for specialized parts of a page. You can’t modify these templates.

**UI Template Maps**

UI template maps govern the overall behavior and appearance of UIs.

When you create a user interface, you select a UI template map, which determines how the UI is constructed. UI template maps maintain the mapping between types of model nodes (such as standard items, option classes, reference models, and supplemental-structure features) and the control templates that allow users to interact with the nodes. Examples of such mapping are as follows:

- If an option class is defined as having mutually exclusive options (meaning that only one option at a time can be selected), it’s mapped to a radio button group control template.
- If an option class isn’t mutually exclusive, it’s mapped to a check box group control template.
- A required model reference is mapped to an item selection control template.

UI template maps group control templates into UI pages that represent major components of a configurator model, such as option classes. UI template maps also determine the navigation style between pages.
The Default User Interface

If you don't define any of your own UIs for a configurator model, the model uses a default UI at run time.

- The default UI is created dynamically at run time if no generated UI is specified, using UI templates and UI template maps.
- The default UI reflects any model changes, and doesn't need to be refreshed.
- The Single Page Navigation UI template map governs the default UI. This choice is predefined. You can't select a different UI to be the default UI.
- The default UI is also used for a configurable product model for which no configurator model was created.
- There's only a single default UI in use at a time. The same default UI is used for all configurator models that need one at run time.

Generated User Interfaces

You can create a user interface for a configurator model by generating it, which is an action that automatically builds a UI based on the model structure, using one of a set of templates that determine the appearance and interactive behavior of the UI.

- Generating a UI is an optional part of model definition. If you don't generate any UIs for your model, then the default UI is used at run time.
- When you generate a UI, you select a UI template map that imparts a distinct look and feel to the UI, including navigation style. After you create the UI, you can't change the UI template map that it uses.
- When you generate a UI for a model that includes referenced models, then UIs are generated for any referenced models that don't already have their own UIs.
- You can suppress a particular model structure node from appearing in UIs that you generate by deselecting the Display in user interface check box on the UI Presentation tab of the Details region for that node. This setting doesn't suppress the node in existing UIs.

Generating a User Interface

Generated UIs are created on the User Interfaces tab of the Edit Model page of the Configurator Models work area.

1. On the Edit Configurator Model page, navigate to the User Interfaces tab.
2. Select Create from the Actions menu.
3. In the Create User Interface dialog box, enter a name for the new UI, and select a UI template map.

Predefined UI template maps are provided for each of the navigation styles, in two versions for each style:

- Template map with ordinary selection controls.
- Template map with enhanced selection controls, which show more detail about the state of the selected items. For example, an icon indicates whether an item was selected by the end user or by a rule.
- The UI template map named Single Page Navigation for Test UI with Enhanced Selection Controls is used when you test the behavior of a model with the Test Model operation. This template should never be chosen for a UI that's intended for run time use with end users.
4. Click the **Save and Close** button.
5. A new UI is now generated automatically, following the selected template map that associates the structure of your configurator model with UI elements.
6. Select the new UI in the User Interfaces list. On the Overview tab for the UI, you can edit the name and description. You can also choose the applications and languages for which your user interface is applicable.
7. Optionally, to verify that the behavior of the generated UI is what you expect, test the model, using the **Test Model** button.
8. If you made further changes to the UI, click **Save**, to save them.

**Testing a User Interface**

To verify that the behavior of a generated UI is what you expect, test the model.

To test a user interface:

1. Click the **Test Model** button. You don't have to be on the User Interfaces tab for the model.
2. In the Test Model dialog box, ensure that you select the new generated user interface that you intend to test, in the **User Interface** field. The selected UI isn't necessarily the UI that you're editing. By default, the previously tested UI is selected.
3. Make selections among the configuration options, and navigate through the pages of the UI. Observe how the UI functions in presenting configuration choices.

When testing a user interface, consider the following criteria.

- How effective is the navigation style (which is associated with the UI template map you chose) in reaching all parts of the model in the way that the end user expects for the product?
- Does the generated set and sequence of UI pages (which is determined by the UI template map and the model structure) enable the end user to locate and configure the most important elements of the model easily and efficiently?
- Do the generated headings and captions (which are derived from the node descriptions in the model structure) guide the end user in understanding what's to be configured?
- Are the default UI controls (which are generated by control templates) appropriate to interacting with items?

If any of these elements of the generated user interface are insufficient for your purpose, consider modifying the generated UI.

**How You Synchronize UIs with Structure**

The changes to product item structure in the Product Information Management work area must be reflected in user interfaces for the affected snapshots and models.

When a product item changes, you refresh snapshots for that item. These changes can affect the item-based structure of any corresponding configurator models created from those snapshots, and consequently affect any user interfaces created for those models.

To account for model changes, you must create a new workspace and add the affected model and the updated snapshots to that workspace. UIs aren't automatically refreshed to synchronize with all model structure changes. The following list explains what you need to know, or to do, to keep your UIs current with the product changes in the Product Information Management work area.

- When a UI is initially generated, it includes by default all the nodes in the model. If model nodes have been deleted, or have become ineffective (meaning that they represent items that are end-dated as of run time),
they’re automatically filtered out of your UI without further action by you. They aren’t displayed at run time, and display conditions using such nodes are ignored.

- If new option classes, model references, or individual items are added to the existing model and should be visible in the UI, then you must add them individually to the existing UI.

However, the existing UI continues to work without any changes for the following additions:

- New items that are added to an existing option class.
- New options that are added to an existing option feature.
- New transactional item attributes that are added to an existing item (excepting the model item itself).

- If the product changes from a snapshot refresh involve changes such as instantiation type, or minimum and maximum quantities, then existing UIs will continue to function, but may not provide the best user experience, and may allow the creation of invalid configurations, or prevent the creation of certain configurations that would be valid.

- If the configuration behavior for a node changes, then you might have to change the type of control template that renders the node in the UI. For example, if an option class that originally had mutually exclusive options now allows multiple selections, then you might want to change the control template for the node’s page item from a radio button group to a check box group.

- If supplemental structure nodes have been reordered in the model structure, then you must reorder them in the existing UI, if they’re explicitly displayed in the UI. The option nodes of an option feature aren’t affected.

### Multichannel User Interfaces

Applicability parameters allow a model to use multiple UIs, each targeted to a different channel of use. Applicability helps you present the UI that’s most appropriate to the context.

- You may need to configure the same model in multiple host applications, each having different UI requirements.
  - Host application A is used by self-service customers with elementary knowledge of your product line. You might need to present a simplified UI for Product X that guides the user through each step of the configuration, and hides some product details that might be confusing.
  - Host application B is used by internal sales fulfillment staff who are very familiar with your product line. You might need to present a full-featured UI for Product X that exposes every option, in a layout that enables users to reach those options most efficiently.

- You may need to present the same product to the same type of audience, but in different countries. Consequently you need to present the UI in multiple languages.

To provide for such multiple requirements, you can set the applicability parameters for a UI.

### Setting Applicability Parameters

On the Overview tab for the UI, you can choose the applications and languages for which your user interface is applicable.

1. Edit your configurator model and navigate to the Overview subtab of the User Interfaces tab.
2. Under Applicability, select a parameter:
   - **Applications** sets the applications that the UI will be used for. For example, if you select **Order Management**, then the UI will be presented when Configurator is invoked by Oracle Fusion Order Management.
   - **Languages** sets the languages that the UI will be used for. For example, if you select **Korean** and **American English**, then the UI will be presented when Configurator is invoked by applications using one of those languages.

3. The default setting for each parameter is **All**, meaning that the UI is available at run time to all channels.

4. Select the **Selected** setting. The **Select** button becomes enabled.

   By default, the currently selected parameter is **None**. If you leave the setting as None, then the UI will not be available at run time to any of that parameter's options. If no UIs are available, then the default UI is used.

5. Click the **Select** button. The selection dialog box for the parameter presents a list of available options, from which you select one or more to determine the applicability of the UI.

6. If more than one UI has the same applicability parameter settings, then the sequence of UIs in the table on the User Interfaces tab determines which UI will be used at run time.

   To change the sequence in the table of UIs, select a UI then select one of the **Move** commands on the Actions menu.

### Templates, Pages, and Navigation Styles

User interfaces are composed of pages on which UI elements mapped to model structure are placed. The UI pages are associated with model nodes, and the navigation between pages is part of the mapping.

UI template maps determine the navigation style between the UI pages that represent major components of a configurator model. You see a template map when creating a UI. The available navigation styles are:

- Single Page Navigation
- Dynamic Tree Navigation
- Step by Step Navigation

Each template map is available in alternate variants:

- Template map with ordinary selection controls.
- Template map with enhanced selection controls, which show more detail about the state of the selected items. For example, an icon indicates whether an item was selected by the end user or by a rule.

You can't change the choice of template map after creating a user interface. If you need to use a different template map, you must create a new UI using that map.

**Note:** Don't use the Single Page Navigation for Test UI with Enhanced Selection Controls template on UIs intended for end users. That template is designed for use only with the Test Model operation.

### Single Page Navigation

The Single Page navigation style collects all the configurable options of a model onto a single page. If a model has reference models, the user can drill down into the UI for the reference model by clicking the **Configure** control on the reference to the reference model.
The UI-level actions that the end user can select on this page are:

- **Finish**: The configurator engine finishes the configuration, which means that the engine automatically completes the remaining selections that are required for a valid configuration of the model. After the configuration is finished, the configuration session ends, the configuration data is returned to the host application, and the user is returned to the previous location in the host application.

- **Finish and Review**: The configurator engine finishes the configuration, then navigates the user to the Review page, where the user's selections are displayed for review and possible further configuration.

- **Save for Later**: Saves the configuration in the exact state left by the user. The configurator engine doesn't finish the configuration, and the saved configuration may be invalid.

- **Cancel**: Warns the end user about losing any selections made, and returns to the host application without saving any configuration data.

If a model is being configured in a host application, and no corresponding configurator model with a UI exists yet, the Single Page UI is displayed for the end user to configure the model.

**Dynamic Tree Navigation**

The Dynamic Tree navigation style allows end users to navigate to a specific UI page by using the tree links that are displayed in the left pane. When the UI is created, each of these tree links is created as a page.

There's no tree link available to navigate to reference models, but the user can drill down into the UI for the reference model by clicking the **Configure** control on the reference to the reference model.

This navigation style provides the same UI-level actions as the Single Page style.

**Step by Step Navigation**

The Step by Step navigation style allows end users to navigate to a specific step by using a series of linked UI train stops that are displayed at the top of the page. When the UI is created, each of these tree links is created as a page.

There's no train stop available to navigate to a reference model, but the user can drill down into the UI for the reference model by clicking the **Configure** control on the reference to the reference model.

This navigation style provides the same UI-level actions as the Single Page style. In addition to those actions, there are two additional buttons, **Back** and **Next**, which are available to enable navigation to the previous or the next step.

**Running Summary**

To assist a sales end user in maintaining a comprehensive picture of a complex configuration, you can add a running summary pane to each of the navigation style templates.

The running summary templates enable you to display cumulative sales information for all the items in the current configuration that can be selected and ordered.

The following running summary templates are available:

- Running Summary with Item Description, and Amount
- Running Summary with Item Description, Quantity and Amount
- Running Summary with Quantity and Item Description

The summary also includes the total configured net price, excluding charges such as tax and shipping.
To add a running summary to one of your own user interfaces:

1. On the User Interfaces tab of the Edit Configurator Model page, select one of your UIs.
2. On the Overview subtab, select a template from the **Running Summary Template** list.
3. When the UI is tested, the running summary appears on the right side of the page. The items in the summary are updated as you make selections in your configuration.

### Instance Management Table

The configuration of multiple instances of a model can be complex and confusing to the end user. You can use the Instance Management Table template in model user interfaces to provide easier configuration of multiple instances at run time.

The Instance Management Table template is only applicable to model nodes that can have multiple instances. Multiple instances are specified prior to model import, in the Product Information Management work area. Such nodes display the **Instantiability** setting as **Multiple instances** on the Details pane of the model's Structure tab. When you create a user interface, any model nodes that can have multiple instances are rendered by default in the new UI using the Instance Management Table template. You can also create UI page items using the template by adding instantiable nodes from the model tree in the Structure pane.

The following variants of the template are available. You can change the variant by editing the page item for the template in the WYSIWYG pane.

- Default (Instance Management Table from UI Template Map)
- Instance Management Table
- Instance Management Table with Facets

At run time, an instantiable node rendered with the Instance Management Table template is initially displayed as a caption and an **Add** button. When the end user clicks Add, an instance of the node is created, and the instance management table is updated, adding a row that enables the end user to manage the newly created instance. The columns of the instance management table are described in the following table.

<table>
<thead>
<tr>
<th>Table Heading</th>
<th>Instance Behavior</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Name</strong></td>
<td>The default name of the instance of the node, which can be edited by the end user. Each instance is identified by adding a sequence number. Example: [1] [2] [3] .... Instance numbers remain associated with the instances for which they're created. Example: If the user creates instances [1] [2] [3] and deletes instance [2], then the next new instance added is numbered [4], although there are three instances at that moment.</td>
</tr>
<tr>
<td><strong>Quantity</strong></td>
<td>The number of instances of the item that will be managed together from that row of the table. Example: If the Quantity is 4, then 4 instances can be configured identically, deleted together, or duplicated together. The number of instances that can be entered in this field at that moment is displayed as tip text when you click in it. Note that the total quantity for all the instances in the table is governed by the Minimum Quantity and Maximum Quantity of the associated instantiable model node. Example: if the Maximum Quantity is 8, and there are four instances, and the user enters a quantity of 5 for one instance, then the other three instances are each assigned a quantity of 1.</td>
</tr>
<tr>
<td><strong>Configure</strong></td>
<td>The edit control to configure the instance. The configuration that you perform on this instance is applied identically to all of the number of instances in <strong>Quantity</strong>.</td>
</tr>
<tr>
<td><strong>Delete</strong></td>
<td>The edit control to delete the instance. Instance numbers of deleted instances aren't reused.</td>
</tr>
</tbody>
</table>
Table Heading | Instance Behavior
--- | ---
Duplicate | The edit control to duplicate the instance. The new duplicate instance keeps the instance number of its original, and prefixes the instance name with Copy of. Additional duplicates are prefixed with Copy 2 of and so on. Since the instance number of the original is retained in its duplicates, but each duplicate is counted as an instance, a new instance added with the Add button is assigned an instance number that reflects the current total number of instances. Example: If the user creates instance [1] and makes three duplicates of it, then the next new instance added is [5], since there are four instances at that moment of addition. The control is disabled for all instances when the Maximum Quantity of the associated instantiable model node has been reached.

How You Modify User Interfaces

After you generate a UI for a model, you can modify it to better suit the needs of your application.

On the Overview tab of a UI’s details you can see which UI template map it uses, and set its applicability parameters. On the Design tab, a WYSIWYG editor enables you to view and manipulate the elements in a UI.

To modify a UI, you can:
- Change the location of generated UI items
- Add model nodes to the UI
- Add basic UI elements that enhance the appearance of the UI
- Control the visibility of UI elements

Using the WYSIWYG Editor

The Design tab provides the Pages pane for controlling the pages in the UI, the Resource pane for adding model Structure nodes or basic UI elements, and the WYSIWYG editor region where you can interactively see the results of your modifications for the page selected in the Pages pane. The labels of the model-related items in the WYSIWYG region reflect the Description values for the nodes in the model itself.

