Escenic Widget Framework

Developer Guide

3.0.3.154583

escenic

A CCI COMPANY

Table of Contents

1 Introduction

The Escenic Widget Framework is an add-on product for the Escenic Content Engine that greatly simplifies the process of designing publications. Without the Widget Framework, publication design requires considerable HTML and JSP programming skills. With the Widget Framework, publications can be designed using a drag-and-drop interface in Escenic Content Studio (the Escenic content editor).

This manual is a user guide for:

- Template developers who want to extend/modify the Widget Framework to fit their needs
- System administrators who are installing the Widget Framework
- Publication designers who want to use the Widget Framework to design Escenic publications

The prerequisites for using this manual are:

- You have some knowledge of HTML, CSS, JSP, Javascript and Java (but you don't necessarily need to know **all** of these technologies to find the manual useful)
- You are familiar with the general structure of Escenic publications
- You know how to use Content Studio for editorial purposes
- You know how to create new Escenic publications

1.1 What is The Widget Framework?

The Escenic Content Engine is a **template-based** publishing system, in which content production is completely separated from layout design. This allows writers and editors to concentrate on the production of content without needing to think about layout, and allows designers to ensure that a publication has a consistent, well-designed appearance. Web pages are generated by combining content items written and edited using Content Studio with templates written in HTML/JSP.

This approach works well, but it has some disadvantages:

- It requires designers to have HTML and JSP programming skills in addition to design skills
- It makes publication design a relatively slow and error-prone process, with the result that:
  - Publications cannot easily be redesigned for special occasions
  - The production of ad-hoc extra publications is difficult and in general, too costly

The Widget Framework solves this problem by enabling publication designers to assemble templates from a library of predefined template fragments called **widgets**. In this way it is possible to build a complete set of templates for a publication in a fraction of the time it would take to write, test and debug templates by hand.

1.2 What Will You Learn From This Manual?

This manual contains, among other things, information on how to:

Copyright © 2010-2014 Escenic AS

- Install Widget Framework distributions
- Access and use the Escenic Maven repository
- Combine multiple Widget Framework distributions
- Install custom widgets alongside the standard Widget Framework distributions
- Support multiple languages on Widget Framework sites
- Create your own widgets
- Add layout grids to Widget Framework publications
- Import and export widgets and templates
- Modify the behaviour and appearance of the standard widgets in various ways (adding themes, writing your own controllers, overriding messages etc.)

Copyright © 2010-2014 Escenic AS

2 Installation

This chapter describes how to install the Widget Framework Core distribution.

2.1 Preconditions

The following preconditions must be met before you can install the Widget Framework Core:

- Version 5.6 of the Content Engine (plus the Escenic assembly tool) has been installed as described in the Escenic Content Engine Installation Guide and is in working order.
- The following plug-ins have been installed as described in the relevant plug-in guides, and are in working order:
  - **Forum** (3.1 or later)
  - **Menu Editor** (2.1 or later)
  - **Analysis Engine** (2.4 or later)
- You have the required plug-in distribution file (`widget-framework-core-3.0.3.154583.zip`). This file can be downloaded from http://documentation.vizrt.internal/ece-download-matrix/5.6/.

2.2 Conventions

The instructions in the following section assume that you have a standard Content Engine installation, as described in the Escenic Content Engine Installation Guide. *escenic-home* is used to refer to the `/opt/escenic` folder under which both the Content Engine itself and all plug-ins are installed.

The Content Engine and the software it depends on may be installed on one or several host machines depending on the type of installation required. In order to unambiguously identify the machines on which various installation actions must be carried out, the **Escenic Content Engine Installation Guide** defines a set of special host names that are used throughout the manual.

Some of these names are also used here:

assembly-host
The machine used to assemble the various Content Engine components into an enterprise archive or .EAR file.

engine-host
The machine(s) used to host application servers and Content Engine instances.

editorial-host
engine-host(s) that are used solely for (internal) editorial purposes.

The host names always appear in a bold typeface. If you are installing everything on one host you can, of course, ignore them: you can just do everything on the same machine. If you are creating a larger multi-host installation, then they should help ensure that you do things in the right places.

Copyright © 2010-2014 Escenic AS

2.2.1 Installation Procedure

Installing the Widget Framework involves the steps listed below

1. Log in as **`escenic`** on your **assembly-host**.
2. Download the required distribution file from the Escenic Technet web site (http://technet.escenic.com/ewf/article5419.ece). If you have a multi-host installation with shared folders as described in the **Escenic Content Engine Installation Guide**, then it is a good idea to download the distribution to your shared **`/mnt/download`** folder:

   ```
   $ cd /mnt/download
   $ wget http://user:password@technet.escenic.com/downloads/release/5.6/widget-framework-core-3.0.3.154583.zip
   ```

   Otherwise, download it to some temporary location of your choice.
3. If the folder **`/opt/escenic/engine/plugins`** does not already exist, create it:

   ```
   $ mkdir /opt/escenic/engine/plugins
   ```
4. Unpack the distribution file:

   ```
   $ cd /opt/escenic/engine/plugins
   $ unzip /mnt/download/widget-framework-core-3.0.3.154583.zip
   ```

   This will result in the creation of an **`/opt/escenic/engine/plugins/widget-framework-core`** folder.
5. Log in as **`escenic`** on your **assembly-host**.
6. Run the **`ece`** script to re-assemble your Content Engine applications.

   ```
   $ ece assemble
   ```

   This generates an EAR file (**`/var/cache/escenic/engine.ear`**) that you can deploy on all your **engine-host**s.
7. If you have a single-host installation, then skip this step.

   On each **engine-host** where you wish to run the plug-in, copy **`/var/cache/escenic/engine.ear`** from the **assembly-host**. If you have installed an SSH server on the **`assembly-host`** and SSH clients on your **`engine-host`**s, then you can do this as follows:

   ```
   $ scp -r escenic@assembly-host-ip-address:/var/cache/escenic/engine.ear /var/cache/escenic/
   ```

   where *assembly-host-ip-address* is the host name or IP address of your **assembly-host**.
8. On each **engine-host** where you wish to run the plug-in, deploy the EAR file and restart the Content Engine by entering:

   ```
   $ ece deploy
   $ ece restart
   ```

2.2.2 Verify the Installation

To verify the status of the plug-in you have installed, open the Escenic Admin web application (usually located at **`http://`***server*`/escenic-admin`) and click on **View installed plug-ins**. The status of all currently installed plug-ins is shown here, and indicated as follows:

The plug-in is correctly installed.

Copyright © 2010-2014 Escenic AS

- The plug-in is not correctly installed.

2.3 Build the code

After extracting the distribution, you will see several folders. The source code of the widgets can be found in **`misc/widgets`** directory. In order to build the code, you need to have access our maven repository (**`http://maven.vizrt.com`**). Note that this repository is password protected. You need to contact Escenic to get the username/password. After that, you need to configure maven so that it downloads artifacts from this repository. A sample **`settings.xml`** file is provided in **`misc/conf`** folder of each distribution.

When you have access to the repository, you should create the Widget Framework artifacts by running **`mvn clean install`** command in the **`misc/widgets`** directory. It will then add all widgets as artifacts onto your local repository.

After this, you should go to the **`misc/demo`** directory. If you run the **`mvn clean install`** command again, a demo webapp will be available under the **`misc/demo/target`** folder.

Please see the **Escenic Content Engine Installation Guide** for instructions on how to deploy the war file onto your application server.

2.4 Adding Custom Widgets

To add some custom widgets to the **`demo`** web application:

1. Go to **`misc/demo`** folder.
2. Place your widget code in **`misc/demo/src/main/webapp/template/widgets`** folder.
3. For each widget, the content type definition should be placed in a separate **`content-type`** file in 'misc/demo/src/main/webapp/template/widgets/<widget-name>/model' folder.
4. Moreover please add the <ui:group/> definition for Custom widgets in **`misc/demo/src/main/webapp/META-INF/escenic/publication-resources/escenic/content-type`** file.
5. Run **`mvn clean install`** command in **`misc/demo`** folder.

The demo webapp will be created under the **`misc/demo/target`** folder. It will contain the custom widget templates and the **`content-type`** resource will also contain the custom widget definition.

2.5 Creating the publication

The Widget Framework is delivered as a Web Archive, which should be used to create an Escenic Publication.

How to create, build and deploy a publication is described in detail in the **Escenic Content Engine Installation Guide**.

Copyright © 2010-2014 Escenic AS

2.6 Supporting multiple languages

In Widget Framework, application resource files are used to store static texts that appear in the framework. They are used to support multiple languages and provide an easy way to switch between them.

You will have to specify a separate application resource file for each of the languages that you want the Widget Framework to support. The naming convention for these files is **`ApplicationResources_[language-code].properties`**, where [language-code] stands for the ISO 639-1 code of the language. For an example, suppose you want to display Widget Framework in German. Then -

1. Create an **`ApplicationResources_de.properties`** file
2. Store it in the **`[path-to-your-publication]/WEB-INF/classes/com/escenic/framework/`** folder
3. Modify the **`language-code`** section parameter of the publication's Home section and change it to **`de`**(if the parameter is missing then you need to create it)

Note that you don't need to restart ECE to include new application resource files, but if you modify contents of an existing one then you need to reboot the server. If you want to avoid server reboot, then after editing the resource file rename it to something else (i.e., **`messsage.properties`** and then modify the **`javax.servlet.jsp.jstl.fmt.localizationContext`** parameter in the **`web.xml`** file to include the name of the new file. This parameter value is set to **`com.escenic.framework.ApplicationResources`** by default, which you should change to **`com.escenic.framework.[new-file-name]`**.

