Exercise 8

Create custom editors and actions

INTRODUCTION

This document will guide you through the process of creating your own custom **editors** and **actions** within the Backoffice framework.
The Backoffice Framework provides about 30 OOTB[1] editors and 12 OOTB actions, which can be integrated into your custom interface and just need to be configured as required.
In this module, we will look at implementing entirely new **custom** editors and actions using the MVC paradigm that underlies all Backoffice Framework development. We shall see that editors and actions are subtypes of widgets, so this exercise will build on the structures and concepts you have previously learned.

What you will learn

- How to implement and configure a custom Editor
- How an Editor manages its view
- How an Editor obtains data, i.e., value to edit, user input, and configuration parameters
- How an Editor can exert control on the value
  (*during* value editing at the ZK-component level, distinct from Backoffice/SAP Commerce validation)
- Optional: How to implement and configure a custom action

What you will need before starting

1. A good understanding of how to develop editors and actions using the Backoffice Framework. This includes:
   - the model, view, and controller aspects of editors (from the slides)
   - Access to the relevant API class JavaDocs (https://help.hybris.com/1811/api/backofficeframework/index.html)
2. A completed previous exercise
3. A running SAP Commerce server
4. Hot deployment enabled
5. Run the preparation script for exercise 8.1
6. Have a terminal/console window open at the platform directory

INSTRUCTIONS

Part 1 - Custom Editor

Let's implement a custom Editor for the ***publisher*** attribute of the **Book** type. In our cockpit, the field Editor currently being used for ***publisher*** is the default editor for `java.lang.String`, which accepts all values without validation.
We want to create a custom attribute Editor for the ***publisher*** field that blocks numerical digits as the user types characters, to impose more active control than a standard validator. We also want the attribute Editor to interactively impose a limit on the number of characters *and* aid the user by displaying character-count statistics as the user types.
Don't forget to run the preparation target for exercise **8.1** before starting this part.

[1] Means, "Out Of The Box"

1. Before we begin, let's go over some changes our preparation script has made for you to set-up both parts of the exercise:

   Select a book within your Backoffice Book Management cockpit. You will see that now the ***Book.publisher*** and ***Book.rentable*** fields (which were previously visible *only* under the catch-all ***Administration*** tab) now appear in the topmost section of *every* tab of the *Editor Area* (because the preparation script has added both fields to the ***ESSENTIAL*** section, *if* displaying a Book).
   In fact, because both the *Administration* cockpit *and* the *Book Management* cockpit *both* use the ***editor-area*** configuration component to configure their *Editor Area* widgets, *and* because both Editor Areas are displaying a Book, they now both look the same!
   IMPORTANT CONCEPTUAL POINT: if our bookstore also sold *non-Book* Products, *their* appearance in the *Editor Area* would be unaffected – *Editor Area* widgets would continue to display those non-Book Products the same way as before, as mere Products.

   Look inside the preparation-script-updated `bookstorecustombackoffice-backoffice-config.xml` to see how this was achieved.

2. Now, let's observe the behavior of the default `java.lang.String` Editor currently in use for the publisher attribute within your *Book Management* cockpit's *Editor Area* widget: try modifying the *publisher* text field such that it contains numerical digits and click Save. Currently, there is no mechanism in use (not even validation) to prevent you from doing this. The same goes for this field's character count – it's currently unlimited.

   We *could* simply use the Commerce Platform's validation framework to enforce both of these restrictions – honestly, that would make the most sense, since these sort of restrictions are best implemented when bound to the ***Book.publisher*** attribute at the lowest possible level (i.e., the Persistence Layer) to ensure enforcement regardless of where or how the value change is attempted.

   But this exercise is about how to create a custom ***Editor***, so we need ***some*** excuse for doing so. Attribute Editors are all about how to ***present*** information to the user and the ways the user is (or is not) allowed to ***interact*** with the attribute Editor to manipulate its value.

   Having a choice among multiple Editors for the same data type gives us stylistic and/or functional choices; various `java.util.Date` Editors and `java.lang.Boolean` Editors are prime examples of this, but good Editors for these already exist.

