Escenic Widget Framework

# Developer Guide

4.5.2-2

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:

- 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.)

# 2 Installation

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

- A suitable version of the Escenic Content Engine and Escenic assembly tool have been installed as described in the **Escenic Content Engine Installation Guide** and are in working order.
- You have the credentials needed to access Escenic's SW repositories.
- The following plug-ins have been installed as described in the relevant plug-in guides, and are in working order:
  - **Menu Editor** (3.1 or later)
  - **Analysis Engine** (3.0 or later)

### Choosing an installation method

Until recently, all Escenic-supplied components had to be installed by manually downloading and unpacking archive files. This is no longer the case. The Content Engine itself, the assembly tool, the ece scripts and all plug-ins are now available as **`deb`** packages (for installing on Ubuntu/Debian systems) and as **`rpm`** packages (for installation on RedHat/CentOS systems). This is now the recommended method of installing Escenic systems, although it is still possible to use the old method based on manually downloading and unpacking archives. Note, however, that when you are installing a plug-in on an existing Content Engine installation, you should use the same installation method as you used for the Content Engine itself.

Using the new package installation method offers many advantages over the old manual installation procedure:

- It is faster and quicker
- It is significantly less error-prone
- It supports a fully automated upgrade process in which not only the installed packages themselves are upgraded, but also deployed EAR files. An upgrade script automatically checks the EAR files for copies of JAR files that have been updated, and replaces them with the new versions.

The old installation method will continue to be documented for the moment. Components installed using the new method are installed in **`/usr/share/escenic`**, whereas the old method recommends installation in **`/opt/escenic`**, and some other path components are slightly different. To cope with these differences, the following placeholders are used in some paths:

| Placeholder | NEW METHOD path | OLD METHOD path |
|---|---|---|
| *engine-installation* | **`/usr/share/escenic/escenic-content-engine-`***engine-version* | **`/opt/escenic/engine`** |
| *assemblytool_installation* | **`/usr/share/escenic/escenic-assemblytool`** | **`/opt/escenic/assemblytool`** |

When installing the Widget Framework, you should use the same method as you used to install the Escenic Content Engine.

> Note that if you are planning to install the Escenic Content Engine using the new package-based method, you can install the Widget Framework simultaneously by simply including the Widget Framework package name (`widget-framework`) in the `apt-get` installation command. You only need to follow this procedure if you have already installed the Content Engine and now need to install the Widget Framework.

If you use CUE as your Escenic editor, then in addition to installing the Widget Framework, you should also install a matching CUE Widget Framework plug-in into CUE. This plug-in adds template editing capabilities to CUE. Whichever method you plan to use for installing the Widget Framework itself, CUE plugins are always package-based installations. For details, see section 2.9.

## 2.1 Browser Support

The Widget Framework generates modern HTML 5 web pages and makes use of the latest techniques in web page design. Widget Framework publications therefore require the use of modern browsers for satisfactory results. The following browser versions are supported:

| Browser | Versions supported |
|---|---|
| Internet Explorer | Version 8 or higher |
| Firefox | Version 25 or higher |
| Chrome | Version 30 or higher |
| Safari | Version 7 or higher |

Pages cannot be expected to display correctly in older browser versions.

## 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.

## 2.3 Package-based Installation

To install the Widget Framework on an Ubuntu or other Debian-based system, do the following on your **assembly-host** and on each of your **engine-hosts**:

1. Log in as **`root`**.
2. If necessary, add the Escenic **`apt`** repository to your list of sources:

   ```
   # echo "deb http://user:password@apt.escenic.com stable main non-free" >> /etc/
   apt/sources.list.d/escenic.list
   ```

   where *user* and *password* are your Escenic download credentials (the same ones you use to access the Escenic Maven repository). If you do not have any download credentials, please contact Escenic support.
3. Enter the following commands:

   ```
   # apt-get update
   # apt-get install escenic-widget-framework
   ```

On RedHat / CentOS systems, enter the following command as **`root`** on your **assembly-host** and each of your **engine-hosts**:

```
# rpm -Uvh https://user:password:yum.escenic.com/rpm/escenic-widget-
framework-4.5.2-2.x86_64.rpm
```

## 2.4 Old-style Installation

Installing the Widget Framework involves the following steps:

1. Log in as **`escenic`** on your **assembly-host**.
2. Download the Widget Framework distribution from the Escenic software repository. 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 https://user:password@maven.escenic.com/com/escenic/plugins/widget-
   framework/widget-framework/4.5.2-2/widget-framework-4.5.2-2.zip
   ```

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

   ```
   $ mkdir engine-installation/plugins
   ```
4. Unpack the downloaded distribution file:

```
$ cd engine-installation/plugins
$ unzip /mnt/download/widget-framework-4.5.2-2.zip
```

## 2.5 Re-assembling Applications

After installation, you will need to reassemble your web applications:

1. Log in as **escenic** on your **assembly-host**.
2. 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.
3. If you have a single-host installation, then skip this step.

   On each **engine-host**, 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 escenic@assembly-host-ip-address:/var/cache/escenic/engine.ear /tmp/
   ```

   where *assembly-host-ip-address* is the host name or IP address of your **assembly-host**.
4. On each **engine-host**, deploy the EAR file and restart the Content Engine by entering:

   ```
   $ ece stop deploy start --file /tmp/engine.ear
   $ ece restart
   ```

## 2.6 Verifying The Installation

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

The plug-in is correctly installed.

The plug-in is not correctly installed.

## 2.7 Build the code

After installing 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.escenic.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 to your application server.

## 2.8 Upgrading

How you upgrade the Widget Framework plug-in depends upon how you installed it in the first place. If you installed it using the new package-based method as described in section 2.3, then upgrading is very easy. If you installed it using the old method described in section 2.4, then it requires a bit more work.

### 2.8.1 Package-based Upgrade

> You can only use this upgrade method if you have previously installed the Widget Framework plug-in as described in section 2.3. Otherwise you need to follow the method described in section 2.8.2.

All you need to do to upgrade your Widget Framework plug-in is:

1. Read the release notes for your planned upgrade. Make a note of any special tasks that need to be carried out in connection with the upgrades.
2. Log in as **`root`** on your **assembly-host** and on each of your **engine-hosts**, and enter the following commands:

   ```
   # apt-get update
   # apt-get upgrade
   ```
3. Carry out any required upgrade tasks.

### 2.8.2 Old-style Upgrade

> You should only use this upgrade method if you have previously installed the Widget Framework plug-in as described in section 2.4. Otherwise you need to follow the method described in section 2.8.1.

To upgrade the Widget Framework plug-in to version 4.5.2-2:

1. Read the release notes for your planned upgrade. Make a note of any special tasks that need to be carried out in connection with the upgrades.
2. Log in as **`escenic`** on your **assembly host**.
3. Download the distribution file to a temporary location by entering:

   ```
   $ cd /tmp/
   $ wget http://user:password@maven.escenic.com/com/escenic/widget-framework/widget-framework/4.5.2-2/widget-framework-4.5.2-2.zip
   ```

   where *user* and *password* are the user name and password you have received from Escenic.

4. Remove the old version of the Widget Framework plug-in from your Content Engine **`plugins`** folder.

   ```
   $ mv /opt/escenic/engine/plugins/widget-framework/ /tmp/
   ```

5. Unpack the downloaded installation package into the **`plugins`** folder:

   ```
   $ cd /opt/escenic/engine/plugins/
   $ unzip /tmp/widget-framework-4.5.2-2.zip
   ```

6. Re-assemble the Content Engine by running the **`ece`** script:

   ```
   $ ece assemble
   ```

7. **If you are installing everything on one host, then skip this step.**

   Log in as escenic on each of your **engine-hosts** and 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-hosts**, then you can do this as follows:

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

   where:

   ***engine-host-ip-address***
   is the host name or IP address of one of your engine-hosts.

   ***assembly-host-ip-address***
   is the host name or IP address of your assembly-host.

8. On each **engine-host**, deploy the EAR file by entering:

   ```
   $ ece deploy
   ```

   and then restart the Content Engine by entering:

   ```
   $ ece restart
   ```

9. Carry out any required upgrade tasks.

## 2.9 CUE Plug-in Installation

Installing the Widget Framework plug-in involves installing a CUE plug-in as well as the main Escenic plug-in. To do this:

1. Log in as **`root`** on the host on which your CUE editor is installed.
2. Update your package lists:

   ```
   # apt-get update
   ```

3. Download and install the Widget Framework plug-in:

   ```
   # apt-get install cue-plugin-widget-framework-4.2
   ```

### 2.9.1 Upgrading

To upgrade the CUE plug-in:

1. Log in as **`root`** on the host on which your CUE editor is installed.
2. Update your package lists:

```
# apt-get update
```

3. Upgrade the plug-in:

```
# apt-get upgrade
```

## 2.10 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 the **misc/demo/src/main/webapp/template/widgets/***widget-name***/model** folder.
4. Add a **<ui:group/>** definition for custom widgets to **misc/demo/src/main/webapp/META-INF/escenic/publication-resources/escenic/content-type** file.
5. Run **mvn clean install** command in the **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.

## 2.11 Creating A 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.

## 2.12 Supporting Multiple Languages

The static texts displayed by the Widget Framework are stored in an application resource file called **ApplicationResources.properties**. Each publication has its own copy of **ApplicationResources.properties**, located in the following folder:

*publication-path***/WEB-INF/classes/com/escenic/framework/**

The contents of **ApplicationResources.properties** are the same in all publications.

You can add support for a languages other than US English as follows:

1. Make a copy of **ApplicationResources.properties** (download it from one of your publications on the server).
2. Translate the contents to the required language.
3. Rename the translated file to **ApplicationResources_***locale***.properties**, where *locale* is a locale code supported by Java (see http://www.oracle.com/technetwork/java/javase/java8locales-2095355.html). A German application resource file, for

example, could be called **`ApplicationResources_de_DE.properties`** or **`ApplicationResources_de_AT.properties`** (for Austrian German).

4. Upload the translated file back to *publication-path*`/WEB-INF/classes/com/escenic/framework/`. If you have several publications, then upload it to all of them.

Now all you need to do to switch a publication to the new language is to set the **`locale`** section parameter in your publication's home section to the correct locale code (**`de_DE`**, for example). If there is no **`locale`** section parameter in the publication home section, then add it. To switch the publication back to US English, set **`locale`** to **`en_US`**.

> It is not necessary to restart the Content Engine when you add a new application resource file. However, you **do** need to restart the Content Engine if you change the contents of an existing application resource file. You can avoid having to restart the Content Engine in this case by:
>
> 1. Renaming the modified application resource file (from **`ApplicationResources_de_DE.properties`** to **`ApplicationStrings_de_DE.properties`**, for example)
> 2. Modifying the **`javax.servlet.jsp.jstl.fmt.localizationContext`** property in your **`web.xml`** accordingly. By default it is set to **`com.escenic.framework.ApplicationResources`**, so in this case you would want to change it to **`com.escenic.framework.ApplicationStrings`**.

## 2.13 Enabling Video Playback

For any publication that is required to support video playback, the publication **`web.xml`** file must include the following servlet declaration:

```
<servlet>
  <servlet-name>mediaInfoServlet</servlet-name>
  <servlet-class>com.escenic.framework.servlet.MediaInfoServlet</servlet-class>
</servlet>
```

# 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
```

```
            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 templates.

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.

### 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 inContent 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.

#### 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, seesection 3.4.5.

## 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 "**Hello World** widget.

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
```

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

The **Hello World** widget has two views: one called **`default`**, which is a div containing the text "hello world", and another called **`custom`**, which can contain a custom text. This requires the following structure:

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

### 3.3.1 Hello World Content Type

Only the **`custom`** view of the **Hello World** widget requires any configuration, so it only needs one configuration panel in Content Studio (in addition to the General and Advanced panels that all widgets have).

```
<?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>
```

```
    <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**

```
<%@ 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 Implementing Widget Functionality in Java

You can often improve the performance and maintainability of your widgets by implementing part of their functionality in Java rather than doing everything in JSP templates. You do this by creating a **model processor**: a Java class that can modify the widget's model object, and therefore take over some or all of the work done by controller JSP templates.

If a widget has both a model processor and controller JSP templates, then the model processor is executed before the JSP templates.

### 3.4.1 Writing 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.GenericModelProcessor`**
- Override **`updateModel`** method

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

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

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


public class HelloWorldWidgetDefaultViewModelProcessor extends GenericModelProcessor {
  @Override
  protected void updateModel(final Map<String, Object> pWidgetModel,
                             final PresentationArticle pWidgetContent) {

    if (StringUtils.isNotBlank(pWidgetModel.get("greetings").toString())) {
      pWidgetModel.put("greetings", "Welcome");
    }
  }
}
```

You could use then access the contents of the resulting model object from your view JSP template as follows:

```
<h3>${widget.model.greetings}</h3>
```

> Note that the modified widget model is cached, so the model processor is not executed every time a widget is rendered.

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-4.5.2-2.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

#### 3.4.1.1 Configuration

To configure and register the model processor you must:

1. Create a file called **`HelloWorldWidgetDefaultViewModelProcessor.properties`**.
2. Enter the following in the file:

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

   You can specify several view names when you define **`viewNames`**: use a comma to separate them.
3. Create a file called **`HelloWorldWidgetDescriptor.properties`**.
4. Enter the following in the file:

   ```
   $class=com.escenic.framework.descriptor.WidgetDescriptor
   widgetName=helloWorld
   modelProcessor.helloWorld.1=/com/escenic/framework/controller/
   HelloWorldWidgetDefaultViewModelProcessor
   ```

   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/framework/controller/
   HelloWorldWidgetCustomViewModelProcessor
   ```

   (Of course, you would then also need to repeat steps 1 and 2 to configure your **`HelloWorldWidgetCustomViewModelProcessor`** as well.)
5. Create a file called **`DescriptorRegistry.properties`**.
6. Enter the following in the file:

   ```
   $class=com.escenic.framework.descriptor.DescriptorRegistry
   widgetDescriptor.helloWorld=./HelloWorldWidgetDescriptor
   ```

#### 3.4.1.2 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
  |           |  +-HelloWorldWidgetDefaultViewModelProcessor.properties
  |           +-descriptor
  |              +-DescriptorRegistry.properties
  |              +-HelloWorldWidgetDescriptor.properties
  +-mycompany
    +-view
      +-processor
        +-HelloWorldWidgetDefaultViewModelProcessor.class
```

### 3.4.2 Writing a Common Model Processor

It is possible to write a common model processor that works on all widgets. You might, for example, want to read some fields from a panel that is displayed by all widgets.

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

The actual process of writing such a processor is same as that described insection 3.4.1.

#### 3.4.2.1 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 and add the following:

```
modelProcessor.custom-key=/com/mycompany/controller/processor/
CustomCommonModelProcessor
```

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

You may also need to create a **.properties** file for your **CustomCommonModelProcessor** and configure it according to your requirements.

#### 3.4.2.2 Packaging

A common 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 common 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

**Common model processor package structure**

For the common 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
|              +-CustomCommonModelProcessor.properties
+-mycompany
  +-controller
    +-processor
       +-CustomCommonModelProcessor.class
```

### 3.4.3 Writing a Widget Processor

A **widget processor** is a Java class that performs some action every time a widget is rendered. A widget processor can be configured to work either for a single view or for multiple views. A widget processor must

- Extend **com.escenic.framework.controller.processor.GenericWidgetProcessor**
- Override **process** method.

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

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

import com.escenic.framework.controller.processor.GenericWidgetProcessor;
import neo.xredsys.presentation.PresentationArticle;
import javax.servlet.http.HttpServletRequest;
import java.util.Map;


public class HelloWorldWidgetDefaultViewWidgetProcessor extends GenericWidgetProcessor
 {
  @Override
  protected void process(final Map<String, Object> pWidgetContext,
                         final HttpServletRequest pRequest) {

    PresentationArticle article = (PresentationArticle)
 pRequest.getAttribute("article");
    pWidgetContext.put("viewCount", getViewCount(article));
  }
}
```

Assume that the **`getViewCount()`** method returns the number of times the current article has been viewed. Your widget processor adds this value to **`pWidgetContext`**, the **widget context bean**, so that it can then be retrieved by a view JSP template. For example:

```
<p>This page has been viewed ${widget.viewCount} times.</p>
```

The widget context bean provides access to all data associated with the context widget (including its model object). For further information, see section 3.6.

Before a widget processor can be used it must be:

- Compiled
- Added to the web application's classpath

To compile a widget processor you must have **`wf-presentation-4.5.2-2.jar`** in your classpath.

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

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

#### 3.4.3.1 Configuration

To configure and register the widget processor you must:

1. Create a file called **`HelloWorldWidgetDefaultViewWidgetProcessor.properties`**.
2. Enter the following in the file:

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

You can specify several view names in **viewNames**, separated by commas.

