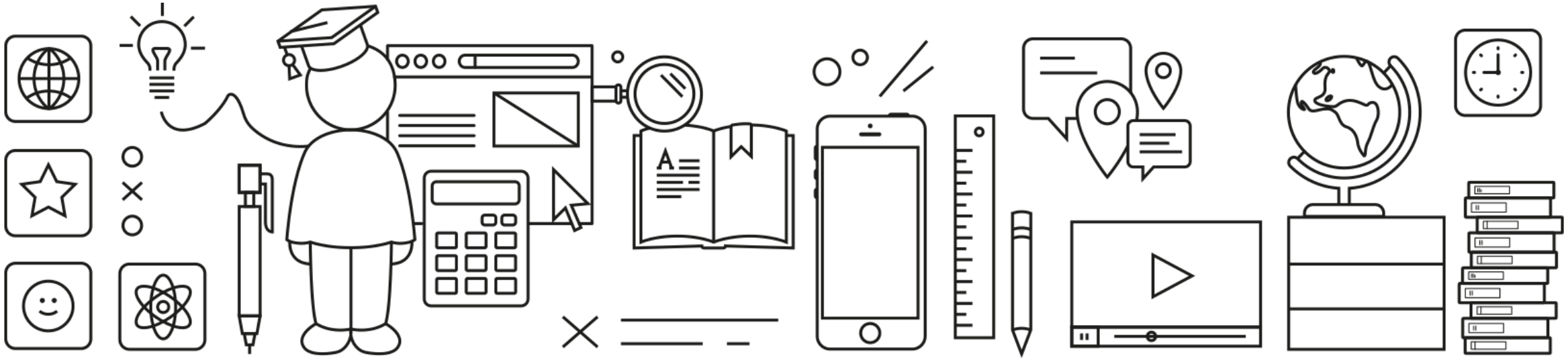
SAP Customer Experience

# SAP Commerce Cloud Backoffice Framework Developer Training

## Widget Context Configuration

THE BEST RUN SAP



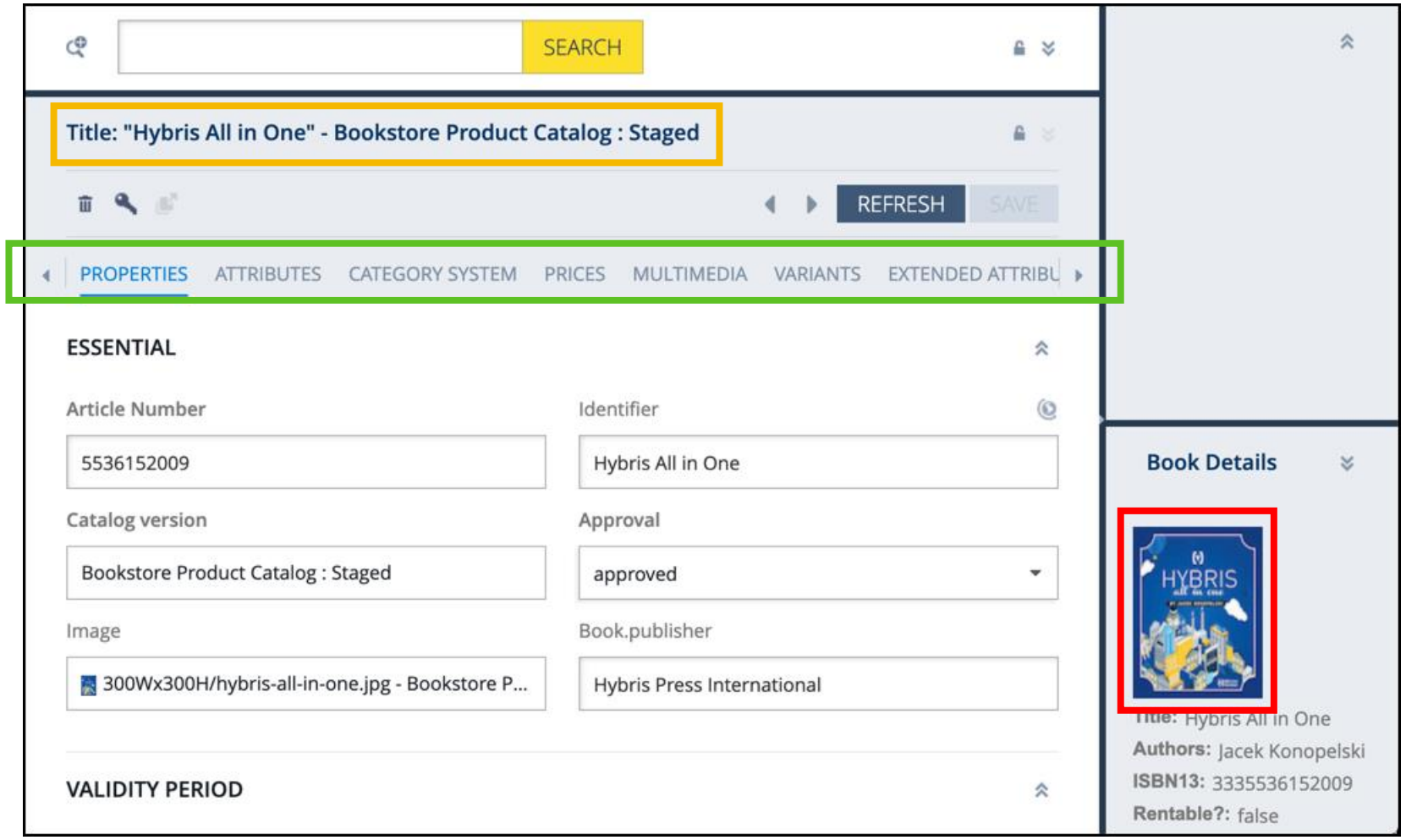
# Backoffice Context

Backoffice Context Configuration
Configuration-Data Types
Context Reuse and Merging
Orchestrate the Context
Configuration Validation
Exercise

# Flashback!

In the Book Management cockpit, you might wonder what determines:

- The label on top of the editor area and on each book in the Books list
- The tabs in the editor widget
- What the preview of a book should look like



## Backoffice Configuration – Recap

- Two types of Configuration: ***1)Application Structure***
  *2)UI Context*

1. **Application Structure**: via `*-backoffice-widgets.xml` files
   - Declarations of ALL widget instances that make up the Backoffice-Framework application – **merged into `widgets.xml`**
   - Each widget's initial "widget settings" (e.g., visibility of its slots)
   - How each widget is nested within another widget's "slot" (i.e., the hierarchal composition of widget instances)
   - Wiring of widgets (i.e., interconnections via widget sockets)

## Backoffice Configuration – Recap

- Two types of Configuration: *1)Application Structure*
  ***2)UI Context***

**2. UI Context** Config. – via `*-backoffice-config.xml` files

- Context-sensitive component-specific configs **merged into `config.xml`**
  - Each widget *instance* has its own special way of being "configured" to affect how its view is generated by the widget controller
  - Allows each instance of the same widget *type* (definition) to adapt what it displays or how it functions to its *context*ual circumstances (e.g., type of data being displayed; privileges of user / role logged in

# Examples of Backoffice *Context* Configuration

Widget views can benefit greatly from Context-Sensitive configurations

Examples:

Q: What part of the system, from the explorer tree, should be accessible to users in a certain role?

- A: Only system administrators should see the *System* node and its subtree

Q: Which fields should be visible for each item type when displayed?