Each UI item has a hidden edit control bar, indicated by a chevron icon. Click the chevron to select a UI item for editing and open the edit control bar.

- Click the pencil icon to edit the properties of the page item.
- Click the X icon to delete the page item from the UI page.
- Click the arrow icons to move the item up or down on the page.

The properties of UI items vary, depending on the type of the item. Commonly-used properties include:
- The internal **Name** of the item, which isn’t displayed at run time.
- The **Associated Model Node** that the UI item represents. The name is read-only. Click the information icon for details about the node.
- The **Template** option for selecting a control template for the UI item.
- The **Page Caption** that’s displayed for the item at run time.
• The **Run Time Conditions** option for defining a display condition on the UI item.

## Changing The Location Of Generated UI Items

The default location of page items in a UI is determined by the UI template map you selected when you created the UI. You can change the default location.

You can make the following changes in the location of page items:

- Move the items up or down on the page by clicking the arrow icons on the edit control bar.
- You can’t cut and paste page items to move them. Instead, delete the item from its original page and add it to a different page.
- In a Step By Step or Dynamic Tree UI, you can add new pages from the Pages pane, then add new nodes to the page from the Structure pane. When you select the node and click the **Add as Page Item** action, the new page item is added just below the item that’s currently selected in the WYSIWYG pane.
- To change the order of UI pages, select the page in the Pages pane, then click the arrow icons on the toolbar.
- To delete UI pages, select the page in the Pages pane, then click the X icon on the toolbar. First be sure that the page doesn’t contain any page items that you want to retain in your UI, though you can add them back elsewhere later.
- To change the generated title for a UI page, select the page with its edit control, or in the Pages pane, and change the **Page Caption** in the Edit Page dialog box.

## Adding Model Nodes to the UI

You can modify the set of UI pages and UI page items that represent your configurator model, which are created when you create a generated UI.

In a generated UI, UI items reflect item-based model structure:

- UI pages are generated for each item-based option class that’s an immediate child of the model root. Since supplemental-structure features must be created on the root node of model structure, those features are generated on the root page of the UI, but you can place them on other pages instead.
- UI page items are generated for each item-based standard item or supplemental-structure feature.
- UI control templates provide controls in the UI for end-user interaction with model nodes.

You can’t move a page item from one page to another in a single operation. Instead, you delete it from its original page and add it to the other page. In the case of page items that are model nodes, you must add them from the Structure tab in the Resources pane. To add a page element for a model node:

1. Select the location to add the new page item. The page item will be added below the currently selected page element. (If the currently selected UI element is a layout region, then the new page item is added as the last item in the layout region.) Consider the size and location of the page element when selecting it, and the effect on the placement of the added page item.

   You select a page element by clicking the chevron icon for its edit control. A contextual label tells you what kind of page element is associated with the edit control.

2. In the model structure tree in the Structure tab in the Resources pane, select the node that you want to add to the page. You may want to view the tree by Description, to ease locating the desired model node.

3. From the context menu, or the **Add to Page** menu, select an available action to add the node to the page.
   - **Add as Page Item** adds the node as an page item, which enables you to select a different control template and display condition.
Add as Header Region adds the node as a header region, which displays the node name as a section header, and enables you to set the control template and display condition for the region. You may want to add the node as a page item under the section header, or add other nodes or UI elements.

Add as Layout Region adds the node as a layout region, which enables you to set the control template and display condition for the region. You may want to choose a form or stack layout for the region.

You can't change the display name of the node, or its children, because those names reflect the Description text of the nodes in the model itself.

**Note:** You can designate those nodes in your model that must have a value or selection before the autocompletion process can run (which is triggered by clicking Finish or Finish and Review at run time). If you select the Prerequisite for autocomplete check box on the UI Presentation tab of the node details, in the Structure view of the Edit Configurator Model page, then an asterisk appears next to the node in the UI when the required value is missing. This indicator appears in any UI, and isn't modifiable as part of a UI, since it's part of the structure and rules of the model, rather than its UI.

## Adding Basic UI Elements

In addition to model-dependent elements that are automatically inserted into a generated UI, you can add basic UI elements that are independent of model structure to enhance the appearance or usability of the UI for end users.

To add a page element for a model node:

1. Select the location to add the new UI element. The element will be added below the currently selected page element.
2. From the list in the UI Elements tab in the Resources pane, select the UI element that you want to add to the page.
3. From the context menu, or the UI Elements toolbar, click Add to Page.
4. In the Add dialog for the added element, enter the desired properties in the Contents group.

   - For a **Text** element, enter:
     - **Text:** Text to be displayed.
     - **Inline Style:** CSS expression. Example:
       
       ```
       font-family:Arial,sans-serif;font-weight:bold;font-style:italic;color:#cc33cc;font-size:24pt
       ```
     - **Style Class** CSS class selector. The style sheet must be accessible to the UI at run time. Example:
       ```
       background-yellow
       ```

   - For an **Image** element, enter:
     - **Image:** Image file, to be uploaded to Configurator environment.
     - **Alt Text:** Provides accessibility text for images.
     - **Inline Style:** CSS expression. Example:
       ```
       position:relative;top:10px;left:-200px
       ```

   - For a **Spacer** element, enter:
     - **Width:** Width of the spacer, in pixels. Do not add px to the number.
     - **Height:** Height of the spacer, in pixels. Do not add px to the number.
     - **Inline Style:** CSS expression. Example:
5. You can change the default associated model node for a basic UI element by clicking the Search and Select control on the Associated Model Node field, and selecting a node that's accessible relative to the UI page. Changing the associated model node enables you to affect basic UI elements based on values or events in other parts of the model. Examples include:
   - You can use the value entered for a text feature during a configuration session as part of the value for the URL parameter of an IFrame basic element. Set the text feature as the associated model node of the IFrame, then use the UI expression #{amn.value} to access the value of that node. Example:
     https://www.mysite.com:1000/perform?taskID=#{amn.value}
   - You can use selection of a value for an option class or option feature during a configuration session to conditionally control the whether an Image basic element appears or not. Set the option feature as the associated model node of the Image, then set the display condition for the Image to be TRUE when the option feature is bound to a value by the occurrence of a selection. Example:
     - Object: Associated model node
     - Attribute: IsBound
     - Value: Equals TRUE

Limitations on Modifying UIs

There are limitations to how you can modify a UI.

- You can't change the UI template map that you selected when creating the UI. Therefore you can't change the navigation style for the UI.
- You can't change the Associated Model Node for a UI item that directly represents a model node. Therefore you can't change the node represented by a particular UI item. But you can change the UI control template selected for the UI item, thus changing its appearance.
- If the model has child referenced models, you can't explicitly specify which of child model's UIs will be used at run time. The child model's UI will be selected based on its applicability parameters.
- You can't modify the configuration summary page.

Related Topics

- CORS
How You Modify UI Elements

After you generate a UI for a model, you can modify individual UI elements in it, to better suit the needs of your application.

Controlling the Visibility of UI Elements

There are several ways to control which elements of a UI appear, and under which conditions.

The most flexible way to control the visibility of UI elements is by using display conditions, which apply to all elements in a UI except pages.

To set a display condition:

1. Select an element of the UI, and click its edit control to open its properties dialog box.
2. Under Run Time Conditions, change the value of the Displayed control from Always to Conditionally.
3. Now the Condition group of options is provided. You define the display condition for the element by selecting options which specify that some attribute of some object in the model has a specified value. When the specified attribute has that value, then the UI element you’re defining is displayed at run time, or when you test the model.
4. Select the Object whose value will trigger the display condition. The default object is the associated model node for the UI element itself. If you select Other model node, then a search control is provided for selecting that node. If you select Configuration session, then the Attribute option lists attributes of the configuration session as a whole at run time, rather than attributes of a model node.
5. Select the Attribute whose value will determine whether the display condition is triggered.
6. Select whether the run time Value for the chosen attribute Equals or Does not equal the value that you select in the last option of the display condition.

As an example, assume that you want the UI to display the message You picked red. when the end user selects the option Red for an option feature named Color. You would:

1. Add a Text element to the UI, positioned near the Color feature.
2. Edit the properties of the Text element and enter You picked red. in the Text field.
3. Select Conditionally for Displayed.
4. Select the object Other model node, then search for and select the supplemental node Red under the option feature Color.
5. Choose the attribute SelectionState.
6. Select the operator Equals and the attribute value Selected.
7. Click Test Model. Navigate to the option feature Color, and select Red. The message You picked red. appears. The same message would appear to the end user at run time.

There are also other ways of preventing model nodes from appearing in a UI:

- You can suppress a particular structure node for a model from appearing in any new UIs by deselecting the Display in user interface check box on the UI Presentation tab of the structure details pane for that node. This won't affect UIs that you have generated before you change this setting, but will affect all UIs that you generate after changing it.
- You can minimize the number of choices presented to the end user in a UI at run time by selecting the Hide excluded items in run time Configurator check box on the details overview section of the User Interfaces tab when editing the model. This setting is independent of model structure, and only affects the particular UI in which you set it. At run time, if options or items are excluded from selection by configurator rules, then they're not displayed in the UI.
Modifying UI Element Captions

You can use the UI expression language to override the default item display name of a given UI item, replacing it with a combination of static text and all associated model node attributes, with the exception of DisplayName.

The following table provides an example of the effect of using the UI expression language.

<table>
<thead>
<tr>
<th>Element of UI caption</th>
<th>Example</th>
</tr>
</thead>
</table>
| Associated model node in model structure                  | Name: CARWHEELS
|                                                           | Description: Wheels                                                   |
| Default display name in UI                               | Wheels                                                                 |
| UI expression                                             | Item Name: #{amn.name} - Item Description: #{amn.description}          |
| Modified display name at run time                         | Item Name: CARWHEELS - Item Description: Wheels                        |

To override the default display name of a UI element:

1. Select the UI element.
   
   You can modify the following UI elements:
   
   - Page Caption of a Page element
   - Header Region Caption of a Header Region element
   - Page Item Caption of a Page Item element
   - List Item Caption of a List Item element, which can be part of a Page Item

   **Note:** When you override the default display name of UI caption elements, then the node property DisplayName isn’t available for the associated model node.

2. Click the Edit control for UI element.
3. In the Edit dialog box, enter a UI expression for the caption of the UI element.
4. Click OK in the Edit dialog, save the UI, and click **Test Model**.
5. In the test UI, the caption appears with the overriding expression text.

Using UI Facets

You can add basic UI text elements in the UI facets of the header of certain UI page items, to augment the information presented to the end user at run time.

A UI facet is an area in the UI item into which you can place additional UI items related to the associated model node for the UI item, or reachable from the associated model node. The following configurator model node types have UI templates that include UI facets:

- Option classes
Templates with facets are available for the following UI page items:

- Boolean Feature with Facets
- Check Box group with Facets
- Check Box group with Quantity and Facets
- Choice List with Facets
- Item Instance Management Table with Facets
- Radio Button Group with Facets
- Radio Button Group with Quantity and Facets

A template with facets contains layout regions arranged in the following default relationship:

- **Header**: template with facets
- **Messages facet**: Compact Stack Layout template containing:
  - Basic UI text elements (in vertical sequence)
- **Information facet**: Flow Layout template containing:
  - Basic UI text elements (in horizontal sequence)
  - Items or options of the associated model node

The layout regions can be changed if they’re not suited for the desired behavior.

Example showing two text elements stacked in the messages facet and four text elements in a flow in the information facet:

```
System Memory
Memory mixing requires firmware version: XXYY123
Your selection of DIMMs exceeds the maximum of: 64
Memory (GB) Included: 0 | Minimum Required: 64 | Maximum Allowed: 2048 | Total Configured: 2608
[0] A8V234 Four 8 GB DIMM
[0] A8V567 Four 16 GB DIMM
[0] A8V890 Four 32 GB DIMM
[18] A8V999 Four 64 GB DIMM
```

To use UI facets for a page item:

1. Select and edit a page item for an associated model node that can use facets.
2. Apply one of the templates that includes facets. After you click OK, the facets are added to the page item.
3. Select the first layout region under the header, which is the messages facet. This region’s template is Compact Stack Layout. Don’t change the template.
4. From the list in the UI Elements tab in the Resources pane, select the **Text** element, then, from the context menu or the UI Elements toolbar, click **Add to Page**.
5. Edit the basic Text element that you inserted into the facet. Add static text in the **Text** field to provide messages about configuring the associated model node to the end user.
6. You can use the UI expression language to dynamically display information about the associated model node, or nodes reachable from the associated model node. You must use expression language attributes that correspond to the node types for which you can use UI templates with facets.

For example, assume that the associated model node is an option feature with Minimum Selections of 2 and Maximum Selections of 5, and you enter the following three expressions in Text field of three basic text elements in the messages facet:

```
Number of options you selected: #{amn.selectedCount}
Maximum number of selections allowed: #{amn.maxSelected}
Minimum number of options selected?: #{ amn.minSelectionSatisfied}
```
When you test the model and select one option, the following text is displayed in the facets:

- Number of options you selected: 1
- Maximum number of selections allowed: 5
- Minimum number of options selected?: false

7. Select the second layout region under the header, which is the information facet. This region’s template is Flow Layout. Don’t change the template.

8. Repeat the steps described for the messages facet, to add static or dynamic text in basic Text elements in the information facet.

9. Save your changes to the UI, and click **Test Model**.

**Caution:** If you add UI items to facets, then later change the page item’s template to a template without facets, then the facets will be removed, and all the UI items that you added to the facets will be lost.

**Related Topics**
- Configurator Model Node Properties

---

**The UI Expression Language**

You can use the UI expression language to dynamically create values for text parameters of basic UI elements based on the associated model node attributes of any basic element UI item.

**UI Expression Language Syntax**

Expressions in the UI expression language consist of a reserved keyword, which refers to an associated model node, qualified with a name that refers to an attribute of the model node.

An example of using the expression language is:

```
The node named #{amn.name} has a maximum quantity of #{amn.maxQuantity}.
```

- All expressions start with `{` and end with `}`.
- The keyword `amn` establishes a reference to the associated model node for a UI basic element.
- Expressions are case-sensitive, including the keyword `amn`.
- The keyword `amn` is always qualified by a reference to an attribute of the associated model node, following this form:
  
  `#{amn.referenceName}`

- References must be to a valid attribute of the associated model node. Valid attributes are enumerated in the user assistance for configurator model node properties. References to attributes that don't apply to the associated model node generate errors when attempt to use them in expressions.
- UI expression attribute references are formed by putting the initial letter of the attribute name into lower case. The prefixes `Is` and `Has` are omitted from expression references.

