Escenic Widget Framework

Developer Guide

2.0.3.136893


vizrt

Disclaimer

Vizrt provides this publication “as is” without warranty of any kind, either expressed or implied.

This publication may contain technical inaccuracies or typographical errors. While every precaution has been taken in the preparation of this document to ensure that it contains accurate and up-to-date information, the publisher and author assume no responsibility for errors or omissions. Nor is any liability assumed for damages resulting from the use of the information contained in this document.

Vizrt’s policy is one of continual development, so the content of this document is periodically subject to be modified without notice. These changes will be incorporated in new editions of the publication. Vizrt may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time.

Vizrt may have patents or pending patent applications covering subject matters in this document. The furnishing of this document does not give you any license to these patents.

Technical Support

For technical support and the latest news of upgrades, documentation, and related products, visit the Vizrt web site at www.vizrt.com.

Last Updated

01.03.2013


vizrt

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 config sections
- Modify the behaviour and appearance of the standard widgets in various ways (adding skins, writing your own controllers, overriding messages etc.)

2 Installation

There are three Widget Framework modules:

Widget Framework Core
The core module provides a standard set of general-purpose widgets and is a required component of any Widget Framework installation: you must always install this module.

Widget Framework Community
The community module provides widgets useful for social, community-based sites. It is an optional component.

Widget Framework Mobile
The mobile module provides widgets useful for mobilized sites. It is an optional component.

All three modules are packaged as Escenic plug-ins and should be installed in the same way as other plug-ins. Each module contains:

- Full source code for all widgets included in the module
- A demo web application

2.1 Preconditions

The following preconditions must be met before you can install any of the Widget Framework modules:

- Version 5.4 of the Content Engine (plus the Escenic assembly tool) has been installed as described in the Escenic Content Engine Installation Guide and is in working order.
- The following plug-ins have been installed as described in the relevant plug-in guides, and are in working order:
  - **Forum** (3.1 or later)
  - **Menu Editor** (2.1 or later)
  - **Poll** (2.2 or later)
  - **Geo Code** (2.4 or later)
  - **Analysis Engine** (2.4 or later)

  Depending on your requirements, you may also need to install the following plug-ins:

  - If you want to provide in-browser editing of your publications, you will need to install the **In Page** plug-in.
  - If you intend to install the **Widget Framework Community** distribution, you will need to install the **Viz Community Expansion**.

- If you intend to install the **Widget Framework Mobile** distribution, you will need to install the **Viz Mobile Expansion**.

- You have the required plug-in distribution file (**`widget-framework-core-2.0.3.136893.zip`**, **`widget-framework-community-2.0.3.136893.zip`** or **`widget-framework-mobile-2.0.3.136893.zip`**). These files can be downloaded from http://technet.escenic.com/ewf/article5419.ece.

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.

2.2.1 Installation Procedure

Installing a Widget Framework module involves the steps listed below. Replace *module* everywhere it appears with the name of the module you are installing (**`core`**, **`community`** or **`mobile`**).

1. Log in as **`escenic`** on your **assembly-host**.
2. Download the required distribution file from the Vizrt Technet web site (http://technet.escenic.com/ewf/article5419.ece). If you have a multi-host installation with shared folders as described in the **Escenic Content**

Engine Installation Guide, then it is a good idea to download the distribution to your shared **`/mnt/download`** folder:

```
$ cd /mnt/download
$ wget http://user:password@technet.escenic.com/downloads/release/5.4/widget-framework-module-2.0.3.136893.zip
```

Otherwise, download it to some temporary location of your choice.

3. If the folder **`/opt/escenic/engine/plugins`** does not already exist, create it:

```
$ mkdir /opt/escenic/engine/plugins
```

4. Unpack the distribution file:

```
$ cd /opt/escenic/engine/plugins
$ unzip /mnt/download/widget-framework-module-2.0.3.136893.zip
```

This will result in the creation of an **`/opt/escenic/engine/plugins/widget-framework-`***module* folder.

5. Log in as **`escenic`** on your **assembly-host**.
6. Run the **`ece`** script to re-assemble your Content Engine applications.

```
$ ece assemble
```

This generates an EAR file (**`/var/cache/escenic/engine.ear`**) that you can deploy on all your **engine-host**s.

7. ---

   If you have a single-host installation, then skip this step.

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

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

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

8. On each **engine-host** where you wish to run the plug-in, deploy the EAR file and restart the Content Engine by entering:

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

2.2.2 Verify the Installation

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

The plug-in is correctly installed.