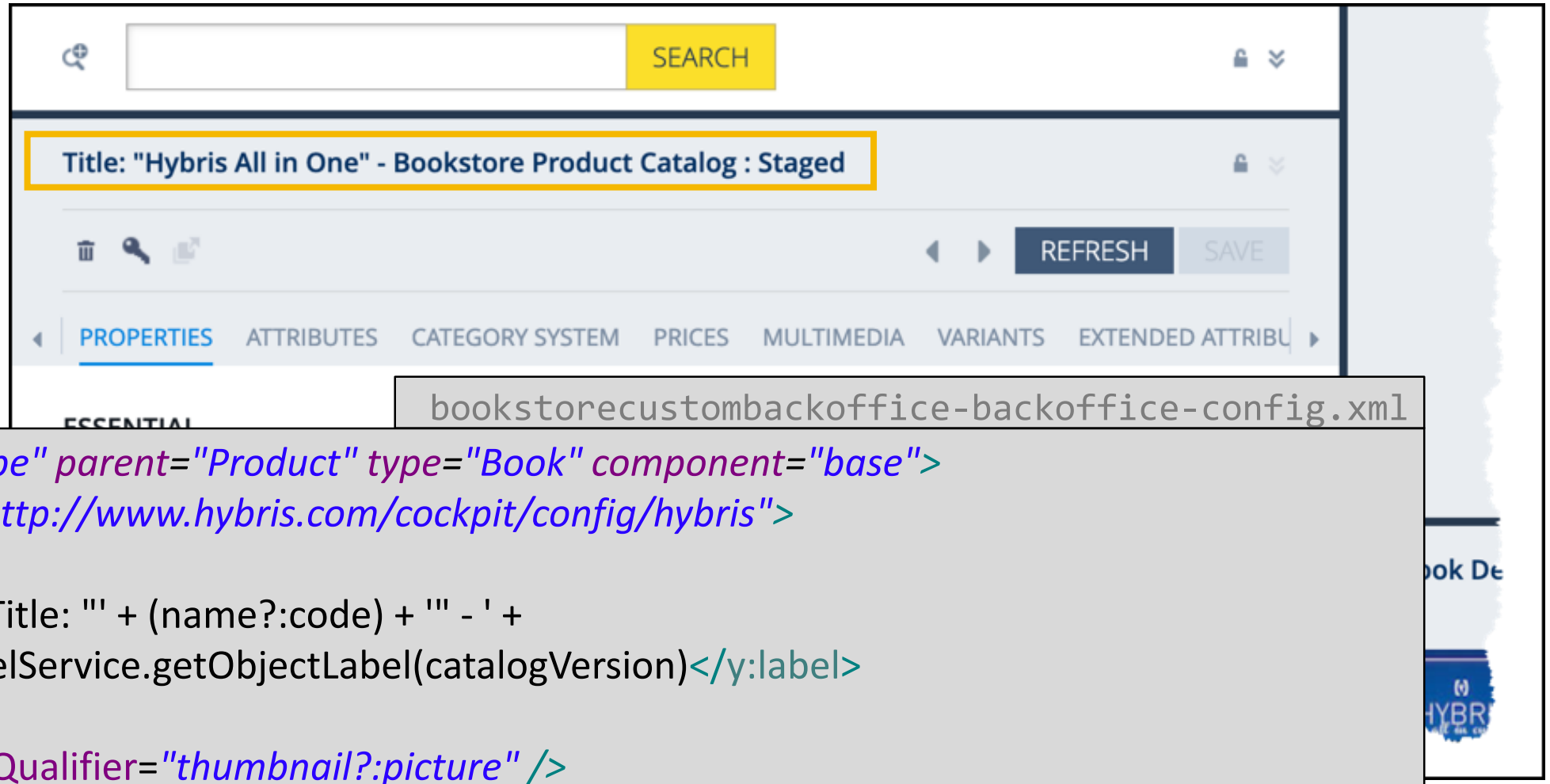
- A: Show thumbnails only for the item types that have an image associated with them

# UI Context Definitions – Recap

- XML attributes of the <context> elements used in `*config.xml`
- Allows “filtering” of which context elements to merge
  - **component** – name (non-unique) of multiple context info elements to be merged in some way
  - **merge-by** – specifies whether to “merge by” ***type*** for a "*type* hierarchy of components”, by ***principal*** for a "*user*/*UserGroup* hierarchy of components, or by ***component*** for a "*component id* hierarchy of components" (see ***parent***)
  - **principal** – config applies only if "current user" is *this* User or in *this* User Group
  - **type** – config applies only if *item type* of current item displayed is *this* type
  - **parent** – Specifies the value of the ***type***, ***principal***, or ***component*** attribute (depending on **merge-by**) of the <context> element(s) "one level higher"
  - custom attributes can be added, with some development work

# An Example of “Base” Configuration

Similar in principle to Java’s toString(): *an item type’s “most basic” way to render*




bookstorecustombackoffice-backoffice-config.xml

```
<context merge-by="type" parent="Product" type="Book" component="base">
    <y:base xmlns:y="http://www.hybris.com/cockpit/config/hybris">
        <y:labels>
            <y:label>'Title: "' + (name?:code) + '" - ' +
                @labelService.getObjectLabel(catalogVersion)</y:label>
        </y:labels>
        <y:preview urlQualifier="thumbnail?:picture" />
    </y:base>
</context>
```

# UI Configuration Precedence

- 3 different versions of the UI configuration:

`mybackofficeextension/backoffice/resources/widgets/mywidget/cockpit-config.xml`

(optional) Configuration of a specific widget. Think of it as the widget's default configuration

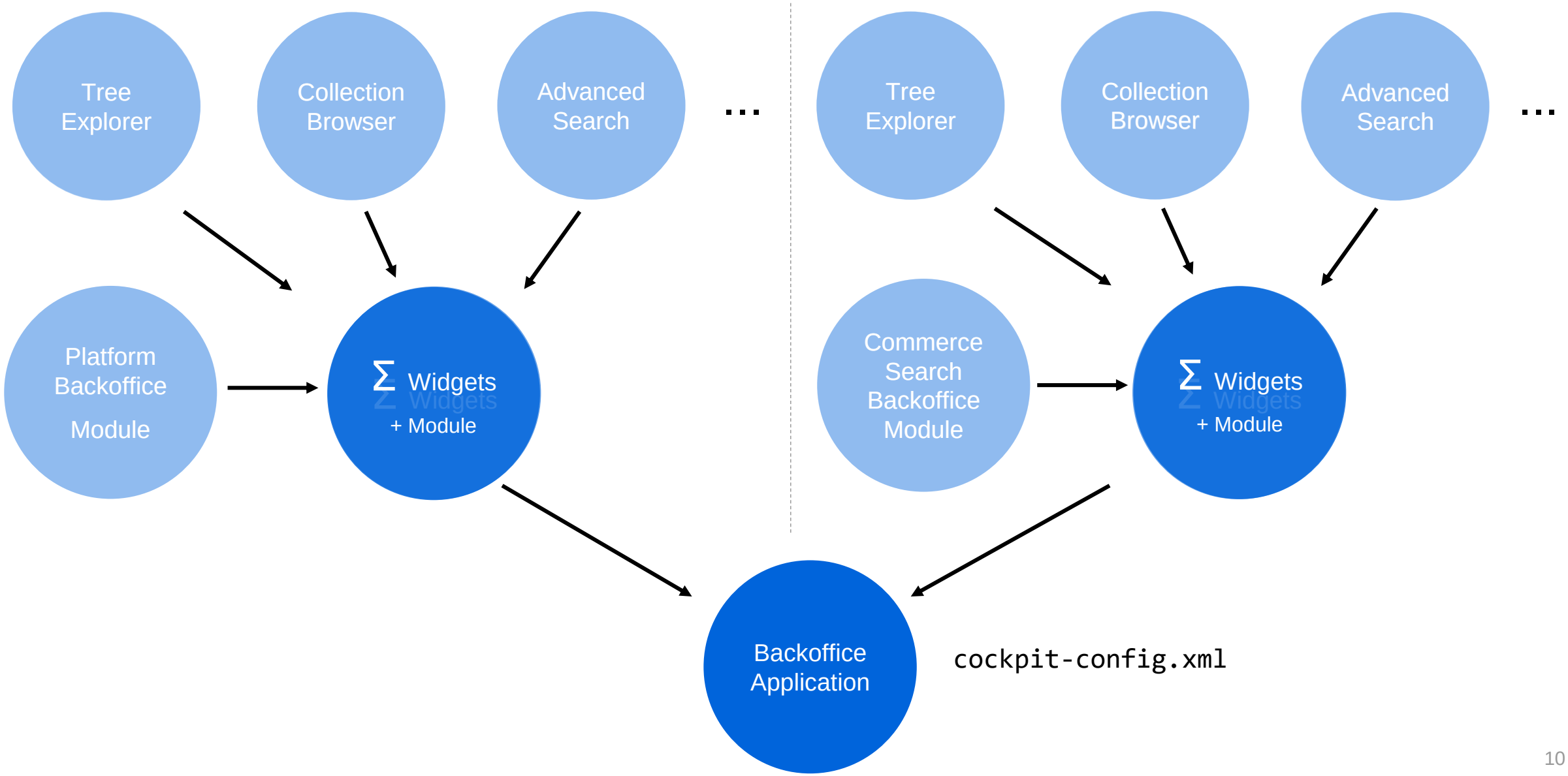
`mybackofficeextension/resources/mybackoffice-backoffice-config.xml`