   CONSIDERATIONS:

   - A Phone Number Format Editor, or Credit Card Number Format Editor, at first glance, *seem* like good candidates, as these would essentially be specialized `java.lang.Integer` editors – they could, e.g., block all non-digit characters and auto-format the *displayed* value with hyphens, parentheses, etc. while keeping the *internal* value purely numerical. Unfortunately, we want to do something with the Book type, and a Book doesn't have these kinds of attributes. Besides, these sort of data formats vary by country, card issuer, etc., so they would be difficult to do properly within a short exercise.
   - ISBN10 or ISBN13 seem like PERFECT candidates!!! – We could continue storing them as `java.lang.String`, but impose the proper formatting of hyphens. But unfortunately, a bit of research proved otherwise:

> *A 13-digit ISBN can be separated into its parts (prefix element, registration group, registrant, publication and check digit), and when this is done it is customary to separate the parts with hyphens or spaces. Separating the parts (registration group, registrant, publication and check digit) of a 10-digit ISBN is also done with either hyphens or spaces.* ***Figuring out how to correctly separate a given ISBN is complicated, because most of the parts do not use a fixed number of digits****.*
> *(https://en.wikipedia.org/wiki/International_Standard_Book_Number)*

Do you get the idea? Going into all of this detail about our thought process above is an attempt to impart a good sense of what a custom Editor is intended *for*. I.e., less about *validation,* (though Backoffice's standard Available Editors certainly enhance the user experience when SAP Commerce Platform's attribute-level Validators are used) and more about *the view and behavior*.

So… Let's just create a custom Editor that blocks numerical digits AND imposes a character limit, BUT with a custom way of interacting with the user (again keep in mind, we could far more *easily* have used built-in validation to achieve both of these goals, BUT with a far less interactive user experience).

Existing Editors are also capable of being configured to enforce a pattern, so it's our custom Editor's ***behavior*** that is of main interest here.

3. Open the following editor definition file that was created for you by the preparation script:

   `/bookstorecustombackoffice/backoffice/resources/editors/publisherEditor/definition.xml`

   This definition file specifies to Backoffice:

   - The official Backoffice **ID** for this **Editor *definition***
     (TODO 8.3a) Assign an ***id*** of ***my.bookstore.backoffice.editor.publisherEditor*** to this Editor definition.
   - What **Commerce type** this Editor is capable of working with
     (TODO 8.3b) Assign the full name of the Commerce type which this Editor is capable of handling. (NOTE: Though we will specifically use it for the ***publisher*** attribute of ***Book***, this Editor is capable of being applied to any attribute of the same type.)
     You can go back to the *Administration Cockpit*, search for ***Book*** under *System→Types*, and check the *Properties* tab to view the ***publisher*** attribute's Commerce type – (atomic types take on the fully-qualified name of their Java backing classes.)
   - What **Java controller** class (more correctly called the ***renderer*** **class**) this Editor uses
     (TODO 8.3c) Provide the fully qualified name of the `PublisherEditor` renderer class that we'll use to implement the Editor functionality (the class has already been created for you by the preparation script)

     > NOTE: As per the slides, Backoffice controller and renderer classes reside NOT in `<extension>/src`, (as it would if it were a Platform service, CronJob, or Populator backing class, etc.), but rather, in `<extension>/backoffice/src`, being that its lifecycle is controlled by the Backoffice framework. Locate it now in your IDE's Project Explorer to determine its fully qualified name.

4. We will get back to the `PublisherEditor.java` class later. For now, it is sufficient to know the Editor's *definition ID* and the controller class' fully qualified name.

   Let's first embed an instance of this Editor into a Widget's view so that, as developers, we have some idea of the environment in which the Editor instance will exist. In a simple case, we would embed this Editor into a Widget's view via the Widget's ZUL file. However, *our* case is not so simple – we want to embed our Editor into the view of an *Editor Area* widget. For *this* Widget type, the view is typically specified *indirectly*, i.e., via context configuration (i.e., in a Widget-specific way, we describe to the Widget what we need shown and how to organize it, and the Widget generates its view components dynamically).