Copyright © 2010-2014 Escenic AS

3 Widget Development

A widget is a package containing all the components needed to provide a useful web page component for use in Escenic publications: JSP files, CSS files, graphics files, Escenic resource files and so on. In order for the widgets to function as free-standing modules they must conform to a strictly defined structure. This makes it easy to merge new widgets into a widget framework installation along with other widgets.

This chapter contains:

- A general description of the widget structure
- A description of how to create a custom view for an existing widget
- A description of how to create a new widget from scratch

3.1 Widget Structure

A widget consists of the following primary components:

- An Escenic content type definition
- A set of one or more themes, each consisting of CSS file and associated graphics files
- A template, consisting of a set of JSP files. The template is internally organized as a **controller** and a set of one or more **views**.

This structure is reflected in a widget's folder tree:

```
webapp/
  META-INF/
    escenic/
      publication-resources/
        escenic/
          content-type
  static/
    theme/
      theme1/
        css/
          theme1.css
        gfx/
          widget-name/
            theme1-graphics-files
      theme2/
        css/
          theme2.css
        gfx/
          widget-name/
            theme2-graphics-files
  template/
    widgets/
      widget-name/
        controller/
          helpers/
            helper1.jsp
```

Copyright © 2010-2014 Escenic AS

```
            helper2.jsp
          controller.jsp
          view1.jsp
          view2.jsp
        view/
          helpers/
            helper1.jsp
            helper2.jsp
          view1.jsp
          view2.jsp
```

These components are discussed in more detail in the following sections.

3.1.1 Content Type

A widget has a standard Escenic content type definition, defined in the usual way in a **`content-type`** resource file. Widgets have content type definitions so that they can be "understood" by Content Studio. This allows Content Studio to be used to:

- Configure widgets
- Define page layouts by adding widgets to config sections.

A widget content type definition:

- Is stored in the widget's **`src/main/webapp/META-INF/escenic/publication-resources/escenic/content-type`** file
- Is a standard Escenic **`content-type`** resource file
- Contains one **`content-type`** element defining the widget, plus the **`field`**, **`field-group`** and other elements it references

A widget content type should usually contain:

- A General **`panel`** containing **`field`**s for common options.
- A Default **`panel`** containing options for the widget's default view.
- An Advanced **`panel`** containing **`field`**s for advanced options

For more information about **`content-type`** resource files, see:

- The Escenic Content Engine Template Developer Guide
- The Escenic Content Engine Resource Reference

3.1.2 Themes

As well as having multiple views, a widget's appearance can be modified by the application of different themes. A theme consists simply of a CSS file and an accompanying set of graphics files (if required). Themes are stored in **`src/main/webapp/static/theme/`***theme-name* folders. Each *theme-name* folder contains a **`css`** folder containing a CSS file and **`gfx`** folder containing any images, Flash animations or other media files required by the theme.

Copyright © 2010-2014 Escenic AS

3.1.3 Templates

The real work of rendering widgets is performed by JSP templates. A widget may be rendered in several different forms called **views**. A widget might, for example, have a **`default`** view and a **`json`** view for rendering the widget as JSON data rather than HTML. The view that is actually used in any particular case is determined by the publication designer, who selects the view in Content Studio. Widgets with multiple views must therefore always include a **`view`** option that allows the designer to make this selection. The **`view`** field that represents this option should always be included in the General panel of the widget's content type definition. There should usually also be a panel to hold the parameters for each view (see section 3.1.1).

The core widgets supplied with the Widget Framework currently all have only one **`default`** view. Despite this, there is always a **`view`** field in the widget's General panel, and a corresponding Default panel, to prepare for possible future expansion.

Templates are stored in a widget's **`src/main/webapp/template/widgets/`***widget-name*`/` folder. This folder contains a **`view`** subfolder that in turn contains one JSP file for each view supported by the widget. There may also be a helpers folder containing additional JSP files used by the main view JSPs.

> If a widget has no **`view`** field, then a default view name is set by the controller framework. The default view name is **`default`**. It is possible to change this default view name by setting the **`defaultViewName`** property in the **`DefaultMapController.properties`** file. If this property is not set then **`default`** is used as the default view name.

3.1.3.1 Widget Code

Widget code is divided functionally into:

- A **controller** that contains all the logic required to produce the values needed to render a widget.
- A **view** that use the values generated by the controller to produce the final output.

Widget views are always implemented in JSP. Controllers, however, may be implemented either in JSP or in Java. These two different kinds of widget are described further in the following sections.

3.1.3.2 Pure JSP Widgets

In versions of the Widget Framework prior to version 3.0, both the view and the controller components were always implemented in JSP: a widget always contained a **`controller`** folder alongside the **`view`** folder containing controller templates. In this kind of widget, the controller templates obtain or calculate all the values required to produce a specific view and write them to a bean ready for use by the view template.

The view templates are stored in a *widget-name*`/view` folder, and it should usually contain just one template for each view. The **`view`** folder may, however, also contain a **`helpers`** sub-folder. This folder can be used to hold templates containing additional code that can be shared between the views.

The controller templates are stored in a *widget-name*`/controller` folder, and should usually contain one **`controller.jsp`** containing common code, plus one template for each view. Like the **`view`** folder, the **`controller`** folder may also contain a **`helpers`** sub-folder for shared code.

For detailed instructions on how to make a pure JSP custom widget, see section 3.3.

Copyright © 2010-2014 Escenic AS

3.1.3.3 JSP/Java Widgets

From version 3.0, the Widget Framework's preferred widget architecture retains the JSP view templates, but implements all the controller logic in Java. There is no **`controller`** folder in the widget's template tree and all the controller functionality is provided by a Java class. All the core widgets supplied with Widget Framework 3.0 or later are JSP/Java-based widgets. It is, however, still possible to make widgets of your own using the pure JSP method.

For detailed instructions on how to write a Java controller for a Java/JSP widget, see section 3.4.

3.2 Creating a Custom View

If you want to change one of the current views offered by the Widget Framework, but do not want to change the controller or any fields, you can simply copy one of the views in the view directory in the widgets into a directory called custom on the top level of the widget, and modify it to your needs.

3.3 Creating a JSP Custom Widget

This section describes how to make a pure JSP widget with a Hello World example.

The layout of a widget is as follows:

```
webapp/
      META-INF/
      static/
        theme/
          theme1/
            css/
              theme1.css
            gfx/
              widget-name/
                a.png
                b.swf
          theme2/
            css/
              theme2.css
            gfx/
              widget-name/
                a.png
                b.swf
      template/
        widgets/
          widget-name/
            controller/
              helpers/
                helper1.jsp
                helper2.jsp
              controller.jsp
              a.jsp
              b.jsp
            view/
              helpers/
                helper1.jsp
                helper2.jsp
```

Copyright © 2010-2014 Escenic AS

```
            a.jsp
            b.jsp
```

In our example we will create two views. One called **default**, which will simply be a div containing the text "hello world", and another called **custom**, where you can have a custom text. We will have the following structure in the example:

```
helloworld/
        controller/
            controller.jsp
            default.jsp
            custom.jsp
        view/
            default.jsp
            custom.jsp
```

3.3.1 Hello World Content Type

The custom view is the only one that needs configuration so we will only add a panel for that. All widgets have a general and advanced panel.

```
<?xml version="1.0" encoding="UTF-8"?>
<content-types
      xmlns="http://xmlns.escenic.com/2008/content-type"
      xmlns:ui="http://xmlns.escenic.com/2008/interface-hints"
      xmlns:rep="http://xmlns.escenic.com/2009/representations"
      xmlns:doc="http://xmlns.vizrt.com/2010/documentation" version="4">

  <content-type name="widget_helloworld">
    <ui:label>Hello world Widget</ui:label>
    <ui:description>The widget that shows a list of articles of a particular type</
ui:description>
    <ui:title-field>title</ui:title-field>

    <panel name="general">
      <ui:label>General</ui:label>
      <ui:description>The basic configuration fields for hello world widget</
ui:description>
      <field  name="title" type="basic" mime-type="text/plain">
        <ui:label>Name</ui:label>
        <ui:description>The name of the widget</ui:description>
        <constraints>
          <required>true</required>
        </constraints>
      </field>
      <field name="view" type="enumeration">
        <ui:label>View</ui:label>
        <ui:description>The view to be used to render the widget</ui:description>
        <enumeration value="default">
          <ui:label>Default</ui:label>
        </enumeration>
        <enumeration value="custom">
          <ui:label>Custom</ui:label>
        </enumeration>
        <ui:value-if-unset>default</ui:value-if-unset>
      </field>
    </panel>
```

Copyright © 2010-2014 Escenic AS

```
    <panel name="custom">
      <ui:label>Custom</ui:label>
      <ui:description>The custom configuration fields for the hello world widget</
ui:description>
      <field name="customText" type="basic" mime-type="text/plain">
        <ui:label>Custom Text</ui:label>
        <ui:value-if-unset>Hallo everyone</ui:value-if-unset>
      </field>
    </panel>

    <summary>
      <ui:label>Content Summary</ui:label>
      <field name="title" type="basic" mime-type="text/plain">
        <ui:label>Name</ui:label>
      </field>
    </summary>
  </content-type>
</content-types>
```