3. Create a file called **HelloWorldWidgetDescriptor.properties**.
4. Enter the following in the file:

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

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

```
widgetProcessor.helloWorld.2=/com/mycompany/view/processor/
HelloWorldWidgetCustomViewWidgetProcessor
```

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

5. Create a file called **DescriptorRegistry.properties**.
6. Enter the following in the file:

```
$class=com.escenic.framework.descriptor.DescriptorRegistry
widgetDescriptor.helloWorld=./HelloWorldWidgetDescriptor
```

#### 3.4.3.2 Packaging

A widget 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 widget processor class
   - The Content Engine's package naming conventions
2. Pack it in a JAR file

##### Widget Processor Package Structure

For the widget 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
|           |  +-HelloWorldWidgetDefaultViewWidgetProcessor.properties
|           +-descriptor
|              +-DescriptorRegistry.properties
|              +-HelloWorldWidgetDescriptor.properties
+-mycompany
```

```
+-view
  +-processor
    +-HelloWorldWidgetDefaultViewWidgetProcessor.class
```

### 3.4.4 Writing a Common Widget Processor

It is possible to write a widget processor that will work for all widgets.

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

The actual process of writing such a processor is the same as that described insection 3.4.3.

#### 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 and add the following:

```
$class=com.escenic.framework.controller.impl.DefaultMapController
widgetProcessor.custom-key=/com/mycompany/controller/processor/
CustomCommonWidgetProcessor
```

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

You may also need to create a **`.properties`** file for your **`CustomCommonWidgetProcessor`** and configure it according to your requirements.

#### 3.4.4.2 Packaging

A common 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 common 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

**Common widget processor package structure**

For the common 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
|              +-CustomCommonWidgetProcessor.properties
+-mycompany
  +-controller
    +-processor
      +-CustomCommonWidgetProcessor.class
```

### 3.4.5 Writing a Java Controller

The default Java controller component runs on every widget execution to process widget information and invoke processors. If your requirement cannot be met by the implementing processors mentioned above, you can 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.

#### 3.4.5.1 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.

#### 3.4.5.2 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.
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**.

## 3.5 Modifying an Existing Controller

In version 4.5.2-2 of the Widget Framework , most of the controller functionality has been moved to Java. 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). This means that JSP controllers can still 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.

### 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.

For further information about the Widget Framework API, take a look at the **Escenic Widget Framework 4.5.2-2 API documentation**.

### 3.6.3 Decorating ResultItems

To decorate **ResultItem**s you must write a Java decorator class that:

- Extends **com.escenic.framework.decorator.OnDemandResultItemDecorator**
- Overrides **OnDemandResultItemDecorator**'s **createExtraInfo** and **createRelated** methods.

The following example shows a **RelatedVideoDecorator** that decorates a **ResultItem** with related videos.

```
package com.escenic.framework.widget.common;

import com.escenic.framework.decorator.OnDemandResultItemDecorator;
import com.escenic.framework.presentation.ResultItem;
import neo.xredsys.presentation.PresentationElement;
import java.util.ArrayList;
import java.util.Collections;
import java.util.List;

public class RelatedVideoDecorator extends OnDemandResultItemDecorator {
private int mMaxItem;
```

```
private String mKey;

public class RelatedVideoDecorator extends OnDemandResultItemDecorator {
  private int mMaxItem;
  private String mKey;

  public RelatedVideoDecorator(final ResultItem pResultItem,
                               final String pKey,
                               final int pMaxItem) {
    super(pResultItem);
    mMaxItem = pMaxItem > 0 ? pMaxItem : Integer.MAX_VALUE;
    mKey = pKey.toLowerCase();
  }

  @Override
  protected Object createExtraInfo(final String pKey) {
    return null;
  }

  @Override
  protected List<PresentationElement> createRelated(final String pKey) {
    if (pKey.equals(mKey)) {
      // retrieve all related videos
      List<PresentationElement> relatedVideos = new ArrayList<>();
      relatedVideos = getRelatedVideos(mMaxItem);
      return relatedVideos;
    }
  return null;
  }
}
```

You also need to write a Java processor class that invokes the decorator class you have created to decorate the **`ResultItem`**.

This class must:

- Extend **`com.escenic.framework.controller.processor.ResultItemDecoratorProcessor`**
- Override **`ResultItemDecoratorProcessor`**'s **`createResultItemTransformer`** method

This **`RelatedVideoDecoratorProcessor`** class, for example, invokes the example **`RelatedVideoDecorator`**.

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

import com.escenic.framework.controller.processor.ResultItemDecoratorProcessor;
import com.escenic.framework.decorator.ResultItemDecorator;
import com.escenic.framework.presentation.ResultItem;
import com.escenic.framework.widget.common.RelatedVideoDecorator;
import com.google.common.base.Function;
import java.util.Map;

public class RelatedVideoDecoratorProcessor extends ResultItemDecoratorProcessor {
  private int mMaxItems;

  @Override
  protected Function<ResultItem, ResultItemDecorator>
      createResultItemTransformer(final Map<String, Object> pWidgetContext) {
```

```
      return new Function<ResultItem, ResultItemDecorator>() {
        @Override
        public ResultItemDecorator apply(final ResultItem pResultItem) {
          ResultItemDecorator decoratedItem = new RelatedVideoDecorator(
            pResultItem, "videos", getMaxItems());
          return decoratedItem;
        }
      };
    }

    public int getMaxItems() {
      return mMaxItems;
    }

    public void setMaxItems(final int pMaxItems) {
      mMaxItems = pMaxItems;
    }
  }
```

Now create a **properties** file called **RelatedVideoDecoratorProcessor.properties** and add the following:

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

maxItems=5
```

Finally, register the above model processor. For further information about model processor configuration, please see section 3.4.1.1.

The **ResultItem**'s related videos will now be accessible from JSPs as follows:

```
${requestScope.resultItem.related.videos}
```

## 3.7 Widget Properties

Widget properties are named values stored in widget instances. When a widget is edited inContent 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://docs.escenic.com/ece-resource-ref/5.7/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
```

### 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 Including Javascript Code

You can include Javascript code in your widget in the following ways:

**Inline code**
You can simply include in-line Javascript in your widget. It will work, but it is not recommended. If a widget containing in-line code is included several times on a page, then the code will be duplicated, making your pages slower to load and less efficient.

**Use the jscripts folder**
Put all your Javascript code in files in your webapp's **`jscripts`** folder. The Widget Framework will then merge all the files it finds in this folder (in alphabetical order) into a single file, together with any Javascript files in the WAR files your publication depends on. For detailed information about this process see section 11.2.1.

> Merging your files in this way does not cause any problems with interference between widgets. **`document.ready()`** and other events will all be handled as expected.

**Use RequireJS**
RequireJS is a Javascript module loader that manages dependencies and provides efficient on-demand loading of Javascript. If you want to use RequireJS to manage your Javascript code, then you should **not** put your Javascript files in your webapp's **`jscripts`** folder. You are free to use any other location (for example **`static/js`**). For further information about the use of RequireJS, see the RequireJS documentation.

## 3.9 Javascript Execution for Delayed Page Content

Some Widget Framework features depend on delayed loading of page content:

- Lazy loading (see section 9.6)
- In-line linked content (see the **Link Settings** option of the Teaser widget, for example, in the **Escenic Widget Framework Core Widgets Reference**).

In both cases, pages have content that is is not loaded to the DOM at the normal time, but loaded later using AJAX functionality. This means you have to be careful about when any Javascript code that you want to operate on this delayed content is invoked. If the code is invoked too early then it may not work because the elements it addresses have not yet been loaded.

Code that is invoked for a widget on the **`$(document).ready()`** event **inside the widget itself** 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 script will not work, however, if the script is loaded from the document and **awesome-slideshow** is lazy-loaded - because the addressed element will not be present when the **$(document).ready()** event fires. And the **$(document).ready()** event will never work for in-line loaded content.

The Widget Framework therefore provides a custom event called **pageContentReady()** which is triggered whenever delayed content is added to the DOM. You can use this event to ensure that code is executed at the right point, immediately after the content it addresses has been loaded:

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

## 3.10 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** that generates the required HTML markup.
- Javascript code for determining which image representation image is required. Two different methods are available:

  **adaptive**
  This is the default method. It calculates which image representation to load based on the **img** element **width**.

  **picturefill**
  This is a slightly simpler method based on 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.

### 3.10.1 Configuring Responsive Image Handling

You can control the Widget Framework's responsive image handling by setting a number of section parameters (usually in a publication's root section).These section parameters are associated with content profiles (see chapter 5), 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`**
You can set this parameter to one of the following values:

**`adaptive(default)`**
For all **visible** images, representations are selected and loaded based on the **`width`** specified in the **`img`** element. For **hidden** images (for example, images that are not initially visible in a carousel), low quality representations are loaded initially. Once such a hidden image is actually displayed, the proper representation is loaded and used to replace the low-quality representation. You can control the details of how the low quality proxy image representations are generated using the **`image.proxy.`***xxx* section parameters described in section 3.10.2.

**`picturefill`**
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.

The default policy is **`adaptive`**.

**`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`**.

If **`image.policy`** is set to any other value, then this parameter is not used.

**`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. The default value is **`320,480,640,767`**.

If **`image.policy`** is set to any other value, then this parameter is not used.

### 3.10.2 Configuring Image Preloading

If you enable responsive images, then images are loaded **after** the rest of the document has been loaded and displayed. Loading the images at the end often results in an annoying "page bounce" effect, where the text on the page is moved around and reformatted as the images are loaded. You can prevent this happening by enabling the Widget Framework's **image preloading** feature.

Image preloading prevents page bounce by loading alternative low-bandwidth images together with the page, and then replacing them with the proper full images at the end.

You can choose between two different image preloading methods:

**proxy**
Low-quality representations of each image are loaded in-line, and then replaced by the full images after the document has finished loading.

**placeholder**
Small placeholder images are loaded in-line, and then replaced by the correct images after the document has finished loading. By default, the placeholder images are transparent images located in the webapp's **static/img/placeholder/** folder.

You can control preloading by setting a number of section parameters (usually in a publication's root section).These section parameters are associated with content profiles (see chapter 5), so that different content profiles can have different preload settings.

The image preloading section parameters are:

- **wf.contentprofile.***content-profile***.image.preload**
- **wf.contentprofile.***content-profile***.image.proxy.size.scale**
- **wf.contentprofile.***content-profile***.image.proxy.enable.imagesize.min**
- **wf.contentprofile.***content-profile***.image.proxy.size.min**
- **wf.contentprofile.***content-profile***.image.proxy.size.max**

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

The parameters should be configured as follows:

**image.preload**
You can set this parameter to one of the following values:

**proxy**
Preload low-quality initial images.

**placeholder**
Preload static placeholder images.

**none (default)**
Do not preload any images.

**image.proxy.size.scale**
If **image.preload** is set to **proxy** and **image.policy** (see section 3.10.1) is set to **grid**, then this value is used as a percentage scale to calculate size of the proxy image. A value of **30**, for example means that if the actual image width is **500**, the proxy/placeholder image width will be **150**. The default value is **30**.

**image.proxy.enable.imagesize.min**
If **image.preload** is set to **proxy** and **image.policy** (see section 3.10.1) is set to **grid**, then proxy images will only be used for images that are wider than the minimum specified here. If you specify **200**, for example, then proxy images will only be loaded for images that are more than 200 pixels wide. The default value is **200**.

**image.proxy.size.min**
If **image.preload** is set to **proxy** and **image.policy** (see section 3.10.1) is set to **adaptive** or **picturefill** then this parameter specifies the minimum width of the proxy image. The default value is **50**.

**`image.proxy.size.max`**
If **`image.preload`** is set to **`proxy`** and **`image.policy`** (see section 3.10.1) is set to **`adaptive`** or **`picturefill`** then this parameter specifies the maximum width of the proxy image. The default value is **`200`**.

## 3.11 Widget Packaging

Widgets are packaged as Maven WAR artifacts. The Widget Framework includes two artifacts for assisting in the development and packaging of widgets:

**`widget-framework-widgets`**
A master artifact that acts as the parent POM for all widget artifacts and manages the process of building widgets.

**`widget-framework-webapp-sdk`**
This artifact acts as an SDK for widget development. Including a dependency on this module reduces the number of framework dependencies needed for widget development.

The following **`pom.xml`** snippet shows how to package the example **Hello world** widget described in section 3.3.

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://
maven.apache.org/xsd/maven-4.0.0.xsd">
  <parent>
   <artifactId>widget-framework-widgets</artifactId>
   <groupId>com.escenic.widget-framework</groupId>
   <version>4.5.2-2</version>
  </parent>

  <groupId>custom.widget</groupId>
  <artifactId>widget-hello-world</artifactId>
  <version>$version</version>
  <packaging>war</packaging>

  <dependencies>
   <dependency>
     <groupId>com.escenic.sdk</groupId>
     <artifactId>widget-framework-webapp-sdk</artifactId>
     <version>4.5.2-2</version>
     <type>pom</type>
   </dependency>
    ... other dependencies ...
  </dependencies>

</project>
```

This will result in the production of three artifacts:

**`widget-hello-world-`*`version`*`-blueprint.war`**
This WAR file contains the configuration components of your widget, and is intended for use in blueprints.

**widget-hello-world-*version*-site.war**
This WAR file contains the presentation components of your widget, and is intended for use in site publications.

**widget-hello-world-*version*.war**
This WAR file contains both the configuration and presentation components of your widget, and is intended for use in legacy publications.

## 3.12 Using Custom Widgets

To make use of custom widgets you have created, all you need to do is add them as dependencies to your publications. Exactly how you do this depends on the publication type (site publications, blueprints or legacy all-in-one publications).

### 3.12.1 Using Custom Widgets in Blueprints

In order to use the example **Hello World** custom widget in a blueprint, add the following dependency to your blueprint project's **pom.xml** file:

```
<dependency>
  <groupId>custom.widget</groupId>
  <artifactId>widget-hello-world</artifactId>
  <version>$version</version>
  <type>war</type>
  <classifier>blueprint</classifier>
</dependency>
```

The highlighted **blueprint** classifier ensures that only the required configuration components of the widget are included in the project.

### 3.12.2 Using Custom Widgets in Site Publications

In order to use the example **Hello World** custom widget in a site publication, add the following dependency to your site project's **pom.xml** file:

```
<dependency>
  <groupId>custom.widget</groupId>
  <artifactId>widget-hello-world</artifactId>
  <version>$version</version>
  <type>war</type>
  <classifier>site</classifier>
</dependency>
```

The highlighted **site** classifier ensures that only the required presentation components of the widget are included in the project.

### 3.12.3 Using Custom Widgets in Legacy Publications

In order to use the example **Hello World** custom widget in a legacy all-in-one publication that contains both content and layout, add the following dependency to your project's **pom.xml** file:

```
<dependency>
  <groupId>custom.widget</groupId>
```

```
    <artifactId>widget-hello-world</artifactId>
    <version>$version</version>
    <type>war</type>
  </dependency>
```

This dependency includes no classifier. Therefore, the **`widget-hello-world-`***version*`.war`** artifact that contains both the configuration and presentation components of the widget is included in the project.

# 4 Data Source Extensions

The Widget Framework Data Source component includes a range of different query types that you can use to retrieve items from the Content Engine. If these predefined query types are insufficient for your needs, you can extend the data source by making custom query types of your own.

Once a custom query type has been correctly defined, it can be used in exactly the same way as a built-in query type: it appears as an option in the **+Add query** dialog displayed by a Data Source **Query** tab list, and adds a collapsible form to the **Query** tab if selected, and can be combined with other standard and custom query types to create complex queries if required. (See Queries for a description of the standard query user interface.)

Making custom queries requires Java programming skills and involves the following tasks:

1. Create a query type definition. A query type definition is an XML file containing a complete description of the query and its user interface.
2. Package and deploy the query type definition.
3. Create a handler for the query type. A handler is a Java class that extends the com.escenic.framework.datasource.fetcher.impl.AbstractFetcher interface.
4. Compile, package and deploy your handler class.
5. Create various **`.properties`** files and add them to one of your configuration layers. These configuration layer files are used to:
   - Register the query type definitions you have created
   - Register the handler classes you have created and associate them with correct query types
   - Define additional aspects of the query type user interface such as labels, enumeration options and look-up services

## 4.1 Creating a Query Type Definition

You start defining a custom query type by creating an XML format file defining the input fields you want to be displayed in the query form in Content Studio. You can specify the type of control to display for each field, and you can also exercise some control over the layout of the form by using **`container`** elements to group fields together and arrange them in rows.

The following example shows a simple query type definition containing three fields, two of which will appear in a single row.

```
<?xml version="1.0" encoding="UTF-8"?>
<query name="customQuery" xmlns="http://xmlns.escenic.com/2014/datasource-query">
  <editor>
  <component name="section" />
  <field name="itemCount" type="number" />
  <container style="row" column-ratio="3:1">
    <field name="myField1" type="lookup" source="groupNames" />
    <field name="myField2" type="enumeration" enum="groupIndex" />
  </container>
  </editor>
</query>
```

For a complete description of the query definition format, see section 4.5.

## 4.2 Deploying a Query Type Definition

Before a query definition can be used, you need to package it correctly and deploy it in the correct location. You must package it in a **`.jar`** file, and deploy the **`.jar`** file to your application server (that is, Tomcat) **`escenic/lib`** folder. If you have created several query type definitions, then you can package them all together in the same **`.jar`** file if you wish. For example:

```
com
  mycompany
    datasource
      query
        customQuery1.xml
        customQuery2.xml
        customQuery3.xml
```

## 4.3 Creating a Handler Class

A **query handler** is a Java class that retrieves items from the Content Engine using search criteria entered into a Data Source query form. A query handler must:

- Extend com.escenic.framework.datasource.fetcher.impl.AbstractFetcher
- Override its **`fetch()`** method

The following example shows a skeleton handler class definition:

```
package com.mycompany.datasource.query.handler;

import com.escenic.framework.datasource.fetcher.impl.AbstractFetcher;
import com.escenic.framework.datasource.ContextInfo;
import com.escenic.framework.datasource.model.SourceDefinition;
import com.escenic.framework.datasource.model.SourceSettings;

import java.util.List;

public class MyQueryHandler extends AbstractFetcher {

  @Override
  public List<ResultItem> fetch(final SourceDefinition pSourceDefinition,
                                final SourceSettings pSettings,
                                final ContextInfo pContextInfo) {

    // your code here

  }
}
```

The **`fetch()`** method's first parameter is a com.escenic.framework.datasource.model.SourceDefinition object. This object is a bean containing all the values set by the user in the query tab. It has a set of **`get`***Xxx***`()`** methods for recovering

various predefined standard query parameters (**`getSection()`** and **`getContentTypes()`**, for example), plus a **`getFields()`** method that returns a map containing all other custom parameters that you may have defined for your query form. The query form (and therefore the contents of the **`SourceDefinition`** bean) are determined by the query type definition (see section 4.1).

