Exercise 6

Connect your widgets

INTRODUCTION

In this exercise, you will create a new connection between the Book Details widget and the Collection Browser widget in your cockpit. The intention is to add a socket to our widget, allowing *part* of the widget's view (dynamic ZUL components for a book popup image) to be generated only when triggered to do so.

You will also trap global update/delete events to trigger updates in your widget's view if necessary.
To simulate a more real-life situation, starting with this exercise, we will give the task instructions in larger chunks rather than micro-steps with detailed descriptions.

What you will learn

You will learn how to use an adapter widget to facilitate communication between two sockets of incompatible datatype. Also, you'll use `@GlobalCockpitEvents` to trap global *objectsUpdated* and *objectsDeleted* CRUD persistence events.

What you will need before starting

- Complete the previous exercise.
- Enable hot deployment.
- Run the preparation script of this exercise (and refresh your IDE project, if necessary).
- Have a terminal/console window open at the platform directory.

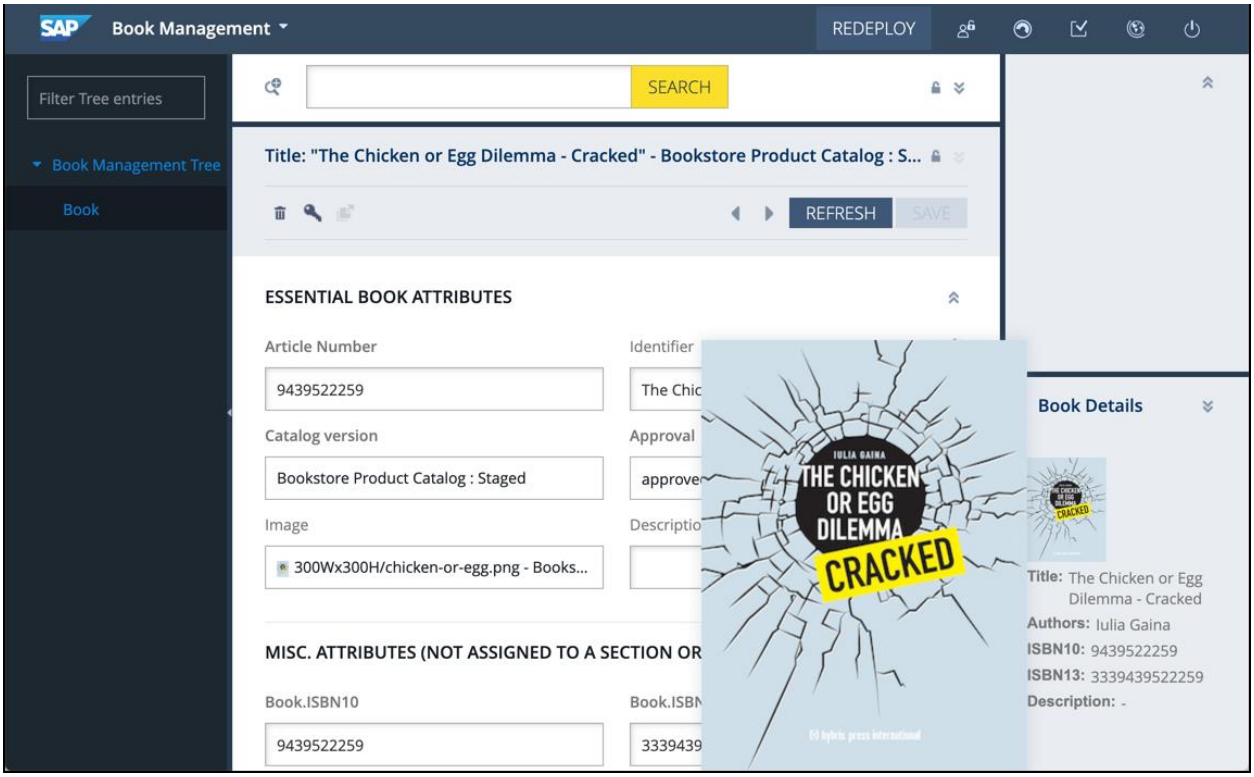
INSTRUCTIONS

1. Open and log in to the Backoffice application (or, if already logged in, log out and back in) to trigger configuration reset.
2. Press the Redeploy button, then refresh the page.

