Exercise 9

Create your own widget

INTRODUCTION

In this exercise, you will create a new *widget definition* and then add an instance of it to your Book Management cockpit. The preparation script of this exercise will take care of several of the necessary tasks for you, mainly to save time. You will be filling in the blanks to complete the widget-creation process.

The new widget will be a simple *Book Details Widget* that displays a few details of a chosen book. We will add an instance of this new widget definition to the right pane of your cockpit's main area.

What you will learn

You will get familiar with different components that comprise a widget. You will learn how these components work together behind the scenes to function as a single Backoffice widget.

What you will need before starting

- Complete the previous exercise.
- Enable hot deployment.
- Run the preparation script for this exercise and refresh your *bookstorecustombackoffice* project.

INSTRUCTIONS

1. First, let's create the definition of the new widget. As part of the preparation, a *widget definition* file has already been partially created for you. The file is located under

   `${HYBRIS_BIN_DIR}/custom/bookstorecustombackoffice/backoffice/resources/widgets/bookDetails`

   Open `definition.xml` and complete the definition using the following settings:

Property	*Value or Tag Attribute*
name	Book Details
controller	`class="my.bookstore.bookstorecustombackoffice.widgets.bookdetails.BookDetailsController"`
view	`src="bookDetails.zul"`
input socket	`id="selectedBook" type="my.bookstore.core.model.BookModel"`

   By setting these properties, respectively, you're assigning the new widget definition a:

   - user-friendly ***name*** for this widget *type*. The technical, unique ID for this widget definition has been set for you in the root element of this definition (i.e., `<widget-definition>`) via its id attribute. Our preparation script set the widget definition ID for you – its value is *my.bookstore.bookstorecustombackoffice.widgets.bookDetails*. (NOTE the naming convention looks a lot like a Java class name, except that ***bookDetails*** starts with an **initial lower-case** letter.)
   - ***controller*** in the form of a Java class whose main job is to generate or modify the widget's view and handle events
   - ***view*** file that defines the initial state of the widget view for the controller to modify
   - ***socket*** that will be used by the widget and its controller to receive `BookModel` objects. The arrival of each object at the widget's input socket will serve as a trigger for our widget controller to generate or modify its view representation so it can display some of the attributes of the received `BookModel` object). We've decided that receiving a **`null`** on this socket should serve as a trigger for our widget controller to *clear* its view.

   NOTE: `definition.xml`, also sets a `<keywords>:` "Book" to aid looking it up.

2. Open the view file that you assigned to your widget, `bookDetails.zul`. It's been created by the preparation script and is located next to the definition file.

   Uncomment the block of ZUL components (normally, you would have written this structure of components from scratch). Familiarize yourself especially with the two inner **`<div>`** elements (i.e., with ***id***s *imgDiv* and *infoDiv*).

   This composition of **<div..>** ZUL elements will serve as the fixed (non-changing, static, base) portion of this widget's UI output, onto which to attach dynamically generated UI fragments created/modified by widget-controller methods. At runtime, during widget-controller instantiation, the ZK framework will ***bind*** these two specific ZUL elements (*imgDiv* and *infoDiv*) to their "matching" widget-controller instance variables. Thes ZK 2-way bindings enable the controller to *access* and/or *modify* these ZUL components' state. For example, the controller can access the component's attributes/values (modified in the browser by a user), or access the component's child elements (as they are added or removed by JavaScript), OR the *controller* can make those same modifications *to* the component (via changes to the bound Java instance variable) thus dynamically changing its contents to produce the widget instance's output. (ZK will magically synchronize those changes back to the browser)

   The **<div..>** "sclass" (i.e. style class) attribute is the ZK framework's version of the standard "class" attribute in HTML. It allows a style class to be assigned to the component. Here, the style

class is defined inside `default.css` located next to the view file and is defined as the *style* source in the view file.

3. The final ingredient required to create your new *widget definition* is an implementation of the widget controller class whose fully qualified name you already specified. A mostly-implemented widget-controller Java class has already been created and copied-over for you by the preparation script and placed in the following Java folder & package:

   `${HYBRIS_BIN_DIR}/custom/bookstorecustombackoffice/backoffice/src/my.bookstore.bookstorecustombackoffice.widgets.bookdetails`
   (HINT: in Eclipse, the **bookstorecustombackoffice** project might *not* appear to contain a **src** subfolder of a **bookstore** folder – check toward the beginning of the list of project folders, grouped with the Java 'build' folders (i.e., ones whose folder icons hold a 'brown paper package'), for a *source folder* named **bookstore/src**)

   Open `BookDetailsController.java` and have a look at the implementation. Note that this class extends `DefaultWidgetController` and overrides its `initialize()` method.