Once you have a created a query handler class, you need to:

- Create **`.properties`** files to register the handler class in the system
- Package the handler in a JAR file
- Deploy the JAR file in your publication

### 4.3.1 Handler Configuration

Create a file called **`FetcherConfig.properties`**, and register your handler as follows:

```
$class=com.escenic.framework.datasource.FetcherConfig
fetcher.query-type=handler-path
```

where:

- *query-type* is the name of the query type as specified in your query type definition file. For the example file shown in section 4.1 you would enter **`customQuery`**.
- *handler-path* is the path of your handler. For the skeleton example shown in section 4.3 you would enter **`/com/mycompany/datasource/query/handler/MyQueryHandler`**.

Create a properties file for your class called **`MyQueryHandler.properties`** containing the following line at least:

```
$class=com.mycompany.datasource.query.handler.MyQueryHandler
```

If your class has properties that you want to set, you can of course also include these settings in the file.

### 4.3.2 Handler Packaging

A query handler 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

#### Handler Package Structure

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

```
com
  +-escenic
  |  +-servlet
  |   +-default-config
  |     +-com
```

```
|        +-escenic
|        | +-framework
|        |   +-datasource
|        |     +-FetcherConfig.properties
|        +-mycompany
|          +-datasource
|            +-query
|              +-handler
|                +-MyQueryHandler.properties
+-mycompany
  +-datasource
    +-query
      +-handler
        +-MyQueryHandler.class
```

## 4.4 Configuring a Data Source Extension

Once you have created a query definition and query handler, you need to configure your extension. This involves adding various files to one of your Escenic configuration layers (most likely the common layer) in order to provide the Widget Framework with the information it needs about the extension. The configuration tasks you need to carry out are:

- Register the query type
- Define query form labels
- Define enumeration field options (if required)
- Define custom look-up services (if required)

### 4.4.1 Registering a Query Type

If it does not already exist, create a file called *configuration-root*`/com/escenic/framework/datasource/DatasourceEditorConfig.properties` in one of your configuration layers and register your query type as follows:

```
query.number = query-definition
```

where:

- *number* is an integer used to distinguish multiple `query` entries from each other
- *query-definition* is the full path of the query type definition file within the query definition `.jar` file you deployed

You should add one such line for each query type you have deployed in the `.jar` file. So for the example `.jar` file structure shown in section 4.2, you should enter something like this:

```
query.1001 = /com/mycompany/datasource/query/customQuery1.xml
query.1002 = /com/mycompany/datasource/query/customQuery2.xml
query.1003 = /com/mycompany/datasource/query/customQuery3.xml
```

### 4.4.2 Defining Query Form Labels

Labels for your query type definition are defined in a properties file in the common configuration layer. Open *configuration-root*`/com/escenic/framework/ui/Datasource.properties` (or create it if it does not exist), and add label definitions as follows:

```
datasource.field.field-name.label=field-label
```

where:

- *field-name* is the name of a field in one or more of your query type definitions
- *field-label* is the label to be displayed next to this field

You can also define **placeholders** for fields in a similar way:

```
datasource.field.field-name.placeholder=placeholder-text
```

(A placeholder is the dummy text ("Please type your name", for example) sometimes displayed in a text field before the user has entered a value.

To define a label for the query itself, enter:

```
datasource.query.query-name.label=query-label
```

> If two of your query definitions contain fields with the same name, then they will share the same label/placeholder definitions in `Datasource.properties`. If you want them to have different labels, then you must give them different names.

### 4.4.3 Defining Enumeration Options

If any of your query type definitions contain enumeration fields, then you will need to define the options for the enumerations in a properties file in the common configuration layer. Open *configuration-root*`/com/escenic/framework/ui/Enumerations.properties` (or create it if it does not exist), and add option definitions as follows:

```
option.enum-name.order.value=enum-value
option.enum-name.order.label=enum-label
```

where:

- *enum-name* is the name of an enumeration field in one or more of your query type definitions
- *order* is an integer defining the option's position in the list of options. Real numbers containing decimal points are not allowed
- *enum-value* is the internal value of the option
- *enum-label* is the label displayed in the list of options

The label definition entry is optional. If you do not define a label for an option, then the value is used as a label.

For a field definition like this, for example:

```
<field name="sort" type="enumeration" enum="sortByTag" />
```

You might enter:

```
option.sortByTag.1.label=Time modified (ascending)
option.sortByTag.1.value=modified_date_asc
option.sortByTag.2.label=Time modified (descending)
option.sortByTag.2.value=modified_date_dec
option.sortByTag.3.label=Time published (ascending)
option.sortByTag.3.value=publish_date_asc
option.sortByTag.4.label=Time published (descending)
option.sortByTag.4.value=publish_date_dec
option.sortByTag.5.value=relevance
```

Note that since no label is defined for the last option in the above example, the value **`relevance`** will be used as a label.

You can define a default value for an enumeration as follows:

```
default.enum-name=default-value
```

For example:

```
default.sortByTag=modified_date_asc
```

> If two of your query definitions contain enumeration fields with the same name, then they will share the same option definitions in **`Enumerations.properties`**. If you want them to have different option definitions, then you must give them different names.

### 4.4.4 Defining Look-up Services

Data source query forms may contain a special type of field called a **`look-up`** field. A look-up field is one where the user can select a value by typing: as the user types, a list of matching options is displayed, from which the user can pick the required option. The look-up operations that provide this kind of functionality are performed by web services. A number of look-up services are included with the Widget Framework, and you can also create your own look-up services if required (see section 4.4.4.1).

The built-in look-up services provided with the Widget Framework are listed in a configuration layer component called **`/com/escenic/framework/webservice/LookupSources`** as follows:

```
$class=com.escenic.framework.webservice.helper.LookupSources
lookupService.group-names=/webservice/escenic/wf/collection/groups/search/
lookupService.content-types=/webservice/escenic/wf/collection/content-types/search/
lookupService.relation-groups=/webservice/escenic/wf/collection/relation-groups/
search/
lookupService.tags=/webservice/escenic/wf/collection/tags/search/{term}
entryService.tags=/webservice/escenic/wf/collection/tags/name/{value}
lookupService.sections=/webservice/escenic/wf/collection/sections/search/{term}
entryService.sections=/webservice/escenic/wf/collection/sections/name/{section}?
publication={publication}
```

These services are used by the default data source query types, but you can also use them in your own queries:

**`group-names`**
: This service returns information about all section page groups in all publications. Groups that appear in templates are not included.

**`content-types`**
This service returns information about all content types in all publications.

**`relation-groups`**
This service returns information about all relation type groups in all publications.

**`tags`**
This service returns information about tags from all tag structures.

**`sections`**
This service returns information about sections. It is used by the data source query section component (that is **`<component name="section"/>`**) in a query definition file.

If these services are insufficient and you need to create a look-up service of your own, then you can do so. For more about creating your own look-up service, see section 4.4.4.1.

To register your own look-up service with the Widget Framework, open *configuration-root*`/com/escenic/framework/webservice/LookupSources.properties` (or create it if it does not exist), and add a definition that looks like one of the following:

- ```
  lookupService.service-name=service-url/search/
  ```
- ```
  lookupService.service-name=service-url/search/{term}
  entryService.service-name=service-url/name/{value}
  ```

In both kinds of definition, *service-name* is the name that you will use to refer to the service in your query type configuration file and *service-url* is the URL of your service. Which kind of definition you should use depends on what kind of look-up service you have created. For further information, see section 4.4.4.1.

### 4.4.4.1 Creating Your Own Look-up Service

A data source query look-up service is a web service that responds to requests by returning JSON data in a specified format. There are two types of service:

- A **list service** that has a single end point and returns a list of values. The Widget Framework then carries out the actual look-up operation using the returned list.
- An **active service** that does the look-up itself and returns a list that matches the characters entered (so far) by the user. An active service has two end points, for performing different types of look-up.

A list service is simpler to implement and is fine if the collection of objects to look up is relatively small. For look-ups involving very large object collections, it is probably better to create an active service that does most of the work on the server.

#### Defining a list service

A list service has a single endpoint (for example: **`https://mycompany.com/webservice/recipes/search`**), and responds by returning a JSON file of the following form:

```
{"lookUpInfo":[
    {
      "value":"apple-pie-1",
      "label":"Apple Pie",
      "summary": "Just like Mom used to make it"
    },
```

```
    {
      "value":"apple-pie-2",
      "label":"Apple Pie with chili",
      "summary": "For the jaded palate"
    },
    ...
  ]}
```

The Widget Framework retrieves this list from the web service the first time it is needed in a Content Studio session, and caches it. It then uses the cached copy to perform the necessary look-ups when the Content Studio user starts typing in the query form look-up field. The values in the JSON file are used as follows:

**`value`**
: is the value that will actually be stored in the field

**`label`**
: is matched against the characters entered by the user and displayed in the field when the user finally makes a selection

**`summary`**
: is optional. If present, summaries are displayed below the labels in the list of options shown while the user is typing in the field

If the user opens a query form in which a look-up field already contains a value, then the Widget Framework uses the same list to look up the label it needs to display in the field.

### Defining an active service

An active service has two endpoints, and the last part of each endpoint URL is used as a look-up string. For example:

**`https://mycompany.com/webservice/recipes/search/{term}`**, and
**`https://mycompany.com/webservice/recipes/name/{value}`**

Each time the user types a character in the look-up field, the contents of the field are used to replace **`{term}`**, and the Widget Framework sends a **`GET`** request to the resulting URL. Every time the service receives such a request it is expected to use submitted term as a look-up string and return a JSON file. This file has the same structure as the JSON file returned by a list service, but instead of containing containing only entries where **`label`** matches the supplied characters. As the user types more characters, the JSON structure returned will therefore get shorter. The returned structure is used to display a list of options from which the user can pick the required value.

The second **`{value}`** endpoint is used when the user opens a query form in which a look-up field already contains a value. The Widget Framework then replaces **`{value}`** with the field value and sends a **`GET`** request to the resulting URL. The service is expected to respond by returning a JSON structure containing a single record, thereby providing the correct label to display in the field.

If, for example, the Widget Framework sent a request to **`https://mycompany.com/webservice/recipes/name/apple-pie-2`**, then the response might be:

```
{
  "value":"apple-pie-2",
  "label":"Apple Pie with chili",
  "summary": "For the jaded palate"
}
```

The Widget Framework API includes a LookUpInfoclass that you may find helpful in implementing an active lookup service in Java: it provides various useful methods for manipulating and comparing look-up service JSON structures. If you want to use this class then you should add the following dependency to your service's POM file:

```
<dependency>
  <groupId>com.escenic.widget-framework</groupId>
  <artifactId>wf-webservice</artifactId>
  <version>4.5.2-2</version>
  <classifier>lib</classifier>
</dependency>
```

## 4.5 datasource-query

The **datasource-query** schema defines the Widget Framework's query definition format. You can use it define the user interface of custom queries to be used by Widget Framework data sources. You can define the fields to be displayed in the user interface, their type and some aspects of their layout.

The query definition file does not control the execution of the query in any way: that must be implemented in a Java class. Nor does it define the labels displayed in the user interface, which must be defined in properties files added to one of the Widget Framework's configuration layers.

#### Namespace URI

The namespace URI of the **datasource-query** schema is **http://xmlns.escenic.com/2014/datasource-query**.

#### Root Element

The root of a **datasource-query** file must be a **query** element.

### 4.5.1 component

Includes a predefined user interface component in the query editor.

#### Syntax

```
<component
    name="(section|publishingPeriod)"
  />
```

#### Attributes

**name="(section|publishingPeriod)"**
The user interface component to be included.

Allowed values are:

**section**
This component is a standard section selector as used in the data source's built-in queries. It provides a group of 3 fields: **Section selection mode**, **Section** and **Include subsections**.

**`publishingPeriod`**
This component is a publishing period selector as used in the data source's built-in queries. It provides a group of 2 fields: **Start date** and **End date**.

### 4.5.2 container

Includes a container in the query editor. A container can be used to group related fields and control their layout.

#### Syntax

```
<container
    style="row"
    column-ratio="text"?
  >
  <field/>+
  <component/>*
</container>
```

#### Attributes

**`style="row"`**
The fields in the container will be arranged horizontally in a single row. This is currently the only available container style.

**`column-ratio="`*`text`*`"` (optional)**
Specifies the horizontal space to be allotted to each of the fields in the container. If there are three fields in the container then a value of `1:1:2` will allocated 25% of the available width to the first two fields, and 50% to the third.

If this attribute is omitted, then the available horizontal space is divided equally between the fields in the container.

### 4.5.3 editor

Contains a definition of the editor user interface for this query.

#### Syntax

```
<editor>
  <component/>*
  <container>...</container>*
  <field/>*
</editor>
```

### 4.5.4 field

Includes a field of the specified type in the query editor.

#### Syntax

```
<field
    name="text"
    inner="(true|false)"?
    ( type="(text|number|boolean)" | type="enumeration" enum="text" | type="lookup"
 source="text" multiple="(true|false)" )
```

```
    />
```

### Attributes

**`name="`*text*`"`**
The name of the field. The name you specify must be unique among the fields in this editor.

**`inner="(true|false)"` (optional)**
Specifies whether or not the field is a standard, predefined Widget Framework query field, or a custom field. This affects how the field contents are accessed in your Java handler class.

If you specify **`true`**, then the field name must exactly match one of the properties of the `com.escenic.framework.datasource.model.SourceDefinition` class, and will be accessible using its **`get`** method in the Java handler class. If you specify **`false`** or omit this attribute, then the field will be added to **`SourceDefinition`**'s **`Fields`** property, which contains a map of all custom fields. If you specify **`true`**, for a field that does not exactly match one of **`SourceDefinition`**'s properties, then an exception will be thrown at run-time.

For further information about this, see section 4.3.

**`type="(text|number|boolean)"`**
Specifies the type of value the field will accept.

Allowed values are:

**`text`**
The field will accept any text value.

**`number`**
The field will only accept numerical value.

**`boolean`**
The field will be displayed as a check box, only capable of accepting on/off values.

**`type="enumeration"`**
Specifies that the field will be displayed as a drop-down box containing a list of allowed values.

**`enum="`*text*`"`**
The name of an enumeration definition containing a list of options to display. The enumeration must be defined in a properties file (**`/com/escenic/framework/ui/Enumerations.properties`**) in the Widget Framework's global configuration layer. For further information, see section 4.4.3.

**`type="lookup"`**
Specifies that the field will be displayed as a "type-to-select" box where the characters the user enters are used to look up matching system objects from which he can then make a selection.

**`source="`*text*`"`**
The name of the web service that performs the look-up. This name must match a service definition listed in a properties file (**`/com/escenic/framework/webservice/LookupSources.properties`**) in the Widget Framework's global configuration layer. For further information, see section 4.4.4.

**`multiple="(true|false)"` (optional)**
Determines whether or not this lookup field allows multiple entries.

Allowed values are:

**`true`**
Multiple entries are allowed.

**false**
Multiple entries are not allowed. (Default)

### 4.5.5 query

The root element of a Widget Framework data source query definition....

#### Syntax

```
<query
    name="text"
  >
  <editor>...</editor>
</query>
```

#### Attributes

**name="*text*"**
The name of this query. It must be unique among all custom queries defined for this Widget Framework installation.

# 5 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.section.news
      config.default.section.sports
    config.default.article
      config.default.article.type.story
      config.default.article.type.picture
    config.default.topic
    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.section.news
      config.default.section.sports
    config.default.article
      config.default.article.type.story
      config.default.article.type.picture
    config.default.topic
    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.picture
    config.newsletter.topic
    config.newsletter.master
```

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

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 Templates, in the **Widget Framework User Guide**.

## 5.1 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=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.

## 5.2 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
```

## 5.3 Custom Templates

As well as adding complete content profiles under the **config** root section, you can also add custom templates. Here, for example, is a "special" template for sports pages:

```
config
  config.default
    ...
  config.special
    config.special.section
      config.special.section.sports
```

Custom templates are templates that editorial staff can explicitly select from Content Studio by setting a page option (see Story Layout Tab for details). They can be created in exactly the same way as content profile templates. The only difference is that you don't need to create a complete hierarchy of custom templates, you can just create the specific templates you require.

# 6 Adding a New Grid

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

## 6.1 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.

## 6.2 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>
```

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.

## 6.3 Adding grid styles in CSS

The next step is to add CSS entries for the new columns to a CSS file: For example, in **`theme.css`** file (found in **`/static/theme/default/base/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 {
```

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

## 6.4 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"/>
```

```
        <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>
```

## 6.5 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.

# 7 Themes

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

## 7.1 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>static/theme/my-theme/base/css/theme.css</resource>
   </transformer>
   ```

   The Maven Shade plug-in will then merge all the **`theme.css`** files in all the widgets into a single **`theme.css`** file.
6. Compile and deploy your code.