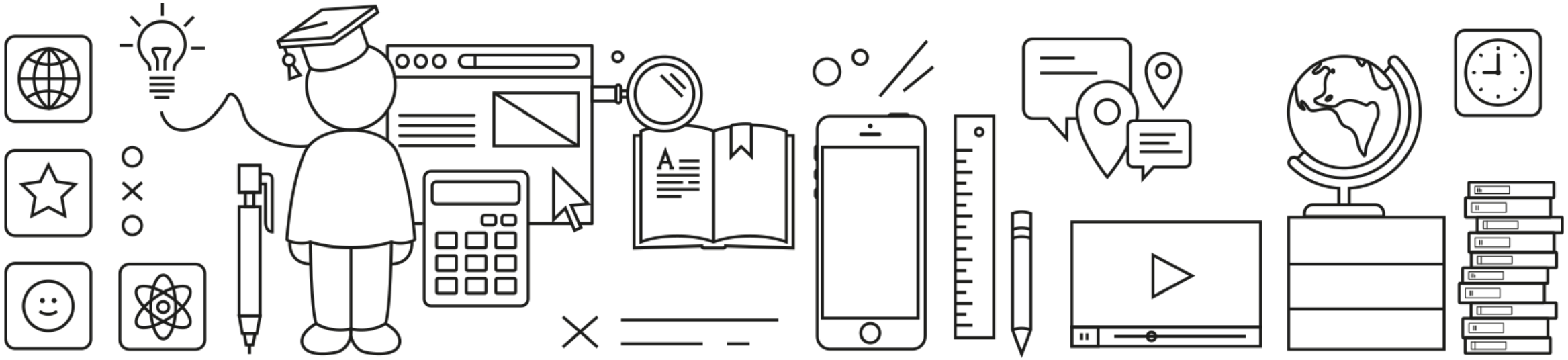
Configuration of components defined in the mybackoffice extension. It can also be loaded at runtime by doing a reset in Application Orchestrator

`cockpit-config.xml`

Merged configuration of the whole `Backoffice` application. You can modify it at runtime from Application Orchestrator --> Show cockpit-config.xml. This file is stored as an SAP Commerce Cloud Media item

# For Instance





# Configuration-Data Types

# Refreshing XML Concepts and Technologies

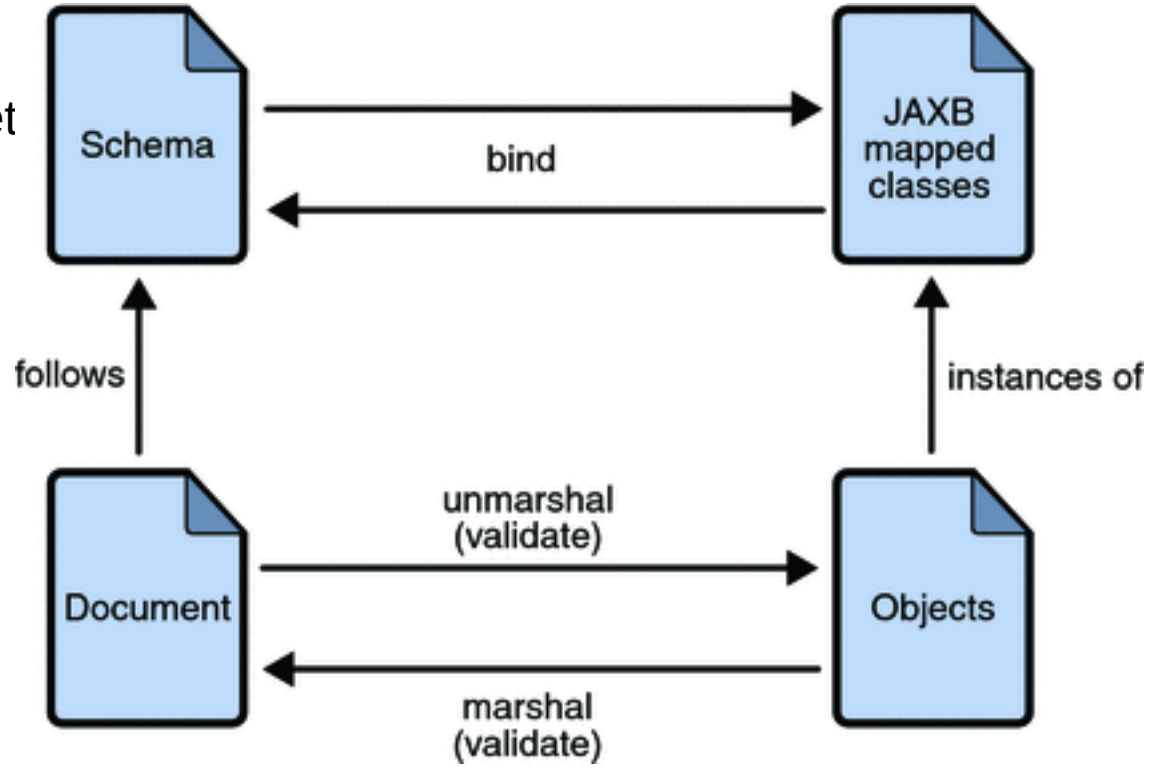
- **XML**: e**X**tensible **M**arkup **L**anguage that defines a set of rules for encoding documents in a format which is both human-readable and machine-readable.

- **XSD**: **X**ML **S**chema **D**efinition specifies how to formally describe the elements in an XML document.

- **JAXB**: **J**ava **A**rchitecture for **X**ML **B**inding (JAXB) allows Java developers to map Java classes to XML representations. Marshal Java Objects into XML and unmarshal XML into Java Objects.