> The reason you need to perform a *Redeploy* is because the preparation script updated/overwrote your widget controller's .java file and performed an 'ant build'. But if you've started your server *after* running the preparation script for this exercise, then a *Redeploy* is not necessary.

3. Go to the *Book Management* cockpit and browse for books. Select one that has a preview image. Then click on the preview image in the Book Details widget. If a larger preview of the image (as in the following screenshot appears), it means that the preparation has been done successfully. If not (at first), try this:

   TIP: Due to subtle issues related to the way that Backoffice dynamically (re)creates its view façades, your widget's pop-up image will probably ***not*** work, if it is your first try after a re-deploy/refresh – but here's a "trick":
   Click on the image (expecting a pop-up but not getting one), then collapse and re-expand your widget's collapsible pane, then click on the image a second time – this should get the popup image to appear. What also seems to work is to click on the image, then click on the *Previous* or *Next* action button (i.e., the ones that you added to your cockpit's *Editor Area*), then click on the image a second time. If you are still having trouble, ask your instructor for help.



To better understand how this popup works, we need to look at what this exercise's preparation script has changed. Look inside your newly modified **`bookDetails.zul`**, (within **bookstorecustombackoffice**, filepath:

`/backoffice/resources/widgets/bookDetails/bookDetails.zul`. It now contains a new **<popup..>** ZK component with id ***bookImagePopup***. Our controller will manipulate this component via a ZK-bound instance attribute to add or remove child elements to affect that portion of the widget view.

4. Now, from the list of books in the result list, select a book that ***doesn't*** have a preview image, e.g. "Withering Heights".

   You should see an error box on the screen and a Java exception in the Terminal/DOS window, which occur because the popup is attempting to display a non-existent image.

   In the next few steps, we remedy this by adding a new widget socket that tells the widget whether or not there's an image available to preview, and an adapter widget to connect to this socket. NOTE, we **know** that there are much better approaches to solving this problem (in fact, this approach actually *creates* a subtle problem). But we needed an excuse to show you how to work with adapter widgets. So here goes…

5. Open `BookDetailsController.java`
   (recall, it's in project **`bookstorecustombackoffice`**, in build folder **`/backoffice/src`**,
   in Java package **`my.bookstore.bookstorecustombackoffice.widgets.bookdetails`**)
   and familiarize yourself with its `handleSelectedBook(..)` method, because in the next step, you'll be asked to implement a similar method.

6. In `definition.xml`, create a new input socket whose *id* is `allowPreview` of type `java.lang.Boolean` for the Book Details widget. For compatibility with the code provided for you in this training, please do NOT use the primitive type `boolean`, nor the short form of the class type name, i.e., `Boolean` (remember, when specified within XML, Java class names must be fully qualified).

   Now, in the controller class, use `handleAllowPreview(Boolean allowPreview)` as the signature for the new controller method which will handle inputs from this socket. To actually *cause* input-socket events to be handled by this method, remember to use the appropriate annotation to designate this method as the handler for the socket events that the ***allowPreview*** input socket will generate. (Suggestion: just before doing this, define a `private static final String` class constant to hold the socket's name – locate the TODO comments for this step in the Java code).
   Now, on to the body of the handler method…
   The preparation script has updated your controller class – it now includes a method called `populateImagePopup()` which handles generating the popup ZUL elements for you, and `clearImagePopup()` which clears-out all existing ZUL child elements of `<popup..>`.
   Now, the `handleSelectedBook()` method has a call to `populateImagePopup()`. The problem is that creating a popup makes no sense (and causes an error) if there is no image to display for a popup.