3.3.2 JSP example code

controller/controller.jsp

```
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
  <%@ taglib uri="http://java.sun.com/jsp/jstl/functions" prefix="fn" %>

  <%-- create the map that will contain relevant field values --%>
  <jsp:useBean id="helloworld" type="java.util.Map" scope="request"/>

  <%-- access the fields that affect all views--%>
  <c:set target="${helloworld}" property="styleClass" value="helloworld"/>
```

controller/default.jsp

```
<%-- Needs to be here but can be kept empty --%>
```

controller/custom.jsp

```
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
  <%@ taglib uri="http://java.sun.com/jsp/jstl/functions" prefix="fn" %>

  <%--declare the map that will contain relevant field values --%>
  <jsp:useBean id="helloworld" type="java.util.Map" scope="request" />

  <c:set target="${helloworld}" property="customText"
 value="${fn:trim(element.content.fields.customText.value)}"/>
```

view/default.jsp

```
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>

  <%--declare the map that will contain relevant field values --%>
  <jsp:useBean id="helloworld" type="java.util.Map" scope="request" />

  <div class="${helloworld['styleClass']}">
    Hello world!
  </div>
```

view/custom.jsp

Copyright © 2010-2014 Escenic AS

```
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>

  <%--declare the map that will contain relevant field values --%>
  <jsp:useBean id="helloworld" type="java.util.Map" scope="request" />

  <div class="${helloworld['styleClass']}">
    <c:out value="${helloworld['customText']}"/>
    <input type="button" id="btnHello" value="Click Me"
onclick="printHello('${helloworld['customText']}')" />
  </div>
```

3.4 Writing a Java Controller

It is possible to write a custom Java controller for a widget. This chapter contains an outline of the steps needed to write and deploy such a custom controller.

3.4.1 Writing the Java Class

The custom controller must extend the class **`com.escenic.framework.controller.AbstractController`** and override whatever methods are necessary to customize the controller functionality in the way you require. It is also possible to implement the interface **`com.escenic.framework.controller.Controller`** directly, but then it is not possible to reuse the common logic that is provided by **`AbstractController`**.

For information about the operations performed by **`AbstractController`** and the methods that can be overridden, please see the Javadoc for this class.

3.4.2 Using the Custom Controller

This description is based on the assumption that:

- You have written the class **`com.escenic.framework.controller.impl.CustomController`**, which extends the class **`com.escenic.framework.controller.AbstractController`**.
- This class is in a separate Maven module and packaged in a jar file.
- You want to use this class as the controller for one of the standard widgets.

To make use of the class you have written:

1. Make sure that the Maven module for the custom controller contains a **`.properties`** file called **`CustomController.properties`** in the folder **`/src/main/resources/com/escenic/servlet/default-config/com/escenic/framework/controller/impl`**. The file must have the following contents:

   ```
   $class=com.escenic.framework.controller.impl.CustomController
   ```

2. Run:

   ```
   mvn clean install
   ```

   in the Maven module folder. This will generate a **`.jar`** file and install it in your local repository.
3. Download and extract the Widget Framework Core distribution.
4. Choose one of the core widgets for modification - the **`code`** widget for example.

Copyright © 2010-2014 Escenic AS

5. Create a **.properties** file called **ControllerFactory.properties** in the **widget-core-code/src/main/resources/com/escenic/servlet/default-config/com/escenic/framework/controller/factory** folder.
6. Open **ControllerFactory.properties** for editing and enter the following:

   ```
   controller.code=/com/escenic/framework/controller/impl/CustomController
   ```

7. Change directory to the distribution's **misc/widgets** folder.
8. Run the following command.

   ```
   mvn clean install
   ```

9. Change directory to the distribution's **misc/demo** folder.
10. Open **pom.xml** for editing.
11. Add a dependency to the **.jar** file containing the custom controller and relevant configuration files. The dependency must be added in **compile** scope to ensure that the **.jar** file containing the custom controller is present in **WEB-INF/lib** folder of **demo.war** after the build.
12. Run:

    ```
    mvn clean install
    ```

    During the build, all the **ControllerFactory.properties** files in various modules will be merged. The merged file will contain the line that you added to the **ControllerFactory.properties** file for the **code** widget.
13. Deploy the **demo.war** webapp created in **misc/demo/target**.

3.4.3 Writing a Model Processor

Instead of writing a controller JSP you can write a Java class, referred to here as a **model processor**. A model processor can be configured to work either for a single view or for multiple views. A model processor must

- extend **com.escenic.framework.controller.processor.AbstractModelProcessor**
- override **processModel** method.

You might, for example create a model processor called **HelloWorldWidgetDefaultViewProcessor** for your **Hello World** widget's **default** view:

```
package com.mycompany.view.processor;

import com.escenic.framework.controller.processor.AbstractModelProcessor;
import neo.xredsys.presentation.PresentationArticle;
import org.apache.commons.lang.StringUtils;
import javax.servlet.http.HttpServletRequest;
import java.util.Map;


public class HelloWorldWidgetDefaultViewProcessor extends AbstractModelProcessor {
@Override
protected void processModel(final Map<String, Object> pWidgetContext,
                            final HttpServletRequest pRequest) {

          Map<String, Object> model = (Map<String, Object>)
 pWidgetContext.get("model");
          if (StringUtils.isNotBlank(model.get("greetings").toString())) {
            model.put("greetings", "Welcome");
```

Copyright © 2010-2014 Escenic AS

```
            }
        }
    }
```

pWidgetContext is the context widget bean, and provides access to all data associated with the context widget. For further information, see section 3.6.

Before a model processor can be used it must be:

- Compiled
- Added to the web application's classpath

To compile a model processor you must have **wf-presentation-3.0.3.154583.jar** in your classpath.

Once you have created a model processor class, you need to:

- Create a set of properties files to declare the model processor and register it in the system
- Package the model processor in a JAR file
- Deploy the JAR file in your publication

3.4.3.1 Configuration

To configure and register the model processor you must:

1. Create a file called **HelloWorldWidgetDefaultViewProcessor.properties** in the *configuration-root*/**com/escenic/servlet/default-config/com/escenic/framework/controller** folder in one of your configuration layers.
2. Enter the following in the file:

   ```
   $class=com.mycompany.view.processor.HelloWorldWidgetDefaultViewProcessor
   viewNames=default
   ```

   You can specify several view names in **viewNames**, separated by commas.
3. Create a file called **HelloWorldWidgetDescriptor.properties** in the *configuration-root*/**com/escenic/servlet/default-config/com/escenic/framework/descriptor** folder.
4. Enter the following in the file:

   ```
   $class=com.escenic.framework.descriptor.WidgetDescriptor
   widgetName=helloWorld
   modelProcessor.helloWorld.1=/com/escenic/mycompany/view/processor/
   HelloWorldWidgetDefaultViewProcessor
   ```

   Note the **.1** on the last line above: you can specify more than one model processor for a widget. If, for example, you wanted to add a second model processor to handle a **custom** view then you would need to add another entry like this:

   ```
   modelProcessor.helloWorld.2=/com/escenic/mycompany/view/processor/
   HelloWorldWidgetCustomViewProcessor
   ```

   (Of course you would then also need to repeat steps 1 and 2 to configure your **HelloWorldWidgetCustomViewProcessor** as well.)

Copyright © 2010-2014 Escenic AS

5. Register the widget descriptor you have created by entering the following line in the **DescriptorRegistry.properties** file in *configuration-root*/**com/escenic/servlet/default-config/com/escenic/framework/descriptor**:

```
widgetDescriptor.helloWorld=./HellowWorldWidgetDescriptor
```

3.4.3.2 Packaging

A model processor class and all the properties files that configure it must be correctly packaged in a JAR file before you can deploy it. To package it you must:

1. Copy the files into a folder structure that matches:
   - The package name of your model processor class
   - The Content Engine's package naming conventions
2. Pack it in a JAR file

Model Processor Package Structure

For the model processor example shown earlier, you would need to create a JAR file with the following structure:

```
com
+-escenic
|  +-servlet
|   +-default-config
|     +-com
|       +-escenic
|         +-framework
|           +-controller
|           |  +-HellowWorldWidgetDefaultViewProcessor.properties
|           +-descriptor
|              +-DescriptorRegistry.properties
|              +-HelloWorldWidgetDescriptor.properties
+-mycompany
  +-view
    +-processor
       +-HelloWorldWidgetCustomViewProcessor.class
```

3.4.4 Writing a Generic Processor

It is possible to write a processor that will work for all widgets. You might, for example, want to read some fields from a panel that exist in all widgets.

Let's assume you want to write a processor called **CustomGenericProcessor** that will belong to the package **com.mycompany.controller.processor**.

The actual process of writing such a processor is same as that described in section 3.4.3.

Copyright © 2010-2014 Escenic AS

3.4.4.1 Configuration

Once you have written your processor, you need to include it in the configuration of the main controller for all widgets. By default the Widget Framework uses **`DefaultMapController`** as the controller for all widgets, unless you have provided your own custom controllers for specific widgets.

To add your processor to the **`DefaultMapController`** you need to create a **`DefaultMapController.properties`** file in *configuration-root*`/com/escenic/servlet/default-config/com/escenic/framework/controller/impl/` and the following:

```
processor.custom-key=/com/mycompany/controller/processor/CustomGenericProcessor
```

> You can add more than one processor in the **`.properties`** file. They will then be executed in the order they appear.