The following table summarizes attribute references in the UI expression language.
UI Expression Language Reference | Refers To | Example Expression
--- | --- | ---
name | The name of the associated model node. | #{amn.name}
description | The description of the associated model node. | #{amn.description}
quantity | The quantity of the associated model node. | #{amn.quantity}
boundQuantity | Attributes named like IsBoundQuantity | #{amn.boundQuantity}
hasChildren | Attributes named like HasChildren | #{amn.hasChildren}
value | The value of a transactional item attribute, if the associated model node is an attribute under the root model node. | #{amn.value}
suppAttrs[’<suppAttrName>’] | The value of the named supplemental attribute. | #{amn.suppAttrs[’color’]}
userAttrs[’<UDAttrGroup>.<UDAttrName>’] | The value of the named user-defined attribute. | #{amn.userAttrs[’PhysicalAttributes.Color’]}

Restrictions on the UI Expression Language
The following restrictions apply to UI expression language expressions.

- Expressions are case-sensitive.
- Expression attribute references must be to attributes that are valid for the associated model node. This validation is also applied if you change the associated model node for an element.
- The expression language can be used with all model node system, user and supplemental attributes, but only for the text parameters of basic UI elements. The basic UI elements are: Text, Image, Spacer, and IFrame. The text parameters are: Text, Inline Style, Style Class and URL.
- References to elements of a collection or list aren’t valid. Example of invalid reference: #{amn.children[0].name}.
- References to transactional item attributes are only valid when the associated model node of a basic UI element is a transactional item attribute under the root node of the model. Valid reference expression: #{amn.value}.
- If an attribute value of the associated model node isn’t available at run time, the displayed value is blank. If an expression can’t be evaluate at run time because an attribute value isn’t available, the displayed value is N/A.

Uses for the UI Expression Language
Uses for the expression language include the following:

- Creating a Text basic UI element in a user interface page to dynamically display the minimum and maximum values of its associated model node. Example:
• Creating an IFrame basic UI element where the IFrame's URL is dynamically augmented to provide query parameters. Example:
  https://www.mysite.com:1000/perform?taskID=#{amn.value}

• Storing a central set of Cascading Style Sheets (CSS) in a model node's supplemental attribute and using the expression language to generically refer to the CSS from a basic UI element's Inline Style attribute. Example:
  #{amn.suppAttr['InlineStyle']}

Related Topics
  • Configurator Model Node Properties

How You Use Images for Selections

You can represent items with images in place of their names, in the run time UI, enabling the end user to make simpler and more intuitive selections.

Adding Images to Items

To enable selectable item images, you must first provide the images to the desired items. The images should reflect the possible selection states of the item: available for selection, selected, or excluded.

You can add selectable images to the following types of model nodes:

• Standard items of an item-based options class
• Options of a supplemental option feature

To add selectable images to an item:

1. In a workspace, open the model draft containing the item.
2. On the Structure tab of the Edit Configurator Model page, select the node to which you want to add selectable images.
3. Select the UI Presentation tab of the Details region
4. Under Item Selection Images, there are controls for adding the images for the Primary (available but unselected), Selected, and Excluded states of the node.
5. Click the icon in the Primary field to add an image.

You can't add Selected or Excluded images without having a Primary image.

6. Use the Add Image dialog box to locate and add the Primary image file for the node.
7. Use the same procedure to add images for the Selected and Excluded images.
8. Repeat the addition process for the other nodes of the option class or option feature, where desired.

You can remove the Selected or Excluded images as desired. If you remove the Primary image, then the other images are automatically removed. You can change an image at any time.
Adding Selectable Images in UIs

To display selectable images to users at run time, you must add them to a user interface.

To add selectable images to a UI:

1. On the User Interfaces tab of the Edit Configurator Model page, create a new UI, or select the existing UI, on which you want to display selectable images.
2. In the WYSIWYG editor on the Design subtab, select the edit control bar for the page item representing the option class or option feature that you provided selectable images for.
3. Click the control for editing properties of the page item.
4. Open the list in the Template field, and search for one of these templates:
   - Selectable Image Group
   - Selectable Image Group with Header
5. In the Edit Page Item dialog box for the template that you selected, set the values for desired properties in the Contents group.
   - Images Per Row is the number of item images displayed at run time in a horizontal row on the UI page.
   - Inline Style is an optional CSS style expression applied at run time to the entire Selectable Image Group template.
   - Option Inline Style is an optional CSS style expression applied at run time to each selectable image.
6. Save your model changes and click Test Model.
7. In the test UI, the images displayed for the options of the option class or option feature depend on the selection state of the options:
   - The Primary image is displayed if the option is available, and not selected or excluded.
   - The Selected image is displayed if the option is selected by a rule or user selection.
   - The Excluded image is displayed if the option is excluded by a rule.
8. If any images are missing from the Item Selection Images of the Structure view, then the following substitutions are made at run time:
   - If the Primary image for an item is missing, then a graphic placeholder icon is displayed.
   - If the Selected image is missing, and the item is selected, then the Primary image is displayed, enclosed by a rectangle.
   - If the Excluded image is missing, and the item is excluded, then the Primary image is displayed, in dimmed shading.
9. If your UI uses the Selectable Image Group with Header template, then the Description of the option class or option feature and the Description of the selected option are displayed together as a header for the group of option images.

How You Visualize Configurations

You can define user interface elements that enable end users to visualize how an object changes visibly in reaction to actions they take during a configuration session.

To provide visualization of objects in a configurator model:

- Design the visualization model
• Add an IFrame element and enable it to receive model node changes
• Communicate model node changes to the external visualization tool

Design the Visualization Model
Visualization requires that you provide a model in some external 2D or 3D visualization tool. This visualization model should correspond to all or part of the configurator model for the product that your end users are configuring.

Since many parts of a configurator model can change while users are configuring a product, it's important to determine the scope of the model node changes needed to support the visualization. If a visualization only supports a component of a model, then the scope of the model node changes should be reduced to that component only and not the entire model.

Add an IFrame Element and Enable It To Receive Model Node Changes
Visualization takes place within an inline frame of a user interface. You create the inline frame with an IFrame basic UI element. By default, an inline frame displays an accessible internal or external web site. To visualize configured objects, you enable the IFrame element to receive model node changes.

Consult the related topic about modifying user interfaces for more details on adding an IFrame basic UI element. For adding visualization, there are a few special things to do:

1. Select the location to add the IFrame element. It will be added below the currently selected page element.
2. From the list in the UI Elements tab in the Resources pane, select the IFrame element and click Add to Page.
3. In the Add IFrame dialog box, select the associated model node for the IFrame. Choose carefully, because the scope of the node controls which model node changes are passed to the IFrame.
4. Select Receives model node changes.
5. Select the scope from which changes to the model should be communicated to the IFrame: Associated Model Node and Subtree or Associated Model Node.
6. In the URL field of the Contents group, enter the URL and parameters for the external visualization site that you're integrating with. For example:

   https://myserver.com/viewer.html?scs=model_data/mountainbike_blue.scs

   Remember that if the URL is outside the current domain, then you must use Cross-Origin Resource Sharing (CORS).
7. Enter other desired properties in the Contents group.

   **Tip:** If a user interface has multiple UI pages, you can add an IFrame to each page, so that the end user can keep the visualized object in view throughout the configuration session. Depending on your use case, you could determine whether to display different components of your model on different UI pages, or the entire model on every page.

Model Node Changes That Are Passed to the IFrame
When end users select items or enter values for items in the IFrame’s associated model node or subtree, Configurator passes the changes in the values of certain attributes to the IFrame, as JSON payloads of standard name-value pairs. When these model node change payloads are received by the IFrame, the values are passed to the external visualization tool identified by the IFrame’s URL. Be aware that no information is passed back from the IFrame to the configurator model.

These static attributes are passed to the IFrame for every associated model node or subtree node:

• NodeType
- DisplayNamePath
- HasTransactionalAttributesFlag (only for node types imported from the Product Information Management work area)

It's useful to know which attributes are passed to the IFrame, and from there to the external visualization tool, because these attributes can be used in the JavaScript code that matches configurator model nodes with visualization model nodes, as shown in the code fragment provided here.

The NodeType determines which dynamic attributes are passed to the IFrame:

<table>
<thead>
<tr>
<th>NodeType</th>
<th>SelectionState</th>
<th>Quantity</th>
<th>InstanceCount</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>OPTION_CLASS_ITEM</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>STANDARD_ITEM</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>MODEL_ITEM</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>ROOT_MODEL_ITEM</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>COMPONENT_PORT</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
</tr>
</tbody>
</table>

Communicate Model Node Changes to the External Visualization Tool

Configurator communicates model node changes in standard JSON payloads of name-value pairs to the IFrame, internally using the JavaScript method `window.postMessage()`. These payload messages must be interpreted by the visualization tool to properly display model node changes in the visualization IFrame.

For example, assume that a configurator model of a bicycle includes an option class and standard items with these display names:

- Mechanical Options.Brakes
- Basic Alloy Bicycle Brake Set
- Dual Compound Side Pull Bicycle Brake Set
- Delux Cantilever Brake System

In the HTML file used by the visualization tool (which you specified in the URL field of the IFrame) you might write the following JavaScript statements to match the display names of the configurator model nodes with the corresponding nodes of a visualization model that you have built. The sample code below sets the color of a node of the visualization model when the matching configurator model node is selected by the end user in a configuration session.

```javascript
if (displayNamePath.startsWith("Mechanical Options.Brakes")) {
    if (selectionState === "SELECTED") {
        if (displayNamePath.includes("Basic Alloy Bicycle Brake Set")) {
            var color = new Communicator.Color('255', '255', '255');
            model.setNodesFaceColor([18, 19, 20, 21], color);
        } else if (displayNamePath.includes("Dual Compound Side Pull Bicycle Brake Set")) {
            var color = new Communicator.Color('100', '100', '0');
            model.setNodesFaceColor([18, 19, 20, 21], color);
        } else if (displayNamePath.includes("Delux Cantilever Brake System")) {
            var color = new Communicator.Color('200', '100', '100');
            model.setNodesFaceColor([18, 19, 20, 21], color);
        }
    }
}
```
model.setNodesFaceColor([18, 19, 20, 21], color);
}
else if (displayNamePath === "Mechanical Options.Brakes" && selectionState === "SELECTABLE") {
  model.unsetNodesFaceColor([18, 19, 20, 21]);
}
...

Related Topics
  • Item-Based Model Structure
  • CORS

FAQ for Model User Interfaces

How can I rename a page caption?

By default, a UI page has a heading derived from the Description of the associated model node. To change the heading displayed at run time, edit the page, on the Design subtab of the User Interfaces tab for the model. Select the page, in the Pages pane, and click the Edit control in the toolbar. In the Edit Page dialog box, change the default text that was generated for the heading, in the Page Caption field. The new page caption will be displayed at run time, and when you test the model.

Keep in mind that UI pages are not tightly connected to the model. You can add, delete, and reorder pages without affecting the model structure. Similarly, you are allowed to change the page captions that are generated from node names in the model, because you are not changing the model structure. However, UI page items are tightly connected to the model, and you cannot change their page captions in by editing a UI. To change the captions of item-based nodes, you would have to change their Description values in the Product Information Management work area and refresh the snapshots that include them. To change the captions of supplemental structure nodes, you can change their Name values in the structure details of the model.
7 Connectors

Overview of Connectors

Connectors are a type of supplemental structure that you can add to a configurator model.

- Connectors provide the basis for making connections from a model to related items outside that model. Use a connector when the configuration of a product requires you to include an item that isn't part of the product's structure.
- For example, when configuring the model for a computer, you might also want to add a warranty as part of the order, although warranty items aren't part of the computer model. A connector provides the means for including the warranty in the configuration.
- You define a connector in a model's structure, with a target that's an item class. Before defining the connector, you must import the items in that item class.
- The connections are created at runtime, as part of the configuration of a model. The items associated with the target item class are connectable to the model, through the connector.

When To Use Connectors

Products are often configured from items in a bill of materials that are related by specific rules to create a defined, limited model structure. But if you need to configure products that depend on items that aren't part of a single model's structure, you might want to use connectors.

Here are examples of models for which you might use connectors to include separate but related items in a configuration of the model:

<table>
<thead>
<tr>
<th>Configurator Model</th>
<th>Related Item Connected to Model</th>
<th>Conditions for Connection</th>
</tr>
</thead>
<tbody>
<tr>
<td>Network server</td>
<td>Warranties</td>
<td>Exclude warranties for 12 months. Allow only warranties for 18 or 24 months.</td>
</tr>
<tr>
<td>Network server</td>
<td>On-site installation service</td>
<td>Automatically added as a related item on a quote or order.</td>
</tr>
<tr>
<td>Desktop computer</td>
<td>Cables</td>
<td>No restrictions.</td>
</tr>
<tr>
<td>Car</td>
<td>Trailer</td>
<td>Can optionally add only one trailer to a car.</td>
</tr>
<tr>
<td>Car</td>
<td>Warranties</td>
<td>Can add up to four warranties.</td>
</tr>
</tbody>
</table>
Here are some user stories where connectors could be the best solution:

- **User story 1:** Create a quote or order for a configured item with separate related items.

  You are configuring a network server and have selected the required components of the configuration. Certain components that you have selected exclude warranties of 12 months, and only 18 or 24-month warranties are allowed. You should be able to select the available warranties from within the Configurator user interface. On completing the configuration, you should see the network server and the warranties that you selected as related quote or order lines. Also note that these warranties are not modeled in the Product Development work area as part of the configured item's component structure.

- **User story 2:** Modify a quote or order for a configured item with separate related items.

  You have an existing quote or order for which you're re-configuring a network server and have changed the selected components of the configuration. Certain components that you have selected now require an on-site installation service. The on-site installation service should automatically be added as a related item to the quote or order and you should be able to select additional available services from within the Configurator user interface. On completing the configuration, you should see the modified network server and the new on-site installation service as a separate but related line in the quote or order. Also note that these services are not modeled in the Product Development work area as part of the configured item's component structure.

- **User story 3:** Validate a quote or order on or before submission

  You have an existing quote or order which includes a configured network server with an on-site installation service line item and you have removed the on-site installation service line item. Certain components that you had previously selected for the network server require an on-site installation service. Before you submit the quote or order, you select an action to verify it one last time. You will then be notified that an on-site installation service is required and it will be added back to the quote or order as a separate line item. (Note that this validation might also occur automatically when the quote or order is submitted.)

### How You Create Connectors

You create connectors in a way that's mostly similar to creating other supplemental structure. But before you create a connector you must import the item class and item that a connector depends on.

### Import Items Associated with an Item Class

Because a connector has an item class as a target, you must import the items from the intended target item class before you can create the connector to that target. If your item class includes an item class hierarchy for your related items, then you can import all the items in the hierarchy as well, to enable connections to them.

To import items associated with an item class:

1. Identify the item class associated with the items that you want to connect to your model.
2. On the Manage Snapshots page of the Configurator Models work area, select **Import Items Associated with an Item Class** from the Actions menu.
3. In the **Import Item Snapshots** dialog box, select the **Organization** that the items belong to and the **Item Class** of the related items that you want to connect to.

**Note:** To be connectable, an item must have the attribute **Order Management Indivisible** set to **Yes**.
4. By default, only the items in the selected item class itself will be imported. If you also want to connect to all of the items associated with all of the child item classes of the selected item class, select **Include items associated with child item classes**. You can also select a particular child item class to import by itself.

