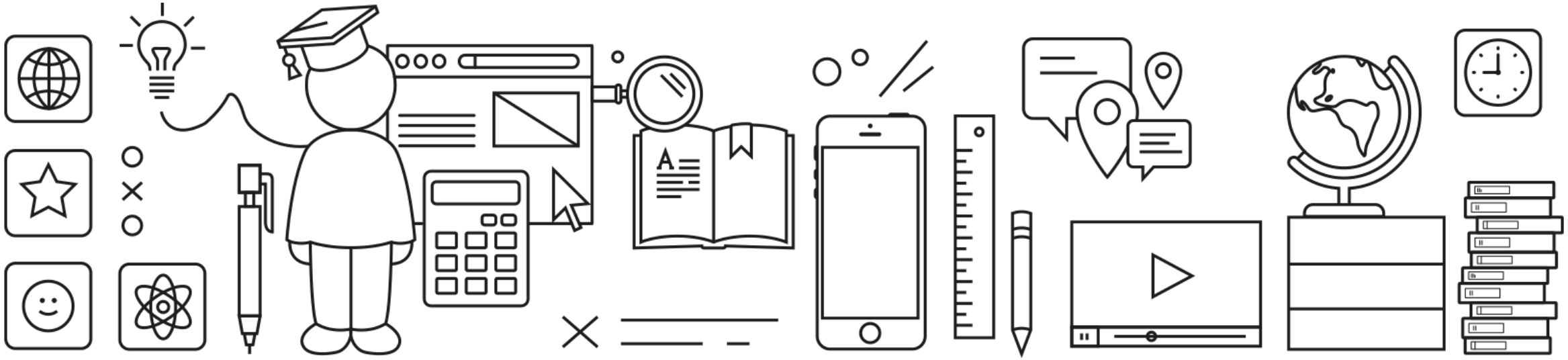
SAP Customer Experience

SAP Commerce Cloud Backoffice Framework Developer Training

Actions and Editors

THE BEST RUN SAP



Introduction

Introduction
Available Editors Gallery
Editors in Widgets
Creating an Editor
Available Actions Gallery
Actions in Widgets
Creating an Action
Editors and Actions Communication
Exercise

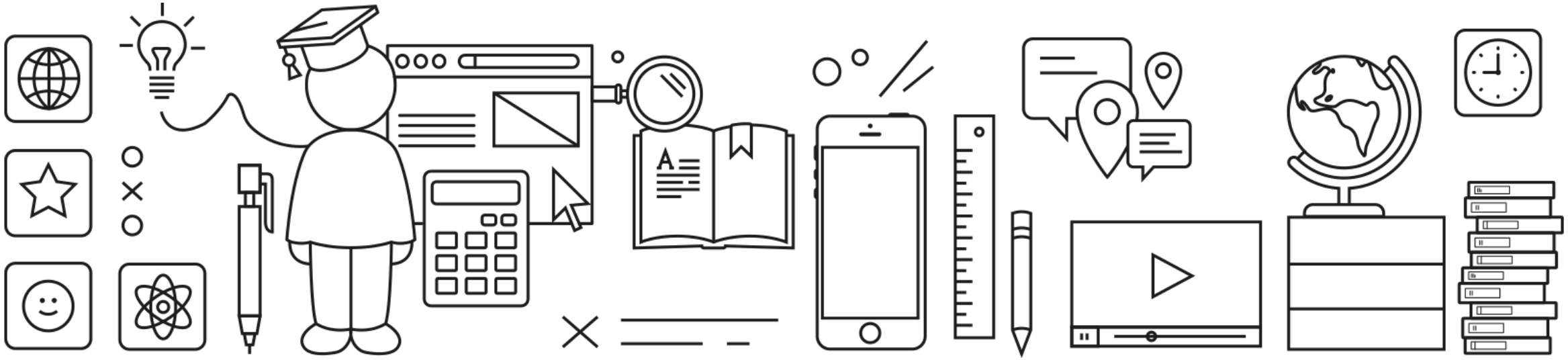
Editors and Actions

- **Actions and Editors are components that can be used INSIDE widgets**

- **Editors** manage the input, display, and handling of a single value of a given type
  - **Key Concept:** *Display and Handle Data* within a Widget's view
  - E.g. **Boolean data types** can use an editor that displays on/off radio buttons to the user
  - E.g. **Date data types** can have an input with format and validation or a custom date picker
- **Actions** are responsible for the invocation of custom code within Backoffice
  - **Key Concept:** *Do Something in Java*, triggered by clicking on an icon
  - E.g. **creating** a new type-based Item (e.g. a Product, Order, or Customer)
  - E.g. **changing** the state of a Product's availability from 'available' to 'unavailable'
  - Often invoked using a button control or menu option

Editors and Actions

- The Backoffice Framework provides standard Editors and Actions OOTB
- Process of *defining* custom Editors and Actions is similar to defining a Widget (...but Editors and Actions are NOT specialized kinds of Widgets)
- Actions and Editors can also have context-based UI configurations
- Actions and Editors are not *socket-aware,* but they can be made socket-capable using STUBS (This will be discussed later)

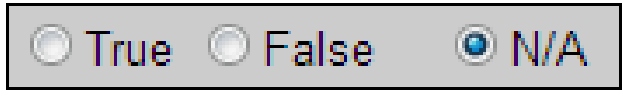


Available Editors Gallery

Introduction
Available Editors Gallery
Editors in Widgets
Creating an Editor
Available Actions Gallery
Actions in Widgets
Creating an Action
Editors and Actions Communication
Exercise