If your **`CustomGenericProcessor`** is a Nursery component then you will also need to create a **`.properties`** file for it and configure it according to your requirements.

3.4.4.2 Packaging

A generic processor class and all the **`.properties`** file needed to configure it must be correctly packaged in a JAR file before you can deploy it. To package it you must:

1. Copy the files into a folder structure that matches:
   - The package name of your generic processor
   - The package naming conventions required by the Content Engine's architecture
2. Pack it in a JAR file using an archiving utility that is capable of creating JAR files

Generic processor package structure

For the generic processor example shown earlier , you would need to create a JAR file with the following structure:

```
com
+-escenic
|  +-servlet
|   +-default-config
|     +-com
|       +-escenic
|       |  +-framework
|       |   +-controller
|       |    +-impl
|       |      +-DefaultMapController.properties
|       +-mycompany
|           +-controller
|            +-processor
|              +-CustomGenericProcessor.properties
+-mycompany
  +-controller
    +-processor
       +-CustomGenericProcessor.class
```

Copyright © 2010-2014 Escenic AS

3.5 Modifying an Existing Controller

In version 3.0.3.154583 of the Widget Framework, most of the controller functionality has been moved to Java space. However, the framework still supports JSP controllers. These controllers are invoked after the Java controller has read the general and view-specific fields from the relevant panels in the widget and put them in the map (the map, in turn, is available in request scope). So, the JSP controllers can be used to execute any custom logic.

3.6 Using The Widget Context Bean

During the process of rendering a widget, the Widget Framework maintains a widget context bean in the request scope called **`widget`**. The **`widget`** bean is a **`java.util.Map`** by default. It contains information about the current context widget and provides access to the intermediate data processed by the Controller and ModelProcessor. The **`widget`** bean is removed at the end of the widget's life cycle.

Note that if a nested widget is loaded from another widget context then **`widget`** represents the nested widget if retrieved from nested widget rendering code.

The **`widget`** bean has the following properties:

- **`${widget.model}`** - A bean (**`java.util.Map`** by default) created from the widget's content field values. The string representation of a widget content field value can be retrieved from this bean using the field name as property name.
- **`${widget.viewName}`** - The name of the widget's selected view.
- **`${widget.properties}`** - A bean (**`java.util.Map`** by default) created from the widget's properties
- **`${widget.widgetContent}`** - The widget content item as a **`PresentationArticle`** object.
- **`${widget.widgetName}`** - The name of the widget.
- **`${widget.contentResult}`** - The Data Source result of a Data Source client widget (for example, a Teaser widget)
- **`${widget.invokingWidget}`** - The widget bean of the invoking widget if the current context widget is invoked by another widget. A Teaser View widget, for example, is invoked by a View Picker widget. Teaser Views do not have a Data Source of their own, they are supplied with data by their invoking View Picker. So if you want to access the Data Source results from a Teaser View widget, you have to use **`${widget.invokingWidget.contentResult}`**.

> In Widget Framework version 2.2.0 and earlier a bean was created based on the widget name. For a widget called **Hello World**, for example, the Widget Framework controller would create a bean called **`helloWorld`**. This bean was used to hold widget field values and processed intermediate data. Use of this bean is now deprecated and it will be removed in a future version of the Widget Framework. You are therefore advised not to use it. Use **`${widget.model}`** instead.

Please note that we do not recommend adding objects directly to the **`request`** scope in order to pass values into JSP pages in a widget. Please use the **`widget`** context bean instead.

Copyright © 2010-2014 Escenic AS

3.6.1 Using The contentResult Bean

The **`widget`** bean's **`contentResult`** property only contains data if the context widget is a **Data Source-based widget** or a view invoked by a Data Source-based widget. At present, that means that the context widget must either be a Teaser widget or a Teaser View widget.

When present, the **`contentResult`** bean contains all the content items returned by the widget's Data Source, plus information about which of the returned items the widget should render. It has the following properties:

`contentResult.resultItems`
The actual content items returned by the Data Source. You can retrieve them as follows:

For a Teaser widget:

```
<c:set var="articleList" value="${widget.contentResult.resultItems}"/>
```

For a Teaser View widget:

```
<c:set var="articleList"
 value="${widget.invokingWidget.contentResult.resultItems}"/>
```

The value returned is a **`List<ResultItem>`**.

`contentResult.offset`
The position in the **`contentResult.resultItems`** list from which the context widget is to start rendering.

`contentResult.count`
The number of content items that the context widget is to render.

`contentResult.totalItems`
The total number of content items in the **`contentResult.resultItems`** list.

3.6.2 Using The ResultItem Beans

The **`ResultItem`** beans returned by **`${widget.contentResult.resultItems}`** or **`${widget.invokingWidget.contentResult.resultItems}`** have the following properties:

`resultItem.articleId`
The content item ID.

`resultItem.content`
The content item as a **`PresentationArticle`** bean.

`resultItem.actualItem`
The content item's content.

`resultItem.publishedDate`
The date the content item was published.

`resultItem.lastModifiedDate`
The date the content item was last modified.

`resultItem.type`
The type of the content item: **`summary`**, **`searchResult`** or **`popular`**. This value can be used to determine how the content item is rendered.

`resultItem.priority`
The priority of the content item.

Copyright © 2010-2014 Escenic AS

For further information about the Widget Framework API, take a look at the **Escenic Widget Framework 3.0.3.154583 API documentation**.

3.7 Widget Properties

Widget properties are named values stored in widget instances. When a widget is edited in Content Studio, its properties are displayed in a **Widget properties** field on the widget's **Advanced** tab, where they can be edited by the publication designer.

The property values are exposed in the **`widget`** context bean where they can be accessed and used by template developers.

3.7.1 Defining New Properties

You can define properties for widgets by including a **`feature`** resource in a **`properties`** folder located alongside the widget's **`view`** and **`controller`** folders:

```
widget1
  controller
  view
  properties
    feature
widget2
  controller
  view
  properties
    feature
```

The feature resource must be a standard feature resource as described in the **Escenic Content Engine Resource Reference** (see http://documentation.vizrt.com/ece-resource-ref/5.6/feature.html). In this file you can both define properties and set default values for them.

A widget property must be named and defined in accordance with the following convention:

`wf.props.`*widget-name*`.`*property-name*`=`*value*

The following feature resource, for example, defines the properties **`title`**, **`title.class`** and **`wrapper.bgurl`** for a widget called **`helloWorld`**:

```
wf.props.helloWorld.title = Hello World Widget
wf.props.helloWorld.title.class=myclass
wf.props.helloWorld.wrapper.bgurl=gfx/helloWorld/background.png
```

3.7.2 Overriding Core Widget Properties

You can override the default values of existing core widget properties in exactly the same way as you create properties of your own - by adding a **`properties/feature`** resource to the widget that contains definitions of the properties you want to override. To set the default value of the Teaser widget's **`field.markup.tag.subtitle`** property to **`h4`**, for example, you would need to add a **`teaser/properties/feature`** resource containing the following property definition:

```
wf.props.teaser.field.markup.tag.subtitle = h4
```

Copyright © 2010-2014 Escenic AS

3.7.3 Editing Widget Instance Properties

All of a widgets properties (core widget properties and user-defined properties are displayed in Content Studio in a **Widget properties** field on the widget's **Advanced** tab. They appear in this field without the **`wf.props.`***widget-name***`.`** prefix:

```
title = Hello World Widget
title.class=myclass
wrapper.bgurl=gfx/helloWorld/background.png
```

The Content Studio user can then:

- Change the values of displayed properties
- Add custom properties of his/her own

Custom properties added in this field do not need the **`wf.props.`***widget-name***`.`** prefix.

3.7.4 Accessing Widget Properties

Properties are exposed in the **`widget`** context bean. You can find all of a widget's properties in **`${widget.properties}`** and access them as in the following example:

```
${widget.properties['title']}
${widget.properties['title.class']}
${widget.properties['wrapper.bgurl']}
```

3.8 Lazy loading and Javascript

The Widget Framework supports lazy loading of widgets, groups and areas. Any page fragments that are lazy-loaded are only loaded if required for the current device, and only after the main page document has been loaded. This means you have to be careful when invoking Javascript functions to operate on lazy-loaded fragments. If you invoke them on the document ready event then they may not work in many cases because the addressed element is not yet present in the DOM.

If a function is invoked for a widget on the document ready event **inside the widget itself**, then it will work even if the widget is lazy-loaded. For example:

```
<div class="widget awesome-slideshow" id="slideshow-1">
    <!--content omitted-->
</div>

<script type="text/javascript">
  $(document).ready(function(){
     $(".awesome-slideshow").awesomeSlideshow();
  });
</script>
```

The above function call will not work, however, if it is made in the page's **`head`** and **`awesome-slideshow`** is lazy-loaded - because the addressed element will not have been loaded when the page's **`ready`** event fires.

The Widget Framework therefore provides a custom event called **`fragmentReady`** that fires after all lazy-loaded fragments have finished loading. So calling this function from the page's **`head`** as follows:

Copyright © 2010-2014 Escenic AS

```
$(document).on("fragmentReady", function(){
  $(".awesome-slideshow").awesomeSlideshow();
});
```

will work whether or not **`awesome-slideshow`** is lazy-loaded.

3.9 Using Responsive Images

The Widget Framework has the ability to switch image representations to suit the available space on different devices and in different browser window sizes).