   Move this method invocation from `handleSelectedBook()` to `handleAllowPreview(Boolean)`. The key idea here is that:
   - the arrival of a `Boolean.TRUE` at the *allowPreview* socket should now trigger the *replacement* of `imgDiv`'s existing content with *new* content
     (i.e., invoke `clearImagePopup()` followed by `populateImagePopup()`)
   - the arrival of a `Boolean.FALSE` or `null` at the *allowPreview* socket should now trigger the *clearing-out* of `imgDiv`'s existing content
     (i.e., invoke `clearImagePopup()`)
   - HINT: Notice that in the above situations, `clearImagePopup()` is invoked either way
   - HINT: The `handleSelectedBook()` method should no longer invoke `populateImagePopup()`

Run `ant build` in a console and, if successful, perform a *Redeploy* and **refresh** your Backoffice app. This is necessary because the new widget definition and the modified controller need to be loaded and made available to the Backoffice app.

Recall that, once you're logged in, redeploying can be done by pressing the **Redeploy** button in your Backoffice app and then refreshing the page.

So far, with these changes, if you click on a book image, nothing happens. This is because creation of the popup now depends on socket input, and no objects ever come in through the socket… because nothing is connected to it. Let's fix that…

7. Using Application Orchestrator, ***try*** to connect
   `selectedItem` of `bookstorecustombackoffice-result-list` (i.e., the Collection Browser widget in our cockpit) whose socket's type at runtime will be `BookModel`
   -- to --
   `allowPreview` of `bookstorecustombackoffice-book-details` (the Book Details widget in our cockpit) whose socket's type is `java.lang.Boolean`.

   You will find that it cannot be done – there are no available sockets of this type in the Book Details widget. We need to perform some special steps (explained here)…

To facilitate connecting our two sockets with incompatible types, we will add an adapter widget in between them. A Condition Evaluator widget can do the job of taking in a `BookModel` object from `selectedItem` socket the Collection Browser widget, and processing it in order to send the desired `java.lang.Boolean` to the `allowPreview` socket of the Book Details widget
Use the Application Orchestrator to insert an instance of Condition Evaluator widget as an invisible child widget of the of `bookstorecustombackoffice-book-details` widget or its parent *collapsible container* widget (which does not have a user-friendly name) occupying the right slot of Book Management's border layout – toggle the visibility of the invisible widget children area via the slash-through-the-eye icon, then look hard for the black **+** icon within the yellow title bar, then *Add & Close* the widget.

TIP: if you are having trouble with "screen real estate" (e.g., if you are using a VM to perform the lab and the widgets are all crammed into the bottom-right corner), you can double-click on a container widget's title bar to display it in its own popup dialog in the center of the screen.

TIP: another "screen real estate" tip is that you can click-and-drag the vertical "divider bar" of your bookstorecustombackoffice cockpit's border layout left to make its "right" region/slot wider, or click-and-drag the horizontal "divider bar" of the Collapsible Container upward to make its "bottom" region/slot taller, or collapse the "middle" region/slot of the Collapsible Container.

Once the adapter widget is added, edit its settings to assign `bookstorecustombackoffice-preview-adapter` as the Widget ID – keep the settings dialog open for the next couple of steps…
Out of the box, this adapter widget's input socket (called ***input***) is of type <T> (i.e., a Java generic type "parameter" named T) and two of its three output sockets: one called ***true*** and one called ***false*** are also of type <T>. Its third output, called ***result***, is of type `java.lang.Boolean`. We can specify a widget's "socket type" for <T> so that we can match it to the desired input type, i.e., `BookModel`.
As was shown in the slides, add a new setting to the adapter widget – it must be called `socketDataType_$T` – the T comes from the generic "parameter" name – and make the

setting's ***type*** (i.e., the the type of value the ***widget setting***, itself, will contain) `java.lang.String`. Once created, set the widget setting's actual ***value*** to `my.bookstore.core.model.BookModel`. NOW this adapter widget instance's input socket type is able to be matched to incoming `BookModel` objects. Keep the settings dialog open a bit longer…

A `BookModel` object contains a reference to its preview image in its `picture` attribute. This adapter can use a SpEL expression to access the `picture` attribute directly and check if it is null. Also, you'll need to make sure that the input object itself is not null. The SpEL expression you need is:

```
#root != null ? picture != null : false
```

Set the ***expression*** setting of `bookstorecustombackoffice-preview-adapter` to the above SpEL expression. Now close the settings dialog and try again to make the two click-and-drag connections: one from the Collection Browser's ***selectedItem*** (output) socket to the adapters ***input*** socket, and one from the adapter's ***result*** (output) socket to the Book Details' ***allowPreview*** (input) socket.