   Open the `bookstorecustombackoffice-backoffice-config.xml` file (in folder `bookstorecustombackoffice/resources/bookstorecustombackoffice/resources`) and locate the *editor-area* context component specific to type *Book* (it is one of the last few entries in the file). Within this entry, nested 4 levels deep, you will find an element specifying that attribute *publisher* should be displayed. (It immediately precedes a similar XML element for attribute *rentable*).

   (TODO 8.4) Modify this publisher entry: first, add an *editor* XML attribute and give it the value of our Editor's *definition id*. Then alter the syntax of this XML element so that it can have a body containing child elements (i.e., convert the XML element syntax from 'short-form empty-element syntax' to 'long-form'). Finally, place the XML fragment containing the two editor-parameters (provided within the TODO comment) into the body of the *attribute* element for *publisher*. Don't forget to save your work.

   These changes will cause *our* custom Editor to be used instead of the default Editor for our attributes type AND the Editor will be able to read-in the custom parameters (i.e., name/value pairs) it is expecting, which the Editor controller (renderer) will use to customize its behavior.

   > NOTE: the details of how you would have known to structure these entries are specific to the design of the *Editor Area* widget controller. The expected format for these configuration entries can be found via `here`, which will provide details and examples on such widget-specific configuration information.
   >
   > The actual parameter names and values (i.e., what parameter names are expected and behavior they cause) for our Editor are specific to our custom Editor's implementation (which you will see later).

5. Now back to our renderer class. Open `PublisherEditor.java` in your IDE.

   (TODO 8.5) Note that `PublisherEditor` extends `AbstractTextBasedEditorRenderer<T>` (where the Java Generics <T> "parameter" corresponds with the Editor's defined *Type* and determines some of your renderer methods' return types and parameter types). In our case, this <T> parameter is set to `String` to correspond to our custom Editor's Commerce type.

   > NOTE: If you were to follow the inheritance hierarchy further, `AbstractTextBasedEditorRenderer` extends `AbstractCockpitEditorRenderer`, which, in turn, implements the interface `CockpitEditorRenderer`. This interface atop the entire hierarchy chain defines exactly one method, `render(..)`.

So, the key concept here is that, first and foremost, an Editor backing class is NOT a specialized kind of Widget. Its main characteristic is that of *Renderer* of one piece – an attribute's worth – of data. So, the key task when defining an Editor class is to override the inherited abstract `render()` method.

The primary job of an Editor's `render(..)` method is to create a new ZK `Component` and attach it to the passed-in `parent` element (i.e., we make our rendered `Component` a child node of the passed-in `parent` element).
(In actuality, the `Component` that the Editor creates can either be a standalone ZK view-component instance or the topmost/root node of an arbitrarily large tree of ZK view components.)

> NOTE: An attribute Editor is basically a "smart" view component embedded somewhere within a Widget's view. You write the Editor's custom renderer class, which typically generates a view-component fragment (typically a composition of ZK elements).
>
> At some point while the enclosing Widget proceeds to render its view, it becomes your Editor's turn to provide its contribution to the Widget's overall view. At this time, the enclosing Widget calls your Editor's `render()` method, passing-in (via the `parent` method parameter) a ZK `Component` reference to be used as an attachment point for your Editor's contribution to the Widget's overall view.
>
> Typically, the Editor's `render()` method performs this attachment via a call to `myEditorNewInputElement.setParent(parent)`. The view fragment your Editor attaches is for displaying its data *to* (and, optionally, receiving interactions and input data *from*) the user. Typically, your Editor's "component" nests several of its own *child* components, enabling your Editor's attached "component" to be quite complex.

The preparation script has provided you with a partially completed `render()` method implementation.

6. The first thing our `render()` method should do is read-in the *editor-attribute* values we provided earlier in the Editor Area's configuration.

   (TODO 8.6) To save you time and confusion, we have provided a line of code within the `render()` method for you to uncomment. Notice that line you uncommented makes a call to the `initializeEditorConfigurationParameters(..)` method. Look at the implementation of this method to see how an Editor renderer accesses its configuration parameters.

7. Still in `render()`, scroll down to where our Editor begins to generate its contribution to the Widget view.

   (TODO 8.7) Uncomment this small block of lines to reveal the creation of a `Vlayout` component whose first child component is a `Textbox` – this is the text box the user will interact with to provide a value to the *publisher* attribute.