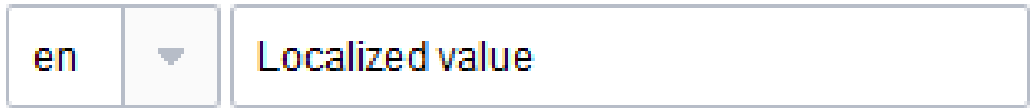
Available Editors Gallery

- **Default Boolean:**



- **Check Box Boolean:**

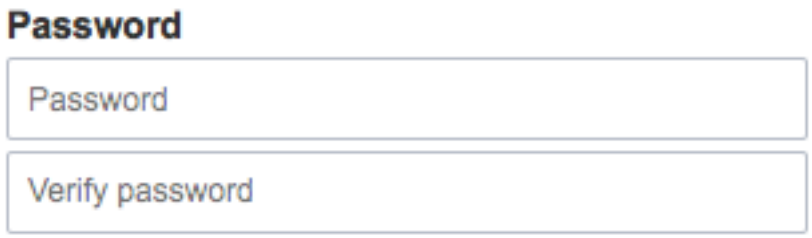
- **Localized Simple Editor:**



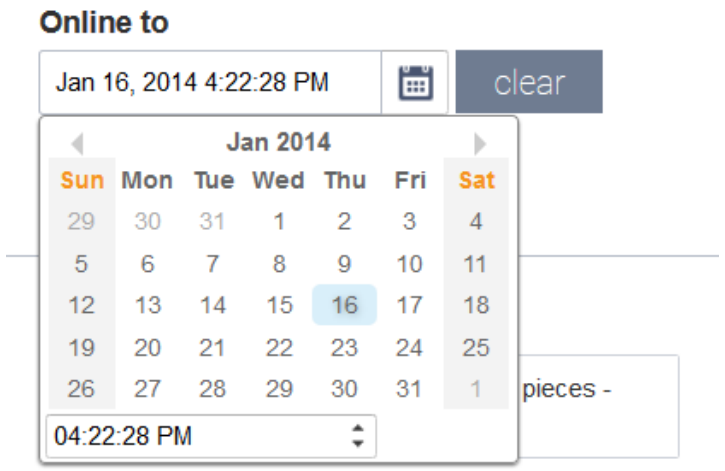
- **Default Big Decimal Editor:**

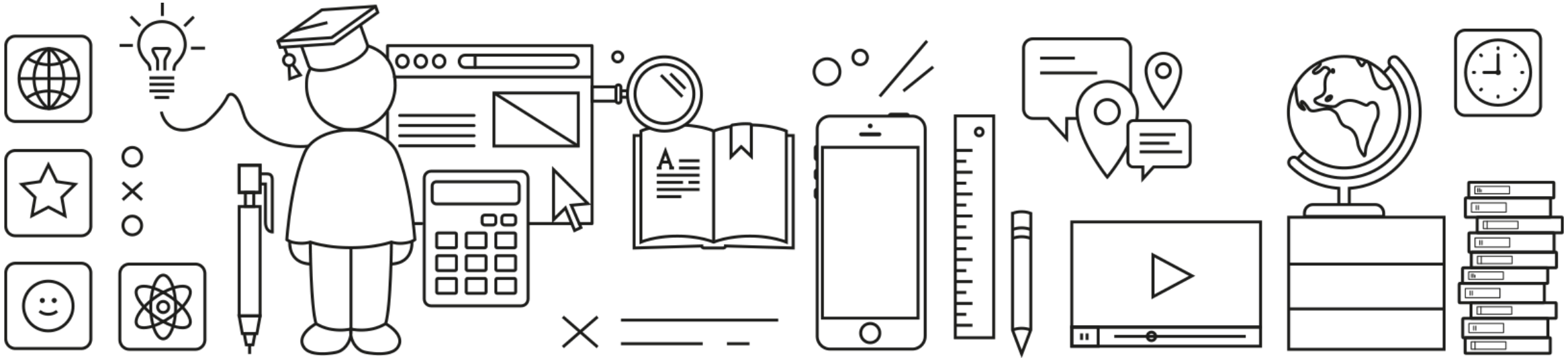
Available Editors Gallery

- **Default Password Editor:**



- **Default Date Editor:**





Editors in Widgets

Introduction
Available Editors Gallery
Editors in Widgets
Creating an Editor
Available Actions Gallery
Actions in Widgets
Creating an Action
Editors and Actions Communication
Exercise

Using a Default (Type-Mapped) Editor In a Widget

- Editor *instances* can be added to a widget via `myWidget.zul`
  - Assign an ID to the view element
  - *Imply* which editor *definition* to use by specifying the data type for the field
  - Each data type is mapped to its default editor in `standard-editors-spring.xml`
  - Backoffice uses the default editor *definition* mapped to this type

myWidget.zul

```
<widget …>
  …
  <label value="First Name:" />
  <editor id="firstNameFieldEditor" type="java.lang.String" />
  …
  <label value="Last Name:" />
  <editor id="lastNameFieldEditor" type="java.lang.String" />
  …
</widget>
```

Using a Specific Editor In a Widget

An *editor definition id* may be explicitly specified, instead of implicitly via type:

- Assign an ID to the instance
- Specify the editor definition to use for the instance
  - Use the defaultEditor attribute
  - Specify the *editor definition id*
  - In this case, Backoffice's WYSIWYG editor is used for this field instead of the default

myWidget.zul

```
<widget ...>
  ...
  <label value="Username:" />
  <editor id="userNameFieldEditor"
          defaultEditor="com.hybris.cockpitng.editor.wysiwyg" />
  ...
</widget>
```

Binding the Editor to a Widget Model Property (via ZUL)

- Bind the editor to a property in the widget model using the *property* attribute
- Two-way binding:
  - The property value will be displayed in the editor upon rendering
  - When a user changes the value in the editor, the model property is automatically updated

MyWidgetController.java

```
WidgetModel model = getModel();
model.put("product", product);
```

myWidget.zul

```
<label value="Product code:" />
<editor type="java.lang.String" property="product.code" />

<label value="Category:" />
<editor type="java.lang.String" property="product.category.name" />
```

Specifying Attribute Editors via Configuration (1)

- Often, in practice, widget types require that you *indirectly configure* ***what*** fields to display and ***how*** to display them (i.e., **editors**, sections, tabs, etc.)
  - Configured via **<context..>** XML entries in ***-config.xml** (as opposed to explicitly specifying the composition of ZUL components)
  - The XML syntax for each **<context..>** element body depends on the JAXB configuration type the widget controller expects