This functionality is provided by:

- A tag in the Widget Framework Core Tag Library, **`wf-core:image`**.
- A Javascript library called **`picturefill.js`** (see https://github.com/scottjehl/picturefill). Inspired by the proposed HTML **`picture`** element (see Responsive Images Community Group), this library uses a set of **viewport breakpoints** to select an appropriately sized image for the current device/browser window size.

  The **`picturefill.js`** library shipped with the Widget Framework is modified slightly to provide better support for the **`wf-core:image`** tag.

3.9.1 Configuring Responsive Image Handling

The Widget Framework's responsive image handling functionality is configured by setting a number of section parameters (usually in a publication's root section).These section parameters are associated with content profiles (see chapter 4), so that different content profiles can have different responsive image policies.

The responsive image section parameters are:

- **`wf.contentprofile.`***content-profile*`.image.policy`**
- **`wf.contentprofile.`***content-profile*`.image.grid.gridViewportWidth`**
- **`wf.contentprofile.`***content-profile*`.image.breakpoints.fluid`**

where *content-profile* is the name of the content profile (**`default`** by default).

The parameters should be used as follows:

`image.policy`
: This parameter can be set to one of two values:

  `picturefill`
  : Enables support for responsive images. Images are selected for small devices (i.e phones) based on the assumption that they will occupy the full width of the screen. For larger devices, including tablets, images are selected based on the assumption that they are displayed in a layout grid. The width of this grid is read from **`image.grid.gridViewportWidth`** (see below).

  `grid`
  : Disables support for responsive images. Fixed-size images are used, in the same way as in Widget Framework systems prior to version 3.0.

Copyright © 2010-2014 Escenic AS

image.grid.gridViewportWidth
If **image.policy** is set to **picturefill** then this parameter is used by the **wf-core:image** tag together with the current element width (calculated by the Widget Framework) to calculate the required image width for larger devices. A typical value would be **1140**.

image.breakpoints.fluid
If **image.policy** is set to **picturefill** then this parameter defines a series of device width breakpoints at which a different size image representation will be selected for display. The specified breakpoints should match the breakpoints used in your Bootstrap CSS. A typical value might be **320,360,480,640,767**.

3.9.2 wf-core:image

Syntax

```
<wf-core:image
  article = "..."?
  articleId = "..."?
  scale = "..."?
  aspectVariant = "..."?
  imagePolicy = "..."?
  styleClass = "..."?
  onClick = "..."?
  width = "..."?
  alt = "..."?
  title = "..."?
/>
```

Attributes

article
A **neo.xredsys.presentation.PresentationArticle**: the content item representing the image to be displayed.

articleId
An article id identifying the image content item to be displayed. The id can be supplied as an **int**, **Integer** or **String**.

scale
The image width specified as a percentage of the available space (that is, the width of the image's enclosing element).

aspectVariant
Specifies the required image aspect. The allowed options are **PORTRAIT**, **LANDSCAPE**, **SQUARE** or **FREE**. If no value is specified, then a default of **LANDSCAPE** is used. (This default can be changed by setting the **defaultAspectVariant** property in **om/escenic/framework/cache/ImageRepresentationCacheManager.properties**.

imagePolicy
Specify **picturefill** to enable responsive image selection or **grid** to disable it.

styleClass
The name of a CSS style (HTML **class** attribute) to be assigned to the generated HTML **img** tag.

onClick
A URL to be assigned to the generated **img** tag.

Copyright © 2010-2014 Escenic AS

`width`
The width value to be used if **`imagePolicy`** is set to **`grid`**. If you specify both **`width`** and **`scale`**, then **`scale`** takes priority.

`alt`
If specified,sets the **`alt`** attribute of the generated **`img`** element. By default the **`alt`** attribute is taken from the source content item.

`title`
If specified,sets the **`title`** attribute of the generated **`img`** element. By default the **`title`** attribute is taken from the source content item.

Copyright © 2010-2014 Escenic AS

4 Content Profiles

The Widget Framework allows you define multiple sets of templates called **content profiles**. The purpose of content profiles is to make it possible for a single publication to be generated in two completely different forms. You could, for example, have a **`default`** content profile for generating your standard web site and a **`newsletter`** content profile for generating a newsletter from the same content.

Widget Framework content profiles (that is, template hierarchies) are organized under the root section **`config`**. By default there is just one content profile, called **`config.default`**, which might look like this:

```
config
  config.default
    config.default.section
      config.default.section.ece_frontpage
      config.default.news
      config.default.sports
    config.default.article
      config.default.article.type.story
      config.default.article.type.simpleVideo
    config.default.master
```

Adding a second **`newsletter`** content profile would result in two parallel template hierarchies like this:

```
config
  config.default
    config.default.section
      config.default.section.ece_frontpage
      config.default.news
      config.default.sports
    config.default.article
      config.default.article.type.story
      config.default.article.type.simpleVideo
    config.default.master

  config.newsletter
    config.newsletter.section
      config.newsletter.section.ece_frontpage
      config.newsletter.news
      config.newsletter.sports
    config.newsletter.article
      config.newsletter.article.type.story
      config.newsletter.article.type.simpleVideo

    config.newsletter.master
```

> Note that the **`config`** section is not itself a template: it is just a container for content profiles.

Copyright © 2010-2014 Escenic AS

The templates that make up a content profile are created by publication designers using Content Studio. For information about this and the rules and conventions governing template naming, organization and inheritance, see the **Widget Framework User Guide**.

4.1 Adding a Content Profile

To add a content profile to a publication:

1. Open the publication's **`WEB-INF/classes/com/escenic/servlet/plugin-config/com/escenic/framework/content/profile/ContentProfileProcessor.properties`** file for editing.
2. Add an entry like this:

   ```
   contextPathSuffix.name=context-path
   ```

   where *name* is the name of your new content profile (**`newsletter`** for example) and *context-path* is its context path (**`newsletter`** for example). All output generated with this profile will then have the specified suffix appended to the publication URL (giving **`http://`***publication-name*`/newsletter`, for example).
3. Define the name of the new content profile's root template by adding a section parameter like this to your publication's root section:

   ```
   wf.contentprofile.name.rootConfig=config.name
   ```

   where *name* is the name of your new content profile (**`newsletter`** for example).
4. Optionally, you can also specify the wireframe for your new content profile by adding a second section parameter:

   ```
   wf.contentprofile.name.wireframe=config.wireframe
   ```

   where *wireframe* is the name of the wireframe you want to use. If you do not specify this parameter then the new content profile will use the default wireframe.

4.2 Using Old Template Hierarchies

Widget Framework versions prior to 3.0 did not support multiple content profiles. A publication had only one template hierarchy, with a root template called **`config`**.

If you are upgrading a publication from a version prior to 3.0 and you do not want to rename the templates, all you need to do is add the following section parameter to the publication's root section:

```
wf.contentprofile.default.rootConfig=config
```

Copyright © 2010-2014 Escenic AS

5 Adding a New Grid

This chapter explains how to add a new grid to the Widget Framework.

5.1 The Example Grid Specification

Suppose you want a grid with 4 columns, called **Four Column Config**. The column widths (from left to right) are to be: 140px, 300px, 300px and 140px.

5.2 Modifying The layout-group Resource

The first step is to modify the publication's **`layout-group`** resource by adding a group definition like this:

```
<group name="x140x300x300x140-config" root="true">
    <ui:label>Four Column Config</ui:label>
    <ct:options>
      <ct:field name="inherits_from" type="basic" mime-type="text/plain">
        <ui:label>Inherits From</ui:label>
        <ui:description>Custom configuration section name or id</ui:description>
      </ct:field>
    </ct:options>
    <ui:decorator name="wfItemsResolver"/>
    <area name="meta"/>
    <area name="header">
      <ref-group name="x460x460"/>
      <ref-group name="x700x220"/>
      <ref-group name="x300x300x300"/>
      <ref-group name="x220x220x220x220"/>
      <ref-group name="tabbingGroup"/>
    </area>
    <area name="left">
      <ref-group name="tabbingGroup"/>
    </area>
    <area name="main1">
      <ref-group name="tabbingGroup"/>
    </area>
    <area name="main2">
      <ref-group name="tabbingGroup"/>
    </area>
    <area name="right">
      <ref-group name="tabbingGroup"/>
    </area>
    <area name="footer">
      <ref-group name="x460x460"/>
      <ref-group name="x700x220"/>
      <ref-group name="x300x300x300"/>
      <ref-group name="x220x220x220x220"/>
      <ref-group name="tabbingGroup"/>
    </area>
  </group>
```

Copyright © 2010-2014 Escenic AS

Note the following:

- A grid group name (**`x140x300x300x140-config`** in this case) has a fixed format. It is formed by concatenating the column widths (in order from left to right). Each column width must be preceded by an **`x`** character, and the column width sequence must be followed by the string **`-config`**.
- A grid group must be a root group (that is, the **`group`** element must have a **`root`** attribute and it must be set to **`true`**). This is necessary to ensure that the group is displayed as a page option in Content Studio.
- The group definition uses a request pool decorator named **`wfItemsResolver`**. This decorator ensures that items for a specific area are found using the inheritance mechanism specific to EWF.
- There is a field named **`inherits_from`** defined in group options. This field allows the user to override the configuration section from which the current configuration section inherits.
- All the other group names in this example have been taken from standard **`layout-group`** resource distributed with the Widget Framework.