   As part of rendering this text box, our editor must place an initial value (i.e., the *publisher* value from the *Book* being edited) into the view component. The proper way to obtain

this initial property value from the enclosing Widget is via the `EditorContext`'s `getInitialValue()` method, whose return type is determined by <T> parameter we looked at earlier.

By the way, our Editor class inherits abstract `setValue()` and `getValue()` methods from parent class `AbstractTextBasedEditorRenderer<T>`. These abstract methods must be overridden, and we have provided concrete implementations for you. Have a look at these methods now – their purpose is to allow external callers to treat the Editor instance as if it were a normal view `Component` and "set" or "get" its value (to which the Editor reacts by manipulating or retrieving the values from its own view component(s) or altering its own state variables). In our Editor it is sufficient to simply pass these values to/from our main text box component.

8. Now our `render()` method programmatically generates the rest of the Editor's view fragment.

   (TODO 8.8) Uncomment this large block of code. When render is done, it will have essentially created the following compositional structure of ZK view components, provided here to help you visualize the view the Editor is manipulating:

```
<vlayout id="publisherEditorVlayout">
  <textbox id="publisherAttribTextbox" />
  <label id="noDigitsAllowedLabel" value="NO DIGITS ALLOWED"
                visible="false" style="color:red; font-weight:bold" />
  <hlayout id="charLimitRow" visible="">
    <label value="Chars:" style="color:red" />
    <label id="charsValueLabel" style="color:red; font-weight:bold" value="" />

    <label value="Limit" style="color:red" />
    <label id="charLimitValueLabel" style="color:red; font-weight:bold" value="" />

    <label value="Chars left:" style="color:red" />
    <label id="charsLeftValueLabel" style="color:red; font-weight:bold" value="" />
  </hlayout>
</vlayout>
```

   NOTE: An Editor is allowed to specify its view via a .zul file *instead of* generating this view programmatically via a custom renderer class, but the two approaches cannot be combined. (I.e., we can't use the same approach we used with our Widget). Since we require custom behavior/logic, the custom renderer class is the only option.

9. **(TODO 8.9a)** Finally, uncomment this call to your Editor class' `initViewComponent()` method (inherited from its parent class). If you have time, look closely at this method's implementation. You will see that it initializes the component specified (in our case, the main text box) such that all of this component's ZK events (i.e., user interactions) are passed along to the ZK framework to be handled.

   Our main text box is now bound to the **`EditorContext`** object, and initialized with all the basic listeners for events such as `ON_CHANGE`, `ON_OK`, and `ON_CANCEL`. These listeners suffice for this exercise, so you don't need to add any new listeners. To gain more insight on this, have a look at your Editor class' parent class, `AbstractTextBasedEditorRenderer`, to see what happens inside `initViewComponent()`.

   If we override these parent-class event handler methods (and then call the overridd*en* event method at the end of, and from within our overrid*ing* method interceptor method

(e.g., via `super.onChaneEvent(..)`), then we can, in effect, inject our own custom behavior into our component. We will do this later.

(TODO 8.9b) Finally, uncomment this line of code that attaches this Editor's view fragment to the Widget-provided "attachment" component. We've written plenty about this, so you should already understand what this is all about.

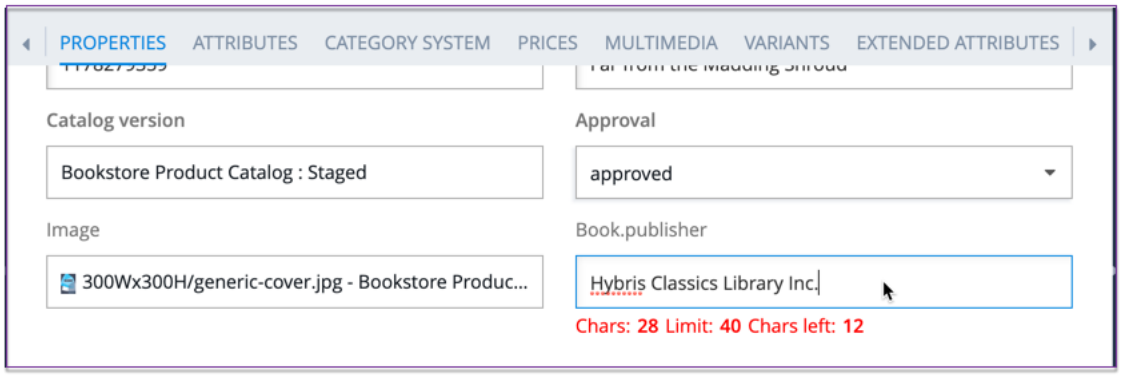
10. So we now have our required `render()`, `setRawValue()`, and `getRawValue()` methods. What's left is to provide custom interactive behavior to our Editor. As mentioned earlier, we provide custom *behavior* to our editor via a technique that sort-of creates an interceptor for ZK-component events.