- Based on this configuration, the widget controller dynamically *generates* ZUL display components (and binds editors to the display components)

Specifying Attribute Editors via Configuration (2)

- For example:

  Advanced Search widget instances expect a JAXB construct containing element

  <field name="*attrib*" editor="*editor.definition.id*" >

mycustombackoffice-backoffice-config.xml

```
<context type="Shoe" component="advanced-search" >
  <advanced-search:advanced-search
      xmlns:advanced-search ="http://www.hybris.com/cockpitng/config/advancedsearch" >
    <advanced-search:field-list>
      ...
      <advanced-search:field  name="waterproof" selected="false"
                    editor="com.hybris.cockpitng.editor.boolean.checkbox" />
    </advanced-search:field-list>
  </advanced-search:advanced-search>
</context>
```

Specifying Attribute Editors via Configuration (3)

Whereas:

Editor Area widget instances expect a JAXB construct containing element
<attribute qualifier="*attrib*" editor="*editor.definition.id*" >

mycustombackoffice-backoffice-config.xml

```
<context merge-by="type" parent="Product" type="Shoe" component="editor-area" >
  <editorArea:editorArea
        xmlns:editorArea ="http://www.hybris.com/cockpitng/component/editorArea" >
    ...
    <editorArea:attribute  qualifier="size"
                editor="my.shoestore.backoffice.editor.customSizeEditor" />
    <editorArea:attribute qualifier="color" />
  </editorArea:editorArea>
</context>
```

Recall: Details of each widget type's configuration data structure at https://help.hybris.com , search term "available widgets"

Specifying an Initial Value via ZUL

- Specify an initial value by adding a *value* attribute

myWidget.zul

```
<widget ...>
  ...
  <editor id="firstNameFieldEditor" type="java.lang.String"
     property="customer.firstName" value="Some text" />
  ...
</widget>
```

Specifying Value-Change Handler via ZUL

- Handling a changed value (returned by an Editor) *without* binding to a property:
- (Handle the **onValueChanged** event)
  - Option 1 - define binding from the widget's view

myWidget.zul

```
<widget ...>
  ...
  <editor id="userNameFieldEditor" type="java.lang.String"
      onValueChanged="widgetController.doSomeThing()" />
  ...
</widget>
```

special variable to reference the actual bound Widget Controller instance

Specifying Value-Change Handler via Java Annotation

- Handling a changed value (returned by an Editor) *without* binding to a property
  - Option 2 - Define **onValueChanged** event handling in the widget's controller with *@ViewEvent*

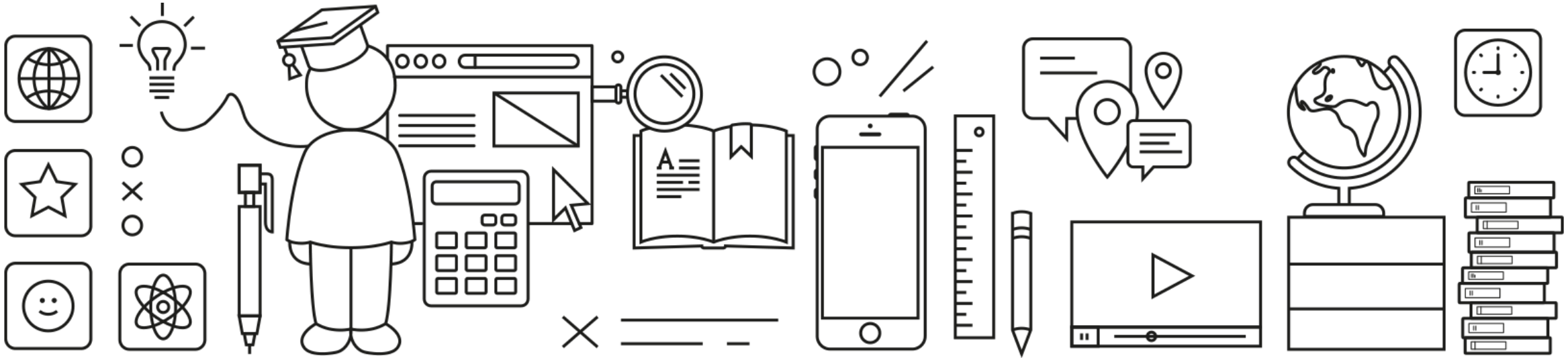
myWidget.zul

```
<widget ...>
  ...
  <editor id="userNameTextField" type="java.lang.String" />
  ...
</widget>
```

MyWidgetController.java

```
public class MyWidgetController extends DefaultWidgetController
{
  private Editor userNameTextFieldEditor;

  @ViewEvent(componentID="userNameTextField", eventName="onValueChanged" )
  public void doSomething()
  {
    final String tmp = (String) userNameTextFieldEditor.getValue();
    // do something
  }...
}
```



Creating an Editor

Introduction
Available Editors Gallery
Editors in Widgets
Creating an Editor
Available Actions Gallery
Actions in Widgets
Creating an Action
Editors and Actions Communication
Exercise

Creating an Editor 1 – Definition File

- Definition file:

bookstorebackoffice/backoffice/resources/widgets/**editors/mySimpleTextEditor**/definition.xml

```
<editor-definition id="com.corp.cockpitng.editor.mysimpletexteditor" >
    <name>My Simple Text Editor</name>
    <description>Simple text editor</description>
    <!--The type of the value this editor is capable of handling. Used for type-mapping -->
    <type>java.lang.String</type>
    <editorClassName>org.myextension.editor.simpletext.MySimpleTextEditor</editorClassName>
</editor-definition>
```

Creating an Editor 2 – Rendering via Custom Class

- Editor class must implement CockpitEditorRenderer:

MySimpleTextEditor.java

```
public class MySimpleTextEditor implements CockpitEditorRenderer<String>
{
  public void render(Component parent, EditorDefinition editorDefinition,
                                    EditorContext<String> context, EditorListener<String> listener)
  {
    Textbox editorView = new Textbox();
    editorView.setValue(context.getInitialValue());
    . . .
    editorView.addEventListener(Events.ON_CHANGE,
      new EventListener<Event>() {
        public void onEvent(final Event e) throws Exception
        {
                . . .
              }
      }
    );
    editorView.setParent(parent);
    . . .
```