# Loading Widget Configuration using JAXB

- JAXB converting XML to Java (and vice-versa)
- The idea is to generate the classes using JAXB and let from XML into Java Objects



# Configuration-Data Types

- A *configuration-data type*, *configuration type*, for short, is a composed type in XML and can be used for holding context configuration data
- Each *configuration type* is defined inside an XML Schema Definition (XSD) file
- JAXB is used to auto-generate Java classes (JAXB-annotated POJOs) that correspond to XML data conforming to your XSD types
- At runtime, we use JAXB to convert from XML data to instances of their corresponding Java JAXB classes

# The “Base” Configuration

base.xsd

```xml
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
                xmlns="http://www.hybris.com/cockpit/config/hybris"
                targetNamespace="http://www.hybris.com/cockpit/config/hybris"
                elementFormDefault="qualified" >

    <xs:element name="base" type="base"/>

    <xs:complexType name="base">
        <xs:all>
            <xs:element name="labels" type="labels" minOccurs="0"/>
            <xs:element name="preview" type="preview" minOccurs="0"/>
        </xs:all>
    </xs:complexType>

    <xs:complexType name="labels">
        <xs:all>
            <xs:element name="label" type="xs:string" minOccurs="0"/>
            <xs:element name="description" type="xs:string" minOccurs="0"/>
            <xs:element name="iconPath" type="xs:string" minOccurs="0"/>
        </xs:all>
        <xs:attribute name="beanId" type="xs:string"/>
    </xs:complexType>
    <xs:complexType name="preview">
        <xs:attribute type="xs:boolean" default="false" name="fallbackToIcon" />
        <xs:attribute type="xs:string" name="urlQualifier" />
    </xs:complexType>
</xs:schema>
```

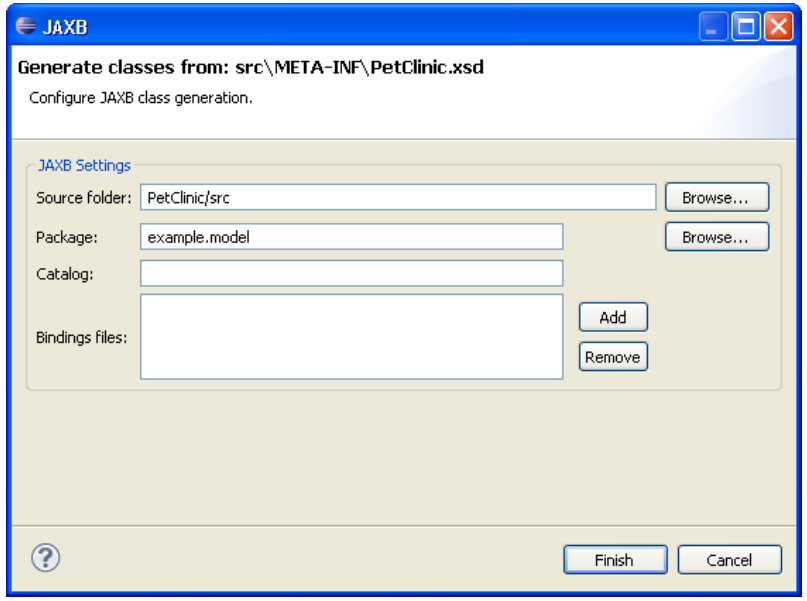
JAXB

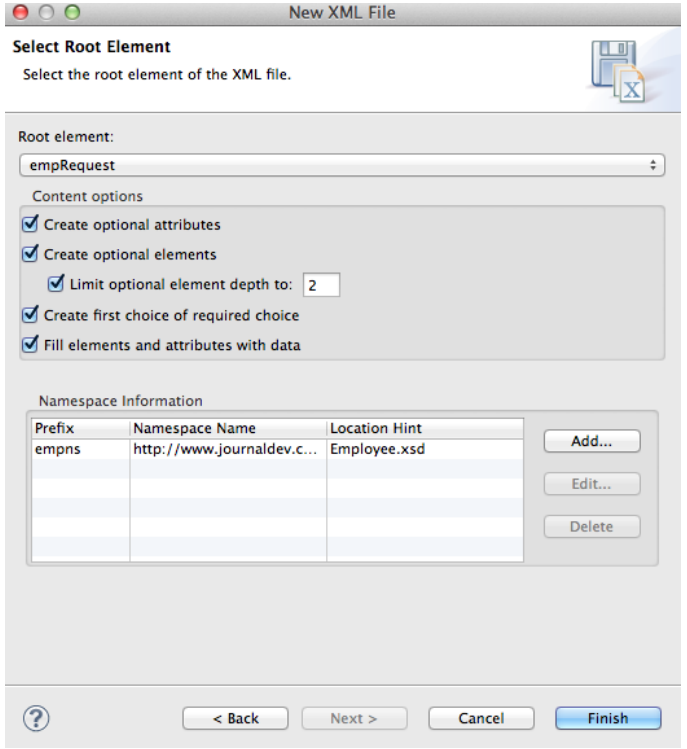
Base.java

```java
...
@XmlAccessorType(XmlAccessType.FIELD)
@XmlType(name = "base", propOrder = {} )
@XmlRootElement(name = "base")
public class Base {

   protected Labels labels;
   protected Preview preview;

   public Labels getLabels() {
      return labels;
   }


   public void setLabels(Labels value) {
      this.labels = value;
   }

   public Preview getPreview() {
      return preview;
   }

   public void setPreview(Preview value) {
      this.preview = value;
   }

}
```

# Creating New Configuration Types

- The three initial steps:
  1. Create your own XSD file that defines the widget configuration
  2. Use JAXB (via your IDE) to generate the Java classes based on the XSD file using JAXB
  3. Use JAXB (via your IDE) to generate an XML file based on the XSD and fill the configuration values



# Instantiating Configuration Types

- Place the generated configuration data (filled-in XML data structures) inside your custom Backoffice extension `*-backoffice-config.xml` file within a **context** element.

- Define the **component** attribute value (i.e., provide a name for your configuration ***context*** entry)

- Optionally, define any context such as **type** and **principal** attribute values

bookstorebackoffice-backoffice-config.xml

```
<context merge-by="type" parent="Product" type="Book" component="base">
    <y:base xmlns:y="http://www.hybris.com/cockpit/config/hybris">
        <y:labels>
            <y:label>'Title: "' + (name?:code) + '" - ' +
                @labelService.getObjectLabel(catalogVersion)</y:label>
        </y:labels>
        <y:preview urlQualifier="thumbnail?:picture" />
    </y:base>
</context>
```

# Instantiating Configuration Types – Alternate Style

- Instead of specifying all attributes in a single element (as in the first example below), <context> elements can also be nested (as in the second example). This can sometimes improve readablility:

bookstorebackoffice-backoffice-config.xml

```
<context component="base" merge-by="type" type="Book" parent="Product" >
    <y:base xmlns:y="http://www.hybris.com/cockpit/config/hybris">...</y:base>
</context>
<context component="base" merge-by="type" type="ReadingLight" parent="Product" >
    <y:base xmlns:y="http://www.hybris.com/cockpit/config/hybris">...</y:base>
</context>
```

bookstorebackoffice-backoffice-config.xml

```
<context component="base" merge-by="type" >
    <context type="Book" parent="Product" >
        <y:base xmlns:y="http://www.hybris.com/cockpit/config/hybris">...</y:base>
    </context>
    <context type="ReadingLight" parent="Product" >
        <y:base xmlns:y="http://www.hybris.com/cockpit/config/hybris">...</y:base>
    </context>
</context>
```

# Relation between XSD and Configuration

**cockpit-config.xml**

```
<context component="tree" ...>
    <tree:tree xmlns:tree="...">
    <tree:nav-node id="..." ...>
    ...
</context>
```

**Create widget's default configuration from generated XML**

**Generated XML Configuration**

```
<?xml version="1.0" encoding="UTF-8"?>
<tns:tree title="" xmlns:tns=...>
    <tns:nav-node ...>
...
```

**Generate XML**

**tree.xsd**

```
<xs:schema ...>
    <xs:element name="tree">
    ...
        <xs:element name="nav-node">
...
```

**Create contextual widget configuration from generated XML**

**<extension>-backoffice-config.xml**

```
...
<context component="tree" ...>
    ...
</context>
<context component="tree" principle="admin" ...>
    ...
</context>
...
```

**Generate JAXB Class**

**Tree.java -> Tree.class**

```
@XmlAccessorType(...)
@XmlRootElement(name ="tree")
public class Tree {
...
```

## Reading the Widget Configuration

- In the widget's controller (examples to follow):