    (TODO 8.10) Take a VERY close look at how this method manipulates the values and states of our Editor's view components. Upon an Editor Area SAVE, whatever value was applied to our text box via `setText()` will be applied to our bound ***Book*** property (i.e., ***publisher***).

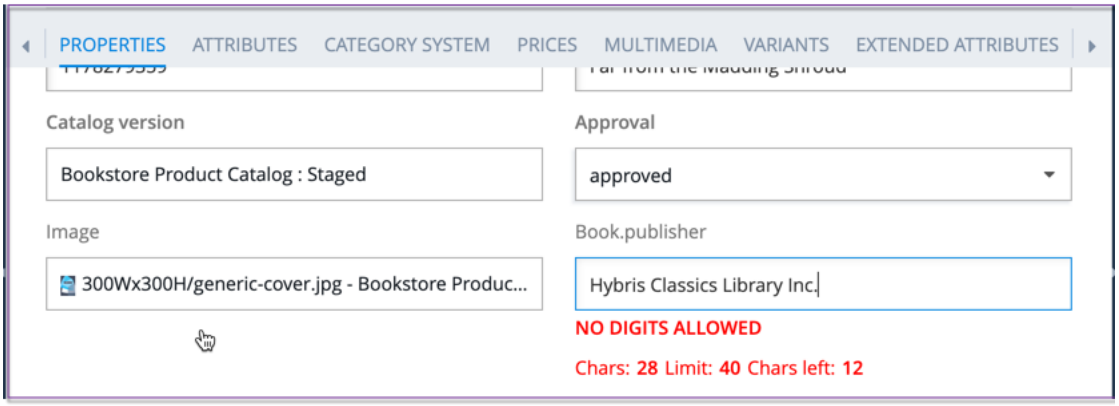
11. Execute an `ant build`, redeploy, and refresh. Logout and Login as admin, or open Application Orchestrator to reset cockpit-config.xml.

    You will be actively prevented from entering forbidden characters or exceeding the Editor's configured length constraint

    To test, select a book from the list and attempt to enter a value containing digits into the ***Book publisher*** field. You should see the following in your *Editor Area* as you type:



…and you should see the following if you attempt to enter a digit anywhere within this field (this editor will actively *remove* the offending character(s) and make *visible* a bold, red label to remind you that digits are not allowed. (This label will remain visible while you are still editing the same book.)



If everything worked as expected, then…

Congratulations, you are now ready to move on to the next exercise.

Part 2 (Optional) – Custom Action

Our second task, in which we are going to implement a custom action, comes from a new business requirement: the Bookstore is planning to make some of its inventory available on a rental basis. Each Book has a `Boolean` ***rentable*** attribute, but the book merchandising team need a convenient means of switching a Book's rentability status by means of an Action button within the Book Details widget.

First, run the preparation target for exercise **8.2**, refresh your IDE project if necessary, then perform the following tasks:

12. Open the following action definition created for you by the preparation script:

    `/bookstorecustombackoffice/backoffice/resources/actions/bookRentability Action/definition.xml`

    Notice that the ***id*** of the `action-definition` has already been assigned for you in the root element. This id will be used whenever referring to this Action definition from which instances of this action are derived. Next, in `definition.xml`, locate the TODO comment for this step – add the *fully qualified* name of the `ChangeBookRentabilityAction` class as the `<actionClassName>`.