Determines return type and parameter type for many controller methods

Read Editor parameters (from ZUL) via EditorContext:

`cxt.getParameter(name)`

Creating an Editor 3 – Rendering via .zul File

- An editor can be rendered using a `.zul` file instead of a via custom Java class:

bookstorebackoffice/backoffice/resources/widgets/**editors/mySimpleBoolEditor**/definition.xml

```
<editor-definition id="com.corp.cockpitng.editor.mysimpleboolleditor" >
    <name>My Simple Boolean Editor</name>
    <description>Simple boolean editor</description>
    <type>java.lang.Boolean</type>
    <view src="boolEditorView.zul" />
</editor-definition>
```

boolEditorView.zul

```
<hlayout id="boolEditor">
    <radiogroup id="rgr" />
    <radio label="True" radiogroup="rgr" forward='onCheck=boolEditor.onEditorValueChanged(${true})'
     checked="${arg.initialValue == true}" />
    <radio label="False" radiogroup="rgr" forward='onCheck=boolEditor.onEditorValueChanged(${false})'
     checked="${arg.initialValue == false}" />
    <radio label="n/a" radiogroup="rgr" forward='onCheck=boolEditor.onEditorValueChanged(${null})'
     checked="${arg.initialValue == null}" />
</hlayout>
```

- Here, view events are “forwarded” to the event handlers bound to the root element

Communication with the Widget Model

EDITOR SETUP

```
editor.setProperty("currentObject.title")
```

COMMUNICATION

EDITOR ↔ WIDGET MODEL

EDITOR INITIALIZATION

Get initial value
from widget model

Add model observer →

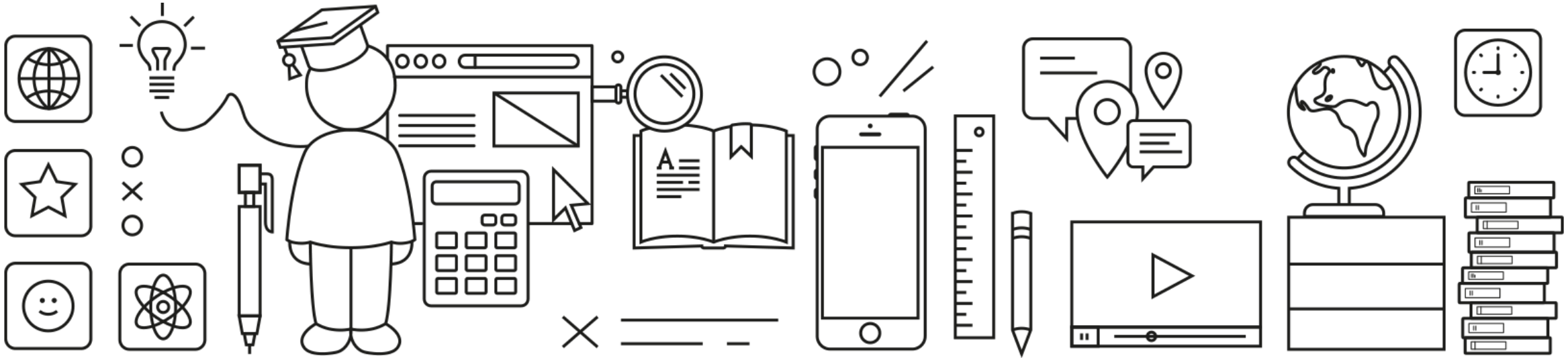
```
widgetModel.addObserver(getProperty(),modelObserver)
```

Call when model changed externally

```
modelChanged()
{
    reload editor
}
```

Create editor listener →

```
onValueChanged() // e.g. User types something
{
    widgetModel.setValue(getProperty(),changedValue)
}
```

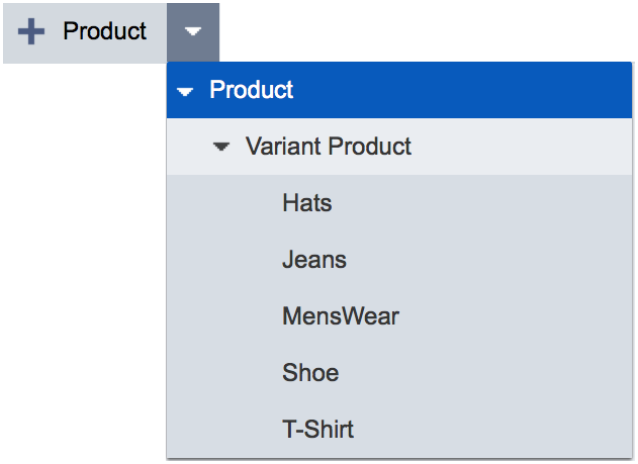


Available Actions Gallery

Introduction
Available Editors Gallery
Editors in Widgets
Creating an Editor
Available Actions Gallery
Actions in Widgets
Creating an Action
Editors and Actions Communication
Exercise

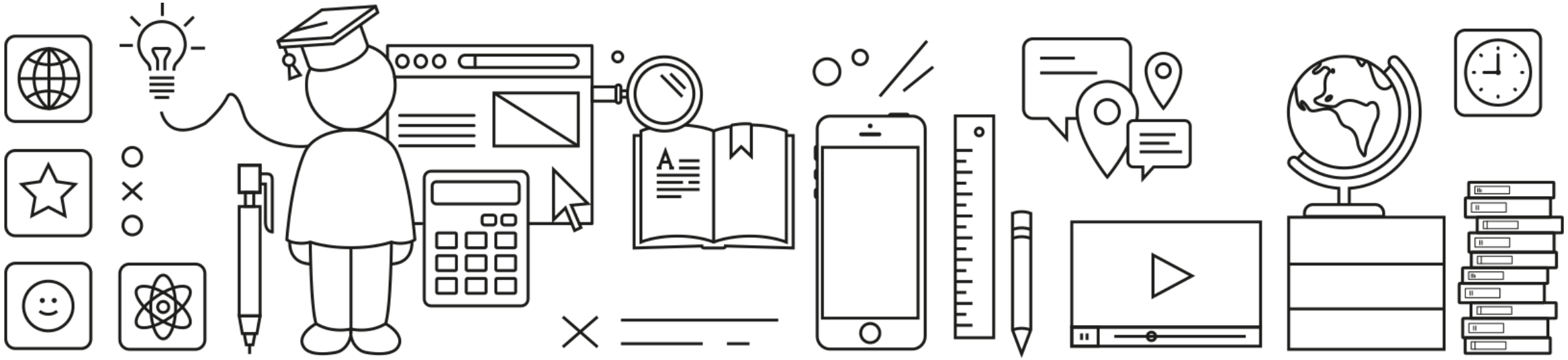
Available Actions Gallery

- ***Create* Action**:



- ***Delete* Action:**

- **Launch *Permission Management*:**



Actions in Widgets

Introduction
Available Editors Gallery
Editors in Widgets
Creating an Editor
Available Actions Gallery
Actions in Widgets
Creating an Action
Editors and Actions Communication
Exercise

Using Actions in Widgets – ADD to Widget – via ZUL

- Add the Action to the widget **view** (any number of <action> elements allowed)
- Bind each action to an ‘attribute’ of the widget model via XML ***property*** attribute
  - Auto-adds an *observer* to the widget model for handling changes
  - Bound attributes are copied/synced into the Action’s ActionContext

myWidget.zul

```
<widget …>
  …
  <action actionId="org.myextension.action.myAction" property="product"/>
  …
</widget>
```

MyWidgetController.java

```
WidgetModel model = getModel();
model.put("product", product);
```

MyActionController.java

```
public ReturnType everyActionMethod ( ActionContext<ProductModel> ctx ) {
  ProductModel boundProduct = ctx.getData();
  …
}
```

Using Actions in Widgets – ADD to Widget – via Config

- A more advanced way to add action(s) to a widget in the view
  - Enables grouping of your actions to control visibility in a context-sensitive way
- Same as basic option, except Action is added *indirectly* to the widget view

myWidget.zul

```
<widget ...>
  ...
  <actions config="myActionsSlotConfig"  group="common"  sclass="yw-actionsSlot" />
  ...
</widget>
```

myExtension/resources/myExtenstion-backoffice-config.xml

```
<context component="myActionsSlotConfig" type="Product" >
        <y:actions xmlns:y="http://www.hybris.com/cockpit/config/hybris">
                <y:group qualifier="common">
                        <y:label>actiongroup.common</y:label>
                        <y:action action-id="org.myextension.action.myAction"
                  property="currentBook" />
                </y:group>
        </y:actions>
</context>
```

Using Actions in Widgets – onActionPerformed – Opt. A

OPTIONAL: You can specify the handler (for an Action's successful `perform()`) from widget's controller via @ViewEvent annotation using `onActionPerformed` event name and id of <action> ZUL component

myWidget.zul

```
<widget ...>
  ...
  <action id="myActionInstance" property="product"
                                actionId="org.myextension.action.myAction" />
  ...
</widget>
```

MyWidgetController.java

```
public class MyWidgetController extends defaultWidgetController
{
    @ViewEvent(componentID="myActionInstance", eventName="onActionPerformed" )
    public void doSomething()
        {
     ...
```

Using Actions in Widgets – onActionPerformed – Opt. B

OPTIONAL: You can specify the handler (for an Action's successful perform()) **from widget's controller** via **@ViewEvent** annotation, `onActionPerformed` event name and id of <actions> ZUL component

myWidget.zul

```
<widget ...>
  <actions id="myActionsInsatnce" sclass="yw-actionsSlot"
           group="common"    config="myActionsSlotConfig" />
</widget>
```

MyWidgetController.java

```
public class MyWidgetController extends defaultWidgetController
{
    @ViewEvent(componentID="myActionsInstance", eventName="onActionPerformed" )
    public void doSomething()    { ... }
```

myExtension/resources/myExtenstion-backoffice-config.xml

```
<context component="myActionsSlotConfig" type="Product" >
  <y:actions>
    <y:group qualifier="common">
      <y:action action-id="org.myextension.action.myAction" property="currentBook" />
    </y:group>
  </y:actions>
```

Using Actions in Widgets – onActionPerformed – Opt. C

- OPTIONAL: You can specify a handler (for a successful Action perform()) from within widget's .ZUL file via the onActionPerformed tag attribute

myWidget.zul

```
<widget ...>
  ...
  <action actionId="org.myextension.action.myAction"  property="product"
            onActionPerformed="my.extensionbackoffice.widgets.MyWidgetController.saveProduct()" />
<!-- OR, to bind one handler for ANY succesful Action within an <actions> group -->
  <actions id="myActionsInsatnce"  group="common"  config="myActionsSlotConfig"
     onActionPerformed="my.extensionbackoffice.widgets.MyWidgetController.saveProduct()" />
  ...
</widget>
```

MyWidgetController.java

```
package my.extensionbackoffice.widgets;

public class MyWidgetController extends DefaultWidgetController {
    public static void saveProduct()  {
        . . .
    }
}
```