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 Changing Themes and Variants, in the **Widget Framework User Guide**.

## 7.2 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.

### 7.2.1 The Sass Filter

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

```
<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`**.

### 7.2.2 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.

### 7.2.3 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.

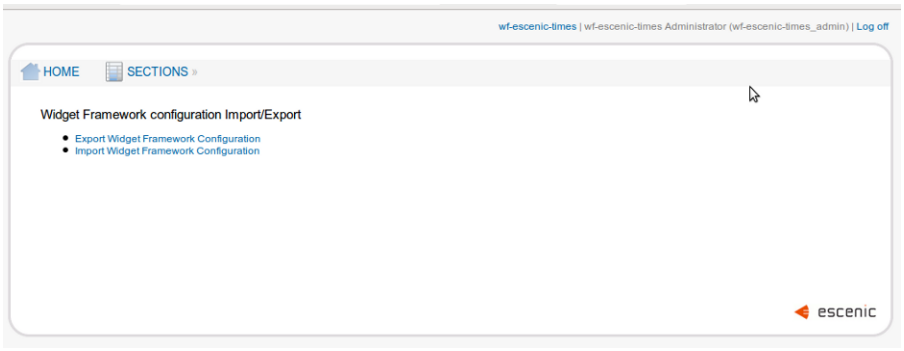
# 8 Export and Import

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.

## 8.1 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.

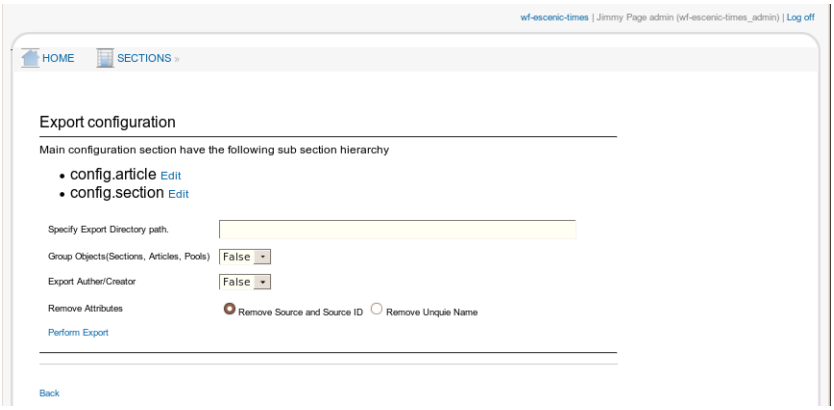
When the plug-in is installed you should get a new menu option in Web Studio. Here you have the options to export and import the configuration.




## 8.2 Syndication Export

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

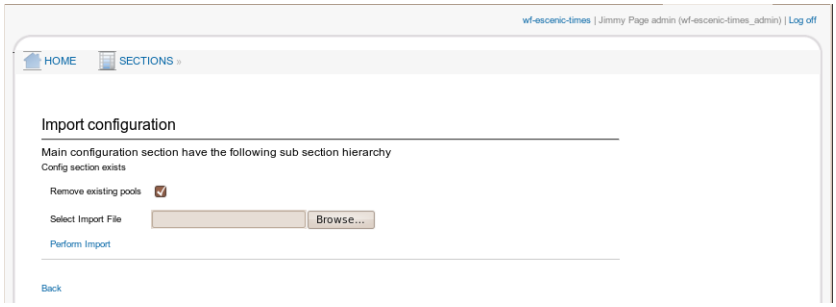
It will be exported to a specified location.



## 8.3 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.



# 9 How To

This chapter contains a variety of useful "recipes" for carrying out specific tasks.

## 9.1 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 templates-> **`/template/framework/group/processor/group-not-found.jsp`**
- Error page when nesting limit exceeds -> **`/template/framework/group/processor/nesting-limit-error.jsp`**

## 9.2 Write a Group Processor

A group processor is a Java class that can be used to process the contents of a template group: add content, filter content or modify it in some way. A group processor must

- Implement **`com.escenic.framework.processor.GroupProcessor`**
- Override **`com.escenic.framework.processor.GroupProcessor`**'s **`process`** method

The following example shows a group processor called **`MyDummyGroupProcessor`**:

```
package com.mycompany.group.processor;

import com.escenic.framework.processor.GroupProcessor;
import neo.xredsys.presentation.PresentationElement;
import javax.servlet.http.HttpServletRequest;
import java.util.Map;

public class MyDummyGroupProcessor implements GroupProcessor {
  protected static final String CUSTOM_ATTRIBUTE_NAME = "myProperty";
  @Override
  public void process(final PresentationElement pGroup,
                      final Map<String, Object> pGroupContext,
                      final HttpServletRequest pRequest) {

    //TODO: Process result and put into the group context
    pGroupContext.put(CUSTOM_ATTRIBUTE_NAME, "Your processed result here");
  }
```

```
}
```

In this case, all **`MyDummyGroupProcessor`** does is to create a dummy group property called **`myProperty`** which can be accessed from JSP templates with the following expression:

```
${requestScope.group.myProperty}
```

Before a group processor can be used it must be compiled and added to the classpath. And you must have **`template-common-4.5.2-2.jar`** in your classpath.

Once you have written a group processor class, you need to:

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

### 9.2.1 Configuration

To configure and register the group processor you must do the following:

1. Write a properties file for example, **`MyDummyGroupProcessor.properties`** and enter the following:

   ```
   $class=com.mycompany.group.processor.MyDummyGroupProcessor

   # Define the components here if you need any for your processor
   ```

2. Create a group descriptor file called **`DummyGroupDescriptor.properties`** and enter the following:

   ```
   $class=com.escenic.framework.descriptor.GroupDescriptor
   groupName=my-group-name

   groupProcessor.my-group-name.1=/com/escenic/framework/processor/
   MyDummyGroupProcessor
   ```

   where *my-group-name* is the name of the group.

   > You can specify more than one processor for a group. To add a second group processor, just add another entry like this:

   ```
   groupProcessor.my-group-name.2=/com/escenic/framework/processor/
   MyAnotherGroupProcessor
   ```

3. Now in order to register your group descriptor with the group descriptor factory, create a file called **`GroupDescriptorFactory.properties`**.
4. Enter the following in the file:

   ```
   $class=com.escenic.framework.descriptor.GroupDescriptorFactory
   groupDescriptor.my-group-name=./DummyGroupDescriptor
   ```

### 9.2.2 Packaging

The group processor class and all the properties files must be 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 processor class
- The Content Engine's package naming conventions

2. Pack it in a JAR file

#### Group Processor Package Structure

For the example processor **`MyDummyGroupProcessor`**, you would need to create a JAR file with the following structure:

```
com
  +-escenic
  |  +-servlet
  |   +-default-config
  |     +-com
  |       +-escenic
  |         +-framework
  |           +-descriptor
  |           |  +-DummyGroupDescriptor.properties
  |           |  +-GroupDescriptorFactory.properties
  |           +-processor
  |              +-MyDummyGroupProcessor.properties
  +-mycompany
    +-group
      +-processor
        +-MyDummyGroupProcessor.class
```

## 9.3 Use A Payment Solution

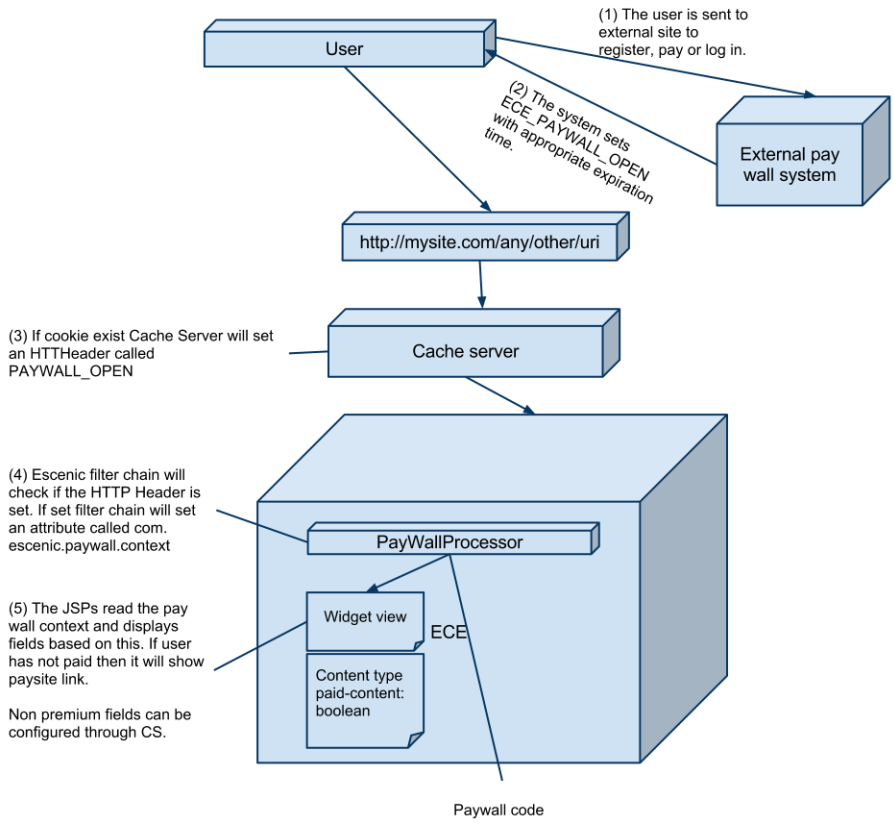
The Widget Framework's **`Story`** content type provides support for premium content. Content items of this type display a **Premium** tab in Content Studio where an editor can mark the content item as premium content. The title and selected other fields can be set to non-premium, making these fields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

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




## 9.4 Override a Core Content Type

A number of content types are included with the Widget Framework (**`Story`** and **`Picture`**, for example). These **core content types** are designed both to work well with the Widget Framework and to provide a flexible "starter pack" of content types that will support a wide range of publication types. Although the core content types satisfy most standard requirements, you may nevertheless feel the need to modify or extend one or more of them in some way.

Should you need modify a core content type, it is actually quite simple to do so. What you actually do is to **override** the content type within a specific publication: you copy the content type definition from the Widget Framework distribution into your publication's **`content-type`** resource, make the changes you require and then build and upload your publication to the Content Engine in the usual way. During the build process, the content type definitions are merged in such a way that your modified version overrides the standard version supplied with the Widget Framework.

The following procedure describes how to change a content type definition for the demo publication supplied with the Widget Framework. The process is, however, exactly the same for any other publication.

1. Unzip the Widget Framework distribution file **`widget-framework-core-4.5.2-2.zip`** somewhere on you computer.

2. In the **misc/widgets/widget-framework-core/src/main/webapp/META-INF/escenic/publication-resources/escenic/** of the unzipped folder tree you will find a **content-type** resource file. Open this file in an editor.
3. Find the content type definition you want to change (let's say the **story** content type).
4. Copy the content type definition.
5. In the **misc/demo/src/main/webapp/META-INF/escenic/publication-resources/escenic** folder of the unzipped folder tree you will find another **content-type** resource file. Open this file in an editor and paste in the copied content type definition.
6. Make whatever changes you require to the pasted content type definition. You might, for example, add a field to it.
7. Save your changes and close both **content-type** files.
8. In the **misc/demo/demo-site** folder, enter:

   ```
   $ mvn clean install
   ```

   This will produce a publication **.war** file in **misc/demo/demo-site/target** called **demo-core-4.5.2-2.war**.
9. Use the **escenic-admin** webapp to create a publication from the **.war** file you have generated or upload it to an existing publication.
10. Open the publication you have created/modified in Content Studio and create a **story** content item. You should be able to see the change you have made.

Note the following points:

- You only need to copy and modify the specific elements you want to change. Even though the **story** content type references lots of fields and field groups defined elsewhere in the **content-type** resource, you don't have to copy them as well. The file you copy into does not need to be a complete **content-type** resource because it's just a collection of overrides that are merged with the main **content-type** resource during the build process.
- You don't need to copy entire content type definitions either. If for example, you just want to add some constraints to a field definition, then you only need to copy the **field-group** the **field** belongs to, and modify the required **field**. Don't copy individual **field** elements though, since they don't have unique names and therefore cannot be merged correctly.

## 9.5 Override layout groups

The Widget Framework includes a predefined **layout-group** resource designed to work with the supplied core widgets and core content types. It contains **group** elements that define the logical structure of both publication section pages and section page templates.

You can modify the **layout-group** resource actually used by a publication in the same way as you modify the **content-type** resource: by adding overrides to a publication **layout-group** resource. This publication **layout-group** is then merged with the default **layout-group** when the publication is built. The procedure is just the same:

1. Unzip the Widget Framework distribution file **widget-framework-core-4.5.2-2.zip** somewhere on you computer.

2. In the **misc/widgets/widget-framework-core/src/main/webapp/META-INF/escenic/publication-resources/escenic/** of the unzipped folder tree you will find a **layout-group** resource file. Open this file in an editor.
3. Find the element you want to change (let's say the **topStories group**).
4. Copy the element.
5. In the **misc/demo/src/main/webapp/META-INF/escenic/publication-resources/escenic** folder of the unzipped folder tree you will find another (empty) **layout-group** resource file. Open this file in an editor and paste in the copied element.
6. Make whatever changes you require to the pasted element. You might, for example, add a **field** to its **ct:options** element.
7. Save your changes and close both **layout-group** files.
8. In the **misc/demo/demo-site** folder, enter:

   ```
   $ mvn clean install
   ```

   This will produce a publication **.war** file in **misc/demo/demo-site/target** called **demo-core-4.5.2-2.war**.
9. Use the **escenic-admin** webapp to create a publication from the **.war** file you have generated or upload it to an existing publication.
10. Open the publication you have created/modified in Content Studio and create a **story** content item. You should be able to see the option you have added to the content item's **topStories** group.

You should as a rule only create overrides that add elements to your publication's **layout-group** resource: removing or renaming elements is likely to break core widgets and/or content types.

The **layout-group** resource not only defines the structure of section pages and templates, it also determines where users are allowed to place widgets and content types on pages and templates. It prevents users placing widgets anywhere on ordinary section pages, for example. Each **area** element contains an **allow-content-types** element that in turn contains a list of **ref-content-type-group** referencing **content-type-group** elements. For example:

```
<area name="topStories-area">
  ...
  <allow-content-types>
    <ref-content-type-group name="contents"/>
    <ref-content-type-group name="custom-contents"/>
  </allow-content-types>
</area>
```

A **content-type-group** contains a list of **ref-content-type** elements that reference **content-type**s defined in the **content-type** resource. The supplied **layout-group** resource contains definitions of three **content-type-group**s:

**core-widgets**
: This group is for ordinary widgets such as the Teaser widget. Content types referenced in this group can be placed in any area of a template **except** the **Meta** area. They may not be placed anywhere on an ordinary section page.

**meta-widgets**
: This group is for meta widgets such as the Search Config widget. Content types referenced in this group can only be placed in the **Meta** area of a template. They may not be placed anywhere else.

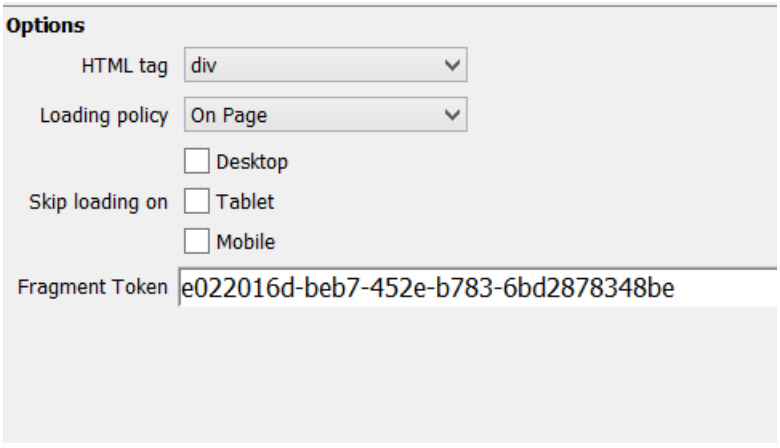
**contents**
: This group is for ordinary content types such as Story. Content types referenced in this group can only be placed on ordinary section pages.

The **area** elements in the **layout-group** resource contain references to three additional **content-type-group**s that are not defined: **custom-widgets**, **custom-meta-widgets** and **custom-contents**. If you have created custom widgets or content types of your own, then you can use these **content-type-group**s to control where they may be placed. You need to add definitions of the required groups along with other layout group overrides to your publication's **layout-group** resource. For example:

```
<groups xmlns="http://xmlns.escenic.com/2008/layout-group"
        xmlns:ct="http://xmlns.escenic.com/2008/content-type"
        xmlns:wf="http://xmlns.escenic.com/2014/wf-config"
        xmlns:ui="http://xmlns.escenic.com/2008/interface-hints">
  <!-- group definition for content section -->
  <content-type-group name="core-widgets">
    <ref-content-type name="widget_myCustomTeaser"/>
  </content-type-group>
  <content-type-group name="meta-widgets">
    <ref-content-type name="widget_MyCustomMetaWidget"/>
  </content-type-group>
  <content-type-group name="contents">
    <ref-content-type name="myCustomStory"/>
  </content-type-group>
</groups>
```

## 9.6 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 Lazy Loading, in 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>
  ...
</area>
```

## 9.7 Modify the URLs of Link and Binary Content Items

The Widget Framework provides two **decorators** that you can use to modify the URLs of certain types of content item.