13. Having defined our Action above, we now need to bind it to the Book Details widget property ***selectedBook***. HOWEVER, we're going to do this in a slightly special way…

    NORMALLY, we would add the bound `<y:action>` element directly to the `bookDetails.zul` file. However, this would result in the Action button ALWAYS being part of the ***Book Details*** widget view.

    INSTEAD, we want our widget to be context-sensitive and ONLY make the ***bookRentabilityAction*** part of its view if a `Book` being displayed.

    SO, we place a *special*, *configurable* `<actions config="bookdetailsActionsConfig">` element (instead of the usual `<action>` element) into the `bookDetails.zul`, which will replace itself with the *actual* view layout (if any), from the body of a `<context component="bookdetailsActionsConfig" type="SomeCommerceType">` entry in a `*-backoffice-config.xml` file. This allows the actual layout to be different depending on context.

    So… let's do this… open `bookDetails.zul` and locate the TODO comment for this step – inside the ***rentalStatus*** div, add an `<actions>` tag with ***id*** of ***actionSlot***, and set ***config*** to the value ***bookdetailsActionsConfig***.
    Also add tag attributes `group="common"` and `sclass="yw-actionSlot"`.

14. Go to your extension's UI context config file:

    `/bookstorecustombackoffice/resources/bookstorecustombackoffice-backoffice-config.xml`

    Locate the TODO comment for this step – search for the context component ***bookdetailsActionsConfig*** near the bottom of the file – it was added for you by the preparation script. Within the body of the `<y:actions>` element is the `<y:action>` tag which needs to be completed by the addition of two attributes to bind it to the Action:

    a) the `action-id` takes the full definition ***id*** of the ***bookRentabilityAction*** (remember – it's in the root element of this action's `definition.xml`)

    b) the `property` attribute, which we'll set to ***currentBook***, which is the name of the property/attribute that the ***Book Details*** widget will place in its ***widget model*** for the ***bookRentabilityAction*** to bind to. (Remember, an Action binds to a widget via a named 'attribute' placed by the widget into its own ***widget model***).

15. Now go back to `BookDetailsController.java` and revisit its `handleSelectedBook()` method.

    The code already assigns the currently displayed Book to the private `selectedBook` attribute of the controller, but to bind to our Action, we need to expose this Book to the ***widget model***. Locate the TODO comments (8.15a, 8.15b, 8.15c, and 8.15d) associated with this step, and add a call the controller's `getModel().put()` method to set a ***widget model*** attribute named ***currentBook*** whose value is the `BookModel` reference that was passed into the `handleSelectedBook(…)` method (i.e., the one that was assigned to ***selectedBook***). NOTE: the only reason we decided on the name ***currentBook*** is to make clear that the way binding works has nothing to do with a widget's instance-attribute names or bean-property names. OPTIONAL (best practice): instead of using three individual String literals for the value "currentBook" (which could easily end up mismatching, causing hard-to-find errors and creating potential maintenance headaches), define and use a single String constant inside the `BookDetailsController` class – perhaps, name it `WIDGET_MODEL_ATTRIBUTE_CURRENT_BOOK`, and use the constant instead, letting the compiler auto-complete its name and disallowing even the slightest mismatch.

16. For behavioral consistency, add corresponding widget-model changes (*similar* to the previous step) to the `handleObjectsUpdatedEvent()` method and to the `handleObjectsDeletedEvent()` method.

17. To make it a little easier for you to see what's going on during the rest of this exercise, alter which Book attributes are displayed by your Book Details widget.
    Go back to: `/bookstorecustombackoffice/resources/bookstorecustombackoffice-backoffice-config.xml`, and locate the ***bookdetailsConfig*** context entry. In the body of this entry, comment-out (or delete) the properties ***ISBN10*** and ***description***, and add property ***rentable*** with label ***Rentable?*** .

18. Because we're not *refreshing* an Action class, but adding a brand new one to Backoffice (this seems to *only* apply to brand new Action classes and not Widgets or Editors), you are going to need to restart, so stop the server, perform an `ant build`, restart, and log in to Backoffice as admin. Take a look at the lower portion of the Book Details panel (if you can't see the Action button/icon, scroll this widget upward or move the pane divider up, if needed, to expose the lower region of the widget). Your Action 'button' should appear as only the greyed-out tool-tip text, "Change Rentability Status".