1. Instantiate and initialize a `DefaultConfigContext` object via its constructor:
   - Specify the configuration **`component`** *id* to request
   - Optionally, specify **`type`** and/or **`principal`** (either via constructor or setter methods) to enable filtering when Backoffice merges configuration components

2. Call `getWidgetInstanceManager().loadConfiguration()`
   - Pass-in a reference to the initialized `DefaultConfigContext` instance
   - Pass-in the JAXB type (i.e., java.lang.Class<Type>) representing the `JAXB` class to return (and, thus, the <context> body's XSD type to expect)

# Reading the Widget Configuration, cont.

To *obtain* a configuration from Backoffice:

BookDetailsController.java

```
...
final DefaultConfigContext configContext = new     DefaultConfigContext(getWidgetSettings().getString(WIDGET_SETTING_CONFIG_CONTEXT));
...
baseCfg = Optional.ofNullable(getWidgetInstanceManager().loadConfiguration(configContext, Base.class));
...
```

It’s also possible to *store* a configuration:

BookDetailsController.java

```
getWidgetInstanceManager().storeConfiguration(configContext, Base.class)
```

# Reading-in the Widget Configuration – Diagram

**definition.xml**

- unique ID
- view
- controller
- sockets
- settings
  - ***config Id***

**Tree.java -> Tree.class**

```
@XmlAccessorType(…)
@XmlRootElement(name="tree")
public class Tree {
… }
```

**Merged cockpit-config.xml**

```
…
<context component="tree" …>
   …
</context>
<context … principle="admin"  …>
   …
</context>
…
```

**Marshalling / Unmarshalling**

**JAXB instance**

**TreeController.java**

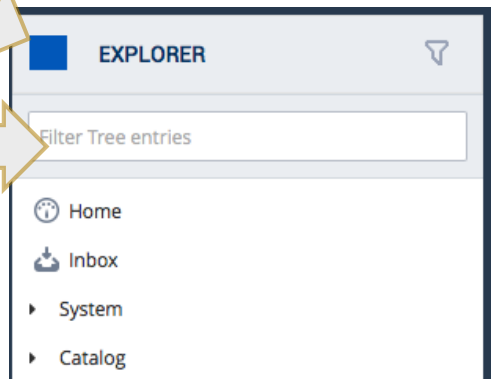
```
…
Tree treeConfig = WidgetInstanceManager
                    .loadConfiguration(ConfigContext, config type)
```

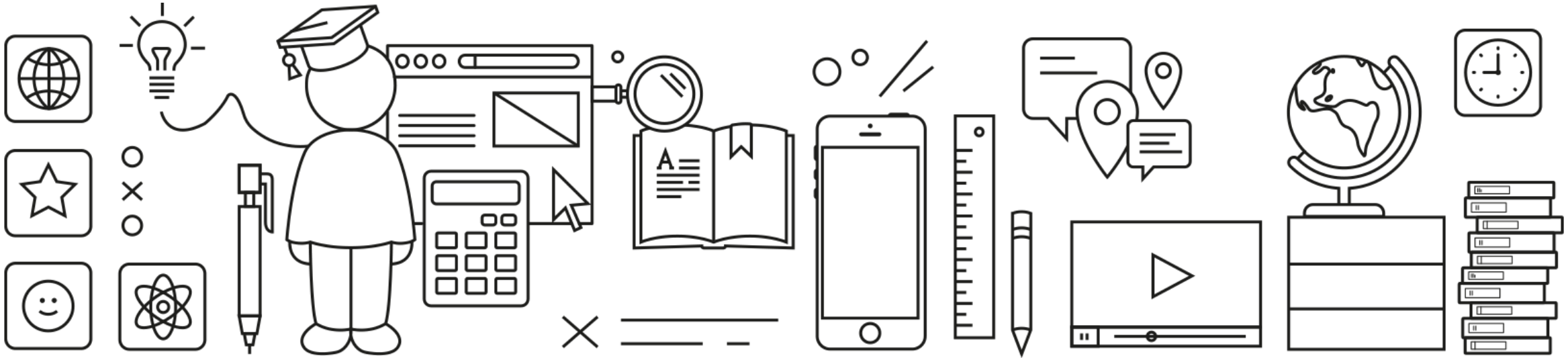
*<manipulate treeConfig and render>*

**view.zul**

```
…
<widget …>
  …
  <panel …>
  <caption …>
    <div>
…
```

**Widget Instance**





# Context Reuse and Merging

## Reusing and Merging Widget Configurations

- Use **`merge-by`** and **`parent`** <context> element XML attributes:
  - **`merge-by`**: <context> attribute specifying the kind of merge, thus the kind of *parent* the merge should be using
    - Possible values: **`type`** | **`principal`** | **`component`** | ~~**`module`**~~
      *(**`module`** is deprecated – value set automatically based on containing extension)*
  - **`parent`**: <context> attribute specifying the next context item in the hierarchy of config components to be merged

    *(examples follow)*

## Example 1 of Merging Widget Configurations

```
<context        component="abc"  merge-by="type"
                type="Book"  parent="Product" >
```

If your widget controller's DefaultConfigurationContext instance was **{component: abc;  type: Book}**

1. Backoffice would find the matching <context> element
2. Merge-in (inherit) the context info for this element's *parent* config element "in the hierarchy, **{component: abc;  type: Product}**
3. Merge-in (inherit) the context info for (presumably) *the next parent* config element, **{component: abc;  type: GenericItem}**, i.e., the *item type* that *this* context element specified (presumably) as *its* parent in the **type** hierarchy, and so on.

## Example 2 of Merging Widget Configurations

```
<context        component="xyz"  merge-by="principal"
                principal="admin"  parent="admingroup" >
```

If your widget controller's DefaultConfigurationContext instance was **{component: xyz;  principal: admin}**

1. Backoffice would find your <context> entry where both values match
2. Merge-in (inherit) the context info for this element's *parent* config element, **{component: xyz;  principal: admingroup}**
3. Merge-in (inherit) the context info for (presumably) *the next parent* element in the hierarchy, **{component: xyz;  principal: backofficeadmingroup}**, i.e., the *User Group* or *Role* that *this* context element specified as *its* parent in the **principal** hierarchy (i.e., the *User Group* or *Role* of which *admingroup* is a member), and so on.

# An Illustrated Example Merging Widget Configurations

shoestorebackoffice-backoffice-config.xml

```
<context component="ShoeDetails" principal="productManagers"
    <property name="size" />
    <property name="brand" />
</context>

<context component="ShoeDetails" principal="stockManagers">
    <property name="width" />
</context>
```

principal="stockManagers"

```
{
    property name="size",
    property name="brand"
}
```

```
{
    property name="size",
    property name="brand",
    property name="width"
}
```

*Merged view-configuration* data a particular widget's controller receives.