If you have IDE errors in your BookDetailsController.java, such as it not being able to find the DefaultWidgetController and several other imports, this is due to the Backoffice project not including and exporting some of its libraries in its build path settings. Edit the build path for the Backoffice project, where you will find the missing jar files under web/webroot/WEB-INF/lib. Add the missing libraries as JAR files on the Libraries tab, and don't forget to also export the libraries on the Order and Export tab.

4. Inside `BookDetailsController`, we decided to define two methods, `populateImgDiv()` and `populateInfoDiv()`, for the widget to clear-out and/or (re) generate its dynamic UI markup output. If you look at these methods' implementations more closely, you'll notice that, although they dynamically create some UI content (look inside the private helper methods that these two populator methods invoke), these new ZUL elements are not yet "attached" to the widget's current view in any way, so they are not yet visible. Let's take care of this now…

   In the controller class, locate the TODO comment for this step (5.4a). Declare two new instance attributes whose names match the *id* of the `<div/>` tags we mentioned earlier (*imgDiv* and *infoDiv* in the .ZUL file), and whose type corresponds to the ZK Component Class for such a ZUL element (i.e., `org.zkoss.zul.Div`) – the mere presence of such a declaration (i.e., an instance attribute with name and type matching the ZUL element) will cause the ZK framework to "*wire"* (bind) the variable to the ZUL component, a technique which you may recall from the slides. Once this *wiring* is achieved, we can manipulate the state of this ZUL *div* component from Java code.

   Before proceeding, make ***sure*** that Java type of your new `imgDiv` and `infoDiv` instance attribute are of type `org.zkoss.zul.Div` (i.e., make ***sure*** that your IDE's Java editor automatically added an **`import`** directive for the *intended* **`Div`** class/type, `org.zkoss.zul.Div`, and not `org.zkoss.zhtml.Div`).

   Once the two instance attributes are declared, you can safely uncomment the lines of code that manipulate their corresponding ZUL elements (these lines are marked via TODO comments 5.4b through 5.4f, – uncomment them all now).

   Next, notice how `populateImgDiv()` and `populateInfoDiv()` always call their corresponding helper methods, `clearImgDiv()` and `clearInfoDiv()` even when merely *changing* our widget's view, because it's easiest to just clear-out each **<div..>** component first, then rebuild its contents from scratch. Have a look inside these populator methods as examples – now that you've wired your two attributes to the **<div..>** view components, the `clearImgDiv()`

and `clearInfoDiv()` methods are able to manipulate these elements by clearing out all of their child elements. Later in the `populateImgDiv()` and `populateInfoDiv()` methods, our widget's new dynamic content is created in the form of newly instantiated and composed ZUL objects attached as child elements to these **div**s.

To summarize: because you *wired* the controller's `divXyz` instance attributes to their corresponding ZUL view elements, the ZK framework applies this component's runtime Java changes to the widget's current display state in the browser: i.e., the ZK framework essentially performs a 2-way synchronization.

5. We're almost done. But one important piece is still missing. Yes, right! We need to trigger our controller "populator" methods to be invoked whenever a `BookModel` object appears on the ***selectedBook*** input socket, because so far, all of those socket input objects are being ignored. We need a method to become a "handler" for this socket's input ***events*** – this is specified via an annotation, which designates the method as an input-socket-event handler for the specified socket. We've already created the method for you: `handleSelectedBook(final BookModel book)`.

   Inside `BookDetailsController`, and right above `handleSelectedBook(..)` method (locate the TODO comment for this step, 5.5a), add an annotation that connects this method to the `selectedBook` socket. Refer to the slides to see an example of how an input-socket-event annotation is created. (Note that there's a static constant called `SOCKET_SELECTED_BOOK` – marked with a TODO comment for step 5.5b, – that holds the socket ID for this purpose (It's *best practice* to work with socket IDs via constants). Now, save your work.

6. You've completed the creation of your widget with some basic functionality, and now is time to build your system.

   Thanks to availability of hot deployment in the Backoffice framework, you don't need to stop your server if it is running. All you need to do is to run the following inside your platform directory from another terminal/DOS window:

   `ant build`
   (don't forget `setantenv.bat` *or* `. ./setantenv.sh` *to prep the environment if it's new)*

   When the build finishes, click on the **Redeploy** button on the top right corner of your Backoffice app and ***refresh* the page**.