Using Actions in Widgets – onActionPerformed – Opt. D

- OPTIONAL: You can specify a handler (for a successful Action perform()) from within widget's .ZUL file via the onActionPerformed tag attribute

myWidget.zul

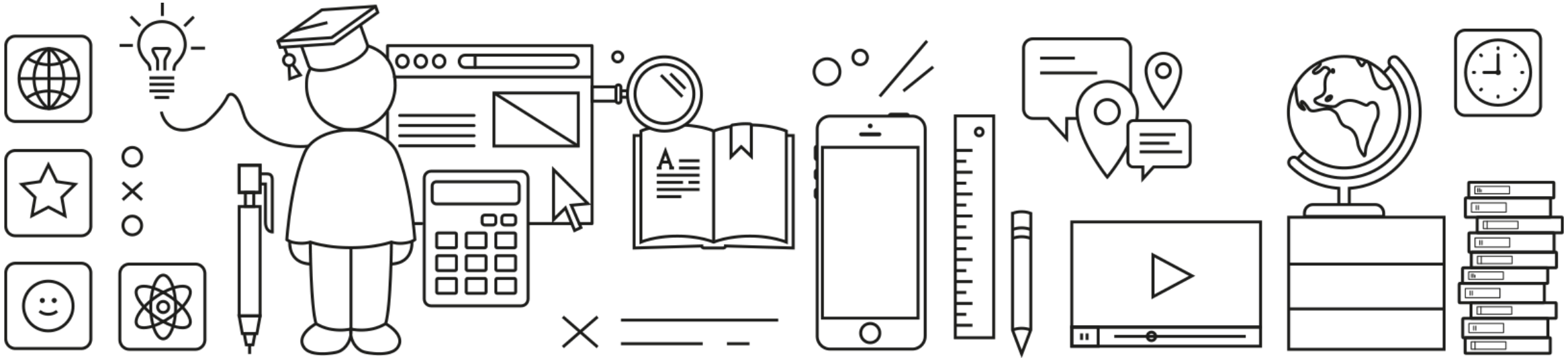
```
<widget ...>
  ...
  <action actionId="org.myextension.action.myAction"  property="product"
              onActionPerformed="widgetController.saveProduct()" />
<!-- OR, to bind one handler  for ALL successful Actions in an <actions> group -->
  <actions id="myActionsInsatnce"  group="common"  config="myActionsSlotConfig"
        onActionPerformed="widgetController.saveProduct()" />
  ...
</widget>
```

special variable to reference the actual bound Widget Controller instance

MyWidgetController.java

```
package my.extensionbackoffice.widgets;

public class MyWidgetController extends DefaultWidgetController {
    public void saveProduct()  {
        . . .
    }
}
```



Creating an Action

Introduction
Available Editors Gallery
Editors in Widgets
Creating an Editor
Available Actions Gallery
Actions in Widgets
Creating an Action
Editors and Actions Communication
Exercise

Creating an Action 1 – Definition File

- Definition file:

bookstorebackoffice/backoffice/resources/widgets/**actions/myAction**/definition.xml

```
<action-definition id="org.myextension.action.myaction"  ... >
        <name>MyAction</name>
        <description>MyAction</description>
        <actionClassName>org.myextension.actions.MyAction</actionClassName>

        <iconUri>icons/hwicon.png</iconUri>
        <iconHoverUri>icons/hwicon_hover.png</iconHoverUri>
        <iconDisabledUri>icons/hwicon_disabled.png</iconDisabledUri>

</action-definition>
```

Creating an Action 2 – Implementation Class

Action class implements CockpitAction<I, O>

- I specifies the Input type (of data attached to ActionContext)
- O specifies the Output type (of data that will be attached to ActionResult)
  - Method signature: ActionResult<O> perform(ActionContext<I> ctx)

CreateSummary.java

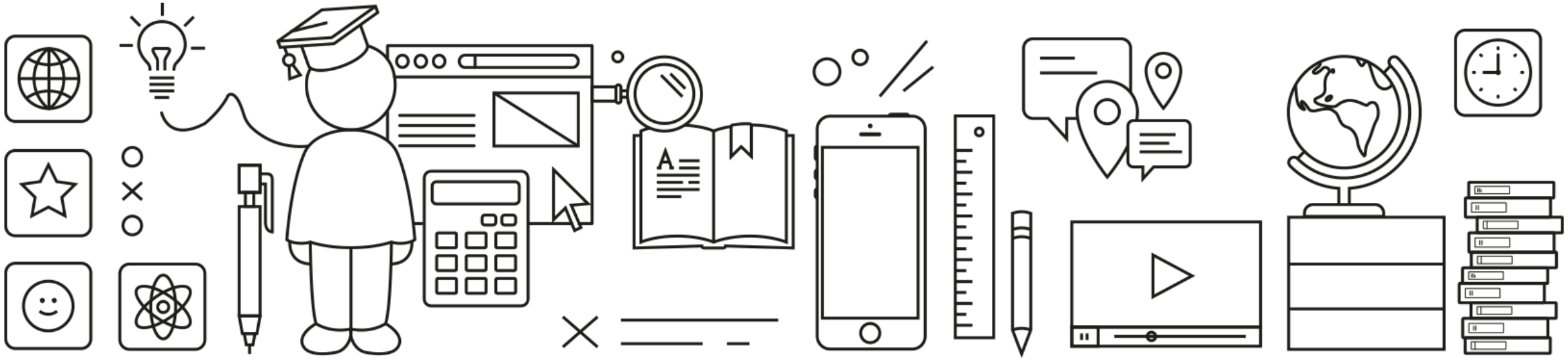
```
public class CreateSummary implements CockpitAction<BookModel, String>
{
  @Override
  public ActionResult<String> perform(ActionContext<BookModel> context)
  {
    …
```

Creating an Action 3 – Obtaining Data

- Action's *attached data* can be obtained from its context:

```
@Override                                                                    MyAction.java
public ActionResult<String> perform(final ActionContext<BookModel> context )
{
        ActionResult<String> result = null;
        final BookModel ctxBook = context.getData();
        final String bookSummary = generateSummary(ctxBook);
        if (bookSummary != null) {
                result = new ActionResult<String>(ActionResult.SUCCESS, bookSummary );
        }
        else {
                result = new ActionResult<String>(ActionResult.ERROR);
        }
        Messagebox.show(result.getData() + " (" + result.getResultCode() + ")" );
        return result;
}
```

perform() returns a new ActionResult instance:
- first constructor arg: ActionResult.***SUCCESS*** or ActionResult.***ERROR***
- second constructor arg: returned (payload) data



Editors and Actions Communication

Introduction
Available Editors Gallery
Editors in Widgets
Creating an Editor
Available Actions Gallery
Actions in Widgets
Creating an Action
Editors and Actions Communication
Exercise