(this widget wants to know, "in this contextual situation, what properties should I display, and in what order?" )

```
{
  component="ShoeDetails"
  principal="stockManagers"
  type=""
}
```

DefaultConfigContext instance used for loading configuration

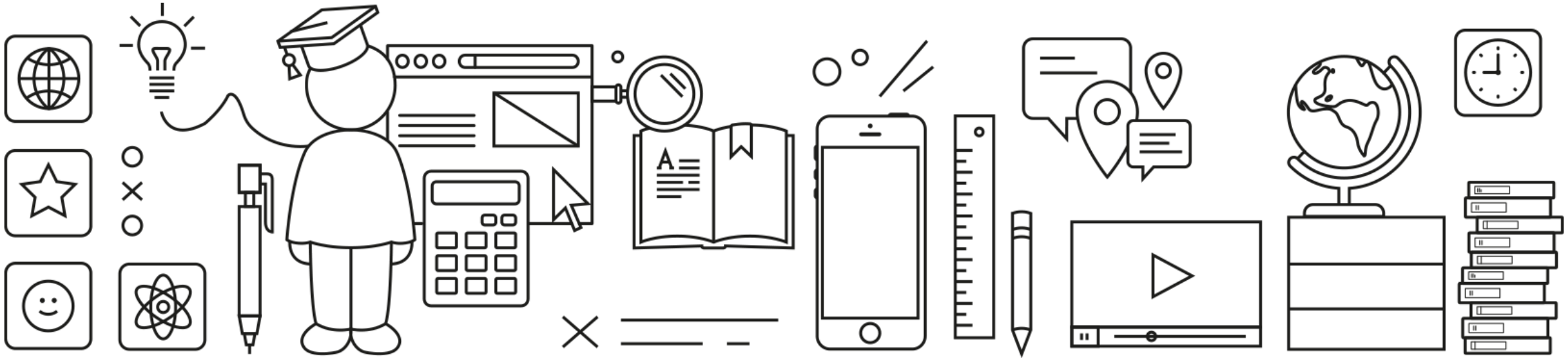
# What Properties are Mergeable?

- When defining a *configuration type*, ***mergeable*** properties should be specified.

- Add the `@Mergeable` annotation to the property list inside the JAXB-generated root configuration class. Set the **`key`** to a property name inside the **`List<AttributeType>`** of attributes

```
@Mergeable(key = "code")
protected List<AttributeType> attributes;
```

- `AttributeType` has a property called `code`, so Backoffice will consider its value when merging for each `AttributeType` inside the `attributes` list.

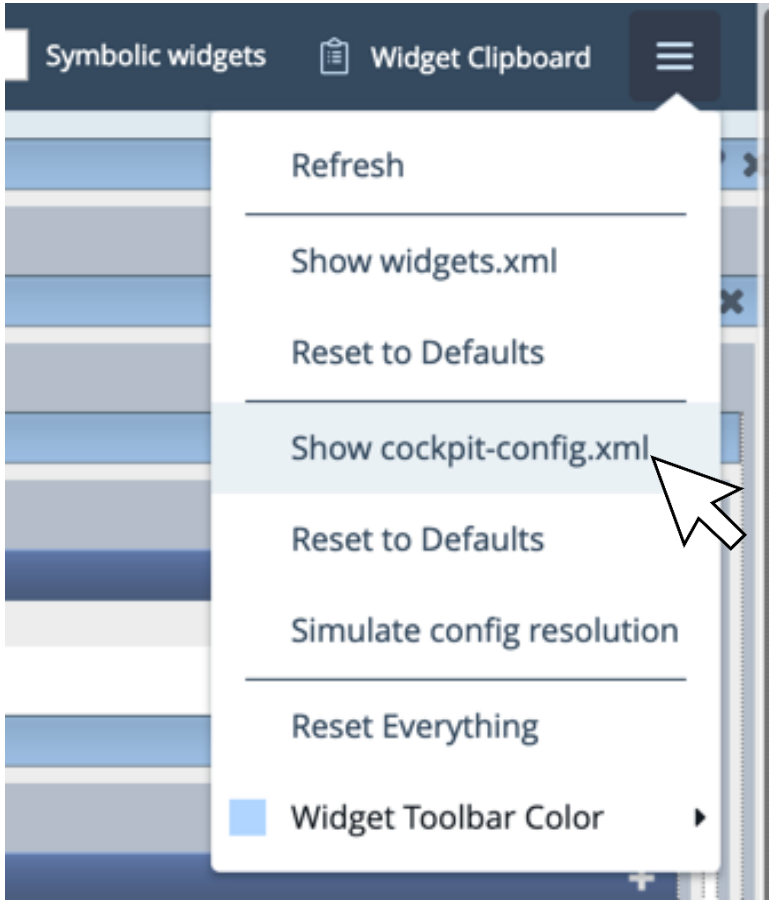


# Orchestrate the Context

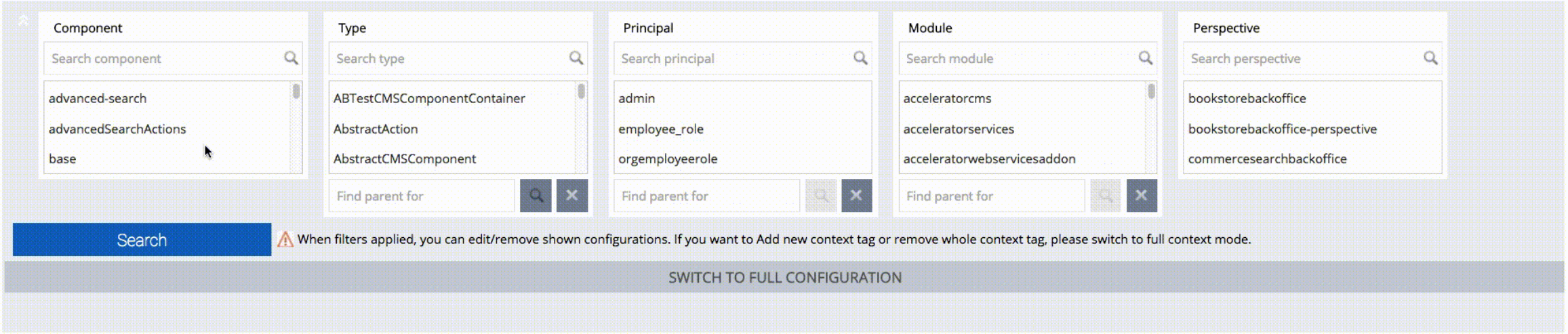
# Changing Widget Configuration On The Fly

- Using Application Orchestrator mode, you can modify widget configuration and see the changes at *runtime* (without restarting the server)

- Useful and fast way to test your widget's configuration



# Cockpit-config Filters




```xml
<?xml version="1.0" encoding="UTF-8"?><config xmlns="http://www.hybris.com/cockpit/config">
  <context type="com.hybris.cockpitng.util.Range" component="base">
    <y:base xmlns:y="http://www.hybris.com/cockpit/config/hybris">
      <y:labels beanId="rangeLabelProvider"/>
        </y:base>
  </context>
  <context type="java.lang.String" component="simple-list">
    <ysl:simple-list xmlns:ysl="http://www.hybris.com/cockpitng/config/simplelist">
      <ysl:name field="#root"/>
      <ysl:description field="class.canonicalName"/>
        </ysl:simple-list>
  </context>
  <context type="java.lang.Object" component="listview">
    <list-view:list-view xmlns:list-view="http://www.hybris.com/cockpitng/component/listView">
      <list-view:column label="hmc.value" qualifier="toString()"/>
        </list-view:list-view>
```

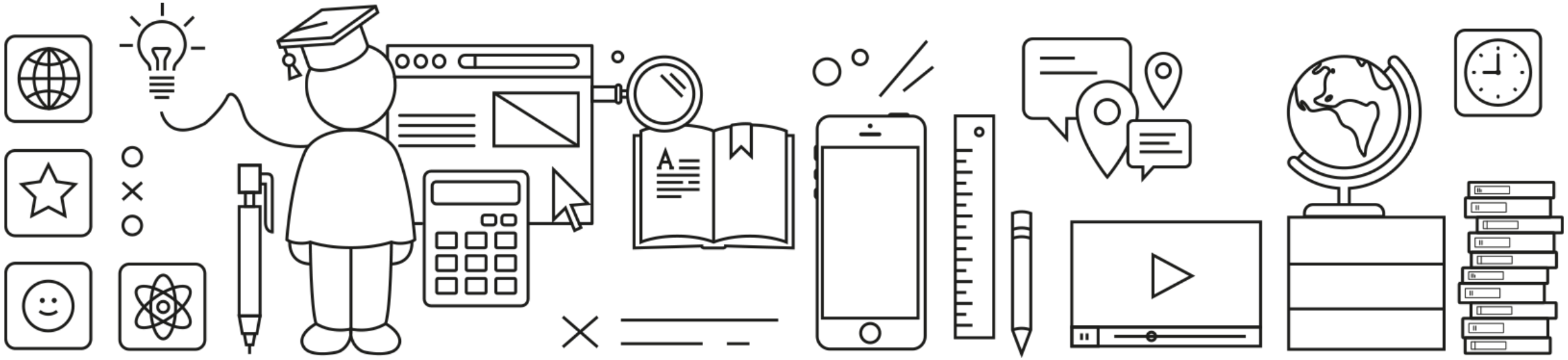
You can filter the configuration file by providing the relevant values for the available attributes!