- The plug-in is not correctly installed.

2.3 Build the code

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

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

After this, you should go to the **`misc/demo`** directory. If you run the **`mvn clean install`** command again, a demo webapp will be available under the **`misc/demo/target`** folder. For core, community and mobile distributions, the demo war file name will be 'demo-core-<version>.war', 'demo-community-<version>.war' and 'demo-mobile-<version>.war', respectively. Here <version> specifies the version of Widget Framework that you are using.

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

2.4 Create a Maven Repository for Your Company (Optional step)

If you don't have access to maven.vizrt.com, then you need to configure your own maven repository.

When using Widget Framework for development, each developer can either create his/her own local Maven repository on their machines or you may set up one for all the developers to use. Regardless of having a local repository or one that is accessible by everyone, it allows you to depend on the desired version of the desired widgets in your publication POM.

In this section, we will explain how to set up a company Maven repository with all the Widget Framework artifacts. The steps involved are simple:

1. Install a web server
2. Populate it with the Widget Framework Artifacts
3. Configure your development/build environments to use your own repository.

Commands with a hash (**#**) should be executed as the root user (**sudo su** on Ubuntu), whereas commands with a dollar sign (**$**) in front of them, should be executed with normal user privileges.

2.4.1 Install a Web Server

You may use any web server for this. The example below will use the Apache HTTPd running on a Debian based system (such as Ubuntu) as an example.

First off, install the Apache web server.

```
# apt-get install apache2
```

2.4.2 Add Widget Framework to Your Repository

Download and extract the latest Widget Framework in the flavour you want (currently, you can choose between "core", "community" and "mobile"). Below, we have used the "community" version as an example:

```
$ cd /tmp/
$ wget http://user:password@download.escenic.com/release/widget-framework/x.y/widget-framework-community-x.y.z.zip
$ unzip widget-framework-community-x.y.z.zip
$ cd widget-framework-community-x.y.z//misc/widgets/
$ mvn install
```

With the default Maven configuration, this will install all the Escenic Widget Framework and dependent artifacts into the user's local repository, located at **$HOME/.m2/repository**.

Switch to the administrative user and create a symbolic link to this repository or alternatively, copy it.

```
# cd /var/www/
# ln -s /home/<user>/.m2/repository /var/www/repo
```

You should now be able to access your repository at http://myhost/repo/

If you cannot access your repository in a web browser, only getting a 404 error message, Apache's log files may give some clues. They are on Debian based systems located in **/var/log/apache2/**

For trouble shooting the Apache configuration, please see the excellent Apache 2.2 documentation.

2.4.3 Use Your Local Repository

In all your development and build environments, you can now instruct Maven to use your local repository by adding this to your Maven **settings.xml** file (located in **$M2_HOME/conf/settings.xml** by default):

```
<?xml version="1.0" encoding="UTF-8"?>
<settings xmlns="http://maven.apache.org/SETTINGS/1.0.0"
          xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
          xsi:schemaLocation="http://maven.apache.org/SETTINGS/1.0.0
          http://maven.apache.org/xsd/settings-1.0.0.xsd">
```

```
[..]
<profiles>
  <profile>
    <id>wf-development</id>
    <activation>
    <activeByDefault>true</activeByDefault>
    </activation>
    <repositories>
      <repository>
        <id>my-wf-repo</id>
        <url>http://myhost/repo</url>
        <releases>
          <enabled>true</enabled>
        </releases>
        <snapshots>
          <enabled>true</enabled>
        </snapshots>
      </repository>
    </repositories>
  </profile>
</profiles>
[..]
</settings>
```

You may also specify this directly in the project POMs, but having it in the global Maven configuration (**`settings.xml`**) is recommended as it will apply to any Widget Framework project you work on.

Be sure to remove any references to the obsolete https://psprojects.escenic.com in any old POMs you may have, including the **`widget-framework-x.y.z/maven/pom.xml`**

2.5 Adding Custom Widgets

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

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

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

2.6 Using core and community widgets together

The core and community widgets are distributed with separate plug-ins. Each distribution contains full source code for relevant widgets, as well as a demo publication containing only those widgets. If you want to build a publication using both core and community widgets, please follow the instructions below :

- Download both **Widget Framework Core** and **Widget Framework Community** distributions from technet.escenic.com and install both of them as standard ECE plug-ins.
- Go to **misc/widgets** directory for each of the extracted archives and run **mvn clean install** command. Make sure that the build is successful.
- Then go to the **misc/demo** folder of the community distribution. Edit the pom.xml and add the dependencies to all core widgets. If you're not sure how to add these dependencies, please check the pom.xml in **misc/demo** folder of the **Widget Framework Core** distribution.
- Moreover, please add the <ui:group/> definition for core widgets in **misc/demo/src/main/webapp/META-INF/escenic/publication-resources/escenic/content-type** file of the community distribution.
- Run **mvn clean install** in **misc/demo** folder of the community distribution. After the build is successful, take the war file from the **misc/demo/target** folder and create a publication using it. This war file should contain the templates, as well as content type definitions, of both core and community widgets.

