



Exercise 7

Configure the context

INTRODUCTION

This exercise gives you a glance into the power of UI context configuration within the Backoffice framework.

Like all the previous exercises, the changes you make in this exercise will further improve the *Book Management* cockpit. First, you will make the Book Details widget's view more configurable by creating a custom configuration data type (XML and Java) that specifies which `BookModel` properties are to be displayed in the widget view – currently, these properties (Title, ISBN10, ISBN13, and Description) are hard-coded into the controller.

Through a simple exercise, you will see how pre-existing bits of configuration can be leveraged to great effect – you can reuse the existing configuration of widgets instances as the basis for new, context-sensitive, *alternate* configurations for the same widgets. As you will see later, that will save you significant effort.

What you will learn

By the end of this exercise, you'll know how to create a new custom configuration XML type (data structure) from scratch. You'll be able to use the new custom data structure to provide configuration to your widget controller thus customizing your widget's view.

Leveraging existing UI configurations in Backoffice can be quite handy. In the second part of the exercise, you'll learn how to use the powerful merging mechanism of the Backoffice framework to, in effect, inherit or override large chunks of UI configuration, saving yourself vast amounts of work.

What you will need to do before starting

- Complete the previous exercise
- Start the SAP Commerce server
- Enable the “Redeploy” button (hot deployment)
- Run the preparation script of exercise 7
- Have a terminal/console window open at the platform directory.

INSTRUCTIONS

Part 1 – Define and Use a New, Custom Widget-Configuration XML Type

1. Each widget *type* can have its own, unique XML format, determined by that widget's developer, for providing it UI-configuration info. Such XML fragments become the bodies of `<context...>` entries placed inside `<extensionName>/resources/<extensionName>-backoffice-config.xml`, where each `<context...>` entry is intended for a specific widget type.

We want to use this approach to configure our *Book Details* widget UI, letting us specify which Book *properties* to display, and what *label* to use for each property. As the widget controller generates view content to output, it will obtain this configuration XML from Backoffice, convert the XML into a Java object (for convenience), and make use of this configuration info when generating the view.

The JAXB (Java Architecture for XML Binding) approach fits this purpose well. In JAXB, we usually begin by creating an XML Schema Definition file (XSD) that specifies the XML structure we want for our widget's configuration. To save you time and effort, the preparation script has provided this file for you:

```
${HYBRIS_BIN_DIR}/custom/bookstorecustombackoffice/backoffice/resources/schemas/config/details.xsd
```

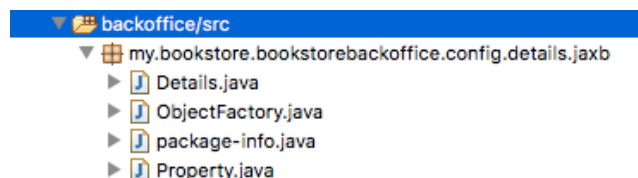
Open this file and have a look at it. Later in the exercise, you will need to be familiar enough with this structure to create an appropriate XML body for your `<context...>` entry.

2. JAXB is used to generate Java classes (based on our XSD), that handle the XML-to-Java conversion. The Eclipse IDE gives us a convenient way to access this functionality, which you've already seen in the slides.

In Eclipse, right-click on the .xsd file and choose *Generate → JAXB Classes* (if you do not see a JAXB option, refer to the grey (i) info box that follows). Once in the *Generate* dialog, provide the following info:

- **source folder:** `bookstorecustombackoffice/backoffice/src`
- **package:** `my.bookstore.bookstorecustombackoffice.config.details.jaxb`
- leave all of the other fields blank

Once done, you should have a new Java source-code package (placed in the **source folder** you specified above) with four new classes as shown the following image:



If, after clicking *finish* button on the JAXB class generation wizard, you get an error saying

Error: Could not find or load main class com.sun.tools.internal.xjc.XJCFacade

, you'll need to configure the installed JREs in your IDE. This error indicates that your Eclipse IDE is using a JRE for its Java libraries, whereas for JAXB to be able to generate Java code, a JDK is required.

If you don't know how, follow the instructions provided for you in a separate instruction document called "Adding JDK" that's located next to this instruction file.



What if you don't see the **JAXB Classes...** option under **Generate**? Well, then you are probably using a *Java* edition of Eclipse instead of a *JEE* edition (which is a much larger download). Here's how to add JAXB support to Eclipse. ☺

It's rather simple:

1. Go to the **Help** menu in your IDE
2. Select **Install New Software...**
3. In the **Works with:** field, check the dropdown to see if it already has an entry for the version of Eclipse we provided you – if so, click on it:
[Oxygen] - <http://download.eclipse.org/releases/oxygen>
If not, manually type into this field:
<http://download.eclipse.org/releases/oxygen>
4. Then, in the resulting package list, locate the **Web, XML, Java EE and OSGi Enterprise Development** node, expand it, and “check” ONLY the node that includes JAXB (e.g., Dali Java Persistence Tools – EclipseLink **JAXB** Support)
5. Continue the wizard until this package is installed
6. Restart your IDE (when eventually prompted) and you are good to go!