7. The new widget, "Book Details", is now ready to use.

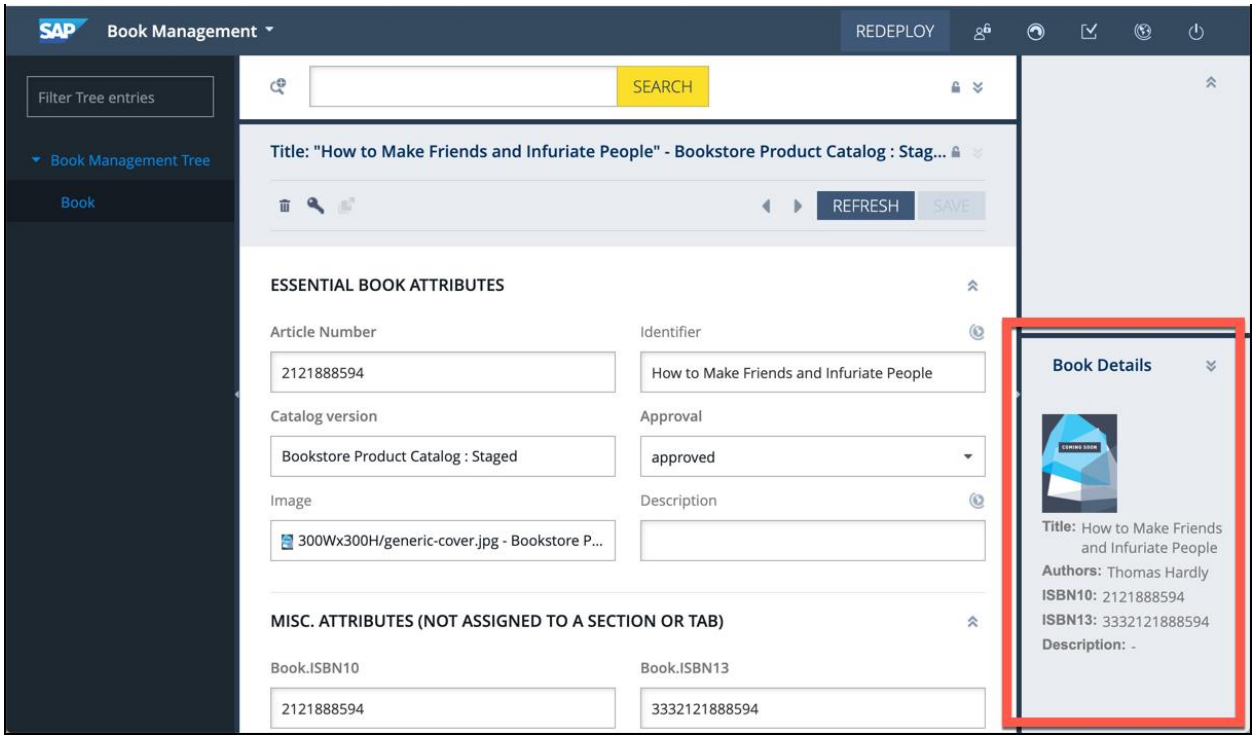
   Go to the Application Orchestrator and "reset `widgets.xml` to defaults" to load the changes made for you by the preparation script of this exercise.

8. Still in the Application Orchestrator, switch to the *Book Management* tab and, locate the ***bottom*** slot of the *Collapsible Container* widget that was added to the right pane by the preparation script. On this ***bottom*** slot, click the white **+** icon to add an instance of your new widget. In the *Book* category of the selection list, click on *Book Details*, then click on "Add & Close" (it's easier to connect it via the click-and-drag--chain-link technique). Once added, edit the settings of this Book Details widget instance (via its "pencil" icon): give this widget instance the sensible, unique ID of ***bookstorecustombackoffice-book-details*** so that you can find it later in widgets.xml, and click on "Close".

   Now, (via the chain-link click-and-drag technique) connect the appropriate output socket of the **bookstorecustombackoffice-result-list** *Collection Browser* widget instance to your new widget's input socket.

   Once you're done, exit Application Orchestrator.

9. To see if the new widget is working, try selecting a book in the “Book Management” cockpit. Then you should see a preview of the book on the right pane of the cockpit, similar to the following:



As we saw in previous exercises, changes that we make through Application Orchestrator can be lost just by a configuration reset (which, in turn can be triggered simply by logging out and back in, due to the way we set-up our development environment). To make these changes persistent, we would normally need to copy our changes (done via Application Orchestrator) from `widgets.xml` to our extension’s corresponding configuration file (in our case, we would copy our changes from `widgets.xml` into `bookstorecustombackoffice-backoffice-widgets.xml`). If you were to do this, you’d go into Application Orchestrator, open widgets.xml, select all, and copy the contents into a text editor. You would then search for all occurrences of your new widget’s ID (e.g., bookstorecustombackoffice-book-detail) to find the XML that adds this widget instance to the cockpit (with the widget’s settings), then find the XML entry for each connection to/from this widget. All of these fragments of XML would have to be copied into your extension’s `<extensionName>/resources/<extensionName>-backoffice-widgets.xml` file.

The required changes have already been placed into the file by the preparation script, but they are commented-out. All you need to do is to uncomment the two XML blocks that are indicated via *TODO* comments for this step. To verify that the changes are in place, please log out and log back into the Backoffice application to force a reset the configuration. If everything still looks the same, you've done the exercise correctly.

Congratulations, you are now ready to proceed to the next exercise!