2.7 Using core and mobile widgets together

The mobile widgets are required if you want to enable mobile support for your site. The mobile widgets are distributed as part of a separate plug-in which contains the full source code for the widgets, along with a demo publication. If you want to build a publication using both core and mobile widgets, you need to follow the instructions below :

- Download both **Widget Framework Core** and **Widget Framework Mobile** distributions from technet.escenic.com and install both of them as standard ECE plug-ins.
- Go to **misc/widgets** directory for each of the extracted archives and run **mvn clean install** command. Make sure that the build is successful in each case.
- Then go to the **misc/demo** folder of the mobile distribution. Edit the pom.xml and add the dependencies to all core widgets. If you are not sure how to add these dependencies, please check the pom.xml in **misc/demo** folder of **Widget Framework Core** distribution.
- Moreover, please add the <ui:group/> definition for core widgets in **misc/demo/src/main/webapp/META-INF/escenic/publication-resources/escenic/content-type** file of the mobile distribution.
- Ensure that the maven-shade-plugin configuration in pom.xml is correct so that css files for both core and mobile widgets are merged properly.
- Please run **mvn clean install** in **misc/demo** folder of mobile distribution. After the build is successful, take the demo webapp war file from the **misc/demo/target** folder and create a publication using it. This war file should contain the templates, as well as content type definitions, of both core and mobile widgets.
- Please download **Viz Mobile Expansion** and install it as an ECE plug-in. Additionally, you'll need a valid license from mobiletech, which needs to be placed in tomcat's **lib** folder

- You need to make sure that the values for various properties in **/WEB-INF/classes/config/core.properties** in the deployed webapp are correct in your context.

2.8 Using core, community and mobile widgets together

The mobile widgets are required if you want to enable mobile support for your site. The mobile widgets are distributed as part of a separate plug-in which contains the full source code for the widgets, along with a demo publication. If you want to build a publication using all core, community and mobile widgets, you need to follow the instructions below :

- Download all **Widget Framework Core**, **Widget Framework Community** and **Widget Framework Mobile** distributions from technet.escenic.com and install all of them as standard ECE plug-ins.
- Go to **misc/widgets** directory for all of the extracted archives and run **mvn clean install** command. Make sure that the build is successful in all case.
- Then go to the **misc/demo** folder of the community distribution. Edit the pom.xml and add the dependencies to all core widgets and all mobile widgets. If you are not sure how to add these dependencies, please check the pom.xml in **misc/demo** folder of **Widget Framework Core** distribution.
- Add the dependency to **widget-framework-mobile** at the top of the **<dependencies>.**
- In the **<transformers>** of the **<configuration>** of **maven-shade-plugin** add the following **<transformer>**.

```
<transformer
  implementation="org.apache.maven.plugins.shade.resource.AppendingTransformer">
  <resource>skins/escenic-times/mobile/css/escenic-times.css</resource>
</transformer>
```

- Open **misc/demo/src/main/webapp/WEB-INF/web.xml** file and add the **<filter>**, **<servlet>**, **<filter-mapping>**, and **<servlet-mapping>**that is required for mobile. If you are not sure how to add these, please check the **web.xml** of the mobile distribution.
- Moreover, please add the <ui:group/> definition for core widgets and mobile widgets in **misc/demo/src/main/webapp/META-INF/escenic/publication-resources/escenic/content-type** file of the community distribution.
- Ensure that the maven-shade-plugin configuration in pom.xml is correct so that css files for both core and mobile widgets are merged properly.
- Please run **mvn clean install** in **misc/demo** folder of community distribution. After the build is successful, take the demo webapp war file from the **misc/demo/target** folder and create a publication using it. This war file should contain the templates, as well as content type definitions, of all core, community and mobile widgets.
- Please download **Viz Mobile Expansion** and install it as an ECE plug-in. Additionally, you'll need a valid license from mobiletech, which needs to be placed in tomcat's **lib** folder

- You need to make sure that the values for various properties in **`/WEB-INF/classes/config/core.properties`** in the deployed webapp are correct in your context.