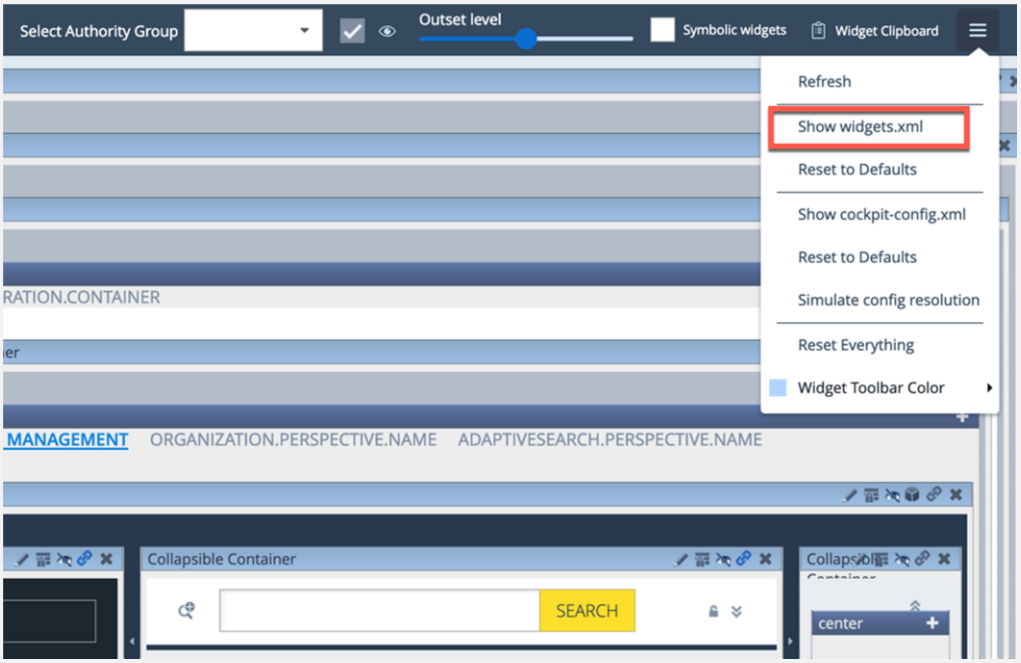
8. Exit Application Orchestrator and check what happens if you choose a book without any image. You should no longer see any error messages (neither in Backoffice or in the console output). Also verify that a preview still pops up if a book *does* have an image – again, it may be helpful to navigate through two or three books via the previous or *next* navigation arrows).

9. To maintain the new mashup, like previous exercises, transfer the changes you made in the mashup of the cockpit from `widgets.xml` (in the orchestrator) to `bookstorecustombackoffice-backoffice-widgets.xml`.
Note that apart from the widget instance definitions, there are also widget connections that you should transfer.

How to find the XML equivalent of your application (widgets.xml) configuration?

When you change your application configuration (mashup) in the application orchestrator, the changes are immediately translated into XML and saved into a SAP Commerce media object persisted in the database. This media object is, however, recreated every time the configuration is reset (by restarting your server, updating your commerce platform, or invoking a reset trigger).

A copy of this media item is directly accessible inside the application orchestrator, under the orchestrator's menu:



If you need only a particular piece of the configuration, you have to find that yourself manually inside `widgets.xml`. We suggest copy-and-pasting the whole body of widgets.xml into a text/xml editor of your choice, and then looking for the required configuration. In this case, you'd perform a search in the text editor for "bookstorecustombackoffice-preview-adapter".