Socket-Aware Editors & Actions 1 – Definition

- Editor or Action socket configuration inside `definition.xml` file
  - The XML syntax is the same as that for a widget
  - The *actual* socket-awareness is added in the implementation class

definition.xml

```
<editor-definition …>
   <name>Example Text Editor</name>
   <editorClassName>com.my.corp.backoffice.editors.ExampleEditor</editorClassName>
   …
   <sockets>
      <input id="testInput"/>
      <output id="testOutput"/>
   </sockets>
</editor-definition>
```

Socket-Aware Editors & Actions 2 – Custom Class

- Backing class must implement AbstractComponentWidgetAdapterAware:

```
public class ExampleEditor
            extends AbstractComponentWidgetAdapterAware
            implements CockpitEditorRenderer<Object>
{
  public void render(final Component parent,
    final EditorContext<Object> context, final EditorListener<Object> listener)
  {
    addSocketInputEventListener("testInput", new EventListener<SocketEvent>()
    {
      public void onEvent(final SocketEvent event)
      {
        label.setValue("Got " + event.getData() + " from widget " + event.getSourceWidgetID());
            }
          })
          ...
     sendOutput( "testOutput", context );
          ...
  }
}
```

Socket-Aware Editors & Actions 3 – Socket Methods

In the example on the previous slide:

- Use `addSocketInputEventListener("testInput", context)` to receive data through input socket
- Use `sendOutput("testOutput", context)` to send data through output socket

Socket-Aware Editors & Actions 4 – Connections

- An action/editor is NOT a widget (it's a component *within* a widget):
  - Virtual sockets not supported
  - It has no widget instance ID necessary for a <widget-connection>
    - Instead, a "STUB widget ID" is used, formed by prepending the action/editor definition ID with "STUB_"

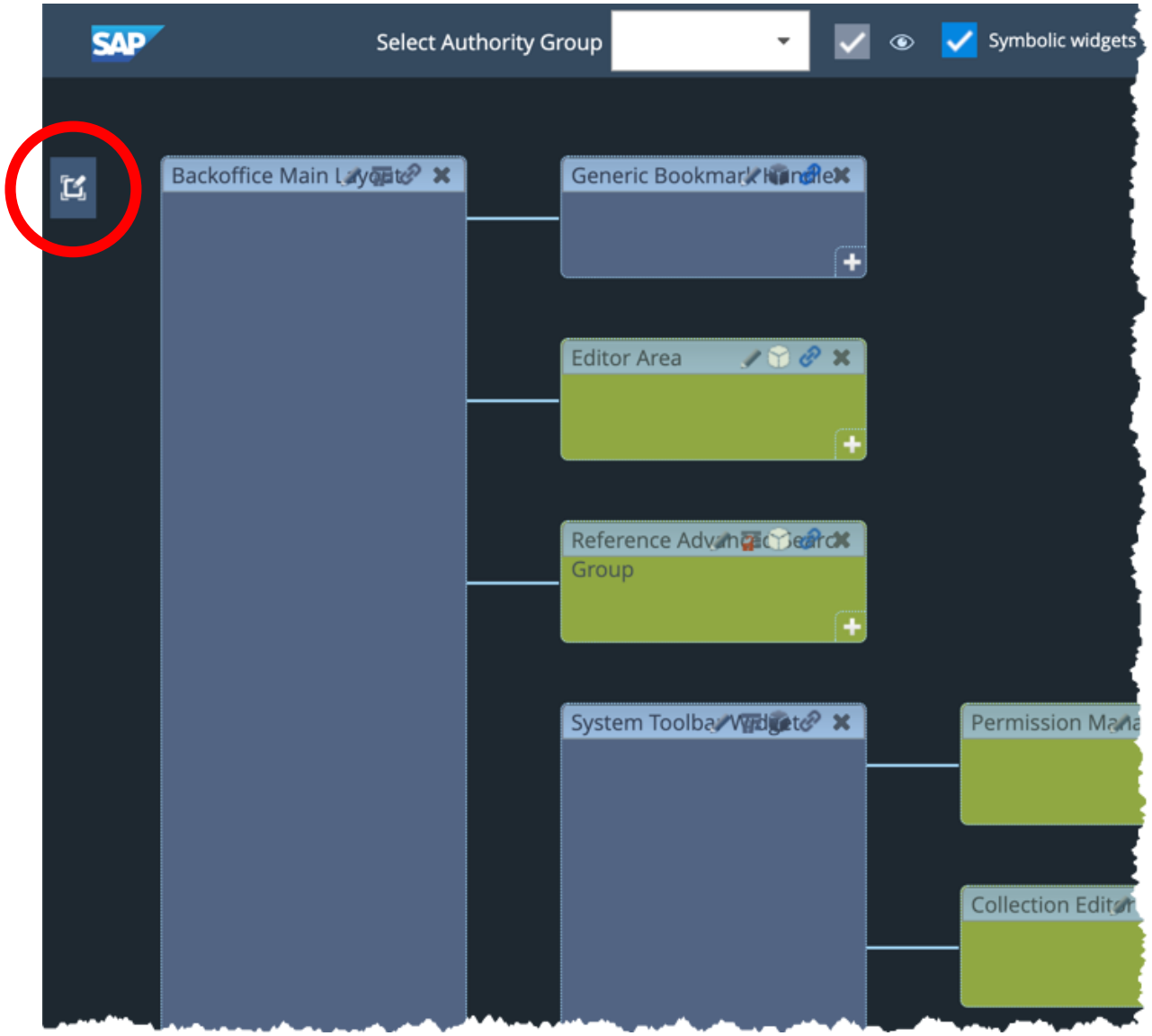
```
<widget-connection
        sourceWidgetId="STUB_com.corp.cockpitng.editor.mycustomereditor"
        outputId="objectToEdit"
        targetWidgetId="borrowerEditorArea"
        inputId="inputObject" />
```

  - Because this stub ID is based on the action/editor *definition* ID, an action/editor connection applies to ALL INSTANCES of the action/editor definition
  - Conversely, all widgets that use an instance of this action/editor will get the same communications behavior from this Action/Editor