2.9 Inpage Editing Support

If you want inpage editing support, you will first need to install In Page plug-in. Additionally, you'll need to ensure that the following section parameters are set correctly :

- inpage.enabled
- inpage.use.html5.edit.rich.text
- inpage.use.html5.edit.simple.text
- inpage.webservice.url

2.10 Creating the publication

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

How to create, build and deploy a publication is described in detail in the "Escenic Content Engine Installation Guide", available on http://technet.escenic.com/engine/54/

2.11 Supporting multiple languages

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

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

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

Note that you don't need to restart ECE to include new application resource files, but if you modify contents of an existing one then you need to reboot the server. If you want to avoid server reboot, then after editing the resource file rename it to something else (i.e., **`messsage.properties`** and then modify the **`javax.servlet.jsp.jstl.fmt.localizationContext`** parameter in the

web.xml file to include the name of the new file. This parameter value is set to **com.escenic.framework.ApplicationResources** by default, which you should change to **com.escenic.framework.[new-file-name]**

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 skins, 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
  skins/
    skin1/
      css/
        skin1.css
      gfx/
        widget-name/
          skin1-graphics-files
    skin2/
      css/
        skin2.css
      gfx/
        widget-name/
          skin2-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
WEB-INF/
```

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

3.1.1 Content Type

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

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

A widget content type definition:

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

A widget can have several **views** (HTML presentations), each of which may have its own configuration options. The **`field`** elements in a widget **`content-type`** should be organized to reflect this structure, with one **`panel`** containing **`field`** elements for setting the options of each view. In addition, the content type should contain:

- A General **`panel`** containing **`field`**s for common options, including an option that allows the designer to select which view to use.
- An Advanced **`panel`** containing **`field`**s for advanced options
- A Caching **`panel`** containing **`field`**s for caching 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 Skins

As well as having multiple views, a widget's appearance can be modified by the application of different skins. A skin consists simply of a CSS file and an accompanying set of graphics files (if required).

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 **`horizontal`** and a **`vertical`** view, or **`simple`**, **`default`** and **`complex`** views. The view that is actually displayed in a publication is determined by the publication designer, who selects the view in Content Studio. A widget 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**).

Widget JSP templates are divided functionally into:

- **Controller** templates that contain all the logic required to produce the values needed to render a widget.
- **View** templates that use the values generated by the controller templates to produce the final HTML output.

The idea is that the view templates should ideally contain only HTML plus the JSTL expressions required to retrieve values generated by the controller templates. All the hard work should be done by the controller templates. The controller obtains all the values that will be required to render the selected view, and writes 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.

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.

3.2 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 custom widget

In this section we will illustrate creating a widget with a Hello World example.

The layout of a widget is as follows:

```
webapp/
            META-INF/
            skins/
```

```
        skin1/
          css/
            skin1.css
          gfx/
            widget-name/
              a.png
              b.swf
        skin2/
          css/
            skin2.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
      WEB-INF/
```

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

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

3.3.1 Hello World Content Type

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

```
<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>
```

3.3.2 JSP example code

index.jsp

```
<%@ taglib tagdir="/WEB-INF/tags/escenic-widget" prefix="widget" %>

  <widget:view widgetName="helloworld"/>
```

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']}"/>
  </div>
```

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" class="java.util.HashMap" scope="request"/>

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

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)}"/>
```

4 Adding a New Grid

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

4.1 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.

4.2 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.

4.3 Modifying grid.css

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

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

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

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

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

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

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

4.4 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>
```

4.5 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.

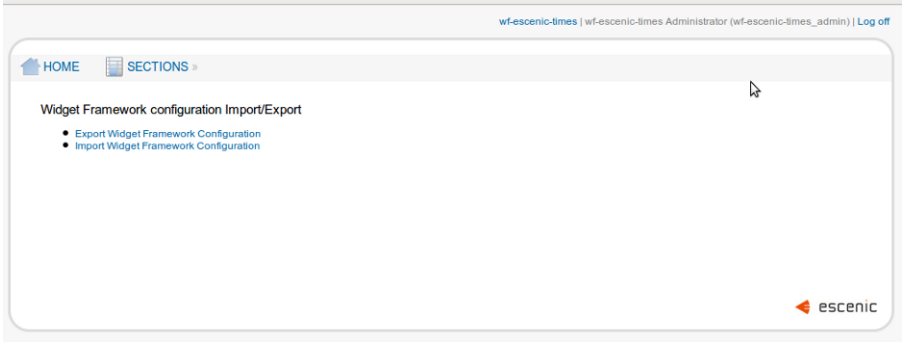
5 Export and Import

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

5.1 Syndication plug-in

To be able to export and import the configuration sections 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.