Clicking on a link to a content item in an Escenic web site will normally result in the content item itself being displayed, and this is normally the desired result. In certain cases, however, it is not the desired result:

- In the case of a "link" content item that just holds a link to some external URL, for example, you would most probably want the click to lead straight to the external URL rather than to an intermediate page containing nothing but another link.
- Similarly, for a "binary" content item that just holds a reference to a PDF document, for example, you would most probably want the user's click to lead directly to the PDF document.

That is what these two decorators are for. To use them you have to add **`ui:decorator`** elements to the appropriate content type definitions in your publication's **`content-type`** resource, as described in the following sections.

For more information about decorators in Escenic, see Decorators, in the **Escenic Content Engine Advanced Developer Guide**.

### 9.7.1 binaryURLDecorator

The **binaryURLDecorator** can only be used with a content type that meets the following criteria:

- It has a field of type **link**
- The name of the **link** field must be **binary**

To use the **binaryURLDecorator**, add a **ui:decorator** element to the content type definition in your publication's **content-type** resource as follows:

```
<content-type name="mybinary">
  <panel ...>
    <field name="binary" type="link"/>
  </panel>
  ...
  <ui:decorator name="binaryURLDecorator"/>
  ...
</content-type>
```

The **binaryURLDecorator** will now modify the **url** property exposed by **mybinary** content items in the presentation layer. Instead of returning the content item's URL, the **url** property will return the value entered in the **link** field.

> **binaryURLDecorator** is used by the core content type **Document** that is supplied with the Widget Framework.

### 9.7.2 hyperlinkURLDecorator

The **hyperlinkURLDecorator** can only be used with a content type that meets the following criteria:

- It has a field of type **uri**
- The name of the **uri** field must be **url**

To use the **hyperlinkURLDecorator**, add a **ui:decorator** element to the content type definition in your publication's **content-type** resource as follows:

```
<content-type name="mylink">
  <panel ...>
    <field name="url" type="uri"/>
  </panel>
  ...
  <ui:decorator name="hyperlinkURLDecorator"/>
  ...
</content-type>
```

The **hyperlinkURLDecorator** will now modify the **url** property exposed by **mylink** content items in the presentation layer. Instead of returning the content item's URL, the **url** property will return the value entered in the **url** field.

> **hyperlinkURLDecorator** is used by the core content type **Hyperlink** that is supplied with the Widget Framework.

## 9.8 Configure Tag Page URLs

**Tag pages** are automatically generated pages created from tagged content items. When a user clicks on a tag keyword link, the corresponding tag page is displayed. The tag page usually contains a brief article about or definition of the subject of the tag, followed by teasers and links leading to tagged content items.

Tag page URLs have the following structure by default:

```
publication-url/tags/tag-structure-name/tag-term
```

The URL of the tag page for the term **bangladesh** in the **places** tag structure, for example, would be:

```
www.example.com/tags/places/bangladesh
```

The tag page URLs are generated by a component called **TagURLHelper**. You can configure this component to generate different tag page URLs by adding a **TagURLHelper.properties** file to your common configuration layer (**/etc/escenic/engine/common/com/escenic/framework/tag/TagURLHelper.properties**) and setting the following properties:

**path-prefix**
: This property determines the prefix placed after the publication URL and before the tag structure name. If, for example, you specified **path-prefix=tagged-as** then the example URL shown above would change to:

  ```
  www.example.com/tagged-as/places/bangladesh
  ```

**tagStructurePathPrefix.*tag-structure-name***
: These properties can be used to change the URL string used for specific tag structures. If, for example, you specified **tagStructurePathPrefix.places=geographical**, then the example URL shown above would change to:

  ```
  www.example.com/geographical/bangladesh
  ```

  You can enter several such properties in order to redefine the URL strings for multiple tag structures.

A sample **TagURLHelper.properties** is included in the Widget Framework distribution (**/misc/siteconfig/com/escenic/framework/tag/TagURLHelper.properties**).

### Including sections in tag page URLs

Prior to version 3, the tag pages generated by the Widget Framework (which were called topic pages) had URLs that included section components (**www.example.com/sport/topic/bangladesh**, for example). You can force the Widget Framework to continue generating these kinds of URLs by adding a **TopicProcessor.properties** file in publication's **WEB-INF/classes/com/escenic/servlet/plugin-config/com/escenic/framework/topic/TopicProcessor.properties** and setting the following property:

```
sectionSpecificTopicURIEnabled=true
```

A tag page for **bangladesh** in the **places** tag structure will then have a URL something like this:

```
www.example.com/sport/tags/places/bangladesh
```

## 9.9 Load static resources from a different location

By default, Widget Framework loads static resources (e.g. JS, CSS) from **static** folder inside publication. In order to load them from a different hosting location(e.g. CDN), you can configure the following section parameter:

```
wf.resource.static.rootUrl=http://cdn.site.com/static/publication/
```

## 9.10 Configure Analysis Engine

In order for a Widget Framework publication to be able to make use of the Escenic Analysis Engine plug-in, you need to set three section parameters in the publication's root section. The parameters must be set as follows:

```
eae.logger.url=http://hostname:port-number/analysis-logger/Logger
eae.queryservice.url=http://localhost:port-number/analysis-qs/QueryService
eae.logger.tag.enabled=true
```

where *hostname* is the name of the host and *port-number* is the number of the port on which the Analysis Engine's Logger component is installed and configured to run.

## 9.11 Expose Properties to JavaScript

**`/com/escenic/framework/frontend/FrontEndConfig`** is a special configuration layer object that you can use to create universally available properties. You can use it to define properties that are available in:

- The global Content Engine configuration layers
- The publication-specific application configuration layer
- Javascript code executed in the browser

The way this works is that:

- Any property defined in the global **`FrontEndConfig`** object is automatically copied to a similar object in the application layer
- Any property available in the application layer **`FrontEndConfig`** that has a name with the prefix "**`property.`**" is automatically made available to Javascript code running in the browser

This means that you can override the global setting of a **`FrontEndConfig`** property by redefining it in your application configuration layer. It also means that you can create publication-specific **`FrontEndConfig`** properties by defining them in the application configuration layer. They will then only be available in that publication.

To create global **`FrontEndConfig`** properties open *configuration-root*`/com/escenic/framework/frontend/FrontEndConfig.properties` in one of your global configuration layers (or create it if it does not exist) and add your property definitions:

```
myFirstProp=1
property.mySecondProp=2
```

If you do this in your common configuration layer (usually located in **/etc/escenic/engine/common**), the properties will be available in all the global configuration layers and also in all your publication applications. If, for some reason, you want **myFirstProp** to have a different value in one of your publications, then you can achieve this by creating another *configuration-root*/**com/escenic/framework/frontend/FrontEndConfig.properties** file in the publication's application configuration layer (that is, in the publication's **WEB-INF/localconfig** folder):

```
myFirstProp=1.5
```

Note that you only need to specify the property you want to override.

You can also create properties that are specific to this publication by adding them to this file:

```
myFirstProp=1.5
property.myFirstPublicationProp=one
```

**myFirstProp** doesn't have a "**property.**" prefix, and is therefore only available to server-side Java and JSP code. **property.mySecondProp** and **property.myFirstPublicationProp**, on the other hand, will also be available to client-side Javascript via a map variable called **FrontEndConfig**:

```
console.log ( 'mySecondProp=' + FrontEndConfig['mySecondProp']);
console.log ( 'myFirstPublicationProp=' + FrontEndConfig['myFirstPublicationProp']);
```

The value you assign to a **FrontEndConfig** property can be any of the following:

- Literal text, as shown previously:

  ```
  myFirstProp=1.5
  property.myFirstPublicationProp=one
  ```

- A JSON string:

  ```
  property.myBox= {"title": "Title","name": "Name", "width": 500, "height": 500}
  ```

  A property defined using JSON in this way will appear as a standard Javascript object on the client. You must use double quotation marks (") around field names and values.

- JSP expression language:

  ```
  property.myArticleTitle=el{article.title}
  ```

  Note that you have to use the keyword **el** to introduce the expression instead of the **$** sign usually used in JSP. Otherwise, you can use JSP expression language in exactly the same way as you would in your JSP code. You can reference any beans that would be accessible from server-side JSP code, and you can make use of expression language operators in the usual way. In addition, you can access section parameters as follows

  ```
  property.mySectionParam=el{sectionParams.section-parameter-name}
  ```

- A mixture of literal text and JSP expression language, for example:

  ```
  property.myContextArticleTitle=el{article.title} in el{section.name}
  ```

You can also combine JSON syntax and the JSP expression language to construct Javascript objects containing run-time values. For example:

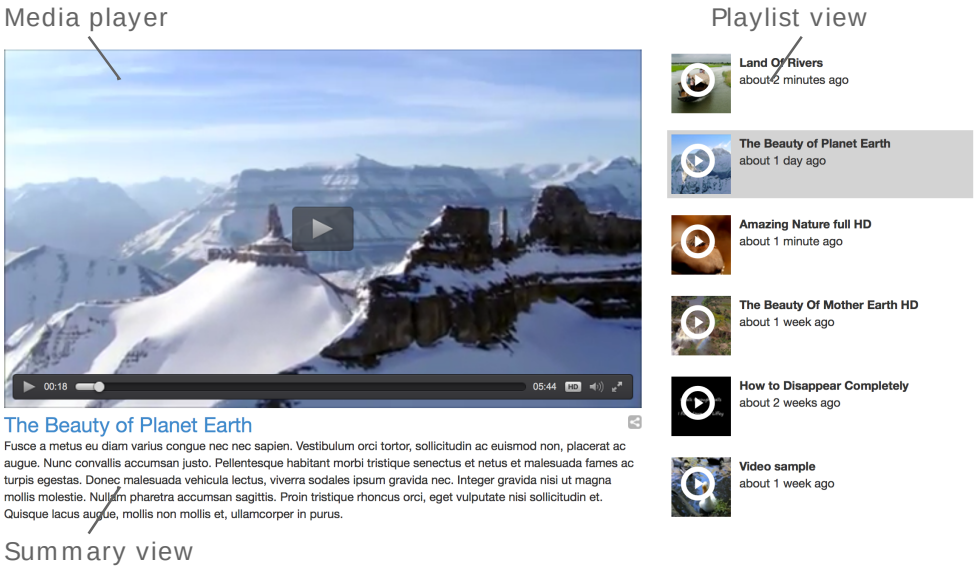
```
property.myArticle= {"title": "el{article.title}","section": "el{section.name}"}
```

For general information about application configuration layers and how they differ from the global configuration layers, see Publication Webapp Properties.

## 9.12 Control Content Visibility in a Gallery Group

The Gallery template group is specially designed to support the display of a combined media player and playlist on publication pages. Unlike other template groups, it can contain a Data Source (along with a Media widget and one or more Teaser/View Picker widgets. The Data Source returns a list of media content items for rendering in a playlist Teaser/View Picker widget. The user can then select media items from the playlist, which are played by the Media widget. For a full description of how a Gallery template group is used, see Define a Media Player and Playlist in the **User Guide**.

The Gallery group can contain more than one Teaser/View Picker widget: typically a second widget is used to render a summary for the currently selected media item so that a title and lead text can be displayed under or alongside the Media widget's video player as shown here:




Both Teaser widgets get their content from the same Data Source, but a summary widget needs to display only the currently selected content item. This can be achieved by making use of the **`wf-gallery-item`** and **`wf-gallery-selected-item`** CSS classes which are automatically assigned to Gallery content items by the Widget Framework. You need to include something like this in your publication CSS:

```
.media-summary .wf-gallery-item {
          display: none;
       }

.media-summary .wf-gallery-selected-item {
         display: block;
       }
```

and make sure that the **`media-summary`** class is added to your summary view widget's **Widget wrapper settings** / **Style Class** field (on the **Advanced** tab). (The class name **`media-summary`** is just an example, it is not supplied by the Widget Framework.)

## 9.13 Prevent the Display of Undecorated Media Content

The definitions of the Widget Framework's Video and Audio content types include a decorator declaration – either **`<ui:decorator name="audioArticleDecorator"/>`** for Audio content items or **`<ui:decorator name="videoArticleDecorator"/>`** for Video content items. These declarations mean that a Java component called a decorator is called during the rendering of the content items, if available. The **`audioArticleDecorator`** and **`videoArticleDecorator`** decorators specified in these declarations are provided by the Escenic Video plug-in, so if your Content Engine installation does not include the Video plug-in, the declarations are ignored.

However, if you do use the Video plug-in to manage your media content, then these decorators are an essential component of the rendering process. If, for any reason, the required decorator is unavailable or fails in some way during rendering, then the media item being published will not be rendered correctly and will not work.

In order to prevent such failures resulting in the display of defective media content on your site, you can set a switch that requires the decorator to be present and to execute successfully. If a failure occurs, then the audio or video content item affected will not be displayed at all rather than being displayed in a defective state.

> If several media items are published on the same page and one of them fails and is therefore not displayed, the other items on the page will not be affected and will still be displayed.

To set this switch you need to:

1. Create content type overrides for the Video and Audio content types, as described in section 9.4.
2. Add a **`fail-on-error="true"`** attribute to each of the media content types' decorator declarations. For the Video content type, for example:

   ```
   <ui:decorator name="videoArticleDecorator" fail-on-error="true"/>
   ```

3. Save your changes.

> Note that this change will not have any effect unless your media content is published via the Video plug-in.

## 9.14 Configure the Content Field Widget

The Content Field widget renders one selected field of the current content item. A publication designer determines which field the widget is to render by setting the widget's **Field** option. The **Field** option is a collection field that offers the user a list of possible fields to choose from.

By default, the list contains all the fields in the **`default`** panel of the Story content type. This is sufficient for many purposes, since the **`default`** panels of the other content types supplied with the Widget Framework mostly have a subset of the fields available in the Story content type's **`default`** panel. Some content types, however, have fields in their **`default`** panel that don't appear in the Story **`default`** panel. You may also have created your own content types that have fields with other names in the **`default`** panel.

If you want to be able to render these additional fields using the Content Field widget, then you need to ensure that they appear in the list of options displayed by the Content Field widget's **Field** option.

You can add the fields of content types other than Story to the Content Field widget's **Field** option by adding a **widget.contentField.contentType** section parameter to the root section of one (or more) of the site publications at your installation. The value of the parameter should be a comma-separated list of content-type names. For example:

```
widget.contentField.contentType=myImprovedStory,myEvenBetterStory
```

Field names from all of the content types you specify will then be added to the **Field** option's list. You only need to add the **widget.contentField.contentType** section parameter to one of your publications as its effect is global. You can add it to several publications, but you can't control its effect by doing so. All your publications are searched for the presence of this parameter, and fields from all the content types you specify in the various publications are added to the **Field** option's list.

Please note that:

- You can only add the **widget.contentField.contentType** section parameter to a site publication – not to a blueprint. Adding this parameter to a blueprint has no effect.
- The content type names you add to **widget.contentField.contentType** must be the actual **name** attributes as specified in the **content-type** resource, not the labels displayed in Content Studio.
- If you are wondering which Story panel is the **default** panel, it's the one labeled **General** in Content Studio. All content types have a panel with the internal name **default**, but some of them have a different label.
- You can only add fields from a content item's **default** panels. Fields on other panels cannot be rendered by the Content Field widget.

## 9.15 Split a Legacy Publication into Site Publication and Blueprint

A new Widget Framework project structure was introduced in version 3.6 in order to cleanly separate content (site publications) from site design (blueprints). Prior to this, both templates and content were held together in a single "all-in-one" publication. This legacy publication structure is still supported, but you may wish to convert your legacy publications to the new structure in order to simplify future maintenance.

If your publication includes any custom widgets, then before you can split the publication, you first need to restructure your custom widget Maven projects. Each project must be modified to specify **widget-framework-widgets** as its parent project. This ensures that when the project is built it will generate three artifacts instead of one: one artifact containing the design-related aspects of the widget, for use in blueprints, one containing the editorial aspects of the widget for use in site publications, and one containing everything for use in legacy publications. For full details, see section 3.11.

Once you have restructured any custom widgets, to split a legacy publication you need to do the following:

1. Export the legacy publication's **config** section and all its subsections (see Export Pubication Content). Export all content items from the section tree, and export section pages along with the sections.
2. Apply the XSL transformation supplied in section 9.15.1 to the exported XML files. This transformation is needed to ensure that section collection fields in some widget types (Search Component and Heading widgets, for example) are converted properly.

3. Set up a new Maven project for the blueprint. The project must include dependencies on:
   - Any layout-related custom content types or **`layout-group`** definitions used in the original legacy publication
   - **`widget-framework-blueprint`** and **`widget-pack-blueprint`** (see section 11.1.2)
   - If your publication includes any custom widgets, the widgets' blueprint-specific dependencies (see Using Custom Widgets in Blueprints)
4. Build the new project.
5. Create a new publication using **`escenic-admin`** (see New Publications. Make sure that you set **Publication type** to **configuration**.
6. Import the content you exported in step 1 to the new blueprint publication (see Import Content.
7. Convert your legacy publication's Maven project to a site publication project. This involves:
   - Removing dependencies on the layout-related custom content-types and **`layout-group`** definitions you copied to the blueprint project. The site project should only retain dependencies on editorial level custom content types and **`layout-group`** definitions.
   - Removing the dependency on **`wf-demo-resources`**, which is no longer required.
   - Replacing the existing Widget Framework dependencies with dependencies on **`widget-framework-site`** and **`widget-pack-site`** (see section 11.1.1).
   - If your publication includes any custom widgets, replacing the existing custom widget dependencies with site publication dependencies (see Using Custom Widgets in Site Publications)
8. Build and deploy the modified project.
9. Update the publication resources of your legacy publication, thereby converting it into a new-style site publication (see Update Resources).
10. Set up the connection between the new site publication and its blueprint (see Set Publication Blueprint).