5. Click **Submit**. A scheduled process is submitted. Make note of the scheduled process request ID.

6. In the Scheduled Processes work area, examine the log file for the **Import Items Associated with an Item Class** process. The log file contains the names of the items imported from the item class you chose. If they haven't been previously imported, then the process imports:
   - The specified item class itself
   - All standard items that are associated with the specified item class
   - If **Include items associated with child item classes** is selected, then all of the child item classes, and all of their associated items

After the import, you can query the imported items and item classes on the Manage Snapshots page.

   **Note:** Each import is limited to a batch of 5000 item snapshots. If that batch size is exceeded, the import process status is set to Warning. Continue to run the **Import Items Associated with an Item Class** action, with the same parameters, until all your items are imported. Check the list of imported items in the log file of the scheduled process to determine when you're finished. A list less with than 5000 items means there are no more items left to import.

7. You can refresh the item class snapshots in the usual way, to reflect item changes to existing items, or addition of new items associated to the item class, in the Product Information Management work area.

Create Connectors on Models

You add connectors to model structure, similar to the way that you add other supplemental structure. To enable the addition of related items to a mode at runtime, you add connectors to model structure.

1. If you haven't already done so, import all the items associated with the intended target item class.
2. In a workspace that you have locked, open a model for editing.
3. On the Edit Configurator Model page, select the root node of the model. Select **Create > Connector**. You can only create connectors on the root node.
4. In the Create Connector dialog box, you select a value for the **Target** of the specified **Target Type**. Search for and select the target item class. The search list contains all the item class snapshots that have been imported into the Configurator Models work area.

After a connector is defined, all imported items associated with the target item class become items that are connectable to the model at runtime.

   **Caution:** Targeting an item class or item class hierarchy containing a very large number of items can result in significant impacts to performance. If you target the Root Item Class, you can't select **Include items associated with child item classes**.

5. Enter values for **Minimum Connections** and **Maximum Connections**, to determine the number of instances of this connection that may be added at runtime. These fields represent the minimum and maximum number of actual related items which can be added to a configuration.

6. Enter values for **Minimum Quantity** and **Maximum Quantity**, to determine the sum quantity of connected items that may be added at runtime through this connector. These fields represent the minimum and maximum sum quantity of all the related items added at runtime to a configuration.
7. After you create a connector, you can modify the minimum and maximum connections and quantities, but you can’t change the target.

8. There’s currently no representation of connectors in a user interface. The UI Presentation setting is reserved for future use.

9. After you create the connector, it appears in the model tree, at root level. The connections that can now be made in a runtime configuration session are between related items and the model as a whole.

Related Topics

- Snapshots
- How You Import Items
- How You Refresh Snapshots
- Supplemental Model Structure

How You Use Connectors

You use the connectors that you created in your model structure to make connections, during a runtime configuration session, with items outside the structure of your model.

In this example, users who are ordering a car may want to also order a trailer to use with the car.

Here are the entities used in this example, and the roles they play in adding a trailer to an order for a car:

<table>
<thead>
<tr>
<th>Entity</th>
<th>Role</th>
</tr>
</thead>
<tbody>
<tr>
<td>Configurator model</td>
<td>The car being ordered. The example model is:</td>
</tr>
<tr>
<td></td>
<td>• CAR4DRSDN - 4 Door Sedan</td>
</tr>
<tr>
<td>Item class</td>
<td>The class of orderable trailers, created in the Product Information Management work area. The example item class is:</td>
</tr>
<tr>
<td></td>
<td>• Trailers - Trailers</td>
</tr>
<tr>
<td>Standard items</td>
<td>The set of orderable trailers, created as items of the Trailers item class, in the Product Information Management work area. The example set of standard items is:</td>
</tr>
<tr>
<td></td>
<td>• 2WHL_BSTL_TRLR_CART - 2 Wheel Black Steel Trailer Cart</td>
</tr>
<tr>
<td></td>
<td>• 4WHL_BSTL_TRLR_CART - 4 Wheel Black Steel Trailer Cart</td>
</tr>
<tr>
<td></td>
<td>• CO_TRLR - Carry-On Trailer</td>
</tr>
<tr>
<td></td>
<td>• HDHM_CARGO_TRLR - Heavy Duty Hitch Mount Cargo Carrier</td>
</tr>
<tr>
<td></td>
<td>• WM_UTIL_TRLR - Wire Mesh Utility Trailer with Ramp Gate</td>
</tr>
<tr>
<td>Snapshots</td>
<td>The snapshots are the copies of the item class and its standard items, with the same names, imported into the Configurator Models work area.</td>
</tr>
</tbody>
</table>
The connector provides the basis for connecting the car model to the standard items in the Trailers item class. The example connector is:

- Trailer Connector

In the absence of UI templates for connected items, the option feature and its options represent the imported item class and its items.

- The example option feature has the same name as the item class: Trailers
- The options have the same names as the items, such as: HDHM_CARGO_TRLR

The text feature holds the name of the option feature's option that the user selects at runtime, and represents the connected item. The example text feature is:

- Selected Trailer Item

Sets the value of the text feature to be the name of the option feature option (representing the connected item) selected at runtime. The example statement rule is:

- Updated Selected Trailer Item

Adds a connectable item to the configuration, using the name of the selected option feature option to connect to the corresponding standard item. The example extension rule is:

- Add Trailer

The steps to put these elements of the example together are:

1. Import the snapshots.
2. Create the supplemental structure.
3. Create the statement rule.
4. Create the connector.
5. Create the extension rule.
6. Test the model.

Import the Snapshots

On the Manage Snapshots page of the Configurator Models work area, select **Import Items Associated with an Item Class** from the Actions menu.

- **Note:** The data shown in this example is hypothetical, for demonstration purposes.

- Search for and select the item class: Trailers.
- Don't select the **Include items associated with child item classes** option, so that the import contains only the items associated with the Trailers item class, and not from its child item classes.
- The resulting imported standard items include:
  - HDHM_CARGO_TRLR - Heavy Duty Hitch Mount Cargo Carrier
  - WM_UTIL_TRLR - Wire Mesh Utility Trailer with Ramp Gate
Create the Supplemental Structure

Since there’s currently no pre-defined user interface support for connectors, you can represent the Trailer item class as an option feature, and the associated items as its options. You can modify the default user interface for the options as desired, for instance by using selectable images for the options. At runtime, the end user can select these options in the user interface and an extension rule adds the actual related items to the running summary.

In the CAR4DRSDN model, create an option feature and its options, with these settings:

- Name: Trailers. For clarity, this is the same name as the item class.
- Minimum Selections: 0
- Maximum Selections: 1
- Create an option that matches the name of each of the standard items in the Trailers item class, such as HDHM_CARGO_TRLR. You can use ADFdi integration to create the options, if there are a large number of them.

In the CAR4DRSDN model, create a text feature with these settings:

- Name: Selected Trailer Item
- Maximum Length: 100 (or whatever is long enough to accommodate the names of your standard items).

Create the Statement Rule

In the CAR4DRSDN model, create a statement rule, with these settings:

- Name: Updated Selected Trailer Item
- Rule Class: Constraint
- In the Rule Text, create a series of REQUIRES statements, one for each option of the Trailers option feature.
  - On the left-hand side of the rule, insert an option of the Trailers option feature.
  - On the right-hand side of the rule, insert the text feature, Selected Trailer Item, and assign the name of the Trailers option as its value.

The CDL statements in the Rule Text field should look like this:

```
'CAR4DRSDN'.'Trailers'.'HDHM_CARGO_TRLR' REQUIRES 'CAR4DRSDN'.'Selected Trailer Item' = "HDHM_CARGO_TRLR"
'CAR4DRSDN'.'Trailers'.'CO_TRLR' REQUIRES 'CAR4DRSDN'.'Selected Trailer Item' = "CO_TRLR"
'CAR4DRSDN'.'Trailers'.'WM_UTIL_TRLR' REQUIRES 'CAR4DRSDN'.'Selected Trailer Item' = "WM_UTIL_TRLR"
'CAR4DRSDN'.'Trailers'.'2WHL_BSTL_TRLR_CRT' REQUIRES 'CAR4DRSDN'.'Selected Trailer Item' = "2WHL_BSTL_TRLR_CRT"
'CAR4DRSDN'.'Trailers'.'4WHL_BSTL_TRLR_CRT' REQUIRES 'CAR4DRSDN'.'Selected Trailer Item' = "4WHL_BSTL_TRLR_CRT"
```

When the end user at runtime selects an option of the Trailers option feature (such as 'HDHM_CARGO_TRLR'), the rule sets the value of the text feature Selected Trailer Item to the name of the trailer standard item (which would be "HDHM_CARGO_TRLR"). This name is used to create a connector instance that specifies that standard item.

The rule code shown in the example is designed primarily for clarity.
Create the Connector
In the CAR4DRSDN model, create a connector under the root node, with these settings:

<table>
<thead>
<tr>
<th>Field</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>Trailer Connector</td>
</tr>
<tr>
<td>Target Type</td>
<td>Item Class</td>
</tr>
<tr>
<td>Target</td>
<td>Trailers</td>
</tr>
<tr>
<td>Minimum Connections</td>
<td>0</td>
</tr>
<tr>
<td>Maximum Connections</td>
<td>1</td>
</tr>
<tr>
<td>Minimum Quantities</td>
<td>1</td>
</tr>
<tr>
<td>Maximum Quantities</td>
<td>1</td>
</tr>
</tbody>
</table>

The characteristics defined in the connector node govern the behavior of the items related through the connector at runtime:

- The target type governs which set of related items are connectable to the model. For a connector whose target type is item class, the set of related items is the set of items associated with that item class.
- The minimum and maximum connections govern how many different standard items from the target item class can be added to this configuration. In this example, that means how many different trailers can be added to the configuration of the car. Because you only want to allow one trailer per car, you set Maximum Connections to 1. To make the addition of a trailer to the configuration optional, you set Minimum Connections to zero.
- The minimum and maximum quantities govern the sum of all the quantities of standard items in the target item class.

In this example, that means how many total trailers can be added to the configuration of the car. Again, because you only want to allow one trailer per car, you set both Minimum Quantities and Maximum Quantities to 1.

Create the Extension Rule
In the CAR4DRSDN model, create an extension rule, with these settings:

<table>
<thead>
<tr>
<th>Field</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>Add Trailer</td>
</tr>
</tbody>
</table>
Field | Value
--- | ---
Base Node | The text feature **Selected Trailer Item**.
The statement rule sets the value of this feature to the name of the standard item that's selected at runtime.

Instantiation Scope | Instance

Rule Text | Enter the Groovy code sample that follows this table. You can reuse this code for other connectors, with the substitution of your connector name for the name in this example.

Event | postValueChange

Event Scope | Base node

Class | ScriptClass

Method | run()

---

Enter this Groovy code in the Rule Text field:

```groovy
config = cxEvent.configuration
root = config.getRootBomModel()
baseNode = cxEvent.baseNode;
// Retrieve the connector definition from the root model
connPort = root.getChildAt("Trailer Connector");
// Check to see if the connector has any connector instances
if (connPort.getInstanceCount() == 0) {
    // If not then add an instance
    instance = connPort.addInstance();
}
// Retrieve the connector instance
instance = connPort.getInstances().get(0)
// Set its quantity to 1
instance.quantity = 1;
// If the connector instance has an existing connected item defined, then remove it
instance.unsetType();
// Set the connector instance to the new connected item
instance.setType(instance.getAvailableLeafType(baseNode.getTextValue()));
```

The comments in the example code explain what it's doing, but several code statements deserve comment.

- The following statement locates the connector you created, by its name:

```groovy
connPort = root.getChildAt("Trailer Connector");
```

When adapting this example to your own scenario, replace the name of the example connector node (Trailer Connector) with your own connector node name.

- The following statement sets the type of the new connector instance, by using the name of the option corresponding to an imported standard item snapshot (in other words, to the standard item):
instance.setType(instance.getAvailableLeafType(baseNode.getTextValue()));

**Related Topics**
- Supplemental Model Structure
- How You Create Statement Rules
- How You Create Extension Rules
- Create an Extension Rule

## Connectors in Action

Connectors provide behavior that's distinct from other configurator model entities.

### Runtime Configuration Sessions

Connectors affect how you interact with models during a configuration session.

- The extension rules that you define can drive the runtime selection of related items that belong to an item class. You can implement business logic in your rules to select the items. Otherwise, you would have to add the related items to the item structure underlying your models, which would be onerous to build and maintain, considering that related items are commonly related to multiple models.

- During a configuration session, any related items that you add to the configuration through connectors and rules are added as lines in the running summary pane of the user interface. The name and quantity of each related item is displayed. User interface templates for connected items aren't available, so you can't add connected items to your UI modifications.

- When you choose to save a configuration during a session, and later restore the configuration in a new session, connected items are included. Host applications can also save and restore configurations including connected items.

- Be aware that items with decimal or fractional quantities can't be used with connectors. This is governed by the attribute **Order Management Indivisible** in the the Product Information Management work area.

### Connections and Quantities

It's helpful to understand some of the design background when you're using connectors.

- Connector nodes are definitions for what are, in effect, typed runtime instances of the connector entity. The name of the connector node becomes the name of the connector instance type. The characteristics defined in the connector node govern the behavior of that connector type's instances at runtime:

- The item class that's named as the target of the connector provides scope for validating the item being connected to. The item class name is used to provide the model with the items it can connect to at runtime, and only those items. The available items from that item class can be limited and become unavailable for selection, depending on your rules governing the availability of these items.

- The minimum and maximum connections govern how many instances of the connector type can be instantiated during the configuration of the model. In functional terms, those values govern the minimum and maximum number of different standard items from the target item class can be added to a configuration. In the example used elsewhere in this chapter, that means how many different trailers can be added to the configuration of the car.
The minimum and maximum quantities govern the sum of all the quantities of related items. In the example of the car and trailer, that means how many total trailers can be added to the configuration of the car.

Here’s an example of the relationship between connections and quantities. Assume that your model represents a family calling plan for cell phone service. A connector targets the item class for cell phone models, in which items are different phones. You would define the minimum and maximum connections and quantities to govern how the phones can be added as related items to the configuration of the plan model.

<table>
<thead>
<tr>
<th>The plan allows, per family:</th>
<th>Your connector definition specifies:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Up to 3 different phone models, from among:</td>
<td>Minimum Connections = 1</td>
</tr>
<tr>
<td>• xPhone</td>
<td>Maximum Connections = 3</td>
</tr>
<tr>
<td>• Solar</td>
<td></td>
</tr>
<tr>
<td>• Pixie</td>
<td></td>
</tr>
<tr>
<td>Up to 4 total phones</td>
<td>Minimum Quantity = 1</td>
</tr>
<tr>
<td></td>
<td>Maximum Quantity = 4</td>
</tr>
</tbody>
</table>

These different combinations would all be allowed by your connector definition:

<table>
<thead>
<tr>
<th>Quantity</th>
<th>xPhone</th>
<th>Solar</th>
<th>Pixie</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

How You Integrate Connectors

You can return configuration selections and items related by connectors to hosting applications.

REST Resources

You can use the connectedItems REST resource to return the connected items from a specified configuration. The connectedItems resource is a child of the configurations resource.

- A hosting application can call Configurator REST services to retrieve connected items as independent lines.
- Connected items can be retrieved through the `configurations` resource for integration purposes in other applications, such as a quoting or order management system.
- You can retrieve an individual connected item's runtime attributes, such as item number, unit of measure or quantity ordered.