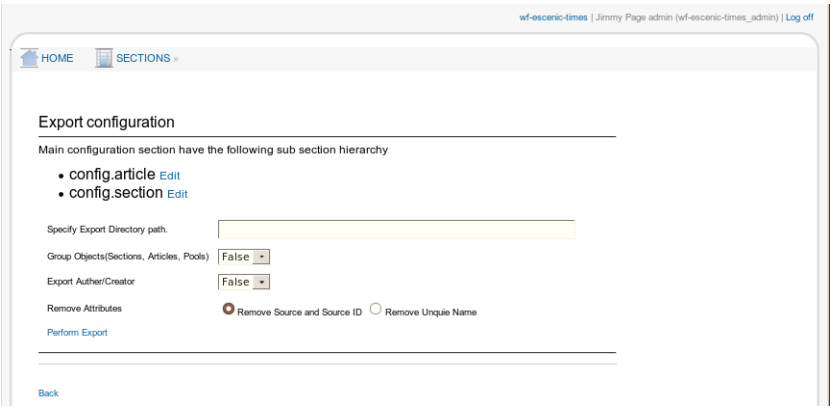
When the plug-in is installed you should get a new menu option in Web Studio. Here you have the options to export and import the configuration.



5.2 Syndication Export

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

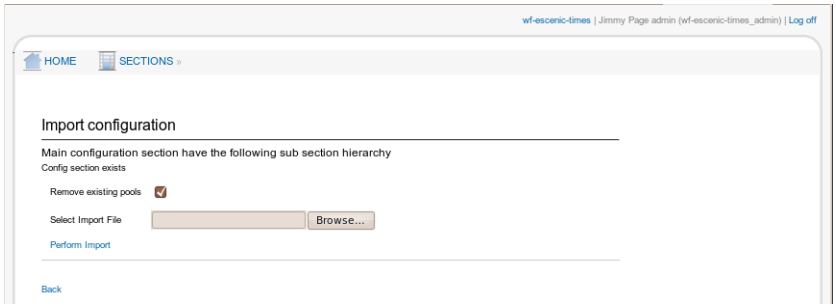
It will be exported to a specified location.



5.3 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.



6 How To

6.1 How to Add New Skin

Widget Framework is shipped with **escenic-times**. But you can add your own skin. The steps involved to add a new skin are as follows-

- Create your skin directory in each widget according to the strucutre of Widget Framework. Say your skin name is **my-skin**. Then you have to create my-skin directory under skin directory of each widget. The path is **<widget>/src/main/webapp/skins**
- You also need to create your skin directory in **widget-framework-core/src/main/webapp/skins** skin and need to add common css and gfx there.
- Add a property **publication.skin.my-skin.title = My Skin** in **widget-framework-core/src/main/resources/com/escenic/framework/ApplicationResources.properties.**
- Add the transformer for skin in your publication pom file which will be used to make final war file. For details, see **section 2.6** and **section 2.7**. If you add this transformer then maven shade plugin will merge all widget's my-skin.css in a single my-skin.css file otherwise it will not merge.

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

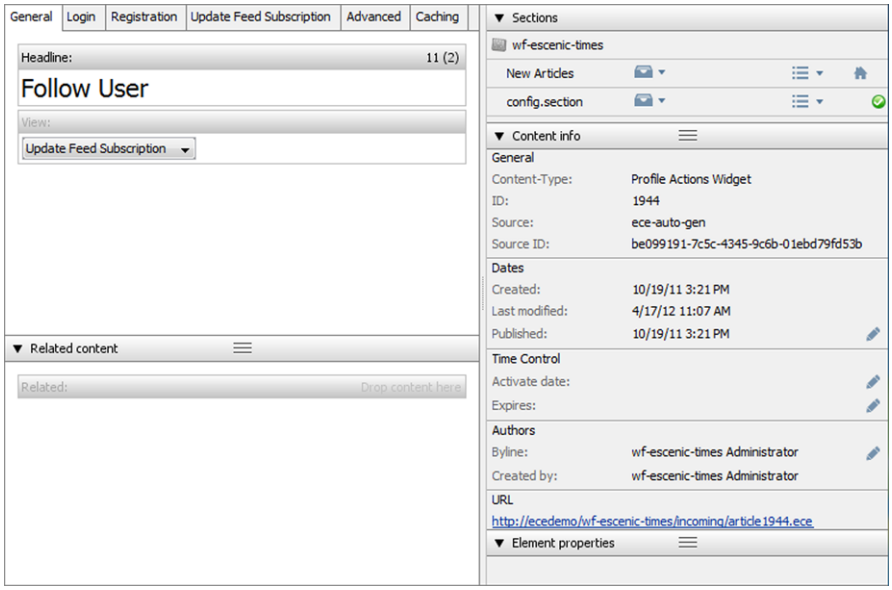
- compile and deploy your code.
- Edit root section parameter from web studio. Set skin variable there e.g **skin=my-skin**