### 9.15.1 Converting Section Collection Fields

Section collection fields in certain widgets and groups will fail to work after conversion unless a small modification is made after export from the legacy publication. The easiest way to ensure that this modification is made in all the required places is to apply the XSL transformation **`migrate-to-3.6.xsl`** to all syndication files exported from the legacy publication before they are imported to the new blueprint publication. You will find **`migrate-to-3.6.xsl`** in the **`misc/migrate/`** folder of the Widget Framework distribution.

If you are upgrading to Widget Framework **3.7.1 or higher**, then you also need to apply the XSL transformation **`migrate-to-3.7.1.xsl`** to all your exported syndication files before you import them to the new blueprint publication. First apply **`migrate-to-3.6.xsl`**, then apply **`migrate-to-3.7.1.xsl`**.

> Note that if you have already upgraded to 3.6 and are now upgrading to 3.7.1 or higher, then you will need to apply the **`migrate-to-3.7.1.xsl`** transformation to your blueprints.

# 10 Configuring Components

The Widget Framework makes use of various third party components, some of which require configuration. This chapter provides some guidance on how they should be configured.

## 10.1 reCAPTCHA

**reCAPTCHA** is a free CAPTCHA service that helps to digitize books, newspapers and old time radio shows. This service is provided by Google. Widget Framework uses **reCAPTCHA** in the **`comments`** widget and **`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. The 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.

## 10.2 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 its **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. It is also important to make sure that the Solr server itself is correctly configured to work well with the Widget Framework.

A Solr schema for defining the content Solr is to index is included with the Widget Framework, and it is preconfigured to provide the basic search functionality required by the Widget Framework. You may, however, need to add some additional configurations to the schema in order to support specific Widget Framework features.

### 10.2.1 Solr Configuration

The Widget Framework uses Solr for a variety of look-up operations, and in a production environment it will exert a heavy load on Solr, especially if your Data Sources use many Solr-based query types such as Query by section, Query by tag and Query by search. You are therefore strongly recommended

to enable Solr response caching. This will significantly reduce the load on Solr and improve its performance.

When configuring Solr you need to ensure:

- That the Solr caches are large enough
- That Solr caches search results for a suitable length of time
- That Solr adds the correct HTTP caching headers to its search query responses

Below are some suggested settings for the Solr configuration file (**`solrconfig.xml`**) that you can start with, which should give reasonable results. If you want to optimize your installation further, then you should consult the Solr documentation.

- ```
  <autoCommit>
    <maxDocs>100</maxDocs>
    <maxTime>300000</maxTime>
  </autoCommit>
  ```

  This tells Solr to cache up to 100 query results for a period of 5 minutes.

- ```
  <query>
    ...
    <queryResultCache
      class="solr.LRUCache"
      size="2048"
      initialSize="512"
      autowarmCount="0"/>

    <documentCache
      class="solr.LRUCache"
      size="20000"
      initialSize="1024"
      autowarmCount="0"/>
    ...
  </query>
  ```

  This sets Solr's internal caches to a suitable size for holding the required number of results.

- ```
  <requestDispatcher handleSelect="true">
    ...
    <httpCaching lastModFrom="openTime" etagSeed="Solr">
      <cacheControl>max-age=30, public</cacheControl>
    </httpCaching>
    ...
  </requestDispatcher>
  ```

  This tells the Widget Framework to cache supplied results for 30 seconds.

With these settings in place, the first time a particular query is submitted to Solr, it will perform the search, save the results in its cache and also return the results to the Widget Framework with an HTTP header containing an instruction to cache the result for 30 seconds. If the same request is repeated within 30 seconds, then the results will be returned from the Widget Framework's cache. If the same request is repeated after 30 seconds have elapsed, then the Widget Framework will again submit a request to Solr, but with an HTTP caching header asking for the response in its cache to be revalidated. If the result is still in the Solr cache, then Solr will return an **`HTTP 304 Not Modified`** response (since this is more efficient than returning the cached response again). The Widget Framework can then use the response in its own cache and revalidate it for another 30 seconds. If the Solr cache has
  ```

expired in the mean time then Solr will perform the requested search again, cache the results and return them with the same caching header.

How much effect this configuration has on Solr performance will obviously depend on usage in your particular installation, and achieving optimum results may require some adjustment. For more information about Solr caching, see the Solr documentation.

### 10.2.2 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.

> Make sure that **enableHttpClientCache** is set to **true** in order to enable caching. This is in fact the default setting in Widget Framework 3.2 and later versions - just make sure you don't override it, otherwise the Widget Framework will be unable to obey the caching instructions included in the HTTP headers returned with Solr results. In earlier versions of the Widget Framework, **enableHttpClientCache** is set to **false** by default, so you must explicitly set it.

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=true
# Sets the maximum number of cache entries the cache will retain.
# maxCacheEntries=3000
# Sets the maximum size of response that will be cached
# maxCacheObjectSize=16384
# max connection per route
# defaultMaxPerRoute=10
# total number of connection
# totalConnection=10
# caching backend storage scheme, for example - /com/escenic/framework/search/solr/
MemcachedStorage
# cacheStorage=/path/to/caching/backend/storage/component
```

#### 10.2.2.1 Search Client Cache Configuration

If you enable caching and don't set the **cacheStorage** property, then search results are cached in local memory by the Content Engine. You can, however, configure the Widget Framework to use an external cache manager by setting the **cacheStorage** property in **HttpClientFactory.properties**. This property must be set to reference a storage component that communicates with the external cache manager. The referenced storage component must also be configured by means of a **.properties** file.

An alternative back end storage component for Memcached, called **MemcachedStorage**, is supplied with the Widget Framework. This component lets you use Memcached (a distributed cache manager) instead of the default local cache. A sample configuration file for this component is provided in **/misc/siteconfig/com/escenic/framework/search/solr/MemcachedStorage.properties**.

To use the **MemcachedStorage** component instead of the default storage scheme, set the **cacheStorage** property in your **HttpClientFactory.properties** file as follows:

```
cacheStorage=/com/escenic/framework/search/solr/MemcachedStorage
```

Then copy **/misc/siteconfig/com/escenic/framework/search/solr/MemcachedStorage.properties** into your configuration layer and set its **servers** property to point to your Memcached servers:

```
# Fill in the Memcached server addresses here
servers=server1:port,server2:port
```

As long as the Memcached servers are correctly configured and working, they will then be used for caching search results.

### 10.2.3 Solr Server Configuration

A default configuration for Solr is included with the Widget Framework. The supplied configuration is:

```
$class=com.escenic.framework.search.solr.SolrSearchEngine
httpClientFactory=./HttpClientFactory
solrServerURI=${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"/>
```

### 10.2.4 Solr Schema Modifications

Some Widget Framework features require small additions to the Solr schema file, which is usually located at **/var/lib/escenic/solr/***ece-instance***/conf/schema.xml**. The required additions are described in the following sections.

> If you make any changes to the Solr schema file, then you will need to regenerate the index. For information on how to do this, see Re-indexing.

#### 10.2.4.1 Priority Sorting Configuration

To enable priority sorting of story content items returned by a **data source** search, the **priority** field of the **story** content-type needs to indexed by Solr. To achieve this you need to add the following XML fragment to **schema.xml**:

```
<field name="priority_enum" type="string" indexed="true" stored="true"/>
```

#### 10.2.4.2 Editorial Layout Control Configuration

To enable editorial teaser layout control (see Editorial Layout Control), you need to add the following XML fragment to **`schema.xml`**:

```
<field name="editorialselection_b" type="boolean" indexed="true" stored="false"/>
```

#### 10.2.4.3 Query by Author Configuration

To enable Data Source **Query by author** (see http://docs.escenic.com/widget-core-reference/4.5/query_by_author.html), and to be able to search for authors by username, you need to add the following XML fragment to **`schema.xml`**:

```
<field name="username_text" type="string" indexed="true" stored="true"/>
<field name="author_username_s" type="string" indexed="true" stored="false"
 multiValued="true" />
```

## 10.3 Google Analytics

Google Analytics is a service offered by Google that monitors web site traffic and generates detailed statistics about the traffic, traffic sources, conversions and sales.

To monitor publications using Google Analytics, you must first set up a Google Analytics account, and obtain the account's **tracking ID**.

All you then need to do to set up monitoring for a publication is to add the following section parameter to the publication's root section:

```
wf.ga.tracker.id=tracking-id
```

where *tracking-id* is your Google Analytics account tracking ID.

You can optionally override the default behavior of Google Analytics by providing a custom tracker object configuration in an additional section parameter:

```
wf.ga.tracker.config=configuration
```

where *configuration* is a JSON data structure something like this:

```
{"cookieDomain": "foo.example.com",
"cookieName": "myNewName",
"cookieExpires": 20000}
```

> You must use **double quotes** around the key names and string values in the JSON data you specify, not single quotes.

For further information, see the Google Analytics documentation here. If you do not specify this parameter then Google Analytics' default **`auto`** configuration is used.

Once you have made these changes to your site, Google Analytics will start tracking all site visitors and recording statistics about their activities. You can then use the analytical tools on the Google Analytics web site to discover all kinds of information about your site users and their activities on your site.

For more details visit http://www.google.com/analytics/.

## 10.4 Media Players

The Widget Framework currently supports three players for audio/video playback: JW Player, Flowplayer and MediaElement.js. The players can be used as follows:

- **JW Player** for audio, video and live streams
- **Flowplayer** for video only
- **MediaElement.js** for audio only

Audio/video playback using any of these players is supported by:

Media widget
: Can be used to play audio, video and live streams using the embedded player on content item pages.

Teaser widget
: Can be used to play audio, video and live streams in several modes (embedded, pop-up and so on) on both content item pages and section pages.

Teaser view
: Can be used to play audio, video and live streams in several modes (embedded, pop-up and so on) on both content item pages and section pages.

Most of the media player configuration parameters are set in a configuration layer object called **`FrontEndConfig`**. This is a special configuration layer object that is able to expose its properties both in publications' application configuration layers and also in the browser, where they can be accessed using Javascript. For more information about this, see section 9.11.

> Not that whichever media player you use, video playback will not work unless you have added a **`MediaInfoServlet`** servlet declaration to your publication's **`web.xml`** file. See section 2.13 for details.

### Player selection

You specify the default player(s) you want to use by adding the following properties to a properties file in the common configuration layer. Open *configuration-root*`/com/escenic/framework/frontend/FrontEndConfig.properties` (or create if it does not exist) and add the following properties:

```
property.videoPlayer=video-player
property.audioPlayer=audio-player
```

where:

***video-player***
: Is set to either **`jwplayer`** or **`flowplayer`**. The default value is **`flowplayer`**.

***audio-player***
: Is set to either **`jwplayer`** or **`mediaelement`**. The default value is **`mediaelement`**.

Specifying these properties in the common configuration layer sets the default media players for all your publications. If you need to use different media players in different publications, you can achieve this by specifying the properties in your publication configuration layers instead of (or as well as) in

the common configuration layer. (See section 9.11 for details of how to create publication configuration layers.)

It is also possible to override these settings by creating custom media player configurations for use on specific pages or with specific widgets. For details, see section 10.4.4.

### 10.4.1 Configuring Flowplayer

The only thing you need to do to use Flowplayer for video playback is to add the following property to *configuration-root*`/com/escenic/framework/frontend/FrontEndConfig.properties`:

```
property.videoPlayer=flowplayer
```

The following features are available using Flowplayer:

- Basic video playback
- Subtitles/closed captioning
- Cue points

For further information about Flowplayer configuration options, see https://flowplayer.org/docs/setup.html#configuration.

### 10.4.2 Configuring MediaElement.js

The only thing you need to do to use MediaElement.js for audio playback is to add the following property to *configuration-root*`/com/escenic/framework/frontend/FrontEndConfig.properties`:

```
property.audioPlayer=mediaelement
```

For further information about MediaElement.js configuration options, see http://mediaelementjs.com/#api.

### 10.4.3 Configuring JW Player

To use JW Player for both video and audio playback, add the following properties to *configuration-root*`/com/escenic/framework/frontend/FrontEndConfig.properties`:

```
property.videoPlayer=jwplayer
property.audioPlayer=jwplayer
property.media.jwplayer.script.location=script-location
property.media.jwplayer.key=license-key
```

where:

***script-location***
: is the URL of the JW Player javascript file. You can use JW player in two different ways: self-hosted, where you use your own copy of the script, or cloud-hosted, where you use JW Player's copy directly.

***license-key***
: You only need to set this property if you have a self-hosted JW Player.

> Cloud-hosted installations may be affected by cross-domain security restrictions. If JW Player needs to access a file that is hosted on a different domain from itself (a VTT file, for example), then you will need to configure the JW Player to allow Cross-domain Resource Loading.

The following features are available using JW Player:

- Basic video playback
- Video quality selection
- Subtitles/closed captioning
- Cue points
- Pre-roll ads
- Social sharing
- In-line player
- Google Analytics integration
- HLS adaptive streaming
- Live streaming

> There are several different editions of JW Player. Some of the above features require the purchase of a Premium or Enterprise edition license.

Some of the additional features offered by JW Player require configuration as well:

- Social sharing (section 10.4.3.1)
- Google Analytics integration (section 10.4.3.2)
- HLS Adaptive Streaming (section 10.4.3.3)

JW Player is highly configurable and offers a large number of options that you can set by adding properties to the **`FrontEndConfig.properties`** file. For details see section 10.4.3.4.

For information on how to configure the Widget Framework to provide JW Player with pre-roll ads, see Configure Video Advertising Services.

#### 10.4.3.1 Social Sharing

JW Player provides a built-in social sharing feature that overlays "share" buttons for popular social media services on the video. The feature is enabled by default, but you can disable it by adding a property to *configuration-root*`/com/escenic/framework/frontend/FrontEndConfig.properties`:

```
property.media.jwplayer.sharing.enabled=false
```

#### 10.4.3.2 Google Analytics Integration

JW Player incorporates a Google Analytics reporting feature. It will, however, only work only if you have a Google Analytics account configured to monitor your site (see section 10.3 for further information about this). You can disable the feature by adding a property to *configuration-root*`/com/escenic/framework/frontend/FrontEndConfig.properties`:

```
property.media.jwplayer.ga.enabled=false
```

#### 10.4.3.3 HLS Adaptive Streaming

If multiple stream qualities are available in the M3U8 manifest provided as source, JW Player automatically starts playing the highest quality stream that fits both the available bandwidth and the screen size. JW Player automatically adjusts the stream if the bandwidth increases or decreases during playback. This feature is enabled by default, but you can disable it by setting a property in *configuration-root*`/com/escenic/framework/frontend/FrontEndConfig.properties`:

```
property.media.jwplayer.hls.enabled=false
```

By default, JW Player disables HLS streaming for Android devices. You can enable it for devices running Android 4.1 and higher by setting a property in *configuration-root*`/com/escenic/framework/frontend/FrontEndConfig.properties`:

```
property.media.jwplayer.options.androidhls=true
```

#### 10.4.3.4 General JW Player Setup Options

You can modify the look and feel of JW Player using setup options provided by JW Player. In order to set these options, you have to add properties to *configuration-root*`/com/escenic/framework/frontend/FrontEndConfig.properties`. You can set any JW Player setup option by adding a property to `FrontEndConfig.properties` that has exactly the same name plus the prefix `property.media.jwplayer.options`:

```
property.media.jwplayer.options.option-name=option-value
```

For example:

```
# set the JW Player 'primary' option to 'flash'
property.media.jwplayer.options.primary=flash

# set the JW Player 'captions.color' and 'captions.fontOpacity' options
property.media.jwplayer.options.captions.fontOpacity=80
property.media.jwplayer.options.captions.color=#CCCCCC
```

Some JW Player setup options are provided with default values by the Widget Framework, for example:

- The `sharing.heading` option is assigned the video title
- The `sharing.link` option is assigned the video URL

You can, however, override these default settings in the same way, by adding properties to `FrontEndConfig.properties`:

```
# Override JW Player sharing heading and link options
property.media.jwplayer.options.sharing.heading=your-custom-heading
property.media.jwplayer.options.sharing.link=your-custom-url
```

### 10.4.4 Custom Media Player Configurations

Should you need to use different media player configurations within the same publication, you can do so by creating **custom media player configurations**. A custom media player configuration is a named media player configuration that you define in Javascript code. Once you have created such a configuration, you can force a Media or Teaser widget (or a Teaser view) to use your custom configuration instead of the default configuration by setting a widget property. This allows you to use

any media player configuration anywhere you want. You can, for example, use several different media player configurations in different locations on the same page.

You can create as many different media player configurations as you want.

#### 10.4.4.1 Defining a Custom Media Configuration

To create a custom media configuration you need to include some Javascript code in your publication (for details of how to do this, see section 3.8). The code required varies depending on what type of media player you want to configure.