3. You can also use built-in tools of the Eclipse IDE to create a convenient *sample XML instance document* based on the new schema. This XML file would contain a skeleton *XML data instance* structured according to the XSD. Having this sample-XML structure is convenient, because you can copy-and-paste it into wherever you need an “instance” of this XML type – you would then simply insert your own data values within the XML tags. Let's create this sample XML file now...

Right-click on `details.xsd` and then choose **Generate → XML File ...** which opens a wizard. In the first step, name the file `sample-details.xml` and place it in the same folder as `sample.xsd`. In the second step, make sure that every box is ticked, as in the image below. By doing that, you tell the XML generator to create sample XML that contains ALL of the schema elements and their attributes.

Select Root Element

Select the root element of the XML file.

Root element:

details

Content options

- ☒ Create optional attributes
- ☒ Create optional elements
- ☒ Limit optional element depth to: 2
- ☒ Create first choice of required choice
- ☒ Fill elements and attributes with data

Namespace Information

Prefix	Namespace Name	Location Hint
tns	http://www.hybris.com/b...	details.xsd

Add...

< Back Next > Cancel Finish

4. After you've completed the wizard, open the XML file and have a look at the `<tns:details>` root element within. You can use this `<tns:details>` element with multiple filled-in `<tns:property>` child elements later-on in your configuration.

Open `bookstorecustombackoffice/resources/bookstorecustombackoffice-backoffice-config.xml` and locate the TODO XML comment for this exercise step. Create a new `<context>` element (Hint: using ctrl-Space in Eclipse's XML editor is a huge convenience) and assign it an appropriate context **component** attribute value of **bookdetailsConfig** so that the Book Details widget controller has a way of specifying what configuration info to retrieve from Backoffice. Specify (via XML attributes) that this context configuration component applies to items of domain type `Book`.

Have this new `<context>` element wrap around the XML configuration content that you want your Book Details widget to retrieve (later), so have this `<context>` element's body be an "instance" of the new XML configuration type you created earlier: copy-and-paste the `<tns:details>` element (without the `<?xml...>` header!) from your `sample-details.xml` and use the generated `<tns:property>` tag as a sample. (This XML element has a **name** attribute and a **label** attribute). Use the following attribute values and save your work:

name (Book attribute) (Exact Match)	label (Display Label) (arbitrary)
name	Title
authors	Authors
ISBN10	ISBN10
ISBN13	ISBN13
description	Description

Now we need to get `BookDetailsController` to retrieve this UI configuration info from Backoffice during view generation. In order for a widget controller to request only relevant and correctly formatted configuration info from Backoffice, it needs the correct UI configuration **component** name.

You have two ways to address this issue.

- One way is to hard-code the configuration component's name within the controller, but this is undesirable because there is no good way to reveal or modify the component name being used by the widget.
 - A second, preferred choice is to add a new *setting* to the widget that holds the component's name, thus exposing its value and making it modifiable. Furthermore, the widget definition can conveniently assign (and expose) a default value.
5. Open the `definition.xml` file for your widget and add a new setting called `propertiesConfigContext`. Specify the appropriate type for the values it may contain (i.e., "String"). Do not specify a default-value here. Save your work.

Now, in `BookDetailsController.java`, define a new `java.lang.String` constant to hold the *name* of this new `propertiesConfigContext` setting (the existing constant `WIDGET_SETTING_CONFIG_CONTEXT`, for the pre-existing widget setting named `configContext`, is a perfect example). To be clear, this constant holds the *name* of the *setting*, which is essentially hard-coded, but NOT the eventual NAME of the UI configuration component. The widget will access this setting by name to obtain the setting's *value*, which will be the name of the context configuration component to retrieve (i.e., the context configuration component you added earlier to `bookstorecustombackoffice-backoffice-config.xml` named ***bookdetailsConfig***).

6. Inside `bookstorecustombackoffice-backoffice-widgets.xml`, add a new setting value to the Book Details Widget, where “key” will be the name of the new setting and “value” will be the component name of the new context element you created earlier in `bookstorecustombackoffice-backoffice-config.xml`. Specify its “type” to be `String`. Note, we *could* have also done this in Application Orchestrator by editing this widget instance's details, but then we would have had to copy the changes into this file, anyway, so this saves a step.
7. Before this exercise, we had a method called `loadConfiguration(String typeCode)` inside `BookDetailsController` that loaded our configuration, but it worked specifically and exclusively with the `Base` JAXB type.