10. We have created a *fairly useful* widget so far. But let's take it one step further! Let's make it responsive to changes that may happen to a Book while a book is on display by our Book Details widget. We want the Book Details view to refresh whenever the details of the book are *updated* (from any cockpit), or to be cleared when the book is *deleted* (from any cockpit), but neither of these CRUD operations would cause the *Collection Browser* widget to send a new `BookModel` object to our *Book Details* widget to trigger our widget to do something with its view, because the *Collection Browser* widget only sends an object when a *changing* its selection. We have to accomplish this another way…
Backoffice-wide CRUD operations on any domain object can be trapped as Global Cockpit Events. In our case, we are interested in an ***objectsUpdated*** or ***objectsDeleted*** event

concerning a `BookModel` object on display, and we want these events to trigger our Book Details widget controller to regenerate the widget's view with the new data, or clear the widget's view, accordingly.
Open `BookDetailsController.java` and define two new methods for handling these events. Name the methods `handleObjectsDeletedEvent()` and `handleObjectsUpdatedEvent()`, accordingly (find the TODO comments for this step in the Java code). Refer to the slides for help on how to add the proper annotation and method parameter to these method headers to trap Global Cockpit Events for CRUD operations.

Now, to implement the handler method bodies…
Keep in mind that these CRUD events fire anytime ANY domain object (or collection of objects) gets updated or deleted ANYWHERE within ANY COCKPIT of our Backoffice. Furthermore, every such event has, attached to it, a reference to every item that was updated, deleted, etc., by that CRUD operation. This so-called ***attached data*** can be used by our event handler method to determine whether or not anything even needs to be done. For example, if the book currently on display by our widget is ***not*** one of the items that was updated or deleted, our widget need not do a thing.

As an example of how to deal with this (and to save you loads of time), our preparation script has provided your controller class with a new helper method, `getEventObjectCurrentlyInView(CockpitEvent e)`, that works as follows:

- The method checks the passed-in CRUD event's *attached data* objects to see if any of them has a PK matching the object currently on display by our *Book Details* widget. If ***not***, then the method returns null, indicating that our widget need not do a thing – if ***so***, the method returns the *attached data* object corresponding to (matching) the one on display.

  To be clear, if our method returns something other than **`null`**:
  - When handling an ***objectsUpdated*** event, the *attached data* object that is returned will be a *new* instance of `BookModel` having the same PK as the instance on display by our widget, but with the latest values from the update operation. We must assign this updated object as our widget's ***selectedBook***, then trigger our widget to refresh its view.
  - When handling an ***objectsDeleted*** event, the *attached data* object that is returned will be a `BookModel` representation/snapshot of the item the moment it got deleted. This is of no real use to us, except to indicate that a matching object was on display by our widget, therefore our widget view must be cleared.

Implement these two handler methods now, with the help of invoking the `getEventObjectCurrentlyInView(CockpitEvent e)` method – HINT: for several subtle reasons, we recommend that each method should initiate a refresh by simulating the two socket events that normally trigger refreshes in our widget: first, invoke `handleSelectedBook(..)`, then invoke `handleAllowPreview(..)`, (of course, with appropriate parameters being passed-in).

(If you are really confused, or short on time, have a quick peek at the corresponding solution file in ***TrainingLabsTool*** for some insight – *but don't mistakenly edit this file, which is a very easy mistake to make, being that its filename is identical to the one you **should** be editing*.)

11. Run `ant build` in console and, if successful, then redeploy and refresh your Backoffice app.

12. Browse the list of the books and select one. In the editor area change the book's title (identifier), and then save it. The new title should immediately show up in the Book Details widget.
    Also try deleting a book – consider first cloning an existing book within the Backoffice Administration Cockpit's Editor Area first, assigning the new Book instance an arbitrary Article Number and Identifier, then delete the clone book. (Don't worry so much about messing up all

your data -- Worst case, we can always just perform a HAC update or re-initialize.) Upon deletion, the Book Details widget view should immediately get cleared.

Congratulations, you can now move on to the next module!