**Flowplayer**

```
WFPlayerPlugins.registerPlayerPlugin({
    name : 'configuration-name',
    type : WFMedia.type.video,
    playerPlugin : FlowplayerPluginFactory({
        your-configuration
    })
});
```

where:

***configuration-name***
is the name of your custom configuration.

***your-configuration***
is a JSON object containing your custom configuration parameters. For details of the structure of this object and the values it can contain, see https://flowplayer.org/docs/setup.html#configuration. Note that: video title, google analytics configuration, posterImageURL is not configurable using the custom configuration.

**MediaElement.js**

```
WFPlayerPlugins.registerPlayerPlugin({
    name : 'configuration-name',
    type : WFMedia.type.audio,
    playerPlugin : MediaElementPluginFactory({
        your-configuration
    })
});
```

where:

***configuration-name***
is the name of your custom configuration.

***your-configuration***
is a JSON object containing your custom configuration parameters. For details of the structure of this object and the values it can contain, see http://mediaelementjs.com/#api.

**JWPlayer**

```
WFPlayerPlugins.registerPlayerPlugin({
    name : 'configuration-name',
    type : WFMedia.type.video,
    playerPlugin : JWPlayerPluginFactory({
```

```
        your-configuration
    })
});
```

where:

***configuration-name***
is the name of your custom configuration.

***your-configuration***
is a JSON object containing your custom configuration parameters. For details of the structure of this object and the values it can contain, see setup options.

**Example**

The following example shows a configuration called **`custom-jwplayer-instance`** that sets up JWPlayer in Flash mode with a width of 50%, a specified Flash file location and a companion ad banner:

```
WFPlayerPlugins.registerPlayerPlugin({
    name : 'custom-jwplayer-instance',
    type : WFMedia.type.video,
    playerPlugin : JWPlayerPluginFactory({
       width: '50%',
       flashplayer:'location/to/flash/file.swf',
       adConfig : {
           comapaniondiv : {
               id : "companionBanner",
               width : 300,
               height : 250
           }
       }
    })
});
```

### 10.4.4.2 Using a Custom Media Configuration

Once you have defined a custom media configuration, you can use it by adding property settings to one of the following widget/view types:

- Media widget
- Teaser widget
- Teaser view

The property settings you need to add are:

```
element.media.videoPlayer=video-configuration-name
element.media.audioPlayer=audio-configuration-name
```

where:

***video-configuration-name***
is the name of a custom video configuration you have created (that is a JWPlayer or Flowplayer configuration)

***audio-configuration-name***
is the name of a custom audio configuration you have created (that is a JWPlayer or MediaElement.js configuration)

Most widget properties are exposed as widget fields in Content Studio. This is not the case for **`element.media.videoPlayer`** and **`element.media.audioPlayer`**. These properties must be manually set. To do so:

1. Open the widget in Content Studio.
2. Click on the **Advanced** tab.
3. Add your property setting(s) to the list of settings in the **Widget properties** field.
4. Publish your changes.

The custom configuration you have specified will now override the default media configuration for that widget, wherever you use it.

# 11 The Publication Build Tool

The Widget Framework includes a **publication build tool** for building your publication WAR files. The publication build tool is actually a collection of various standard and custom components that you can use together with Maven to build publications efficiently and reliably.

The tools are based around the Maven Shade plug-in, a plug-in designed to simplify the creating of **über jars**. An über jar is a Maven package that includes all its own dependencies. In this case the Shade plug-in is used to build a publication WAR file that includes the publication itself plus the Widget Framework and all the widgets the publication depends on.

In addition to the Shade plug-in itself, the publication build tool includes several Shade plug-in resource transformers, both standard transformers and custom transformers created specifically for the Widget Framework. A resource transformer is a component that helps manage the process of merging the contents of the JAR files to be included in an über jar.

The publication build tool also includes its own custom Maven plug-in, **`wf-build-plugin`**, that can be used to modify XML files (publication resources, for example) during the build process.

## 11.1 About The Build Tool

A ready-to-use publication build tool configuration is provided in the default parent project supplied with the Widget Framework: **`com.escenic.widget-framework:wf-project-core`**. All standard publications should use this project as their parent project - they then inherit its publication build tool configuration, and for simple projects no further configuration is required.

The simplest way to set up correctly configured publications is simply to copy and modify the demo publications supplied with the Widget Framework in the **`misc/demo`** folder. If you look at the demo publication's POM file, **`misc/demo/pom.xml`**, you will see that it's parent project is specified as **`com.escenic.widget-framework:wf-project-core`**:

```
<parent>
  <groupId>com.escenic.widget-framework</groupId>
  <artifactId>wf-project-core</artifactId>
  <version>4.5.2-2</version>
  <relativePath></relativePath>
</parent>
```

The demo project contains both a site publication and a blueprint that you can use as the basis for your own sites and blueprints. The project POM file, **`misc/demo/pom.xml`** references two sub-modules, **`demo-site`** and **`demo-blueprint`**, located in the **`misc/demo/demo-site`** and **`misc/demo/demo-blueprint`** folders.

### 11.1.1 Setting up a Site Publication

The demo site publication's POM file, **`misc/demo/demo-site/pom.xml`** declares the following dependencies:

```
<dependency>
  <groupId>com.escenic.widget-framework</groupId>
```

```
    <artifactId>widget-framework-site</artifactId>
    <version>4.5.2-2</version>
    <type>pom</type>
  </dependency>

  <dependency>
    <groupId>com.escenic.widget-framework</groupId>
    <artifactId>widget-pack-site</artifactId>
    <version>4.5.2-2</version>
    <type>pom</type>
  </dependency>
```

These dependencies ensure a site publication has access to all the resources needed to render the publication correctly in the browser. Template design and configuration resources are not included, since they are not required for a site publication.

### 11.1.2 Setting up a Blueprint Publication

A blueprint publication needs to be set up with all the publication resources relevant to template design and configuration. A blueprint usually contains widgets, data sources, templates, etc. It is not recommended to deploy a blueprint, but use as a container of all the templates required to render site publications. Once a blueprint is set up properly, it can be used to manage as many site publications as required. The blueprint should have the following dependencies as demonstrated in **`misc/demo/demo-blueprint/pom.xml`**:

The demo blueprint's POM file, **`misc/demo/demo-blueprint/pom.xml`** declares the following dependencies:

```
  <dependency>
    <groupId>com.escenic.widget-framework</groupId>
    <artifactId>widget-framework-blueprint</artifactId>
    <version>4.5.2-2</version>
    <type>pom</type>
  </dependency>

  <dependency>
    <groupId>com.escenic.widget-framework</groupId>
    <artifactId>widget-pack-blueprint</artifactId>
    <version>4.5.2-2</version>
    <type>pom</type>
  </dependency>
```

These dependencies ensure a blueprint has access to all the resources needed to support Content Studio's template design and configuration functionality. Presentation-related resources are not included. Since a blueprint is never deployed, resources required for presentation and rendering are not required.

In order for a blueprint to be used, it must be set as the active blueprint of a site publication. For instructions on how to do this, see Set Publication Blueprint.

### 11.1.3 Setting up a Legacy All-in-one Publication

To set up a legacy publication that contains both site and blueprint resources, declare the following dependencies in the publication POM file:

```
  <dependency>
```

```
    <groupId>com.escenic.widget-framework</groupId>
    <artifactId>widget-framework-core</artifactId>
    <version>4.5.2-2</version>
    <type>pom</type>
  </dependency>

  <dependency>
    <groupId>com.escenic.widget-framework</groupId>
    <artifactId>widget-pack-core</artifactId>
    <version>4.5.2-2</version>
    <type>pom</type>
  </dependency>
```

### 11.1.4 Additional Dependencies for Plug-ins

Additional widgets and other resources are shipped with the Widget Framework to support certain Content Engine plug-ins. These resources are not included as standard dependencies, however, since not all customers use the plug-ins. If you do use plug-ins that require access to additional Widget Framework resources, then you will need to include these additional dependencies in your projects. Currently, the only plug-in that requires access to additional Widget Framework resources is Live Center.

#### 11.1.4.1 Live Center Dependencies

If you have installed the Live Center plug-in and want to use the Widget Framework to present Live Center-related content, then you need to include the following additional dependencies in your project POM files:

**Additional site publication dependencies**

```
<dependency>
  <groupId>com.escenic.widget-framework</groupId>
  <artifactId>wf-live-center-site</artifactId>
  <version>4.5.2-2</version>
  <type>pom</type>
</dependency>
<dependency>
  <groupId>com.escenic.widget-framework</groupId>
  <artifactId>widget-pack-live-center-site</artifactId>
  <version>4.5.2-2</version>
  <type>pom</type>
</dependency>
```

**Additional blueprint dependencies**

```
<dependency>
  <groupId>com.escenic.widget-framework</groupId>
  <artifactId>wf-live-center-blueprint</artifactId>
  <version>4.5.2-2</version>
  <type>pom</type>
</dependency>
<dependency>
  <groupId>com.escenic.widget-framework</groupId>
  <artifactId>widget-pack-live-center-blueprint</artifactId>
  <version>4.5.2-2</version>
  <type>pom</type>
</dependency>
```

**Additional legacy all-in-one dependencies**

```
<dependency>
  <groupId>com.escenic.widget-framework</groupId>
  <artifactId>wf-live-center-pack</artifactId>
  <version>4.5.2-2</version>
  <type>pom</type>
</dependency>
<dependency>
  <groupId>com.escenic.widget-framework</groupId>
  <artifactId>widget-pack-live-center</artifactId>
  <version>4.5.2-2</version>
  <type>pom</type>
</dependency>
```

## 11.2 Using the Standard Merge Process

This section provides some information about the merging performed by the standard build process, and advice on how to use it when dealing with different types of application file.

If you look in the POM file of the parent **`wf-project-core`** project you will see a large Shade plug-in configuration for correctly merging various publication components into a single WAR file. This merge operation is automatically applied to your publication - you don't need to do anything to make it happen.

You can make use of the publication build tool in two main ways:

- The merge process performed by the build tool mostly involves looking for particular file types in particular locations and then doing the right thing with them: moving them to the right location in the output WAR file, merging the contents into a single file and so on. If you understand what it's doing, then you can make sure that you organize the files in your project correctly so that they will be found and dealt with in the proper way. This chapter contains some guidance on how to do this in relation to specific objectives you might want to achieve, but you can also find out a lot by looking at the configuration in the **`wf-project-core`** POM file.
- You can also modify the standard merge process if necessary. You can do so in the following ways:
  - Add resource transformers to the default Shade plug-in configuration
  - Use **`wf-build-plugin`** to modify some XML files during the build
  - Disable file version generation in shared projects.

  These modifications can be made by adding configuration elements to your publication (or shared project) POM file and are described in section 11.3.

### 11.2.1 Javascript Files

The Widget Framework includes a custom resource transformer for Javascript files called **`com.escenic.framework.transformers.JavaScriptFileMerger`**. This transformer is run as part of the standard publication build process. It merges all the Javascript files in your publication project together with all the Javascript files in the WAR files your publication depends on into one file, for performance reasons.

By default, **`JavaScriptFileMerger`** is configured to:

- Look for Javascript files in your webapp's **jscripts/** folder.
- Generate a merged output file called **wf-site.js**.

Note, however, that **JavaScriptFileMerger** also leaves the original files in their original location, so that they are available for debugging purposes.

All you have to do to make use of this step in the build process, therefore, is make sure that you put all your publication's Javascript files in the webapp's **jscripts/** folder. They will then automatically be included in the merged **wf-site.js**. file in alphabetic order of their file names.

Using a merged Javascript file offers the best performance for production purposes. During development, however, it is usually preferable to use the original individual Javascript files. You can force a publication to use your original Javascript files by setting the following section parameter in your publication's root section:

```
wf.scriptload.devmode=true
```

Widget framework ships with **requirejs**. So you can also load files using requirejs.

### 11.2.2 CSS Files

The Widget Framework uses the standard resource transformer **org.apache.maven.plugins.shade.resource.AppendingTransformer** to merge the CSS files it find in specific **theme** locations in your publication project, in the Widget Framework's framework module and in individual widget modules. The files are merged to theme CSS files. By default, the merge process is configured to handle one theme, called **default** with one variant called **variant**. In order to generate merged output CSS files for this theme, the **AppendingTransformer** looks for input CSS files with the paths **static/theme/default/base/css/theme.css** and **static/theme/default/variant/css/variant.css**.

This means that if you want to add CSS styles to the default theme or override existing styles in the theme, then you need to put your CSS styles in files called **static/theme/default/base/css/theme.css** and/or **static/theme/default/variant/css/variant.css** in your publication project.

You can extend the build process to handle additional themes by adding resource transformer configurations to your publication POM file as described in section 7.1.

### 11.2.3 Resource Files

The user interface labels and messages displayed by widgets are defined in various **ApplicationResources.properties** files included with the Widget Framework. During the publication build process they are merged together into a single **ApplicationResources.properties** file that is written to the publication's **WEB-INF/classes/com/escenic/framework** folder.

If you want to modify any of these labels or messages, all you have to do is:

1. Find the **ApplicationResources.properties** file(s) containing the string definitions you want to change. For a Teaser widget string, for example, you would need to look in the Widget Framework distribution's **misc/widgets/widget-core-teaser/src/main/resources/com/escenic/framework** folder.

2. Open the resource file(s) in an editor.
3. Copy the string definition(s) you want to modify.
4. Paste all the string definitions you want to modify into one new file.
5. Make the changes you want to the copied string definitions.
6. Save the file as **`src/main/resources/com/escenic/framework/ApplicationResources.properties`** in your publication webapp.

Now, when you build your publication, your **`ApplicationResources.properties`** will be merged with the standard **`ApplicationResources.properties`** and your string definitions will override the default definitions.

## 11.3 Modifying the Standard Merge Process

This section contains descriptions of various ways you can modify the standard build process.

### 11.3.1 Adding Resource Transformers

The default POM file supplied with the demo publication includes an empty Shade plug-in configuration that looks like this:

```
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-shade-plugin</artifactId>
  <version>2.2</version>
  <dependencies>
    <dependency>
      <groupId>com.escenic.widget-framework</groupId>
      <artifactId>wf-build-tools</artifactId>
      <version>${wf.version}</version>
    </dependency>
  </dependencies>
</plugin>
```

If you want to add a resource transformer, this is how you do it:

```
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-shade-plugin</artifactId>
  <version>2.2</version>
  <dependencies>
    <dependency>
      <groupId>com.escenic.widget-framework</groupId>
      <artifactId>wf-build-tools</artifactId>
      <version>${wf.version}</version>
    </dependency>
  </dependencies>
  <executions>
    <execution>
      <phase>package</phase>
      <goals>
        <goal>shade</goal>
      </goals>
      <configuration>
```

```
        <transformers combine.children="append">
          <transformer
 implementation="org.apache.maven.plugins.shade.resource.AppendingTransformer">
            <resource>WEB-INF/classes/CustomResources.properties</resource>
          </transformer>
          <!-- more transformers can be added here if required -->
        </transformers>
      </configuration>
    </execution>
  </executions>
</plugin>
```

The **transformers** element must have a **combine.children** attribute set to **append** as shown in the example. This ensures that the additional transformers you have specified are correctly combined with the default transformers configured in the parent POM file. The **transformers** element may contain more than one **transformer** if required.

For information about the purpose of the example shown above, see section 11.3.2.

### 11.3.2 Adding Resource Files

This section describes a specific application of the technique described in section 11.3.1. You might need to do this if you have extended some of the core widgets in some way and need to add your own string definitions. These need to be added in the appropriate locations with each widget, but kept in separate files from the supplied strings. We'll assume that your objective is to generate an output properties file called **com/escenic/framework/custom/CustomResources.properties**. To do this you need to:

1. Find the required widget folders under your Widget Framework distribution's **misc/widgets** folder (**misc/widgets/widget-core-teaser**, for example).
2. Create a **custom** folder in each of the widgets' **src/main/resources/com/escenic/framework** folders.
3. Create a **CustomResources.properties** file in each of the custom folders
4. Add the required property definitions to your **CustomResources.properties** files
5. Run **mvn clean install** in the distribution's **misc/widgets** folder. This will install the updated widget artifacts in your local repository.
6. Add a resource transformer to merge your new resource file, as described in section 11.3.1:

```
<configuration>
  <transformers combine.children="append">
    <transformer
 implementation="org.apache.maven.plugins.shade.resource.AppendingTransformer">
      <resource>WEB-INF/classes/CustomResources.properties</resource>
    </transformer>
  </transformers>
</configuration>
```

Now, when you build your publication, your merged **CustomResource.properties** will be included in the **WEB-INF/classes/com/escenic/framework/custom** folder of your publication WAR file.

### 11.3.3 Using wf-build-plugin

**wf-build-plugin** is a custom Maven plug-in intended to carry out Widget Framework-specific tasks during the publication build process. It is not used in the standard build process, but you can make use of it by adding configuration elements to your publication POM file.

Currently **wf-build-plugin** has only one goal, **xslt**, which is designed to modify XML files (primarily Escenic publication resources) by executing a series of one or more XSL transformations. The **xslt** goal must be bound either to Maven's **package** phase or to a phase that runs after the **package** phase in the Maven build life-cycle.

In order to use **wf-build-plugin** to modify one of your publication resource files, you need to:

- Write one or more XSL transformations that will make the changes you require.
- Save the transformations somewhere in your project.
- Add the necessary executions to the **wf-build-plugin** configuration in your publication POM file.

Two sample XSL transformations are included with the **demo** publication (in the **demo/src/main/resources/xsl** folder):

**sort-groups-alphabetically.xsl**
: Gathers together all **ui:group** elements and sorts them alphabetically by their **ui:label** element.