# Change at Runtime

```
ıg"/>
ıct"/>
'/>
.ine"/>
ted"/>
:iontime"/>
ovalStatus"/>
ıage"/>

' component="editor-area" module="customerreview">
ea="http://www.hybris.com/cockpitng/component/editorArea">
ıunity" position="7">
.user.sections.customerreviews">
```

Close | Store | Validate

Unlike `widgets.xml`, you can modify, *validate*, and *store* your changes to `cockpit-config.xml` inside Application Orchestrator.



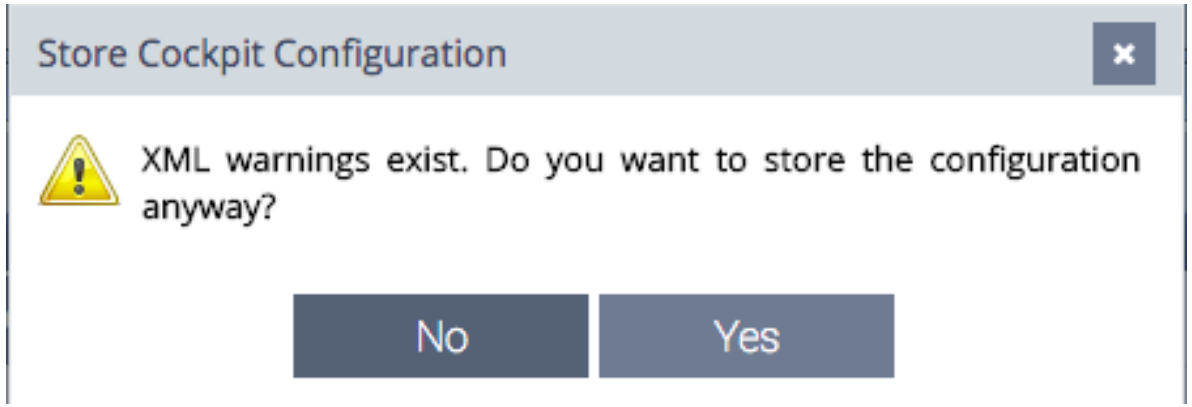
# Configuration Validation

# XSD Validation

- Context/UI configuration can be validated against the XSD of the configuration types
- Done by default when editing in Application Orchestrator
  - Can be controlled through this property:

    `backoffice.cockpitng.validate.cockpitConfig.orchestrator`

  - `true` by default
- If the configuration is not valid you get a warning when trying to store it



# Configuration Validation Everywhere

- Validation is not done by default if you change the configuration files directly rather than through Application Orchestrator
  - Can be controlled through this property:

    `backoffice.cockpitng.validate.cockpitConfig.onstartup`

  - `false` by default

- You'll get a warning in the system console:

```
$ ./hybrisserver.sh
```

```
WARN  [hybrisHTTP1] [SchemaConfigValidator] Warnings occurred while processing configuration file:
SCHEMA VIOLATION: org.xml.sax.SAXParseException;...
```

# Exercise 7

## Exercise 7 – Configure the Context

Take advantage of the power that context configuration gives you by:

- Defining new configuration types
- Reusing (merging-in) existing configuration

## 1. Create a New Configuration Type

Create a new configuration type for the Book Details widget.

It should specify the sequence of Book properties that you would like the widget to display, but only when viewing a Book.

You'll need to:

- Use a schema (provided for you) for the new configuration type
- Use JAXB to convert the schema into JAVA classes

# 2. Define New Configuration

- The Book Details widget already uses configuration data of type Base.
- This existing configuration is used to configure the image and label of a book.

bookstorecustombackoffice-backoffice-config.xml

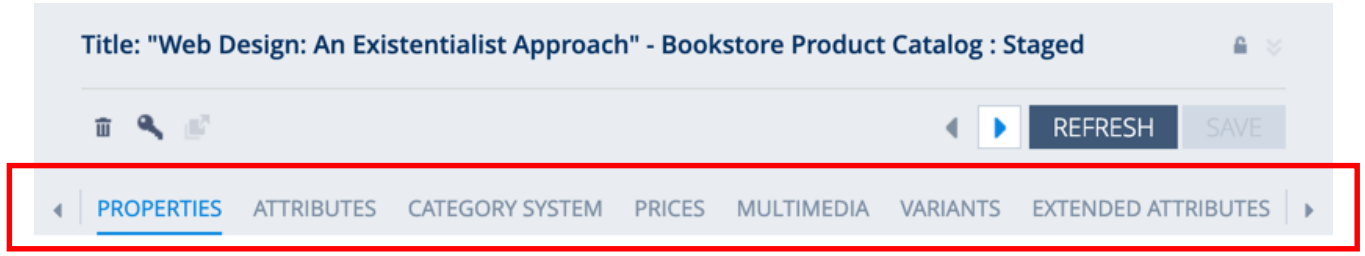
```
<context merge-by="type" parent="Product" type="Book" component="base">
    <y:base xmlns:y="http://www.hybris.com/cockpit/config/hybris">
        <y:labels>
            <y:label>'Title: "' + (name?:code) + '" - ' + @labelService.getObjectLabel(catalogVersion)</y:label>
        </y:labels>
        <y:preview urlQualifier="thumbnail?:picture" />
    </y:base>
</context>
```

- The other properties that are shown by your widget are currently hard-coded into the controller's `render()` method.
- Define (and use) a new XML configuration-data type to allow you to configure which properties should be shown by the Book Details widget.

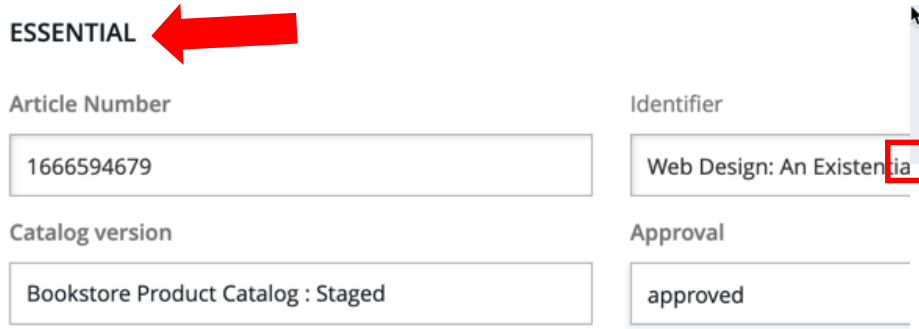
# 3. Compare Configuration for Different Types