19. Go back to the ***ChangeBookRentabilityAction***'s `definition.xml`. Turn your attention to the three empty "icon URI" tags at the bottom of the file.

    We have provided the image files you'll need in the `images` folder adjacent to the definition file – open these images in Eclipse/STS to become familiar with how they look so that things make more sense later. Add the appropriate values to the tags, using `images/…` as the relative path. (Interestingly, those three icon tags are required to be there even if they're empty, otherwise the XML file fails validation).
    Now do another `ant build`, redeploy, and refresh, and observe how the Book Details' UI has changed – now your Action appears as one of the small PNG icons we just added to the Action's `definition.xml`.

20. Now that we got this Action to be visible, it's time to get it to actually DO something!
    Open `ChangeBookRentabilityAction.java`. Our Action class implements the `CockpitAction` interface which specifies a `canPerform()` callback method that Backoffice calls first to find out whether or not there is a programmatically determinable reason `perform()` cannot or should not be called (e.g., because of a technical reason or a business rule). If not, then the Action button shows as being disabled. Let's add some logic to `canPerform()`. Go to the `ActionContext` (via method parameter) and call its `getData()`

method to retrieve the bound object (if any). Have `canPerform()` return `true` only if the returned object is a non-null `BookModel` reference, indicating that the Action is bound to *something*, and that something is a Book.

21. Still in `ChangeBookRentabilityAction.java`, locate the TODO comment for this step – we will implement the key logic of our Action in the `perform()` method.

    The first thing for this Action `perform()` method to do is to get the reference of the current Book bound to the ***Book Details*** widget, which is the Action's parent component. This bound attribute is available to the Action via its `ActionContext`. Access this context via `perform()`'s method parameter. Use the `getData()` method of the `ActionContext` to retrieve the bound attribute (named ***currentBook***, of course) and assign the result to a method variable of type `BookModel`.

22. Still in the `perform()` method of `ChangeBookRentabilityAction.java`, locate the TODO comment for this step.
    Having gotten a reference to the current Book as a `BookModel` object, we can now check its "rentable" status via its `getRentable()` method. If it's null, do nothing. But if it is currently `true`, we want to set it to `false` and vice versa, and then save it back to the platform – i.e., just like any other Model instance we change, we call `modelService.save()` to persist the change.

23. Still in the `perform()` method of `ChangeBookRentabilityAction.java`, locate the TODO comment for this step.
    If a change to a BookModel occurred and the save was successful, publish a Global Cockpit Event to the rest of Backoffice to give notice that this specific book got updated. Refer to the slides of the Widget Communication module if you need help remembering how to do this.

24. Still in the `perform()` method of `ChangeBookRentabilityAction.java`, locate both TODO comments for this step.
    Finally, the `perform()` method must return a new `ActionResult<BookModel>` object – to instantiate such an object, its constructor takes two parameters:
    - The first parameter should be `ActionResult.SUCCESS` or `ActionResult.ERROR` depending on whether or not we caught a `ModelSavingException`;
    - The second parameter should be the BookModel object on which this Action was performed (or attempted.
    Add both return statements, to handle the case of a successful Action and the case of catching a `ModelSavingException` when attempting a `modelService.save()`.

25. To complete the picture of how an Action class works, have a look at the remaining two methods of `ChangeBookRentabilityAction.java` which we have implemented for you:

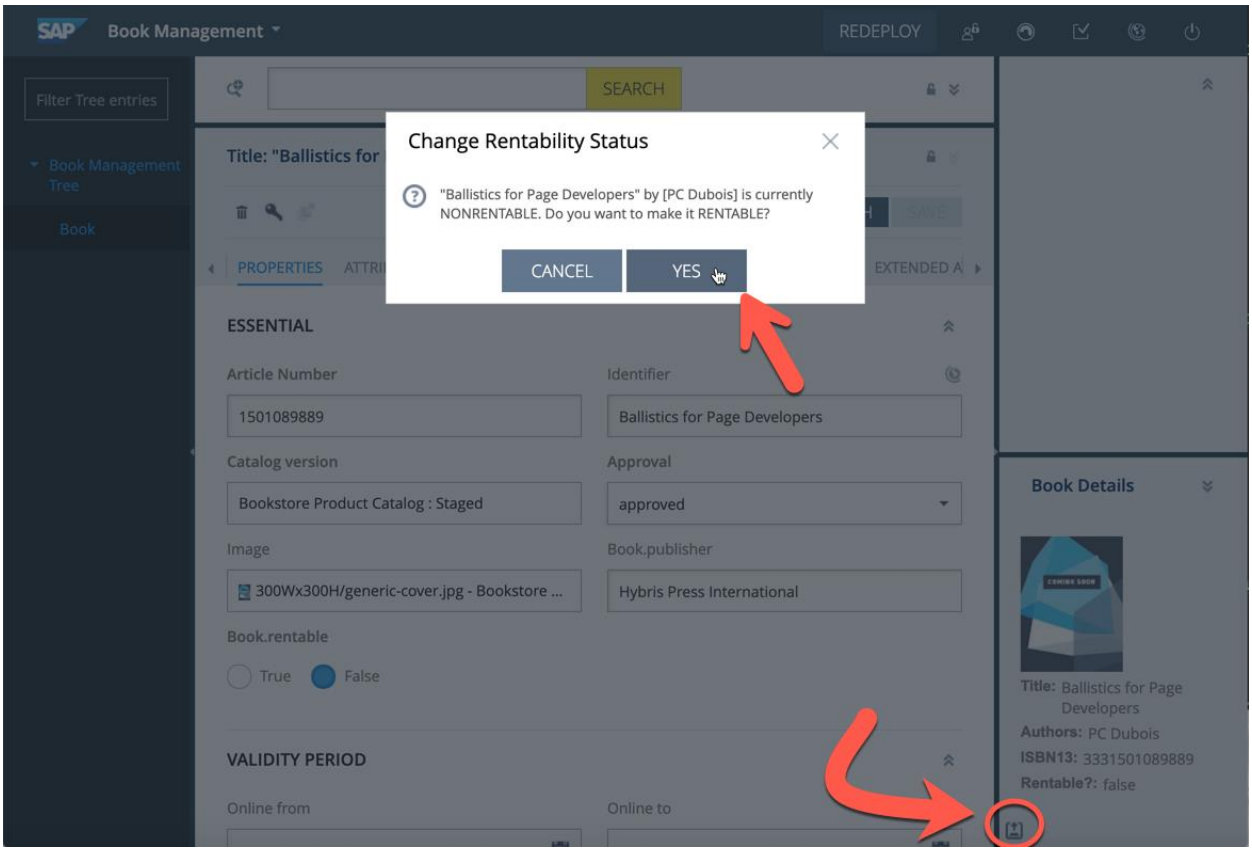
    - `needsConfirmation()` – this callback method is invoked by Backoffice (if `canPerform()` returned `true`) to confirm *with the user* (via Yes/**No** modal popup window) whether or not the Action should be performed. If the response was **Yes**, then the `perform()` method will finally be called.

    - `getConfirmationMessage()` – this callback method is invoked by Backoffice to obtain the message to be displayed by the modal popup confirmation. This method implementation is interesting because it uses the method `ctx.getLabel()` to look-up values from `backoffice/resources/actions/bookRentabilityAction/labels/labels.properties`. When you want to do this in the future, if the labels subfolder does not already exist, create one. Furthermore, this `labels.properties` file is only the default/fallback one for this labels "resource bundle". If you want to override values (for one the same property keys) with locale-specific values, create additional files in this folder using the name `labels_<isocode>.properties`, e.g.,

`labels_es.properties` and `labels_de.properties` for Español (Spanish) and Deutsch (German), respectively.

26. After running `ant build`, clicking Redeploy, logging out and back in, verify your solution by changing the rentability status of a book under the Administration tab of the editor area.

    When the user clicks on the Change Rentability Status Action button/icon, a dialog box will pop up displaying the current Book's rental status, along with its title and author, and asking whether the user wishes to switch to the alternate state. If you click on Yes, the book's ***rentable*** value will change and the Editor Area and Book Details widgets will refresh.



Congratulations – you are now ready to continue with the rest of the exercises.