To enable loading of the new configuration type as well, we'd need to either modify this method so that it could work with either configuration type, or define a second method, e.g., `loadDetailsConfiguration(String typeCode)`, that works specifically with the `Details` type. We chose the first approach and have already modified the method for you via the preparation script of this exercise. Refer to this method as you continue reading...

To enable the `loadConfiguration()` method to work with both `Base` or `Details` configuration types (i.e., the JAXB types corresponding to XML configuration types), we changed this method's signature to handle types generically in a parameterized way. Look at this method now – it now has *three* parameters: the first one is for specifying what configuration component **type** specifier is “in context”, the second one is for specifying the configuration **component** name to filter on, and the third one to dynamically specify the return type of this method – in our case, a JAXB type.

In case you're wondering...

The method's third parameter, `configurationType` is of type `java.lang.Class`, and enables the Generic Methods feature of Java Generics (Java 8). This technique allows you to specify the method's generic TYPE parameter *dynamically* at the moment of invocation (at runtime, as opposed to compile time).

Notice an example of how this method is called from the `populateImgDiv()` method:

- In this method invocation, the third parameter is the `java.lang.Class` reference, `Base.class`, but because the actual reference is known at compile time, the method's invocation's return type (in the Java editor) “magically” changes to `Base`.

- You are about to add a second method call, such that the third parameter is the `java.lang.Class` reference `Details.class` and “magically”, this invocation will have a return type of `Details` even though both method invocations use the same method definition.

8. Modify the preparation-provided lines of code in the `populateInfoDiv()` method (at TODO comment 7.8) to load the second configuration (of JAXB type `Details`) into the (currently commented-out) `propertiesConfig` variable.

9. Further down in the `populateInfoDiv()` no-arg method, find the line of code that calls `populateInfoDiv(BookModel, Base)` (TODO comment 7.9). It currently passes the selected book (i.e., a `BookModel` object) and the `baseConfig` object. Modify this method call so that instead of passing the `baseConfig` object, we pass the new `propertiesConfig` object. Of course, now you must change the method signature of `populateInfoDiv(BookModel, Base)` to `populateInfoDiv(BookModel, Details)`.

10. Currently, the view/UI elements that display the properties in the Book Details widget's view are being generated in a hard-coded manner within the `populateInfoDiv()` method, i.e., the choice of `BookModel` properties to be displayed (and their labels) are hard-coded into this method. Comment-out these lines of code and replace them with the following approach...

Have this `populateInfoDiv(BookModel, Details)` loop through the passed-in `Details` object's `Property` elements to obtain the values of the "name" and "label" to add to the view, thus generating the view. (Note: a quirk of the auto-generated JAXB class is that the method named `getProperty()` method of `Details` returns `List<Property>` instead of an individual `Property` object, as the name implies.)

Within your loop, you can use a helper method we provided:

```
getPropertyValue(String propertyName, BookModel book)
```

to obtain the display values for the `BookModel` properties, instead of using the usual attribute-specific getter methods.

11. Go to your console/terminal and run `ant build`. Once done, go to the Backoffice and do the hot deployment routine: press Redeploy and refresh the Backoffice app.

The Book Details widget should appear as it used to, even though it accomplishes this differently. Change the configuration inside `bookstorecustombackoffice-backoffice-config.xml` and see how that affects the Book Details widget's view. (E.g. remove one of the properties in the XML, perform an Application Orchestrator *Reset to Defaults* on `cockpit-config.xml`, and check that the removed property is no longer displayed in the UI.)

Part 2 – Reuse Existing Configuration

12. Compare the editors for Products in the Administration cockpit and Books in the Book Management cockpit.

You will see that, in the Administration cockpit, the attributes are arranged into multiple tabs as in the following image. In addition, note that regardless of which tab is selected, the same few attributes are always displayed in the topmost section, labelled “Essential” (internally, it is called the *essentialSection*).

The screenshot displays the 'The eCommerce Murderess' - Bookstore Product Catalog : Online editor. The interface includes a title bar, a toolbar with icons and 'REFRESH'/'SAVE' buttons, and a tabbed interface with 'PROPERTIES' selected. The 'ESSENTIAL' section contains fields for Article Number (6824238759), Identifier (The eCommerce Murderess), Catalog version (Bookstore Product Catalog : Online), and Approval (approved). The 'VALIDITY PERIOD' section contains fields for Online from and Online to, both with calendar icons.