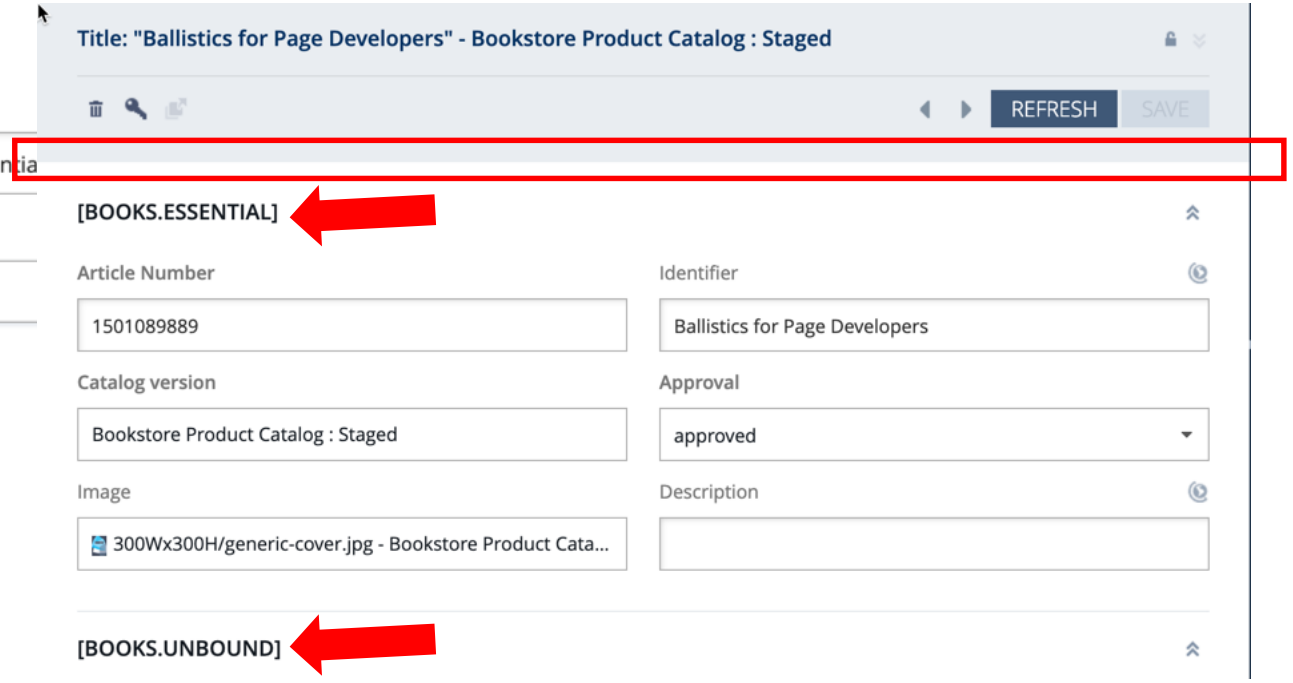
See how the editor for type Product looks in the Admin cockpit

- Compare it with how the editor for type Book looks in the Book Management cockpit.



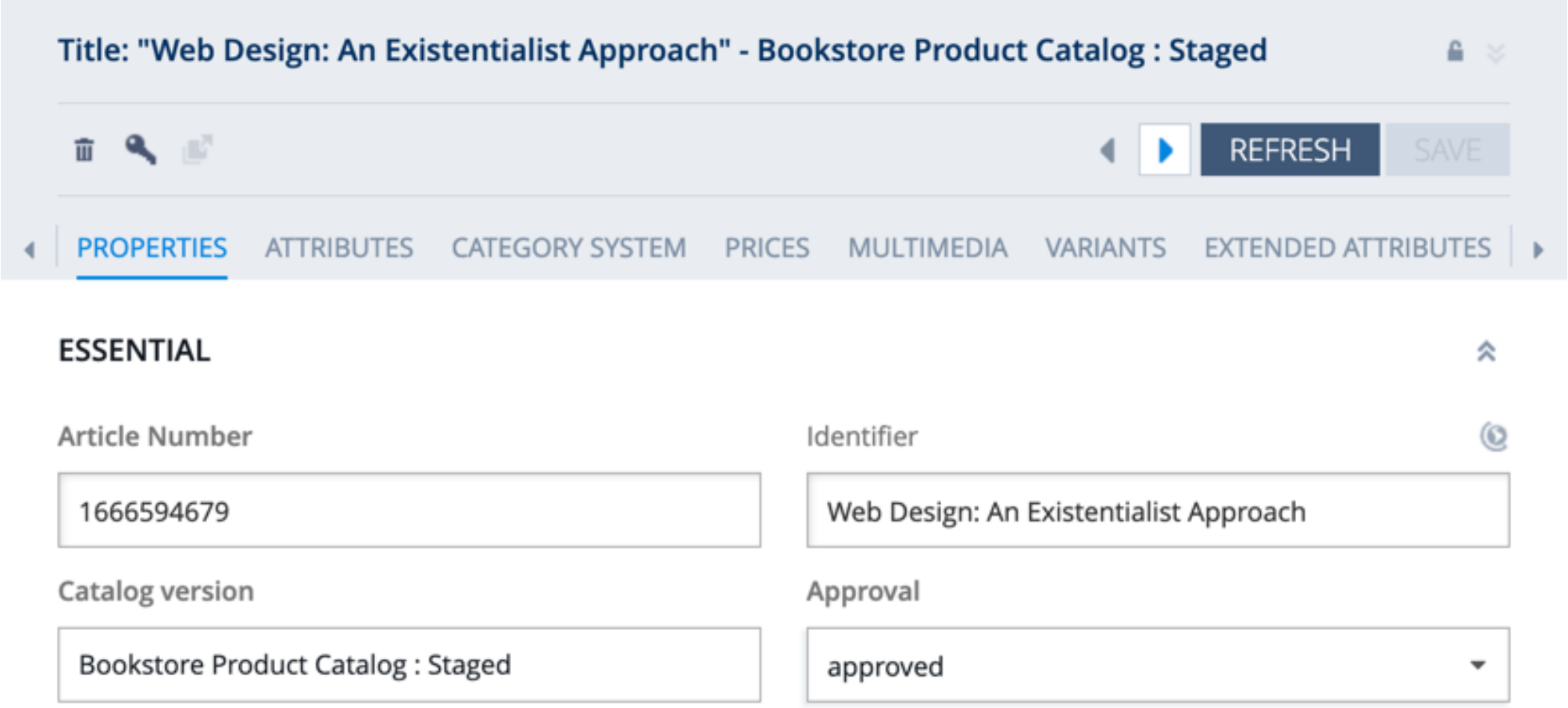

(Notice: no tabs and
no proper section headers)



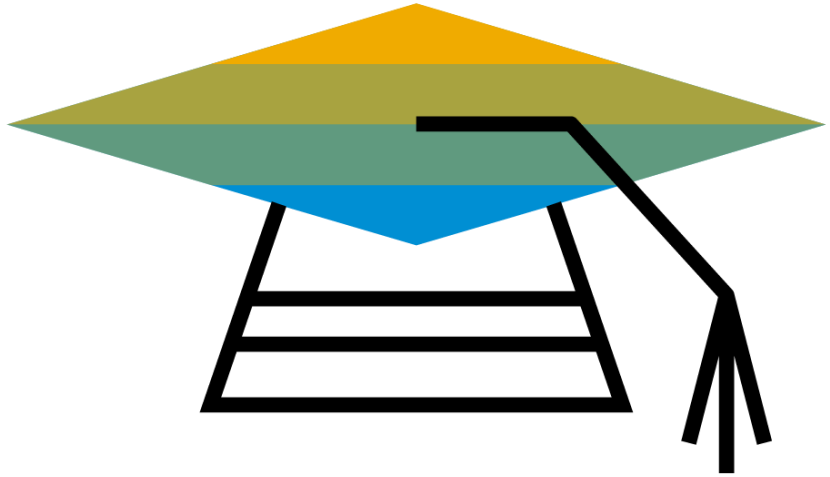



# 4. Reuse Existing Configuration

Reuse Product type’s configuration in the Admin cockpit to improve the Book type’s editor view in the Book Management cockpit.



# Thank you.



THE BEST RUN SAP®