Socket-Aware Editors & Actions 5 – Component Holder

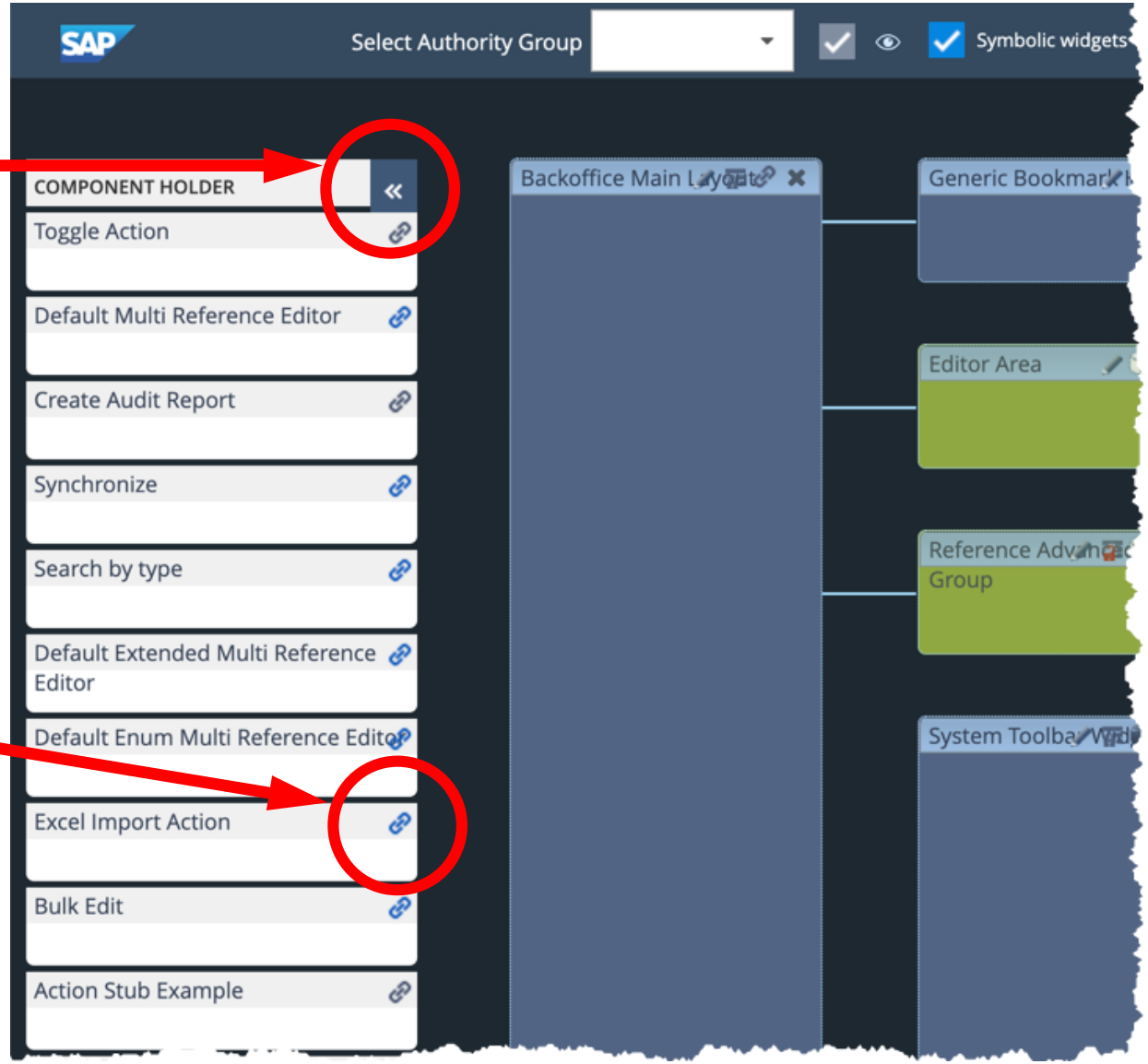
The Application Orchestrator's **Component Holder**:
(click on it to reveal contents)

- Holds all actions/editors that are "socket-aware"



Socket-Aware Editors & Actions 5 – Component Holder

- The Application Orchestrator's **Component Holder**: (expanded – click to close)
  - There is only one visual representation of a socket-aware action/editor for all instances, due to the "STUB widget ID"
- NOTE: An action/editor is ONLY able to connect to a widget instance



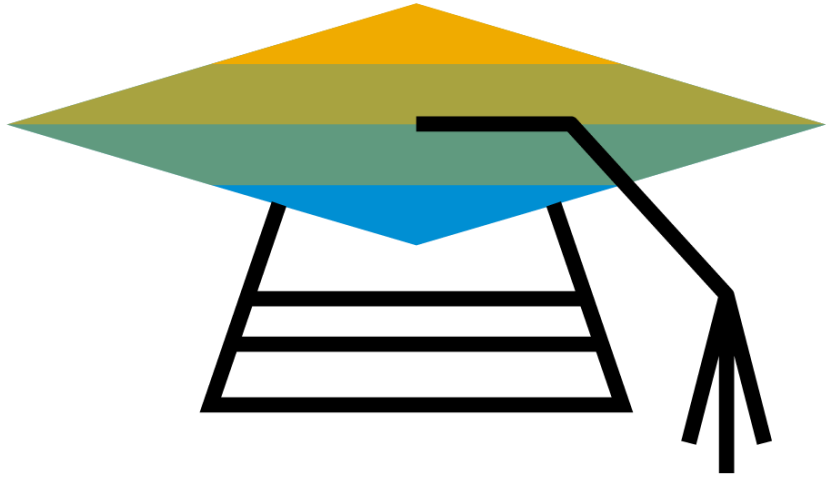
Exercise 8

Exercise 8 – Create Custom Editors and Actions

1. Create a custom editor for the publisher attribute
   - Validates the input text
   - Saves only if the input is validated
   - Gives a warning if numeric values are typed in the field

2. Create an action to toggle the *rentability* of a book
   - Books are by default not rentable
   - Create a button in the Book Details widget that switches the rentability status

Thank you.



THE BEST RUN SAP®