5.3 Modifying grid.css

The next step is to add CSS entries for the new columns to the **`grid.css`** file (found in **`/static/escenic-times/css`**):

```
div.x140x300x300x140-config div#header {
  float: left;
  margin: 0 10px 0 0;
  padding: 0 8px 0 0;
  width: 940px;
}

div.x140x300x300x140-config div#left {
  float: left;
  margin: 0 10px 0 0;
  padding: 0 8px 0 0;
  width: 140px;
}

div.x140x300x300x140-config div#main1 {
  float: left;
  margin: 0 10px 0 0;
  padding: 0 8px 0 0;
  width: 300px;
}

div.x140x300x300x140-config div#main2 {
  float: left;
  margin: 0 10px 0 0;
  padding: 0 8px 0 0;
  width: 300px;
}

div.x140x300x300x140-config div#right {
  float: left;
  width: 140px;
}

div.x140x300x300x140-config div#footer {
```

Copyright © 2010-2014 Escenic AS

```
  float: left;
  margin: 0 10px 0 0;
  padding: 0 8px 0 0;
  width: 940px;
}
```

5.4 Adding A JSP Template for the Grid

Finally, you need to add a JSP template to the **`/template/framework/group`** folder in order to make the new grid work. The name of the template must match the name of the grid (in this case,**`x140x300x300x140-config.jsp`**). Here is an example JSP template:

```
<%@ page language="java" pageEncoding="UTF-8" contentType="text/html; charset=UTF-8"
 %>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<%@ taglib uri="http://www.escenic.com/widget-framework/core" prefix="wf-core" %>

<div id="page" class="x140x300x300x140-config">
  <div id="header">
    <c:set var="area" value="header" />
    <c:set var="items"
 value="${requestScope.configSectionPool.rootElement.areas[area].items}"
 scope="request"/>
    <c:set var="elementwidth" value="940" scope="request"/>
    <wf-core:showItems level="0"/>
    <c:remove var="elementwidth" scope="request"/>
    <c:remove var="items" scope="request"/>
  </div>

  <div id="content">
    <div id="areas">
      <div id="left">
        <c:set var="area" value="left" />
        <c:set var="items"
 value="${requestScope.configSectionPool.rootElement.areas[area].items}"
 scope="request"/>
        <c:set var="elementwidth" value="140" scope="request"/>
        <wf-core:showItems level="0" />
        <c:remove var="elementwidth" scope="request"/>
        <c:remove var="items" scope="request"/>
      </div>
      <div id="main1">
        <c:set var="area" value="main1"/>
        <c:set var="items"
 value="${requestScope.configSectionPool.rootElement.areas[area].items}"
 scope="request"/>
        <c:set var="elementwidth" value="300" scope="request"/>
        <wf-core:showItems level="0"/>
        <c:remove var="elementwidth" scope="request"/>
        <c:remove var="items" scope="request"/>
      </div>
      <div id="main2">
        <c:set var="area" value="main2" />
        <c:set var="items"
 value="${requestScope.configSectionPool.rootElement.areas[area].items}"
 scope="request"/>
        <c:set var="elementwidth" value="300" scope="request"/>
```

Copyright © 2010-2014 Escenic AS

```
        <wf-core:showItems level="0" />
        <c:remove var="elementwidth" scope="request"/>
        <c:remove var="items" scope="request"/>
      </div>

      <div id="right">
        <c:set var="area" value="right" />
        <c:set var="items"
 value="${requestScope.configSectionPool.rootElement.areas[area].items}"
 scope="request"/>
        <c:set var="elementwidth" value="140" scope="request"/>
        <wf-core:showItems level="0" />
        <c:remove var="elementwidth" scope="request"/>
        <c:remove var="items" scope="request"/>
      </div>
    </div>
  </div>

  <div id="footer">
    <c:set var="area" value="footer" />
    <c:set var="items"
 value="${requestScope.configSectionPool.rootElement.areas[area].items}"
 scope="request"/>
    <c:set var="elementwidth" value="940" scope="request"/>
    <wf-core:showItems level="0"/>
    <c:remove var="elementwidth" scope="request"/>
    <c:remove var="items" scope="request"/>
  </div>
</div>
```

5.5 Testing the New Grid

To test the new grid:

1. Update your publication's **`layout-group`** resource.
2. Log intoContent Studio.
3. Verify that the **Four Column Config** grid is available as a page option.
4. Select **Four Column Config** in one of your configuration sections and desk the widgets.
5. Start up a browser and visit a corresponding content section in your publication. The section layout should now reflect the new grid layout you have selected.

Copyright © 2010-2014 Escenic AS

6 Themes

To change the theme and style widget-framework has appropriate facility.

6.1 Add New Theme

The Widget Framework is shipped with one default theme called **`default`**. To add a new theme:

1. Add a folder for your theme to every widget's **`theme`** folder. If your theme is called **`my-theme`** then you would need to add a *widget*`/src/main/webapp/static/theme/my-theme` folder to every widget.
2. Add a corresponding theme folder to the Widget Framework's core **`theme`** folder: **`widget-framework-core/src/main/webapp/static/theme/my-theme`**, for example.
3. Create a **`base`** subfolder in each **`my-theme`** folder you have created. The **`base`** folder should contain all the theme's common **`.css`** and graphics files (in **`css`** and **`gfx`** subfolders). You can also create **variant** folders that have the same structure as the **`base`** folder but with different names. The variant folders can contain modified **`.css`** and graphics files, making it possible for the theme to include variations for use in different sections of a publication. Variants are selected by setting a **`theme.variant`** section parameter in publication sections.
4. Add a property setting like this to **`widget-framework-core/src/main/resources/com/escenic/framework/ApplicationResources.properties`**:

   ```
   publication.theme.my-theme.title = My theme
   ```

5. Add a transformer specification like this to your publication's POM file:

   ```
   <transformer
    implementation="org.apache.maven.plugins.shade.resource.AppendingTransformer">
     <resource>theme/my-theme/css/my-theme.css</resource>
   </transformer>
   ```

   The Maven Shade plug-in will then merge all the **`my-theme.css`** files in all the widgets into a single **`my-theme.css`** file.
6. Compile and deploy your code.
7. Edit root section parameter from web studio. Set theme variable there e.g **`theme=my-theme`**

To use your new theme you will need to set the theme section parameter in your publication's root section. For information on how to do this, see the **Widget Framework User Guide**.

6.2 Dynamic Rendering of Sass

The Widget Framework supports dynamic rendering of Sass files. If you want to use Sass, put your **`.scss`** files in **`theme/`***theme-name*`/base/sass`** folders.

6.2.1 The Sass Filter

The Sass conversion is carried out by a filter specified in **`web.xml`**:

Copyright © 2010-2014 Escenic AS

```
<filter>
  <filter-name>SassCompiler</filter-name>
  <filter-class>com.escenic.sass.SassCompilingFilter</filter-class>
  <!--
  <init-param>
      <param-name>sassLocation</param-name>
      <param-value>some/other/location</param-value>
  </init-param>
  <init-param>
      <param-name>cssLocation</param-name>
      <param-value>some/other/location</param-value>
  </init-param>
  <init-param>
      <param-name>cacheLocation</param-name>
      <param-value>some/other/location</param-value>
  </init-param>
  <init-param>
      <param-name>cache</param-name>
      <param-value>true/false</param-value>
  </init-param>
  -->
</filter>
<filter-mapping>
  <filter-name>SassCompiler</filter-name>
  <url-pattern>/static/theme/default/base/SassToCss/*</url-pattern>
  <!--WARNING: This pattern is provide for default base skin.
  If you change the skin, you have to change here too.-->
</filter-mapping>
```

You can customize the filter by uncommenting and setting the values of the **`init-param`** elements:

`sassLocation`
: The location of the **`.scss`** files to be converted. The default is **`/theme/`***theme-name***`/base/sass/`**.

`cssLocation`
: The location to which the converted **`.css`** files are to be written. The default is **`/theme/`***theme-name***`/base/SassToCss/`**.

`cacheLocation`
: The location of a cache folder (if required). The default is **`/WEB-INF/.sass-cache/`**.

`cache`
: Set to **`true`** if you want converted **`.css`** files to be cached. The default is **`false`**.

6.2.2 Enabling the Use of Sass Files

In order for the **`.css`** files generated by the Sass converter to be used in your publication, you need to ensure that the CSS stylesheet **`link`** in your pages' **`head`** elements are correctly structured. Assuming you use the default set-up, then your **`link`** elements will need to look like this:

```
<link rel="stylesheet" type="text/css" href="${publication.url}theme/${themeName}/
SassToCss/${themeName}.css"/>
```

Note that the link element requests a **`.css`** file, not an **`.scss`** file. The request is intercepted if the request URL matches a URL pattern specified in the **`filter-mapping`** element, and passed to the appropriate filter, which then converts the appropriate **`.scss`** file to generate the requested **`.css`** file.

Copyright © 2010-2014 Escenic AS

6.2.3 Production Set-up

For performance reasons you should remove the Sass filter from your **`web.xml`** configuration on production systems, since the **`.css`** will have been generated and placed in the correct location. There is therefore no need to repeatedly compile the **`.scss`** file.