This configuration is defined via a context component named *editor-area* and is applicable when commerce type *Product* is involved. This context component is defined in the file `platformbackoffice-bacoffice-config.xml`. As the filename implies, it exists in the *platformbackoffice* extension.

You can apply the same configuration to your Book Management cockpit's *editorArea* instance by changing one of its widget settings. The *bookstorecustombackoffice-editor-area* widget instance has a setting called *editorAreaConfigCtx* that currently specifies *bcb-editor-area* as the name of the configuration component for the widget's controller to request.

Before you change this setting, let's first take a look at the *bcb-editor-area* context component which is defined in your `bookstorecustombackoffice-backoffice-config.xml` file. (Search for the entry for configuration component *bcb-editor-area*, for type *Book*.)

Note that this configuration essentially specifies that “for an *editorArea* widget that reads in a configuration component named, *bcb-editor-area*, its *books.essential* section, should display the *Book* attributes: *code*, *name*, *catalogVersion*, *approvalStatus*, *picture*, and *description*.”

Now clear-out the *value* of the widget setting *editorAreaConfigCtx*, exit the Application Orchestrator, and observe that your *editorArea* now displays a few basic tabs when it re-renders (iterate to a different book to force *editorArea* to re-render, with its new configuration). This is the default way the *editorArea* widget configures itself when no external configuration info is given.

Now change the value of the widget setting *editorAreaConfigCtx* to *editor-area*, exit the Application Orchestrator, and observe that your *editorArea* now displays many tabs when it re-renders (iterate to a different book to force *editorArea* to re-render, with its new configuration). In fact, this is the same configuration that the *editorArea* widget in the Backoffice Administration Cockpit uses. To make this change persistent (so that it does not reset every time you log in), change this widget setting in `bookstorecustombackoffice-backoffice-widgets.xml` as well.

13. Let's take this a step further. In our *bcb-editor-area* context component, we displayed the attribute, *picture* (the book's cover image), in the first section of our *editorArea*, but the *editor-area* context component does not do this. Since book-cover images are important to book managers, let's display this additional *picture* attribute in the *essentialSection* of the editor area, if the "current item" is a Book, and not just a Product. In other words, the configuration component *editor-area* will be configured *one way* if the "current item" is a Product, and a slightly different way if the "current item" is a (child type) Book.

The following XML snippet shows how the *essentialSection* of *editor-area* is currently configured for displaying a Product. This configuration context entry was specified in the *platformbackoffice*

`platformbackoffice-backoffice-config.xml`

```
<context merge-by="type" parent="GeneralItem" type="Product" component="editor-area">
  <editorArea:editorArea xmlns:editorArea="http://www.hybris.com/cockpitng/component/editorArea">
    <editorArea:essentials>
      <editorArea:essentialSection name="hmc.essential">
        <editorArea:attribute qualifier="code"/>
        <editorArea:attribute qualifier="name"/>
        <editorArea:attribute qualifier="catalogVersion"/>
        <editorArea:attribute qualifier="approvalStatus"/>
      </editorArea:essentialSection>
    </editorArea:essentials>
    ...
  </editorArea:editorArea>
</context>
```

extension:

In your `bookstorecustombackoffice-backoffice-config.xml` file, locate the TODO comment associated with this step, and create a new "*editor-area*" `<context...>` component entry (i.e., it has the same component name so that it can merge with the one displayed above), but apply it to the type Book whose parent is Product. The element's structure will be the same as this one (to "align" the elements to be merged), except that your entry's *essentialSection* element will only contain the attribute being added or merged-in, i.e., *picture*.



For a complete description of the Editor Area Widget, have a look at its [documentation](#) on SAP Hybris Help. You can also search for Available Widgets in <http://help.hybris.com>.

Once you're done, save your work, then log out of Backoffice and then log in again to trigger the configuration reset. Go to the Book Management cockpit and open a book to edit it. If the editor looks like the image below, that means you've completed the exercise successfully. Note: hovering your cursor over the tiny thumbnail image in the Image field will pop-up a larger image.

Title: "Aspects of the Theory of Sin Tax" - Bookstore Product Catalog : Staged

REFRESH

SAVE

PROPERTIESATTRIBUTESCATEGORY SYSTEMPRICESMULTIMEDIAVARIANTSEXTENDED ATTRIBUTESREV

ESSENTIAL

Article Number

1274653819

Identifier

Aspects of the Theory of Sin Tax


Catalog version

Bookstore Product Catalog : Staged

Approval

approved

Image

 300Wx300H/generic-cover.jpg - Bookstore Product ...

VALIDITY PERIOD

Online from

Online to

Congratulations, you are now ready to work on the rest of the exercises.