**Example**

The following sample URL uses the `connectedItems` child resource of the `configurations` resource to obtain an output payload containing all of the connected items included in the specified configuration. In this example, there are two connected items in the configuration.

```plaintext
https://vision.com/fscmRestApi/resources/11.13.18.05/configurations/300100185429818_300100185429819/child/connectedItems
```

These are the objects of our main interest in the output payload:

<table>
<thead>
<tr>
<th>Example value</th>
<th>Object name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>300100185429818</td>
<td>ConfigHeaderId</td>
<td>Configuration header ID.</td>
</tr>
<tr>
<td>300100185429819</td>
<td>ConfigRevisionId</td>
<td>Configuration revision ID.</td>
</tr>
<tr>
<td>300100185429818_300100185429819</td>
<td>ConfigurationId</td>
<td>ID of the configuration, composed of the configuration header ID and the configuration revision ID. You pass this to the resource to return the connected items.</td>
</tr>
</tbody>
</table>

First example connected item

<table>
<thead>
<tr>
<th>Example value</th>
<th>Object name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>300100185429862</td>
<td>ConnectedItemld</td>
<td>Connected item ID of the first connected item.</td>
</tr>
<tr>
<td>300100181203985</td>
<td>InventoryItemld</td>
<td>ID of the first connected item.</td>
</tr>
<tr>
<td>7YR_PTRAIN_WARR</td>
<td>InventoryItemNumber</td>
<td>Item name of the first connected item.</td>
</tr>
</tbody>
</table>

Second example connected item

<table>
<thead>
<tr>
<th>Example value</th>
<th>Object name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>300100185429864</td>
<td>ConnectedItemld</td>
<td>Connected item ID of the second connected item.</td>
</tr>
<tr>
<td>300100181204014</td>
<td>InventoryItemld</td>
<td>ID of the second connected item.</td>
</tr>
<tr>
<td>WM_UTIL_TRLR</td>
<td>InventoryItemNumber</td>
<td>Item name of the second connected item.</td>
</tr>
</tbody>
</table>
Here is the JSON output payload returned for the connected items:

```json
{
  "items": [
    {
      "ConfigHeaderId": 300100185429818,
      "ConfigRevisionId": 300100185429819,
      "ConnectedItemId": 300100185429862,
      "ParentConfigLineId": 300100185429821,
      "InventoryItemId": 300100181203985,
      "InventoryItemNumber": "7YR_PTRAIN_WARR",
      "InventoryOrganizationId": 204,
      "InventoryOrganizationCode": "V1",
      "UomCode": "Ea",
      "InventoryItemType": 4,
      "Quantity": 1,
      "links": [
        {
          "rel": "self",
          "href": "https://vision.com:443/fscmRestApi/resources/11.13.18.05/configurations/300100185429818_300100185429819/child/connectedItems/300100185429862",
          "name": "connectedItems",
          "kind": "item"
        },
        {
          "rel": "canonical",
          "href": "https://vision.com:443/fscmRestApi/resources/11.13.18.05/configurations/300100185429818_300100185429819/child/connectedItems/300100185429862",
          "name": "connectedItems",
          "kind": "item"
        },
        {
          "rel": "parent",
          "href": "https://vision.com:443/fscmRestApi/resources/11.13.18.05/configurations/300100185429818_300100185429819",
          "name": "configurations",
          "kind": "item"
        }
      ]
    },
    {
      "ConfigHeaderId": 300100185429818,
      "ConfigRevisionId": 300100185429819,
      "ConnectedItemId": 300100185429864,
      "ParentConfigLineId": 300100185429821,
      "InventoryItemId": 300100181204014,
      "InventoryItemNumber": "WM_UTIL_TRLR",
      "InventoryOrganizationId": 204,
      "InventoryOrganizationCode": "V1",
      "UomCode": "Ea",
      "InventoryItemType": 4,
      "Quantity": 1,
      "links": [
        {
          "rel": "self",
          "href": "https://vision.com:443/fscmRestApi/resources/11.13.18.05/configurations/300100185429818_300100185429819/child/connectedItems/300100185429864",
          "name": "connectedItems",
          "kind": "item"
        },
        {
          "rel": "canonical",
          "href": "https://vision.com:443/fscmRestApi/resources/11.13.18.05/configurations/300100185429818_300100185429819/child/connectedItems/300100185429864",
          "name": "connectedItems",
          "kind": "item"
        },
        {
          "rel": "parent",
          "href": "https://vision.com:443/fscmRestApi/resources/11.13.18.05/configurations/300100185429818_300100185429819",
          "name": "configurations",
          "kind": "item"
        }
      ]
    }
  ]
}```
What's the difference between a connector and a model reference?

Model references and connectors both extend the structure of configurator models, but are different entities.

A model reference is a hierarchical link to a child model.

- A model reference is part of the bill of material for the configured product, and appears as the same order line in order management. On the other hand, connected related items are outside the bill of material, and can appear as separate lines in a quote or order. Connected related items can be individually fulfilled, and shipped apart from the configured product.
- A model reference is defined in the Configurator Models work area by importing a snapshot of a hierarchical model item structure that was created in the Product Information Management work area and contains child models, which become referenced models to the parent model.
- During a runtime configuration session, you can navigate the hierarchical model item structure to configure each distinct referenced model. The child models appear on the configuration summary page, if configured.

A connector is a non-hierarchical relationship to an item that exists outside of the model item structure.

- A connector is defined in the Configurator Models work area, as an element of configurator model supplemental structure. The relationship to the related item is established through the item's item class.
- During a runtime configuration session, you can select the related item to be included in the configuration of the bill of materials, although the item isn't part of the model. The related items appear in the running summary pane, but not on the configuration summary page.

Related Topics

- Connected Items REST Endpoints in the REST API for Oracle Supply Chain Management Cloud guide
• References are to models, and connectors are to items, not models.
8 Model Testing

Overview of Model Testing

This chapter covers testing of configurator models.

You can test configurator models using the methods listed in the following table.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interactively</td>
<td>Whenever viewing or editing a model in the Configurator Models work area, you can launch a simulated configuration session that uses the structure, rules, and user interfaces that you have defined for the model.</td>
</tr>
<tr>
<td>Test Service</td>
<td>Using the Configurator Runtime Model Test service, a SOAP web service, you can run Configurator in a non-interactive mode. The service takes as input a payload that creates the configuration, performs one or more configuration operations on the components of the model and closes the configuration. All of these operations are performed without any end user interaction.</td>
</tr>
</tbody>
</table>

Test Models Interactively

You can test a configurator model interactively, at any time during its development cycle, to ensure that its structure, behavior, and appearance are as you intend.

You can test the following aspects of a configurator model, as shown on the Configurator Model details page for viewing or editing a model:

- The structure of the model, as shown on the Structure tab, with any supplemental structure that you have added.
- The configurator rules that you have defined for the model, as shown on the Rules tab.
- Any user interfaces that you have defined for the model, as shown on the User Interfaces tab.

The aspects are all active together during a test session, subject to any test parameters that you apply, or any restrictions that are part of the model, such as:

- Items that are ineffective based on the test session date
- Rules that are disabled or invalid
Test Parameters

In the Test Model dialog box, you can set the test parameters described in the following table.

<table>
<thead>
<tr>
<th>Test Parameter</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Root Model</td>
<td>The model being viewed or edited. Read-only.</td>
</tr>
<tr>
<td>Version</td>
<td>Read-only.</td>
</tr>
<tr>
<td>Session Effective Date</td>
<td>The date and time to use in the test session for considering effectivity. The session effective date must be on or after the Effective Start Date of the workspace that the model is a participant of. Required.</td>
</tr>
<tr>
<td>User Interface</td>
<td>If user interfaces have been defined for the model, the UI to use for the test session, or the default UI. Optional.</td>
</tr>
<tr>
<td>Root Quantity</td>
<td>The quantity of the root model to use for the test session, which affects the relative quantity of child items in the configuration. Required.</td>
</tr>
<tr>
<td>Enable pricing</td>
<td>Enables the calculation and display of prices during a test session. If you enable pricing, then you must choose a pricing strategy that includes pricing rules for this model, in the <em>Pricing Strategy</em> field. Names of pricing strategies are suggested in the field as you enter part of their name.</td>
</tr>
<tr>
<td>Generate trace file</td>
<td>Enables the generation of a trace file. If you enable trace file generation, then you will be prompted to save the file at the end of the test session.</td>
</tr>
</tbody>
</table>

Test a Model

Follow these steps to interactively test a model. The test parameters are described separately.

1. Open a configurator model for editing or viewing.
   
   You must open a specific model. The model can be either locked or unlocked. If the model is not locked by the user who is testing it, that user must first unlock it. The model status can be either Draft or Released. You can be using either the Structure, Rules, or User Interfaces tab; the same test session tests all those aspects of the model. If you’re editing the model, you will be prompted to save any model changes.

2. On the Configurator Model details page, click **Test Model**.

3. In the Test Model dialog box, enter values for the test parameters. When you click **OK**, the configurator runtime test UI opens in a new dynamic tab of the work area window.

4. Interact with the configurator runtime test UI, as if you were an end user. The contents and appearance of the test UI reflect the test parameters that you entered for this test session.

   During the test session, you can navigate UI pages, select options and enter field values, and provide values for transactional item attributes. You can configure referenced models by clicking the **Configure** control, configuring the model, then returning to the root model.
5. When you're finished with your testing, click one of the action buttons to complete the test session. The available actions are similar to, but distinct from, the actions available to the end user in the configurator runtime end user UI.

- **Finish**: The configurator engine finishes the configuration, which means that the configurator engine uses the autocompletion process to automatically complete the remaining selections that are required for a valid configuration of the model. A valid configuration is determined by the attributes defined for the item structure in the Product Information Management work area, and by the configurator rules defined in the Configurator Models work area. After the configuration is finished, the test session ends, and you're returned to the Configurator Model details page.

- **Finish and Review**: The configurator engine finishes the configuration, then navigates to the Review page, where your selections are displayed for review and possible further configuration by clicking Back. Clicking OK returns to the Configurator Model details page.

- **Save for Later**: Saves the configuration in the exact state that you left it. The configurator engine doesn't finish the configuration, and the saved configuration may be invalid.

  You can assign a name to the configuration as you save it, and later restore the configuration for further testing.

- **Cancel**: Warns you about losing any selections made, and exits the test session, returning to the Configurator Model details page.

In all of these session actions, no configuration data is returned to any host application, since there is no host application for a test session.

6. If you enabled the generation of a trace file, then, when you're exiting the test session, a message confirms that the trace file was generated and is available for download. You can open the file or download it for later examination. When you close the confirmation dialog box, the test session ends.

7. Return to viewing or editing the tested model, on the Configurator Model details page.

### Trace Files

You can select the **Generate trace file** option in the Test Model dialog box to produce a trace file that can be provided to Oracle support.

Issues that surface while testing models that have large sets of complex rules can sometimes be difficult and time-consuming to diagnose. If you generate a trace file for a runtime test session, the file helps Oracle support, working jointly with you and Oracle product development, to diagnose the issues and resolve the problem, using constraint technology analysis and other tools and techniques.

The trace file is named after the model being tested, and is in XML format.

### Review Test Configurations

You can review the interim configuration results of a test session before leaving the session.

If you select the **Finish and Review** action during a test session, then you're navigated to the Review page after the configurator engine finishes the configuration. On that page, your selections are displayed for review.

- The orderable model items that have been configured during the session are displayed in a hierarchical tree that can be expanded and collapsed. Values entered for transactional item attributes are displayed in the tree, but values for user-defined attributes aren't displayed.

  Supplemental structure nodes aren't included on the Review page, since they aren't orderable, and would not be returned to a host application after a session.
For each configured item, the page displays the following values:

- **Quantity**: The quantity of the item included in the configuration, either by your selection or because of the action of configurator rules.
- **Unit Quantity**: The quantity of the item included in the configuration for each included unit of the item's parent item.
- **UOM**: The unit of measure for the quantity of the configured item.
- **Your Price**: The price of the item as determined by the applied pricing strategy. Pricing must be enabled for the test session.
- **Amount**: The amount of Your Price multiplied by the amount of Quantity. Pricing must be enabled for the test session.

When you're finished reviewing, click one of the following action buttons:

- **Back**: You're returned to the runtime test UI for possible further configuration.
- **OK**: The test session ends, and you're returned to the Configurator Model details page.
- **Cancel**: Warns you about losing any selections made, and exits the test session, returning to the Configurator Model details page.

### Restore Saved Configurations During Model Testing

You can test the latest configuration for a product model against previously saved configurations, to understand what changed. This ability to review changes in configurations helps you ensure the accuracy over time of your item structure and your configurator model definition.

When testing models, you can:

- **Save configurations during model testing**
  
  When you click **Save for Later** during a configuration session, you can enter a name and optional description for the configuration. You can also record the system-generated configuration header and revision IDs, which you can later use to search for this configuration.

  You can establish a naming strategy for saving configurations, to ease searching among them.

- **Test model changes with previously saved configurations**

  When you start a test session, you can select **Restore a configuration** in the Test Model dialog box, then select the configuration to restore from the **Configuration Name** list. You can search for a configuration by its header ID or revision ID by clicking the **Search** link in the list.

  If you modified the model after the previous test session, you can compare how the modified model behaves against one of your saved configurations.

### Recommendations for Efficiency

Oracle recommends a number of ways in which you can use Configurator more efficiently.
Skipping Validation of Unimported Models

For model items that haven't been imported into the Configurator Models work area, you can optimize the time it takes to validate orders containing configuration lines during order submission as part of an order revision or an explicit call to validate an order from within Oracle Fusion Order Management. You achieve this optimization by causing Configurator to skip the configuration validation when it's deemed unnecessary.

Configuration validation is skipped if there are no changes to the model, or to its item-component-level attributes, as of the configuration effective date on the order, as dictated by the value of the Order Management parameter **Configuration Effective Date**. Configurator checks whether a component item has been deleted or end-dated from the model's item structure, and checks the following attributes for changes in their values:

- **Item-level attributes:**
  - Order Management Indivisible
  - Structure Item Type
- **Child component-level attributes:**
  - Quantity (default)
  - Minimum Quantity
  - Maximum Quantity
  - Optional
  - Mutually Exclusive Options
  - Instantiability (values: Multiple Instances, Optional Single Instance, or Requires Single Instance)

**Example**

Assume that:

- The **Configuration Effective Date** parameter is set to **Ordered date**
- The component-level attribute for a component **Maximum Quantity** is updated to become effective in the future, after the ordered date on the order

Then, if an order revision or explicit order validation call is made, configuration validation is skipped. If the change to the value of **Maximum Quantity** became effective prior to the ordered date, the configuration validation isn't skipped.

This table summarizes the relationship in this example of the Configuration Effective Date parameter, the checked attribute values, and configuration validation:

<table>
<thead>
<tr>
<th>Configuration Effective Date parameter</th>
<th>Value of checked attribute, when effective</th>
<th>Configuration validation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ordered date</td>
<td>after the ordered date on the order</td>
<td>skipped</td>
</tr>
<tr>
<td>Ordered date</td>
<td>prior to the ordered date</td>
<td>not skipped</td>
</tr>
</tbody>
</table>
To control which date is used for the Configuration Effective Date:

1. In the Setup and Maintenance work area, use the Manage Order Management Parameters task:
   - Offering: Order Management
   - Functional Area: Orders
   - Task: Manage Order Management Parameters
2. Select the parameter **Configuration Effective Date**.
3. For a selected business unit, select a value for **Configuration Effective Date**. The values are:
   - Ordered date
   - Configuration date
   - Current date (the default)
   - Requested date

**Related Topics**
- Manage Order Management Parameters
- Valid Component Attributes and Structure Types
- Item Order Management Specifications

### The Configurator Runtime Model Test Service

To efficiently test a large number of models, you can use the Configurator Runtime Model Test service to execute model tests without requiring user input.

The Configurator Runtime Model Test service is implemented as a SOAP web service based on a request and response payload structure. In this service model, a calling application provides an input test model XML payload to the `testModel` public synchronized method of the Configurator Runtime Model Test service, then after executing the requests, the service returns a corresponding result XML payload which the calling application can introspect for the result of the requests.

Features of the test service include:

- Calling applications can launch the Configurator Runtime Model Test service using any SOAP client. The SSL port is enabled by default for this web service, because it’s an external service and will be the default transport for interacting with the web service in the cloud.
- Calling applications can submit model test service requests to the synchronized method `testModel` in a serialized fashion, one request at a time.
- The Configurator Runtime Model Test service can be used in concert with other services or embedded within other applications or services.

The Configurator Runtime Model Test service's input and output payloads use a SOAP XML structure.

The Configurator Runtime Model Test service has been implemented as an autonomous service and doesn’t require any other external data model entities. The service WSDL can be retrieved using the following URL format:

```
https://<hostname>:<port>/fscmRestApi/ConfiguratorRuntimeService?wsdl
```

This service supports only one synchronized method, `testModel`, which has a specific request and response payload structure.
The `testModel` method requires a request input payload which includes the necessary information to:

- Start or restore a configuration
- Perform operations on that configuration
- Query the configuration for attributes
- Save and request a configuration summary

The `testModel` method subsequently returns a response output payload which includes:

- The result of the overall test
- Each individual operation requests
- Query requests
- Configuration summary requests

**Test Model Request Structure**

The following code example shows a test model request in SOAP XML:

```xml
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Body>
    <ns1:testModel xmlns:ns1="http://xmlns.oracle.com/apps/scm/configurator/runtimeService/types/">
      <ns1:request xmlns:ns2="http://xmlns.oracle.com/apps/scm/configurator/runtimeService/">
        <ns2:TestId>1</ns2:TestId>
        <ns2:TestName>test</ns2:TestName>
        <ns2:TestDescription>Test1</ns2:TestDescription>
        <ns2:InitializationParameters>
          <ns2:TestId>1</ns2:TestId>
          <ns2:ParameterId>1</ns2:ParameterId>
          <ns2:CallingApplicationCode>CZ</ns2:CallingApplicationCode>
          <ns2:Header>
            <![CDATA[
              <header>
                <HeaderId>1000</HeaderId>
              </header>
            ]]>
          </ns2:Header>
          <ns2:Line>
            <![CDATA[
              <line>
                <LineId>-22222</LineId>
                <InventoryItemNumber>zCZ-CN82441</InventoryItemNumber>
                <InventoryOrganizationCode>V1</InventoryOrganizationCode>
                <UnitQuantity>1.0</UnitQuantity>
                <UomCode>Ea</UomCode>
              </line>
            ]]>
          </ns2:Line>
          <ns2:CustomParameters>
            <![CDATA[
              <param>
                <WorkspaceName>My Workspace</WorkspaceName>
                <SnapshotBasedModel>true</SnapshotBasedModel>
              </param>
            ]]>
          </ns2:CustomParameters>
        </ns2:InitializationParameters>
        <ns2:OperationRequest>
          <ns2:OperationId>1</ns2:OperationId>
          <ns2:NodePath>zCZ-OC12100 .zCZ-CM1240</ns2:NodePath>
          <ns2:Operation>Select</ns2:Operation>
        </ns2:OperationRequest>
      </ns1:request>
    </ns1:testModel>
  </soap:Body>
</soap:Envelope>
```
The basic structure of a test model request includes the elements described in the following table:

<table>
<thead>
<tr>
<th>Element of the testModel Request</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TestId</td>
<td>The unique identifier for the current test. The same test ID is also used in other elements of the test request to connect them with a given test.</td>
</tr>
<tr>
<td>TestName</td>
<td>A short name for this test</td>
</tr>
<tr>
<td>TestDescription</td>
<td>An optional description for this test</td>
</tr>
<tr>
<td>InitializationParameters</td>
<td>Includes all the information necessary to start a configuration session</td>
</tr>
<tr>
<td>OperationRequest</td>
<td>One or more operations to be performed during this test</td>
</tr>
<tr>
<td>QueryRequest</td>
<td>One or more queries to be performed after an operation request has been performed</td>
</tr>
<tr>
<td>ConfigSummaryRequest</td>
<td>The configuration summary which includes all the orderable items for the configuration</td>
</tr>
</tbody>
</table>

**InitializationParameters**

InitializationParameters is an XML element that models an attribute of the test model request.
InitializationParameters is a required attribute that includes the necessary information to start or restore a configuration session. The following table lists the initialization parameters, both necessary and optional, to start a configuration.

<table>
<thead>
<tr>
<th>Field name in InitializationParameters</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TestId</td>
<td>The unique identifier for the current test.</td>
</tr>
<tr>
<td>ParameterId</td>
<td>A unique identifier provided by the service to refer to this initialization parameters set</td>
</tr>
<tr>
<td>CallingApplicationCode</td>
<td>Indicates the calling application launching the Configurator session. This is used to determine the user interface to use for the session. This code is registered in the Manage Trading Community Source Systems in the case of external non-Fusion system or the FND Application short name (code) for Oracle Fusion Applications. Note: For CPQ there is already a seeded TCA data with application code as ORA_BM_CPQ</td>
</tr>
<tr>
<td>Line</td>
<td>An XML document representing a line to be configured. See the following section for additional information</td>
</tr>
<tr>
<td>Header</td>
<td>(Optional) Represents a quote or order header that contains the Line. This XML document is optional and when present Configurator passes the header information to other services like Oracle Fusion Pricing for pricing integration within a configuration session</td>
</tr>
<tr>
<td>CustomParameters</td>
<td>(Optional) A XML document consisting of additional information to Configurator for use within Configurator Extensions rules</td>
</tr>
</tbody>
</table>

The XML document Line represents a line to be configured. It contains, at a minimum, the information in the following list.

- LineId: Unique identifier for the line being configured
- InventoryItemNumber: Item number of the product or service model item to configure
- InventoryOrganizationCode: Item validation organization code of the product or service model item. This is typically the organization from which items are imported or referenced from external applications.
- UnitQuantity: Quantity of the item being configured
- UomCode: Unit of measure for the item being configured
- ConfigHeaderId: Identifier for the configuration, passed only during reconfiguration, or a session for restoring and validating
- ConfigRevisionId: Identifier for the configuration, passed only during reconfiguration/restore and validation session
- RequestOn (Optional): Date and time used to load the model definition for the configuration session

**OperationRequest**

OperationRequest is an XML element that models an attribute of the test model request.

After a configuration has started, you can perform a series of operations using the test service, similar to the operations that an end user can perform when configuring the model using a runtime user interface. A test model request can have...
one or more operation requests. These operations can range from toggling a Boolean feature, to setting the quantity of a standard item, to adding an instance of a referenced model and configuring it.

The following table lists the attributes of an OperationRequest.

<table>
<thead>
<tr>
<th>Field name in OperationRequest</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TestId</td>
<td>The unique identifier for the current test.</td>
</tr>
<tr>
<td>OperationId</td>
<td>A unique identifier for the operation request</td>
</tr>
<tr>
<td>NodePath</td>
<td>The fully qualified path to the node on which the operation will act on</td>
</tr>
<tr>
<td>Operation</td>
<td>The name of the operation, such as. Select, Toggle, SetValue, or SetQuantity. See the following table for valid operations.</td>
</tr>
<tr>
<td>SequenceNumber</td>
<td>The sequence number to be used during the execution to prioritize this request</td>
</tr>
<tr>
<td>NodeValue</td>
<td>The value to be applied to the node for this test.</td>
</tr>
</tbody>
</table>

The following table lists the operations that you can perform within an OperationRequest, and the node types that you can perform the operations on.

<table>
<thead>
<tr>
<th>Valid Operations for OperationRequest</th>
<th>Node Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Select, Toggle, SetQuantity</td>
<td>Option Class, Option Feature, Standard Item, or Option</td>
</tr>
<tr>
<td>Toggle</td>
<td>Boolean Feature</td>
</tr>
<tr>
<td>SetValue</td>
<td>Text Feature, Integer Feature, Decimal Feature, or Transactional Item Attribute (All types)</td>
</tr>
<tr>
<td>AddInstance, AddInstanceWithQuantity, AddInstanceWithQuantityAndName, RenameInstance, CopyInstance, DeleteInstance, SetQuanity, SetContextPath</td>
<td>Model Node Reference</td>
</tr>
</tbody>
</table>

The operations that can be performed at the configuration level include those in the following list.

- **AutoComplete**: Performs an autocompletion operation on the configuration, to include all the adjustments needed to make the configuration valid.
- **UndoAutoComplete**: Reverts the autocompletion operation.
- AdjustConfiguration: Puts the configuration into adjust mode after autocompletion, allowing changes to the automatically completed selections.
- Save: Saves the configuration and produces a header and revision ID.
- Finish: Finishes the configuration, by saving and closing it.

**QueryRequest**

QueryRequest is an XML element that models an attribute of the test model request.

In order to determine the result of an operation request, you can perform a series of query requests to introspect the node on which the operation request acted, or any other node in the model hierarchy that was changed as a result. For example, if you toggle a Boolean feature using an operation request and subsequently a rule causes a standard item to be selected, then you can add a query request to retrieve the state of that standard item.

The following table lists the fields in a QueryRequest.

<table>
<thead>
<tr>
<th>Field Name in QueryRequest</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TestId</td>
<td>The unique identifier for the current test.</td>
</tr>
<tr>
<td>QueryRequestId</td>
<td>A unique identifier for the query request in the test.</td>
</tr>
<tr>
<td>NodePath</td>
<td>The fully qualified path to the node on which the query operation will act.</td>
</tr>
<tr>
<td>SequenceNumber</td>
<td>The sequence number to be used during the execution to prioritize this request.</td>
</tr>
<tr>
<td>NodeProperty</td>
<td>The property of the node returned in the result payload. See the following table for additional information.</td>
</tr>
</tbody>
</table>

Each query request can include one or many node properties to be retrieved. These node properties can be of either scalar type, such as a string or number value, or a collection or list type, in which the result of the query request will return a list of values. For example, in order to determine the selection state of a standard item, the node property named `Selected` can be requested through a NodeProperty, as illustrated in the following example.

```xml
<ns2:NodeProperty>
  <ns2:PropertyId>1</ns2:PropertyId>
  <ns2:PropertyName>Selected</ns2:PropertyName>
  <ns2:QueryId>1</ns2:QueryId>
</ns2:NodeProperty>
```

The following table lists the fields in a NodeProperty request.

<table>
<thead>
<tr>
<th>Field Name in NodeProperty</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PropertyId</td>
<td>A unique identifier for the property in the request.</td>
</tr>
</tbody>
</table>
Field Name in NodeProperty | Description
--- | ---
PropertyName | The name of the node property to be evaluated. Refer to the list of node properties for the properties supported for each node type.
QueryId | A unique identifier for the query request
CollectionProperty | A list of property requests if the named property being evaluated is a list or a collection type (such as AvailableChildren, SelectedChildren, or SelectableChildren). See the following description of CollectionProperty for additional information.

Similar to the way that you select node properties, you request the scalar properties of a list or collection node property type, by adding the property name to the CollectionProperty element's list of properties, as illustrated in the following example.

```xml
<ns2:NodeProperty>
  <ns2:PropertyId>1</ns2:PropertyId>
  <ns2:PropertyName>SelectedChildren</ns2:PropertyName>
  <ns2:QueryId>1</ns2:QueryId>
  <ns2:CollectionProperty>
    <ns2:PropertyId>1</ns2:PropertyId>
    <ns2:CollectionPropertyId>1</ns2:CollectionPropertyId>
    <ns2:PropertyName>Name</ns2:PropertyName>
  </ns2:CollectionProperty>
  <ns2:CollectionProperty>
    <ns2:PropertyId>1</ns2:PropertyId>
    <ns2:CollectionPropertyId>2</ns2:CollectionPropertyId>
    <ns2:PropertyName>Description</ns2:PropertyName>
  </ns2:CollectionProperty>
</ns2:NodeProperty>
```

The following table lists the elements that identify a CollectionProperty.

Field Name in CollectionProperty | Description
--- | ---
PropertyId | The unique identifier for the property name of the collection node property to be evaluated.
PropertyName | A property name from the collection node property to be evaluated
CollectionPropertyId | A unique identifier for the collection node property

**ConfigSummaryRequest**

ConfigSummaryRequest is an XML element that models an attribute of the test model request.

A test model request can include an optional request to retrieve the configuration summary. The configuration summary can be either brief or full, as controlled by the output mode.
The following table lists the fields of the ConfigSummaryRequest element.

<table>
<thead>
<tr>
<th>Field Name in ConfigSummaryRequest</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TestId</td>
<td>The unique identifier for the current test.</td>
</tr>
<tr>
<td>SummaryId</td>
<td>A unique identifier for the summary request</td>
</tr>
<tr>
<td>OutputMode</td>
<td>The configuration summary output mode. The allowed values are:</td>
</tr>
<tr>
<td></td>
<td>• Brief: Include only configuration-level information such as the validity of the configuration, and the header ID and revision ID of the configuration object.</td>
</tr>
<tr>
<td></td>
<td>• Full (the default): Include the brief output and also all of the orderable items in the configuration.</td>
</tr>
<tr>
<td>SequenceNumber</td>
<td>The sequence number to be used during the execution to prioritize this request</td>
</tr>
</tbody>
</table>

Test Model Response Structure

The model test response payload structure directly corresponds to the test model request payload. For every operation or query request there is a corresponding operation or query result that can be correlated to its request counterpart using the unique operation or query ID.

The test model response includes test level information such as the overall test status and any failure messages that prevented the configuration from starting. This level of information is also available at each operation or query result, which allows for introspection of the results at a much more granular level.

The following code example shows a test model response in SOAP XML:

```xml
 xmlns:wsa="http://www.w3.org/2005/08/addressing">
  <env:Header>
    ...
  </env:Header>
  <env:Body xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-utility-1.0.xsd"
    wsu:Id="Body-WCk1o7Bv0d7m0fziDdW71Q22">
    <ns0:testModelResponse xmlns:ns0="http://xmlns.oracle.com/apps/scm/configurator/runtimeService/types/">
      <ns1:result xmlns:ns0="http://xmlns.oracle.com/apps/scm/configurator/runtimeService/
        xmlns:ns1="http://xmlns.oracle.com/apps/scm/configurator/runtimeService/types/"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:type="ns0:TestModelResponse">
        <ns0:TestId>1</ns0:TestId>
        <ns0:TestName>test</ns0:TestName>
        <ns0:TestDescription>Test1</ns0:TestDescription>
        <ns0:TestStatus>SUCCESS</ns0:TestStatus>
        <ns0:TestFailureMessage xsi:nil="true"/>
        <ns0:OperationResult>
          <ns0:OperationId>1</ns0:OperationId>
          <ns0:NodePath>zCZ-OC12100.zCZ-CM12240</ns0:NodePath>
          <ns0:OperationSelect>1</ns0:OperationSelect>
          <ns0:SequenceNumber>1</ns0:SequenceNumber>
          <ns0:NodeValue>1</ns0:NodeValue>
          <ns0:OperationStatus>SUCCESS</ns0:OperationStatus>
        </ns0:OperationResult>
      </ns1:result>
    </ns0:testModelResponse>
  </env:Body>
</env:Envelope>
```
<ns0:OperationFailureMessage xsi:nil="true"/>
<ns0:TestId>1</ns0:TestId>
<ns0:ContextPath xsi:nil="true"/>
</ns0:OperationResult>
<ns0:QueryResult>
<ns0:QueryId>1</ns0:QueryId>
<ns0:NodePath>zCZ-OC12100.zCZ-CM12240</ns0:NodePath>
<ns0:QueryResultStatus>SUCCESS</ns0:QueryResultStatus>
<ns0:QueryResultFailureMessage xsi:nil="true"/>
<ns0:TestId>1</ns0:TestId>
<ns0:SequenceNumber>2</ns0:SequenceNumber>
<ns0:NodeProperty>
<ns0:PropertyId>1</ns0:PropertyId>
<ns0:PropertyName>Selected</ns0:PropertyName>
<ns0:PropertyValue>true</ns0:PropertyValue>
<ns0:QueryId>1</ns0:QueryId>
<ns0:PropertyResultStatus>SUCCESS</ns0:PropertyResultStatus>
<ns0:PropertyResultFailureMessage xsi:nil="true"/>
</ns0:NodeProperty>
</ns0:QueryResult>
<ns0:ConfigSummaryResult xmlns:ns1="http://xmlns.oracle.com/adf/svc/types/">
<ns0:SummaryId>1</ns0:SummaryId>
<ns0:TestId>1</ns0:TestId>
<ns0:ConfigHeaderId xsi:nil="true"/>
<ns0:ConfigRevisionId xsi:nil="true"/>
<ns0:ValidConfigurationFlag>false</ns0:ValidConfigurationFlag>
<ns0:SequenceNumber>3</ns0:SequenceNumber>
<ns0:RequestOn>2016-12-30</ns0:RequestOn>
</ns0:ConfigSummaryResult>
<ns0:ConfigLine>
<ns0:LineId>1</ns0:LineId>
<ns0:ParentLineId xsi:nil="true"/>
<ns0:InventoryItemId>300100017155319</ns0:InventoryItemId>
<ns0:InventoryOrganizationId>204</ns0:InventoryOrganizationId>
<ns0:InventoryItemType>1</ns0:InventoryItemType>
<ns0:LineQuantity xmlns:tns="http://xmlns.oracle.com/adf/svc/errors/" unitCode="Ea">1</ns0:LineQuantity>
<ns0:UOMCode>Ea</ns0:UOMCode>
<ns0:ConfiguratorPath>300100017155319</ns0:ConfiguratorPath>
</ns0:ConfigLine>
<ns0:ConfigLine>
<ns0:LineId>2</ns0:LineId>
<ns0:ParentLineId>1</ns0:ParentLineId>
<ns0:InventoryItemId>300100017233297</ns0:InventoryItemId>
<ns0:InventoryOrganizationId>204</ns0:InventoryOrganizationId>
<ns0:InventoryItemType>2</ns0:InventoryItemType>
<ns0:LineQuantity xmlns:tns="http://xmlns.oracle.com/adf/svc/errors/" unitCode="Ea">1</ns0:LineQuantity>
<ns0:UOMCode>Ea</ns0:UOMCode>
<ns0:ConfiguratorPath>300100017155319.300100017233297</ns0:ConfiguratorPath>
</ns0:ConfigLine>
<ns0:ConfigLine>
<ns0:LineId>3</ns0:LineId>
<ns0:ParentLineId>2</ns0:ParentLineId>
<ns0:InventoryItemId>300100017232813</ns0:InventoryItemId>
<ns0:InventoryOrganizationId>204</ns0:InventoryOrganizationId>
<ns0:InventoryItemType>4</ns0:InventoryItemType>
<ns0:LineQuantity xmlns:tns="http://xmlns.oracle.com/adf/svc/errors/" unitCode="Ea">1</ns0:LineQuantity>
<ns0:DOMCode>Ea</ns0:DOMCode>
<ns0:ConfiguratorPath>30100017155319.300100017233297.300100017232813</ns0:ConfiguratorPath>
<ns0:SummaryId>1</ns0:SummaryId>
<ns0:InventoryItemNumber>zCZ-CM12240</ns0:InventoryItemNumber>
<ns0:InventoryOrganizationCode>V1</ns0:InventoryOrganizationCode>
<ns0:UnitQuantity xmlns:tns="http://xmlns.oracle.com/adf/svc/errors/" unitCode="Ea">1</ns0:UnitQuantity>
</ns0:ConfigLine>
</ns0:ConfigSummaryResult>
</ns1:result>
</ns0:testModelResponse>
</env:Body>
</env:Envelope>

Related Topics

- Configurator Model Node Properties
- Configurator Models
- Item-Based Model Structure
- Supplemental Model Structure

How You Run the Model Test Service

Calling applications can launch the Configurator Runtime Model Test service using any SOAP client. The instructions here describe the process for creating a web service proxy and data control to run the service within the context of a jUnit test case for Oracle JDeveloper.

The required process is:

1. Create a web service proxy.
2. Create a web service data control.
3. Create a jUnit.

The details for these steps follow.

Creating a Web Service Proxy

The first step is to create a web service proxy using the Oracle JDeveloper Web Service Proxy wizard. Using a web service's WSDL, this wizard creates all the necessary Java objects for the serialization and deserialization of the request and response payloads and calling the web service.

To create a web service proxy:

1. Create a generic application and project in Oracle JDeveloper.
2. Select the project in your application and click File > New.
3. Search for the Web Service Proxy project technology, select it and click OK.
4. Click Next on the wizard's overview page.
5. Leave the Client Style as JAX-WS Style and click Next.
6. Enter the URL to the Web Services Description Language (WSDL) endpoint on the host where the Configurator Runtime Test Model service is deployed (in a cloud deployment this would be on the HTTPS port) then click Next.
7. On the Specify Default Mapping Options step, deselect the **Generate As Async** option (because asynchronous invocation isn’t supported) and click **Next** twice, to navigate through the steps related to port endpoints without changing the options.

8. On the Asynchronous Methods step, be sure to select the **Don't generate any asynchronous methods** option, then click **Next**.

9. On the Policy step, wait for the list of policies to be populated. Select `oracle/wss11_username_token_with_message_protection_client_policy` from the list. Click **Next**.

10. On the Defined Handlers step, click **Next**. Finally, click **Finish**.

11. The wizard creates all the necessary Java objects needed for constructing the request and response payloads needed to interact with the web service. See the description of creating a jUnit for an example of how these objects can be used.

---

**Creating a Web Service Data Control**

A web service data control is an effective way to create a connection using the connection factory object in order to interact with a web service. A data control encapsulates the security policies, credentials and the data model needed when calling the service. With a properly prepared data control, all that’s required for a client to call the web service is to retrieve a service connection using its Java Naming and Directory Interface (JNDI) name. See the description of creating a jUnit for an example of how to call a web service using the service connection factory.

To create a web service data control:

1. Create and select a project where the web service data controls will be created, then click **File New**.
2. Search for the Web Service Data Control project technology, select it, then click **OK**.
3. Enter a name for the data source, which will later be used as the JNDI name for looking up this web service connection. Provide the URL to the WSDL endpoint on the host where the Configurator Runtime Test Model service is deployed (in a cloud deployment this would be on the HTTPS port) then click **Next**.
4. On the Data Control Operations step, select and move the **testModel** operation from the Available operations list to the Selected operations list. Click **Next**.
5. On the Response Format step, click **Next**.
6. On the Endpoint Authentication step, select **ConfiguratorRuntimeServiceSoapHttpPort**, then enter the user name and password for the web service. Click **Next**.
7. Click **Finish**. The web service data control wizard creates all the type XML files and adds a connection reference in the **connections.xml** file of the calling application.

---

**Creating a jUnit**

There are many ways to call a web service. Here, the vehicle described for calling the Configurator Runtime Model Test service is a jUnit. jUnits are an effective way to perform repeatable tests for this service. Also, Oracle JDeveloper has full support for the Apache jUnit 4 framework.

To build a simple jUnit in Oracle JDeveloper,

1. Select an existing project, or create a new project, in your application where you want to create the jUnit.

   **Note:** Ensure that the web service proxy and web service data control projects that you previously created are in this project’s library path. It’s essential that your jUnit have access to the object in those projects.

2. Search for and select the Test Case project technology. Click **OK**.
3. In the **Class Under Test** list, select `<None>`. Click **Next**.
4. Enter a name and package for the jUnit. Select the standard jUnit setup and tear-down methods to be generated. Click **Next** to continue.
5. Click **Finish**. The test case wizard creates the jUnit Java class in your project.

6. The content of the jUnit should be similar to the following code example.

```java
package oracle.apps.scm.cz;
import org.junit.After;
import org.junit.AfterClass;
import static org.junit.Assert.*;
import org.junit.Before;
import org.junit.BeforeClass;

public class MyServiceTest {
    public MyServiceTest() {
    }

    @Before
    public void setUp() throws Exception {
    }

    @After
    public void tearDown() throws Exception {
    }

    @BeforeClass
    public static void setUpBeforeClass() throws Exception {
    }

    @AfterClass
    public static void tearDownAfterClass() throws Exception {
    }
}
```

7. Add a test method to the jUnit that's similar to the following code example.

```java
@Test
public void testMyTest() {
    ConfiguratorRuntimeService_Service configuratorRuntimeService_Service;
    // Insert the name of the file that includes the model test request XML
    File testModelFile = new File("filename.xml");

    TestModelResponse response = null;
    try {
        // 1. Lookup the web service connection using its JNDI name
        Context cxtnew = ADFContext.getCurrent().getConnectionsContext();
        WebServiceConnection wsc = (WebServiceConnection)cxtnew.lookup("WebServiceJNDIName");

        // 2. Get a reference to the web service port
        ConfiguratorRuntimeService_ConfiguratorRuntimeService_Service =
            new ConfiguratorRuntimeService_Service();
        ConfiguratorRuntimeService configuratorRuntimeService =
            wsc.getJaxWSPort(ConfiguratorRuntimeService.class);

        // 3. Create a test model request using JAXB
        JAXBContext jaxbContext = JAXBContext.newInstance(TestModel.class);
        Unmarshaller jaxbUnmarshaller = jaxbContext.createUnmarshaller();
        TestModel testModel =
            (TestModel)jaxbUnmarshaller.unmarshal(testModelFile);

        // 4. Call the web service method
        TestModelResponse testModelResponse =
            new TestModelResponse();
        testModelResponse =
            configuratorRuntimeService.testModel(testModel);