Copyright © 2010-2014 Escenic AS

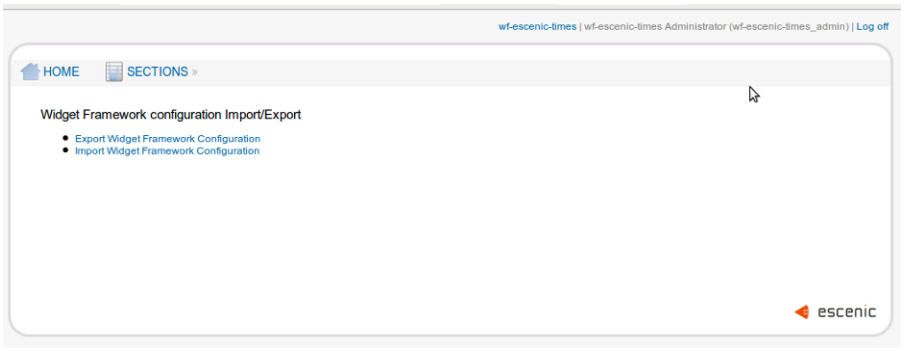
7 Export and Import

This chapter will explain how to export and import the templates and widgets. This can be used to take backup or move the setup to other servers.

7.1 Syndication plug-in

To be able to export and import the templates and the widgets we have created a plug-in to Web Studio. This needs to be installed, and this is done in the same fashion as any other plug-in. You unpack the zip-file in your plug-in directory and run assemblytool.

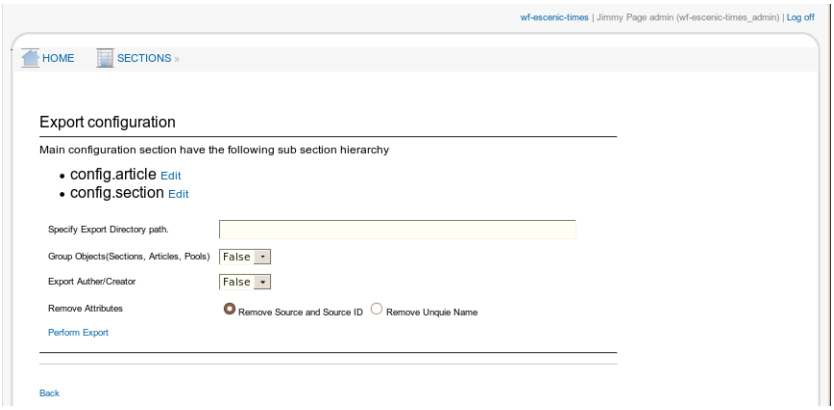
When the plug-in is installed you should get a new menu option in Web Studio. Here you have the options to export and import the configuration.



7.2 Syndication Export

Here you can export your configuration sections and all widgets located within these sections.

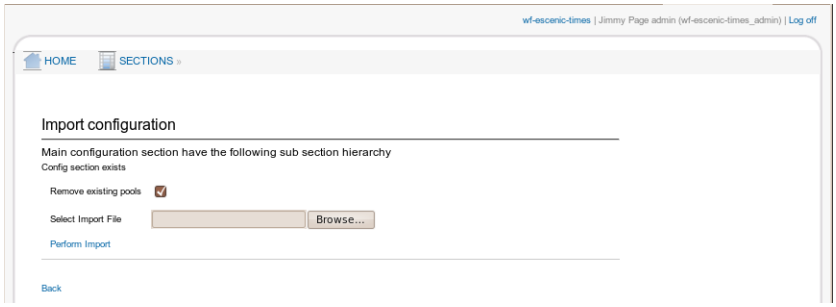
It will be exported to a specified location.



7.3 Syndication Import

When you have a valid export file, this can be imported using the import functionality.

Select the file to import, and click on Perform Import.



Copyright © 2010-2014 Escenic AS

8 How To

8.1 Modify Group and Error Templates

We have moved out template codes from **`showItems.tag`** so that template developer can easily change the template if required. Below is the mapping of moved out jsp pages with groups,errors. If you wish to change these groups, errors template you can change on the following files

- Row group -> **`/template/framework/group/processor/row.jsp`**
- Column group -> **`/template/framework/group/processor/column.jsp`**
- Two column, three column -> **`/template/framework/group/processor/split.jsp`**
- Error page when group template not found-> **`/template/framework/group/processor/group-not-found.jsp`**
- Error page when content (e.g stories) is placed in config pages-> **`/template/framework/group/processor/group-not-found.jsp`**
- Error page when nesting limit exceeds -> **`/template/framework/group/processor/nesting-limit-error.jsp`**

8.2 Use A Payment Solution

The Widget Framework's **`Story`** content type provides support for premium content. Content items of this type display a **Premium** tab in Content Studio where an editor can mark the content item as premium content. The title and selected other fields can be set to non-premium, making these fields will be visible to non-paying site visitors.

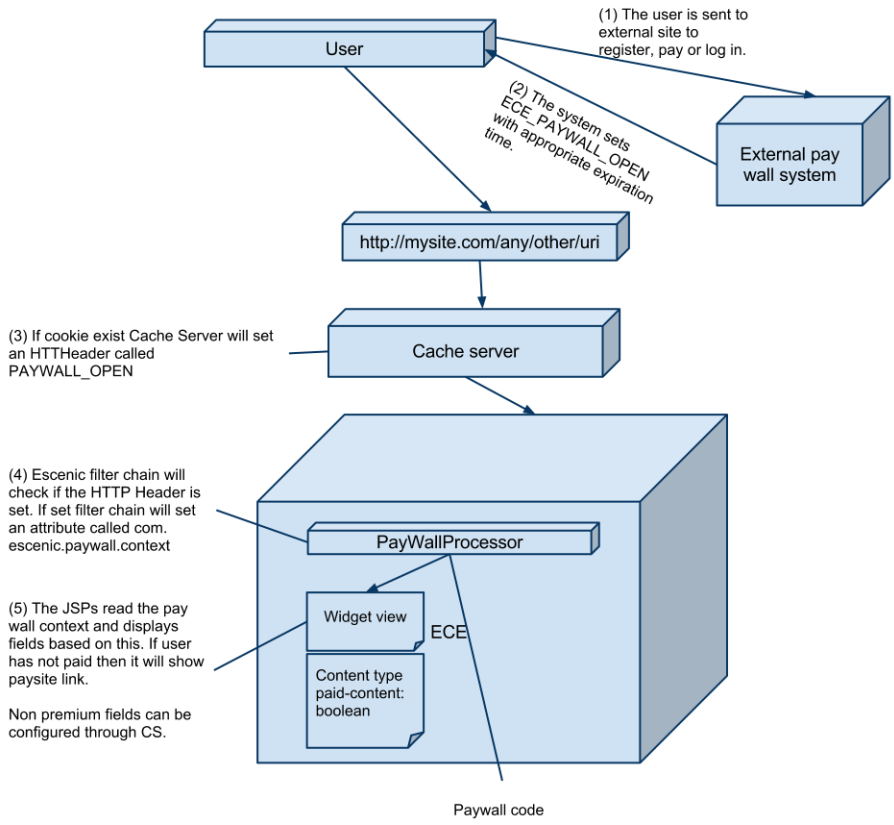
When a visitor clicks on a premium story link, the system checks to see whether or not the visitor has paid for access and only displays the whole content item for paying visitors.

Payment must be handled by a third-party system. The process works as follows:

- When a reader who has not paid clicks on a premium content link, he is offered the option of paying to read the content, and redirected to the payment system if he accepts.
- After paying, the payment system sets a cookie for that user with an appropriate expiration date.
- The user is redirected back to the Escenic site. The Escenic site's **cache server** (e.g Varnish) checks for the existence of the payment cookie. If the cookie is present then it adds a **`PAYWALL_OPEN`** value to the HTTP header.
- The Escenic filter chain checks whether or not the **`PAYWALL_OPEN`** value is present in the header. If it is present, then the filter chain sets a request scope attribute called **`com.escenic.paywall.context`**

Copyright © 2010-2014 Escenic AS

- The template JSP reads the **`com.escenic.paywall.context`** attribute to determine whether or not to display premium fields.



8.3 Override Core Content Type

Sometimes you may need to modify the core content types **(eg. story,picture).** To do this you have to follow the following simple steps:

- Take the widget-framework-core-<version>.zip and unzip it.
- Run a **mvn clean install** in **misc/widgets** folder.
- Now go to the **misc/demo** folder. Add the definition of one core content type (i.e, story) in **src/main/webapp/META-INF/escenic/publication-resources/escenic/content-type** file. And then modify it in your way.
- Run a **mvn clean install** in **misc/demo** folder.
- Take the war file from **target** folder. This war file should contain the modified content-type definition instead of the standard content-type definition. You can verify it by opening the content-type resource using a text editor.
- Upload the war file via **escenic-admin** and see that everything is OK. If your content-type definition contains an error, there will be an error in the merged file too.
- Update a publication and check the custom fields from Content Studio.

Copyright © 2010-2014 Escenic AS

8.4 Load a Widget Using AJAX

In widget framework a mechanism has been provided by which a widget can be updated without refreshing the whole page. Necessary steps to do that are as follows

- A core tag is provided with widget framework named **`widgetAjaxMarkup`**. This tag will render a link element and will keep it hidden. You have to invoke this tag inside widget code. An example of the tag's output is

```
<a href="http://test.escenic.com/
widgetContentId=1234&widgetName=stories&elementwidth=620" rel="refresh"
 id="widgetURL-1234" style="display:none">Reload</a>
```

- Then on any element's (that is inside the widget) **`onclick`** or on other event you should call the javascript utility function **`refreshWidgetContent`** provided with widget framework. This function takes two parameter. One is any DOM element inside the widget and another one is a javascript map. This utility function basically sends an ajax GET request with the url rendered by **`widgetAjaxMarkup`** tag. If you want to pass any other parameter with url to the widget than you have to pass through the javascript map. In response HTML of the widget will be returned and widget will be updated.