**filter-content-type.xsl**
: Removes duplicate **content-type**, **field-group** **relation-type-group** and **ui:group** elements.

So you could, for example, sort the **ui:group** elements in the **demo** publication's **content-type** resource by adding the following to the **wf-build-plugin** configuration in the publication's POM file:

```
<plugin>
  <groupId>com.escenic.widget-framework</groupId>
  <artifactId>wf-build-plugin</artifactId>
  <version>${wf.version}</version>
  <executions>
    <execution>
      <id>run-xslt</id>
      <goals>
        <goal>xslt</goal>
      </goals>
      <phase>package</phase>
      <configuration>
        <xsltJobs>
          <xsltJob>
            <resource>
              META-INF/escenic/publication-resources/escenic/content-type
            </resource>
            <transformers>
              <transformer>
                src/main/resources/xsl/sort-groups-alphabetically.xsl
              </transformer>
            </transformers>
          </xsltJob>
```

```
        </xsltJobs>
      </configuration>
    </execution>
  </executions>
</plugin>
```

**wf-build-plugin** has only one configuration element, **xsltJobs**. **xsltJobs** may contain one or more **xsltJob** elements, each of which must contain:

- A **resource** element containing the path of the XML file to be processed.
- A **transformers** element containing a sequence of one or more **transformer**, each of which contains the path of an XSL transformation to be applied to the resource.

If several **transformer** elements are present then they are applied in sequence, with the output from the first transformation being passed into the second and so on. So you could remove duplicates from the **demo** publication's **content-type** resource as well by adding another **transformer** element as follows:

```
<configuration>
  <xsltJobs>
    <xsltJob>
      <resource>
        META-INF/escenic/publication-resources/escenic/content-type
      </resource>
      <transformers>
        <transformer>
          src/main/resources/xsl/sort-groups-alphabetically.xsl
        </transformer>
        <transformer>
          src/main/resources/xsl/filter-content-type.xsl
        </transformer>
      </transformers>
    </xsltJob>
  </xsltJobs>
</configuration>
```

If you want to process more than one XML file in this way, then you can do so by adding multiple **xsltJob** elements, one for each file.

### 11.3.4 Disabling Version Generation

> The technique described in this section is not intended for use in ordinary publication projects. You should only use it in **shared webapp** projects.

The standard build process defined in **com.escenic.widget-framework:wf-project-core** involves a Shade plug-in execution called **generate-version**. What **generate-version** does is to rename the static files in the project (e.g, Javascript and CSS files), giving them names that depend upon their content. It generates a checksum from the content of each file, and then appends the resulting checksum to the filename. This means that the output static files have names that change every time the content of the file has changed.

Doing this means that content delivery networks and caches can be instructed to cache the static files for a very long time (for maximum efficiency), with no risk of serving old versions of the files. Any change to one of the static files will result in its name changing, so an old version will never be requested or served even if it is still in the cache.

In the standard build process, all Javascript and CSS files in a project are merged to single files, and the resulting files are renamed by **`generate-version`**.

It is sometimes the case that you want several publications to be able to share some common functionality. One way of achieving this is to create a **shared webapp** that provides this functionality, and then create publications based on the shared webapp. In other words, the shared webapp is configured with the Widget Framework's default **`com.escenic.widget-framework:wf-project-core`** as its parent project, and the actual publications are configured with the shared webapp as their parent project.

If you use this method of creating shared functionality, then the **`generate-version`** process should only be performed once, and it should be performed during the publication builds, not the shared webapp build. You therefore need to disable it in the shared webapp build. To disable the execution, add the following **`execution`** element to the Shade plug-in configuration in your shared webapp's POM file:

```
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-shade-plugin</artifactId>
  <version>2.2</version>
  <dependencies>
    <dependency>
      <groupId>com.escenic.widget-framework</groupId>
      <artifactId>wf-build-tools</artifactId>
      <version>${wf.version}</version>
    </dependency>
  </dependencies>
  <executions>
    <execution>
      <id>generate-version</id>
      <phase/>
    </execution>
    ...
  </executions>
</plugin>
```

The empty **`phase`** element in this addition effectively disables the execution of **`generate-version`**.

# 12 Google AMP Support

AMP is a Google project aimed at improving the performance of mobile web content. It consists of:

- A small number of custom HTML tags
- An associated Javascript library
- A set of rules and conventions governing the HTML code allowed in AMP documents

AMP pages are inherently well-suited to display on mobile devices and transport over low-bandwidth connections, and will in general render significantly faster than plain HTML/Javascript pages. They are also given preferential treatment by the Google search engine in some respects (AMP news stories get displayed in the Google news carousel on mobile devices, for example).

At the same time, AMP imposes significant restrictions on what can be included in an HTML document – no Javascript other than the AMP library is allowed, for example – so you might not want to be limited by AMP for the production of desktop pages. It is therefore common practice to generate two web site versions: a default site for desktop/laptop devices, and an AMP site for mobile devices.

The Widget Framework makes it very easy to generate an AMP version of a publication alongside a desktop publication by providing an **`amp`** content profile (see chapter 5). The standard desktop pages are generated using the Widget Framework's **`default`** content profile, and AMP pages are generated using the **`amp`** profile.

For general information about content profiles and how to use them, see chapter 5. The following sections contain specific advice on how to use the **`amp`** profile to produce AMP pages.

## 12.1 Page Styling

The Widget Framework provides Bootstrap-based styling for **`AMP`** pages in the form of a theme called **`amp`**. You can modify the supplied **`amp`** theme in exactly the same way as you would modify any other theme. Note, however, that AMP imposes restrictions on CSS, so any changes you make to the supplied CSS must be AMP-compliant. For general information about themes and how to work with them, see chapter 7.

As required by AMP the CSS is included directly in the HTML rather than in an external file.

### 12.1.1 Bootstrap Support

A full-size Bootstrap library is too large for inclusion in an AMP page, as the AMP standard imposes a size limit on CSS code. The Widget Framework therefore uses a smaller, AMP-compliant version of Bootstrap for AMP pages.

This minimized version of Bootstrap contains:

- Most typographical styles plus a few helper classes such as
  - Quick floats
  - Clearfix
  - Showing and hiding content

The following typographical styles, however, are **not included**:

- Abbreviations
- Addresses
- Blockquotes

- Grid system
- Breadcrumbs
- Labels
- Media object

As Google AMP doesn't allow you to reference external stylesheets, Widget Framework includes those styles inside the page head using **`<style>`** tag as Google suggests. If you need to use any Bootstrap style definition not provided by Widget Framework, you can customize as you see fit and replace the Bootstrap CSS files located in your publication's **`/static/css/amp`** directory with the customized one.

### 12.1.2 AMP Theme

To style Google AMP pages, Widget Framework is shipped with a new theme called **`amp`** and a new variant called **`variant`**.

You can customize the **`amp`** theme by modifying the widget specific **`theme.css`** file located in **`src/main/webapp/static/theme/amp/base/css`** directory. Moreover, if you have some common styles to be added to the **`amp`** theme, you can add those to the core **`theme.css`** file located in **`src/main/webapp/static/theme/amp/base/css`** directory of your project.

To customize the variant of the **`amp`** theme, you can modify the **`variant.css`** file located in **`src/main/webapp/static/theme/amp/variant/css`** directory of every widget. You can also modify the core theme directory's **`variant.css`** file which is in: **`src/main/webapp/static/theme/amp/variant/css`** directory of your project.

## 12.2 Discoverability

AMP is primarily intended to ensure fast load times on mobile devices, and in order to do that, it imposes certain restrictions on the content allowed on a page. This means that in many cases you may want to have two versions of a page: a standard version for display on desktop devices, assumed to have a fast internet connection, and an AMP version that is optimized for speed. It is then important to ensure that both versions of each page are easily discoverable, so that clients are always able to access most appropriate version of a page. For general information about this, see https://www.ampproject.org/docs/guides/discovery.html.

This section describes how to configure the Widget Framework so that your publication ensures the discoverability of both the AMP and default versions of publication pages.

### 12.2.1 Linking Pages

An AMP page's **`head`** element should always include a link to the default version of the same page, if one exists:

```
<link rel="canonical" href="default-version-url">
```

The Widget Framework automatically includes such links in pages rendered using the **amp** content profile.

Similarly, the default page's **head** element should always include a link to the AMP version of the same page, if one exists:

```
<link rel="amphtml" href="amp-version-url">
```

The Widget Framework does not, however automatically include such links in pages rendered using the **default** content profile. You must set the following **section parameter** to include it.

```
amp.enabled=true
```

The code that generates these links can be found in your publication webapp's **/template/framework/amp/head/canonical.jsp** file. By default it produces links that match the default URLs of AMP pages rendered by the Widget Framework (that is, the same as the default page URL plus the suffix **/amp**).

Your system may, however, be configured in such a way your AMP pages don't have the default URLs provided by the Widget Framework – your AMP pages might, for example, be served from a different domain name and/or have a different suffix.

If your AMP page URLs do not have the default format, then modify the generated links by editing the **/template/framework/amp/head/canonical.jsp** template in your webapp. By default, this contains the following code, which produces the default AMP URLs.

```
<%@ page session="false" %>
<@ taglib prefix="wfn" uri="http://www.escenic.com/widget-framework/functions" %>
<link rel="amphtml" href="${wfn:resolveAMPLink(param.pageURL, "amp")}">
```

All you need to do is replace **"amp"** with the correct suffix.

### 12.2.2 Including Metadata

Including sufficient metadata in your AMP pages will ensure they are handled well by search engines and other third party sites that may link to them. You are recommended to add at least the following sets of metadata to your AMP content:

**schema.org**
: schema.org provides a widely-supported vocabulary for adding metadata to web pages. Adding valid **schema.org** metadata to your AMP pages will ensure they are linked to by all major search engines, and will also help to ensure they get added to the Google news carousel for AMP news stories. **schema.org** metadata takes the form of a JSON object that must be wrapped in an HTML **script** element.

**Open Graph protocol**
: Including Open Graph metadata helps to ensure that your content can be handled well by Facebook and other social media apps. Open Graph metadata is encoded as a set of HTML meta elements.

**Twitter Card**
: Including Twitter Card metadata ensures that links to your pages will appear as "cards" in tweets. Twitter card metadata is encoded as a set of HTML meta elements.

Here is an example of a JSP template you might use to generate such metadata for your AMP content items:

```
<script type="application/ld+json">
{
  "@context": "http://schema.org",
  "@type": "NewsArticle",
  "mainEntityOfPage": "${article.url}",
  "headline": "${article.title}",
  "datePublished": "${article.publishedDate}",
  "dateModified": "${article.lastModifiedDate}",
  "author": {
    "@type": "Person",
    "name": "${article.author.name}"
  },
  "publisher": {
    "@type": "Organization",
    "name": "name-of-your organization",
    "logo": {
      "@type": "ImageObject",
      "url": "url-of-your-logo",
      "width": "your-logo-width",
      "height": "your-logo-height"
    }
  },
  "image": {
    "@type": "ImageObject",
    "url":
 "${article.relatedElements.images.items[0].fields.representations.value.small.href}",
    "height":
 "${article.relatedElements.images.items[0].fields.representations.value.small.height}",
    "width":
 "${article.relatedElements.images.items[0].fields.representations.value.small.width}"
  }
}
</script">

<meta property="og:title" content="${article.title}" />
<meta property="og:type" content="article" />
<meta property="og:url" content="${article.url}" />
<meta property="og:image"
 content="${article.relatedElements.images.items[0].fields.representations.value.small.href}" /
>

<meta name="twitter:card" content="summary" />
<meta name="twitter:site" content="@your-twitter-user" />
<meta name="twitter:title" content="${article.title}" />
<meta name="twitter:description" content="${article.fields.leadtext}" />
<meta name="twitter:image"
 content="${article.relatedElements.images.items[0].fields.representations.value.small.href}" /
>
```

To include such metadata in your content items, save the JSP file in your webapp's **`/template/widgets/code/view`** folder (as **`article-metadata.jsp`**, for example). Then create a Code widget, go to its **JSP** tab and set the **Path** field to reference the JSP file. Then place the Code widget in the meta area of your Widget Framework **`article`** templates.

> The **image** object referenced in the JSON metadata above must appear somewhere in the AMP document itself. So make sure you refer to a related image in the JSON that is also rendered in the article page.

> Note that Open Graph and Twitter Card metadata property names include the prefixes **og:** and **twitter:**. Open Graph requires this prefix to be declared and associated with the namespace **http://ogp.me/ns#**, usually in the HTML root element:
>
> ```
> <html prefix="og: http://ogp.me/ns#">
> ...
> </html>
> ```
>
> The **twitter:** prefix does not require any such declaration.

## 12.3 Menus

Any Menu widgets in your templates must be configured to use the AMP view by setting **View** on the **General** tab to **AMP**. The menu will then be rendered in an **amp-sidebar** element which is the appropriate container for navigation elements on an AMP page.

The AMP specification requires **amp-sidebar** to be a direct child of the HTML **body** element (see amp-sidebar). For this reason the following restrictions apply when using Menu widgets with AMP:

- The widget must be placed in a template's outer area
- The outer area's **HTML tag** option must be set to **no wrapper**

## 12.4 Rendering Content Item Body Text

The Widget Framework **amp** content profile renders the body of content items in the same way as other content profiles, with the following exceptions:

- Partial loading of content is not currently supported.
- Pagination of content is not currently supported.

## 12.5 Rendering Images

Google AMP provides a standard element for responsive image rendering called amp-img, which is used by the Widget Framework's **amp** content profile. When the amp content profile is used, all images are rendered using the **amp-img** element.

The **amp-img** element requires a maximum image width to be specified. By default, the **amp** content profile sets this maximum width to **768**. You can, however, modify the maximum width by setting the section parameter **wf.contentprofile.amp.image.width.max**.

The **amp** content profile makes use of an image policy (also called **amp**). This image policy is automatically selected by the **amp** content profile and does not need to be explicitly selected. The **amp** image policy should not be used in conjunction with any other profile: in other words you should never explicitly select the **amp** image policy.

## 12.6 Rendering Video

Google AMP provides a standard element for rendering video called amp-video, which is used by the Widget Framework's **amp** content profile. When using the **amp** content profile, you can use both the Teaser and Media widgets to render video as you would using the **default** content profile. The **amp-video** element does, however, impose a number of requirements and limitations that you must take account of:

**HTTPS only**
The **amp-video** element requires video sources to be available over HTTPS, so your servers must be configured to serve content over HTTPS. If you use the Escenic Video plug-in to transcode video then you must make sure it is configured to serve video over HTTPS (see http://docs.escenic.com/video-guide/3.1/awsclientconfig_properties.html for details).

**Cue point settings ignored**
The **amp-video** element cannot render cue points. The Media widget's **Cue point settings** are therefore ignored.

**Video player settings ignored**
The **amp-video** element always renders video using the HTML 5 **video** element. Any alternative video player settings specified either in widget properties (section 10.4.4.2) or in the **FrontEndConfig.properties** (section 10.4) configuration file are therefore ignored.

**Pre-roll ad settings ignored**
The **amp-video** element does not support pre-roll ads. Any settings related to pre-roll advertising are therefore ignored.

**Gallery settings ignored**
The **amp-video** element does not support video galleries. All gallery-related settings (**Link behavior**/**On click**=**select from gallery** and **Media play mode settings**=**select from gallery** in the Teaser widget, **Auto-advance videos in the gallery** in the Media widget) are therefore ignored. If **Media play mode settings** is set to **select from gallery**, then it falls back to the default setting (**Content page**).

**Pop-up video settings ignored**
The **amp-video** element does not support pop-up video players. **Media play mode settings**=**Pop up box** in the Teaser widget is therefore ignored and the (**Embedded media player**) mode is used instead.

## 12.7 Analytics for AMP Pages

Google AMP provides an amp-analytics component for capturing analytics data from an AMP document. This component is used by the Widget Framework's **amp** content profile for tracking user interactions with the publication's AMP pages.

### 12.7.1 Google Analytics

To enable Google Analytics support in your AMP pages, set up your Google Analytics account, and add the following section parameter:

```
wf.amp.ga.tracker.id=tracking-id
```

where *tracking-id* is your Google Analytics account tracking ID for AMP.

You can optionally override the default behavior of Google Analytics for your AMP pages by providing a custom tracker object configuration in an additional section parameter:

```
wf.amp.ga.tracker.config=tracker-configuration
```

where *tracker-configuration* is a JSON data structure, for example:

```
{"account":"tracking-id",
"title":"My Title",
"clientId":"client-id"
}
```

> You must use **double quotes** around the key names and string values in the JSON data you specify, not single quotes.

The Widget Framework merges these section parameters to generate the final tracker configuration that is passed to Google Analytics. So, if *tracker-id* is not defined in the JSON configuration, it will be collected from the value of **`wf.amp.ga.tracker.id`**. However, note that if *tracking-id* is provided in both **`wf.amp.ga.tracker.id`** and as the value of the **`account`** key in **`wf.amp.ga.tracker.config`**, the latter will take precedence over the former.

For further information about using **`amp-analytics`** and its JSON configuration parameters, see the Google Analytics documentation and the AMP Analytics variables list.

### 12.7.2 Escenic Analysis Engine

In order to enable Escenic Analysis Engine plug-in support for AMP pages, your Analysis Engine's Logger component must be configured to serve data over HTTPS as required by the **`amp-analytics`** element. The **`eae.logger.url`** section parameter in the publication root section must therefore be set as follows:

```
eae.logger.url=https://hostname/analysis-logger/Logger
```