    } catch (Exception e) {
        Assert.fail(e.getMessage());
    }
}
```
8. Be sure to provide the relative path to the model test request XML file and the correct JNDI name to look up the web service connection. The JNDI name is the name you provided when you created the web service data control.

9. Run the Java class as a jUnit within Oracle JDeveloper.

Developer Testing Recommendations

The Configurator Runtime Model Test service is an autonomous SOAP-based web service; it can be called from any SOAP client.

Because the Configurator Runtime Model Test service is deployed as an external service in the cloud environment, the exposed port will be enabled for SSL (Secure Sockets Layer) by default. It is highly recommended that you build and test client applications using the SSL port.

To ensure that the initial SSL handshake between the Java client application (such as jUnit) and the web service is established, you must import the SSL certificate where the Configurator Runtime Model Test service is hosted into the local Java key store where the Java client application is running.

To ensure that the proper SSL certificate for the Configurator Runtime Model Test service is available during the client-server SSL handshake, perform the following steps:

1. Access the Configurator Runtime Model Test service on the SSL port from a standard browser. For an example using Firefox:
   a. Select **Tools > Options > Security**
   b. Click **View Certificate**.

2. Export the server SSL certificate as a certificate in the PEM (Privacy-Enhanced Mail) security format.
   a. On the Details tab of the Certificate Viewer dialog box, click **Export**.
   b. Save the PEM file to a desired location.

   `my_server_com.PEM`

3. Using the Java `keytool` utility, import the certificate file to the local Java keystore where the Java client application is running.

   ```
   keytool -import -trustcacerts
   -file my_server_com.PEM
   -keystore $JAVA_HOME/lib/DemoTrust.jks
   ```

Related Topics

- Configurator Model Node Properties
- Configurator Models
9 Integration

How You Integrate with External Applications

You can integrate Oracle Fusion Configurator with external applications, such as quoting or order capture, to provide an interactive configuration experience for configuring products and services.

Configurator is agnostic about the technology stack of an external application and implements a standards-based approach for a seamless UI integration experience. The integration approach leverages Representational State Transfer (REST) APIs for creating the intent to configure a complex item and for retrieving the final configuration, and an entry point URL for launching the configurator runtime UI, either within an IFrame or within a browser window. REST isn’t a technology, but rather a client-server architectural style in which participants perform actions on data-oriented resources.

Integration Architecture

Integration to launch Configurator from an external application and return after a configuration session consists of the elements described in the following table.

<table>
<thead>
<tr>
<th>Integration Element</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>REST resources</td>
<td>Provide data-oriented services for:</td>
</tr>
<tr>
<td></td>
<td>• Creating an intent to start a configuration session</td>
</tr>
<tr>
<td></td>
<td>• Retrieving configuration details, such as lines, line attributes and messages</td>
</tr>
<tr>
<td></td>
<td>• Performing custom configuration actions.</td>
</tr>
<tr>
<td>External application data model entities</td>
<td>Provide data from the host application that’s needed by Configurator to establish integration, such as a unique identifier for a configuration header.</td>
</tr>
<tr>
<td>Public entry point URL</td>
<td>Provides the entry point for launching a configurator user interface session.</td>
</tr>
<tr>
<td>REST input and output payloads</td>
<td>Carry the specific inputs to and outputs from a configuration session.</td>
</tr>
<tr>
<td>Integration tasks</td>
<td>Actions required to complete an integration.</td>
</tr>
</tbody>
</table>

REST Resources

External applications can leverage these Configurator REST resources to interact with configurations

- **Initialization Parameters**: External applications must use this resource to specify the intent to configure a product or service interactively by passing an input payload which provides details that the configurator uses at run time to start a configuration session.
At a minimum the following information must be posted to this REST resource:

- CallingApplicationCode: The application code registered in the Manage Trading Community Source Systems task in the case of external non-Fusion applications or the FND Application short name (code) for Oracle Fusion applications.
- Line: A JavaScript Object Notation (JSON) document serialized in string format. For each field, a name-value pair corresponds to the sales context line entity definition.

The Initialization Parameters resource is used to load the appropriate model and user interface to launch an interactive configuration session.

The Initialization Parameters resource can be used only once to launch Configurator, then the resource is marked as expired and can’t be reused. The purpose of this restriction is to prevent attacks such as distributed denial of service.

- **Configurations**: External applications must use this resource to retrieve the details of a configuration, using the configuration identifiers that are returned by Configurator after finishing a configuration session.

The Configurations resource consists of one root entity and encompasses three child entities:

- ConfigurationLines: a collection resource that describes all the lines in a configuration.
- ConfigurationLineAttributes: a collection resource that describes all the transactional item attribute lines in a configuration.
- ConfigurationMessages: a collection resource that describes one or more messages added in a configuration.

The Configurations resource by default filters out option class lines when returning configuration results. An external application can request the full result by specifying the OutputMode query parameter value as `full`.

- The Configurations resource supports the following custom actions that can be performed on a configuration:
  - **copy**: Copy a configuration mirroring the copy action on a quote/order line.
  - **validate**: Validate a configuration before submitting a quote or order to Oracle Fusion Order Management for fulfillment to eliminate quote or order errors before submission.

See the REST API for Oracle Supply Chain Management Cloud guide for full details on:

- Descriptions of parameters and query examples for the Initialization Parameters and Configurations REST endpoints
- Examples of use cases for the custom actions

**Note:** The REST API for Oracle Supply Chain Management Cloud guide is the comprehensive reference for working with SCM REST resources, and includes information on getting started with REST APIs.

### External Application Data Model Entities

To persist configuration-related information, all external applications integrating with Configurator must add the following attributes in their entity that represents a configuration line.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>ConfigHeaderId</td>
<td>Configurator-generated unique identifier for a configuration header</td>
</tr>
<tr>
<td>Attribute</td>
<td>Purpose</td>
</tr>
<tr>
<td>---------------------</td>
<td>-------------------------------------------------------------------------</td>
</tr>
<tr>
<td>ConfigRevisionId</td>
<td>Configurator-generated unique identifier for a revision of a configuration</td>
</tr>
<tr>
<td>ConfiguratorPath</td>
<td>Configurator-generated runtime inventory ID path to an item for a configuration line</td>
</tr>
</tbody>
</table>

The combination of ConfigHeaderId and ConfigRevisionId uniquely identifies a configuration. The identification attributes must be passed to Configurator during reconfiguration flows, and to Oracle Fusion Order Management when submitting a quote or order for fulfillment.

In addition to the attributes that identify a configuration, external applications can add the following attributes in their configuration line entity, to synchronize with existing quote or order lines during a reconfiguration session, instead of deleting and recreating the quote or order lines. This information is retrieved using the Configurations resource when you expand the child resources.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>ConfigLineId</td>
<td>Configurator-generated unique identifier for a configuration line</td>
</tr>
<tr>
<td>ParentConfigLineId</td>
<td>Configurator-generated unique identifier to indicate the parent configuration line from a child line</td>
</tr>
<tr>
<td>ConfigLineAttributeId</td>
<td>Configurator-generated unique identifier for a configuration line attribute representing a Transactional Item Attribute associated to an Item.</td>
</tr>
</tbody>
</table>

**Public UI Entry Point URL**

Configurator provides a public entry point URL to launch the configurator runtime UI, either within an IFrame or within a browser window.

The URL entry point to launch the configurator runtime UI is publicly available, and secured to provide access to an authenticated user. The entry point expects the following URL parameters:

- **ParameterId**: An initialization parameter resource identifier generated by the Initialization Parameters REST API while creating the intent to configure, which identifies a set of initialization parameter attribute-value pairs. If this parameter value isn’t specified or is invalid then an error dialog will be displayed to the user and configurator will redirect to the ReturnUrl.

- **ReturnUrl**: A fully qualified URL of the host application to return to after finishing a configuration session. An HTTP 400 error code will be returned if this parameter value isn’t passed.

The entry point URL syntax is:

```
<protocol>://<host>:<port>/fscmUI/faces/ConfigLaunch.jspx
```

Example:

In the case of an error either during or while starting a configuration session, Configurator navigates to the ReturnUrl with an ExitStatus of error. The host application can perform a REST GET operation on the Initialization Parameters resource, with the ParameterId passed as part of the ReturnUrl, to obtain the full error message details.

Integration Tasks

To prepare to call the runtime configurator from a host application, and to prepare and process input and output payloads, follow these high-level steps.

1. Register the external application:

   In the Setup and Maintenance work area, go to the following:
   - Offering: Product Management
   - Functional Area: Product Spoke Systems
   - Task: Manage Trading Community Source Systems

   Then create or register your external application as a source system.

   When an application is registered, you can define applicability parameters for modified user interfaces, in the Configurator Models work area, so that those user interfaces are displayed when the application calls Configurator.

2. Enable the allowed origins white list:

   This step is required if launching the runtime configurator in an IFrame. By default, Oracle Fusion applications will not render contents in an IFrame except when called from the same origin. You can use a configurable white list mechanism to specify the list of allowed external origins that can render contents in an IFrame. You control the allowed origins by setting the profile option `ORACLE.ADF.VIEW.ALLOWED_ORIGINS`, using the Manage Administrator Profile Values task in the Setup and Maintenance work area.

   This step isn’t required for launching the runtime configurator in a browser window.

3. Call the runtime configurator UI, in an IFrame or browser window, using the public entry point URL:

   The public entry point URI is `/fscmUI/faces/ConfigLaunch.jspx`. The URL parameters `ParameterId` and `ReturnUrl` are required.

   The following steps are required for calling the runtime configurator.

   a. Prepare an initialization payload.

      Prepare a REST payload using the JSON format.

      Example, showing the minimum required payload:

      ```json
      {  "CallingApplicationCode":"CZ",
       "Line":{"InventoryItemNumber":"AS100","RequestOn":"2017-01-22T09:09:28-0700","InventoryOrganizationCode":"V1"}
      }
      ```

   b. Post the payload to the Initialization Parameters resource.
Send the payload to the Initialization Parameters REST resource using the POST operation, using the following URI

`/fscmRestApi/resources/latest/initializationParameters`

You can replace `latest` with the string that specifies the desired installed release of Oracle Supply Chain Management Cloud, such as `11.13.18.02`. The string `latest` is a synonym for the latest installed release.

Example, using the cURL command-line tool:

```
curl 'https://acme.cloud.enterprise.com:7000/fscmRestApi/resources/latest/initializationParameters' \  
   -i \  
   -X POST \  
   -H "Content-Type: application/json" \  
   -u myuserid:mypass \  
   -d '{ "CallingApplicationCode": "CZ",  
   "Line": "{"InventoryItemNumber": "AS100", "RequestOn": "2017-01-22T09:09:28-0700",  
   "InventoryOrganizationCode": "V1"}"}'
```

For the payload request shown in the example, the response from the Initialization Parameter resource is like the following:

```
{
   "ParameterId": 300100116074154,
   "CallingApplicationCode": "CZ",
   "Header": null,
   "Line": "{"UnitQuantity":1,"InventoryItemNumber":"AS100","RequestOn": "{2018-01-11T09:28-0700",  
   "InventoryOrganizationCode":"V1","InventoryOrganizationId":204,
   "InventoryItemId":300100016171847},
   "CustomParameters": null,
   "ManualPriceAdjustments": null,
   "PageTitlePrefix": null,
   "ExpiredFlag": false,
   "ErrorMessageText": null,
   "TerminalActionCaption": null,
   "ValidationCannotAlterFlag": true,
   "ValidationFailFastFlag": true,
   "links": [
   {  
   "rel": "self",  
   "href": "https://node001.my.srvr.com:7021/fscmRestApi/resources/latest/initializationParameters/300100116074154"},
   {  
   "rel": "canonical",  
   "href": "https://node001.my.srvr.com:7021/fscmRestApi/resources/latest/initializationParameters/300100116074154"}  
   ]}
```

c. Extract the ParameterId.
The external application must process the response payload from the POST operation to obtain the parameter identifier ParameterId generated by the Initialization Parameters REST resource when initializing the configuration. The value from the example is:

"ParameterId": 300100116074154

d. Call the public entry point URL.

The value obtained for ParameterId is passed as a URL parameter value when calling the public entry point ConfigLaunch.jspx. The resulting example URL, which launches the runtime Configurator user interface in a session that configures the products specified in the initialization payload, is like:

https://acme.cloud.enterprise.com/fscmUI/faces/ConfigLaunch.jspx?
ParameterId=300100089663945&ReturnUrl=https://acme.cloud.quote.com/context/processResults.jsp

4. The end user interactive configuration session occurs at this point.

The end user can complete an interactive configuration session and exit the configurator by using one of the following actions, by clicking the corresponding button in the runtime configurator user interface:

- **Cancel**: The end user can simply cancel the configuration session. A warning dialog is displayed to the end user about losing the changes made to configuration. An exit status of cancel is specified in the ReturnUrl parameter to indicate to the host application that there are no changes to process upon return.

- **Save for Later**: The end user decides to save the configuration to finish it at a later point in time. An exit status of save is specified in the ReturnUrl parameter to indicate to the host application that there are changes to the configuration. The host application can use the configuration identifiers (ConfigHeaderId with ConfigRevisionId) to retrieve the configuration details.

- **Finish** or **Finish and Review**: The end user completes the configuration, by implicitly running the autocompletion process, and returns to the host application. An exit status of save is specified in the ReturnUrl parameter to indicate to the host application that there are changes to the configuration. The host application can use the configuration identifiers (ConfigHeaderId with ConfigRevisionId) to retrieve the configuration details.

5. Process the configuration results:

When the end user completes an interactive configuration session and exits, Configurator navigates to the ReturnUrl that was provided when launching the configuration session, and appends the configuration identifiers and exit status as URL parameter values. The external application must use the URL parameter values to determine the exit status and take the necessary action to navigate to its appropriate page.

- The ExitStatus parameter indicates the exit method. Possible values are save, cancel and error.
- When the exit status is save, the following configuration identifiers are included:
  - ConfigHeaderId: Identifier for the configuration.
  - ConfigRevisionId: Identifier for the configuration revision.

When used together, the combination of ConfigHeaderId and ConfigRevisionId uniquely identify a configuration. Example:

q=ConfigHeaderId=300100112392208;ConfigRevisionId=300100112392209

- If the ExitStatus is save, then the external application must obtain the configuration results by passing URL parameter values to the Configurations REST resource. The URL for the Configurations REST resource is:
The required URL parameters are:

```
q=ConfigHeaderId=000;ConfigRevisionId=000
```

Example URL that queries the Configurations REST resource for a specified configuration and returns all its configuration lines, including option class lines (`OutputMode=full`), and transactional item attribute lines (`expand=ConfigurationLines,ConfigurationLineAttributes`):

```
https://acme.cloud.enterprise.com:7000/fscmRestApi/resources/latest/configurations?q=ConfigHeaderId=300100112392208;ConfigRevisionId=300100112392209;OutputMode=full&expand=ConfigurationLines,ConfigurationLineAttributes
```

This REST resource produces a JSON document in the following structure for one configuration:

```
Configuration
  ConfigurationLines (collection)
  ConfigurationLine [0...n]
  ConfigurationLineAttributes (collection)
  ConfigurationLineAttribute [0...n]
  ConfigurationMessages (collection)
  ConfigurationMessage [0...n]
```

The external application must process the JSON document to create quote or order lines from it.

**6. Reconfiguration flow.**

An external application can perform a reconfiguration of an existing configuration by following the same integration steps as for a new configuration, but must include the configuration identifiers and configurator path information for the root line in the initialization parameters payload. Example, showing the minimum required payload:

```
{
  "CallingApplicationCode":"ABC_CPQ",
  "Line": "{\"InventoryItemNumber\":\"AS100\", \"InventoryOrganizationCode\":\"V1\", \"RequestOn\": \"2016-04-07 13:00:00\"},
  "ConfigHeaderId":300100112392208, "ConfigRevisionId":300100112392209, "ConfiguratorPath": \"987654321\"
}
```

Based on the exit status, an external application can obtain the configuration results using the configuration identifiers. Instead of deleting and recreating quote or order lines, an external application can use the configuration line identifiers to synchronize the existing lines. The important point to note is that Configurator returns the same line identifiers for reconfiguration flows as for new configuration flows. However, the lines in the configuration aren’t ordered, and you can’t provide a sorting order.

**Related Topics**

- REST API for Oracle Supply Chain Management Cloud
- Initialization Parameters REST Endpoints
- Configurations REST Endpoints
- Configurator REST Use Cases
- Product Spoke Systems: Explained
Pricing with Configurator

You can display the prices for standard items and connected items in your user interface. Pricing is performed within Configurator using Oracle Fusion Pricing.

When item prices are displayed during a configuration session, the prices for items or services appear in the running summary pane and the configuration summary page. Connected items and services are displayed as root-level lines in the configuration summary. The price for the connected items and services is included in the total net price for the configuration.

The requirements for using pricing with Configurator are:

- You are running Configurator along with Oracle Fusion Order Management.
- Your items are on a price list, managed in the Pricing Administration work area.
Glossary

argument binding
In a configurator extension rule, the binding of a method argument to a model node or literal value.

autocompletion
A process that automatically provides values for all unbound variables in a configuration, and thus provides values for all possible features in the model that don’t yet have a value assigned during the configuration session.

base node
The model node associated with a configurator extension rule. The actions performed by the extension rule refer to the base node.

baseline
A released version of a workspace participant that becomes the basis for new drafts in other workspaces that are unreleased.

draft
Version of a workspace participant while it is in development, representing changes to the baseline version. When a workspace is released, drafts with changes become versions, and drafts with no changes are removed from the workspace.

effective start date
Date on which the draft modifications to models in a workspace take effect after the workspace is released.

event binding
In a configurator extension rule, the binding of an event to a method defined in the rule text.

event binding scope
In a configurator extension rule, the execution scope of an event binding.

instantiation scope
In a configurator extension rule, specifies how the rule is to be instantiated relative to associated base node instances.

model
Short name for a configurator model, which is a hierarchical structure in Oracle Fusion Configurator that represents a related set of product structures in Oracle Fusion Product Hub.
participant (workspace)
Object contained in a workspace. Draft modifications to participating models become effective when the workspace is released.

refresh
An operation that updates the data in a snapshot to reflect changes in the corresponding product item.

release (workspace)
Making the modifications in a workspace available to run time users, on the effective start date, and preventing further modifications to the drafts in the workspace.

snapshot
A copy of production data that is imported from the Product Information Management work area into the Configurator Models work area at a point in time. Items, item classes and value sets are imported as snapshots.

tip version
The released version of a workspace participant that has no versions defined with a later effective start date. The tip version isn't necessarily the most recently released version, since the release date of a version is independent of its effective start date.

token
The result of translating characters into recognizable lexical meaning. All text strings in the input stream to the parser, except white space characters and comments, are tokens.

version
Sequential revision to a versioned workspace participant. Snapshots are not versioned. A version becomes the baseline for an object when the workspace is released.