Here a sample code snippet is given

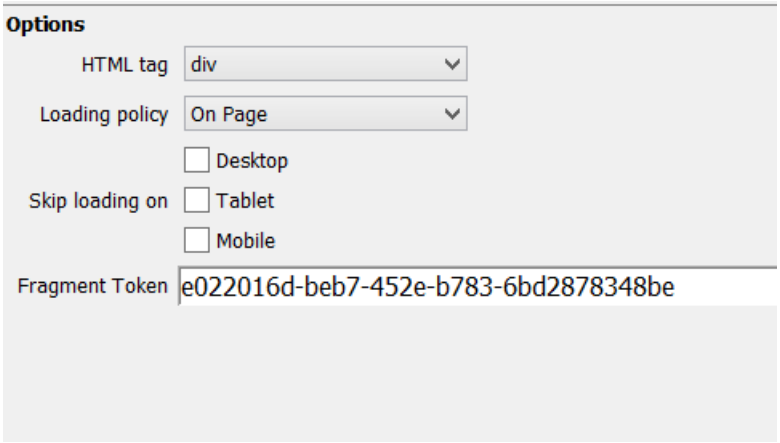
```
<div class="${stories.wrapperStyleClass}">
  <div class="header">
    <h5>
      <c:out value="${stories.widgetName}" escapeXml="false"/>
    </h5>
    <wf-core:widgetAjaxMarkup/>
  </div>

  <img src="reload.png" alt="Reload Widget" onclick="refreshWidgetContent(this)"/>

  <div class="content">
    ..............
  </div>
</div>
```

Utility function **`widgetAjaxMarkup`** basically find closest div having CSS class **`widget`** from the element it is fired on. So, if you remove **`widget`** class from widget markup then widget will not be updated

Copyright © 2010-2014 Escenic AS

8.5 Configure Lazy Loading Options

Lazy loading allows template designers to choose to omit parts of a page depending on the size of the browser window. When editing page templates in Content Studio, areas, groups and widgets offer a set of options to control this functionality:



For usage details, see the **Widget Framework User Guide**.

In order to include the lazy loading configuration options in your own groups and areas, you must include the following code fragment in the group/area definitions in your **`layout-group`** resource.

```
<area name="main">
  <ui:label>Main</ui:label>
  <ct:options scope="current">
    ...
    <ct:field type="enumeration" name="loadingPolicy">
      <ui:label>Lazy loading</ui:label>
      <ct:enumeration value="onPage">
      <ui:label>Disabled</ui:label>
      </ct:enumeration>
      <ct:enumeration value="lazy">
      <ui:label>Enabled</ui:label>
      </ct:enumeration>
      <ui:value-if-unset>onPage</ui:value-if-unset>
    </ct:field>
    <ct:field type="enumeration" name="skipLoadingOn" multiple="true">
      <ui:label>Skip on device</ui:label>
      <ct:enumeration value="large">
      <ui:label>Large</ui:label>
      </ct:enumeration>
      <ct:enumeration value="medium">
      <ui:label>Medium</ui:label>
      </ct:enumeration>
      <ct:enumeration value="small">
      <ui:label>Small</ui:label>
      </ct:enumeration>
    </ct:field>
    <ct:field name="fragmentToken" type="basic" mime-type="text/plain">
      <ui:label>Fragment Token</ui:label>
      <ui:hidden/>
    </ct:field>
  </ct:options>
```

Copyright © 2010-2014 Escenic AS

```
  ...
</area>
```

Copyright © 2010-2014 Escenic AS

9 Configuring Components

Widget Framework used various escenic components and third party components. Some of these components needs configuration. This chapter will depict how to configure those services.

9.1 reCAPTCHA

reCAPTCHA is free CAPTCHA service that helps to digitize books, newspapers and old time radio shows. This service is provided by google. Widget Framework is using **reCAPTCHA** in comments widget, contactForm widget.

To use reCAPTCHA on your site you must have a public key and a private key. You have to register in https://www.google.com/recaptcha/admin/create using your domain to get the keys. You have to add those keys in captcha configuration file which is **`com.escenic.framework.captcha.ReCaptchaConfig.properties`**. If the path does not exist create the path. Content of the ReCaptchaConfig.properties file should be:

```
$class=com.escenic.framework.captcha.ReCaptchaConfig
          publicKey=
          privateKey=
```

You have to add public key in **publicKey** field and private key in **privateKey** field. If you don't use correct public and private key provided by reCAPTCHA based on your domain name then captcha will not be generated.

9.2 Solr

Escenic Content Engine's search functionality is provided by Apache Solr, a Java-based open source search engine that runs as a web application alongside the Content Engine. The Widget Framework Data Source component uses Solr search for it's **Query by search** option, and it is also used for Widget Framework event search functionality. In order for this search functionality to work, the **`com.escenic.framework.search.solr.SolrSearchEngine`** and **`com.escenic.framework.search.solr.HttpClientFactory`** components must be correctly configured.

9.2.1 Search Client Configuration

`HttpClientFactory` is a Content Engine component for defining HTTP client configuration parameters. You should modify it according to your site's needs. A sample configuration file for **`HttpClientFactory`** is provided in **`/misc/siteconfig/com/escenic/framework/search/solr/HttpClientFactory.properties`**:

```
# define the socket timeout in milliseconds, which is timeout for waiting for data
# socketTimeout=5000
# define the timeout in milliseconds until a connection is established
# connectionTimeout=3000
# for enable/disable caching of solr search result
# enableHttpClientCache=false
```

Copyright © 2010-2014 Escenic AS

```
# Sets the maximum number of cache entries the cache will retain.
# maxCacheEntries=1000
# max connection per route
# defaultMaxPerRoute=10
# total number of connection
# totalConnection=10
```

9.2.2 Solr Server Configuration

A default configuration for Solr is included with the Widget Framework. Provided parameters are -

```
$class=com.escenic.framework.search.solr.SolrSearchEngine
httpClientFactory=./HttpClientFactory
solrServerURI=${jndi:java:comp/env/escenic/presentation-solr-base-uri}
restrictedParameters=
defaultRowCount=10
maxRowCount=500
fields=*
```

The most important property is **`solrServerURI`**. This property is mandatory and must reference a JNDI environment variable as shown above. The JNDI environment variable itself must be set to point to the presentation Solr base URI:

```
<Environment name="escenic/presentation-solr-base-uri"
             value="http://server:port/solr/"
             type="java.lang.String"
             override="false"/>
```

Copyright © 2010-2014 Escenic AS

10 Modifying Resource Bundles

It is possible to override the messages defined in the various **`ApplicationResources.properties`** files that come with Widget Framework. In this chapter, we will discuss how to do so in the context of the demo webapps that are part of various Widget Framework distributions.

Please note that special characters (e.g, Arabic characters) in resource bundles should be Unicode-encoded. You can use the **`native2ascii`** tool that is shipped with JDK for the conversion.

10.1 Overriding Messages in Standard Resource Bundles

Let's assume that you want to override some messages that are defined in the standard **`ApplicationResources.properties`** files in the **`Widget Framework Core`** distribution. You need to go through the following steps :

- Go to the **`misc/demo`** directory of the **`Widget Framework Core`** distribution.
- Create an **`ApplicationResources.properties`** file in **`src/main/resources/com/escenic/framework`** folder. Define your customized key-value pairs in this file.
- Run **`mvn clean install`** command in **`misc/demo`** folder. The demo webapp war file will be available under **`misc/demo/target`** folder. The merged **`ApplicationResources.properties`** file will be in **`WEB-INF/classes/com/escenic/framework`** folder within the war. It will contain the custom key-value pairs that you defined earlier.

10.2 Adding New Resource Bundles

Let's assume that you want to add a new resource bundle (**`/com/escenic/framework/custom/CustomResources.properties`**) for the core widgets. You need to go through the following steps :

- Go to the **`misc/widgets`** directory of the **`Widget Framework Core`** distribution. Create **`CustomResources.properties`** files in the **`src/main/resources/com/escenic/framework/custom`** directory in various maven modules and put appropriate key-value pairs in them.
- Run **`mvn clean install`** command in **`misc/widgets`** folder. All the updated artifacts will now be installed in your local repository.
- Go to the **`misc/demo`** directory.
- The shade plugin configuration in the **`pom.xml`** file in **`misc/demo`** folder should be updated so that it merges **`CustomResources.properties`** files during the build. To be specific, the following additional configuration should be added :

```
<transformer
 implementation="org.apache.maven.plugins.shade.resource.AppendingTransformer">
   <resource>WEB-INF/classes/com/escenic/framework/custom/
CustomResources.properties</resource>
</transformer>
```

Copyright © 2010-2014 Escenic AS

- The **pom.xml** file should also be updated so that an additional argument is passed to the class **com.escenic.widgetframework.DemoWebappResourceModifier** during the build :

```
<argument>/com/escenic/framework/custom/CustomResources.properties</argument>
```

- Run **mvn clean install** command in **misc/demo** directory. The demo webapp war file will be available under **misc/demo/target** folder. The merged **CustomResources.properties** file will be in **WEB-INF/classes/com/escenic/framework/custom** folder within the war. It will contain all the key-value pairs that you defined in the first step.

Copyright © 2010-2014 Escenic AS