6.2 To Use Following

To follow **User, Tag, Section or Article** section:

1. Open a **Profile Actions Widget**.
2. Choose **User Feed Subscription** view.
3. In the **Update Feed Subscription** panel choose the option you want to follow. You can follow an user, an article, a section or a tag.

4. Click on **Save** and **Publish** the widget.



5. Now place the widget in appropriate place. (Usually if you want to follow an user then you can place this widget in **`config.section.profile`**, if you want to follow a section you can place this widget in corresponding section config page, to follow an article you can place the widget in **`config.article`** and for tag you should place in config.section.search)

Say you chose an user to follow. Now after placing the widget in config.section.profile if you go to the user's profile page from publication you will find a Follow button. If you click on the button then you have started following the user.

6.2.1 Configurations Needed for Following

- The component **`/com/escenic/framework/community/history/HistoryFeedGenerator`** must be configured correctly for periodic custom feed generation. You have to specify publicationId, duration of update reporting etc there.
- The components **`/com/escenic/framework/community/history/HistoryFeedGeneratorSchedule`** and **`/com/escenic/framework/community/history/mail/HistoryFeedSenderSchedule`** must be specified as services in Initial.properties so that update feed is generated and sent periodically. From the example given in those finds you will find what values you have set.
- The component **`/com/escenic/framework/community/history/mail/HistoryFeedSender`** must be configured correctly so that emails with updates are sent on a periodic basis
- The components **`/com/escenic/framework/community/history/HistoryFeedGeneratorSchedule`** and **`/com/escenic/framework/community/history/mail/HistoryFeedSenderSchedule`** must be specified as services in Initial.properties so that update feed is generated and sent periodically.
- The component **`/neo/io/services/MailSender`** must be configured correctly so that mails can be sent.

You can unfollow an user by going to the user profile page. You will find an unfollow button. Click on the button to unfollow that user.

6.3 To Use Geo Blocking

Using geo blocking feature you can block items based on geographical location. There is two list called **`whiteList`** and **`blackList`**. One list will be active at a time. If you want to allow items only for certain countries then you should use**`whiteList`**. If you want to block only certain countries then you should use **`blackList`**. You will find the list in **`com/escenic/framework/geo/GeoLookupManager`**. Currently this feature is implemented only for video items (**simpleVideo**).

To use geo blocking feature for video you have to follow these steps-

- Configure **`com/escenic/framework/geo/GeoLookupManager`** and set the country code. Knowing the exact country code is very important. You will find the country code in csv format from here. http://geolite.maxmind.com/download/geoip/database/GeoIPCountryCSV.zip
- You must need the geo-ip database file. We are using the open source database provided by **Maxmind** (http://www.maxmind.com). You should place this file to appropriate place and have to specify the path in **`com/escenic/framework/geo/GeoLookupManager`**
- Create a **simpleVideo** and if you want to block that video then check the blocking option.
- Place the video in a group in content section
- Create and configure a video widget. Specify the group name where you placed the video

- **Save** and **Publish** the video widget

Create other two video item which is unblocked and place them in the same group of content section. Say, you choosed to block by **blackList** and in the blackList you specified **BD,US**. So, people from these two countries cannot see the blocked video as they are in black list. But they can see and play the unblocked videos. If users from these two countries goes to the article view of blocked video they will ses a message that **This content is not accessible in your region.**

You should periodically update the geo-ip data file. The open source data file used in `widget-framework` provides 99.5% accuracy and the premium data file provides 99.8% accuracy. See http://www.maxmind.com/app/geolitecountry for more details.

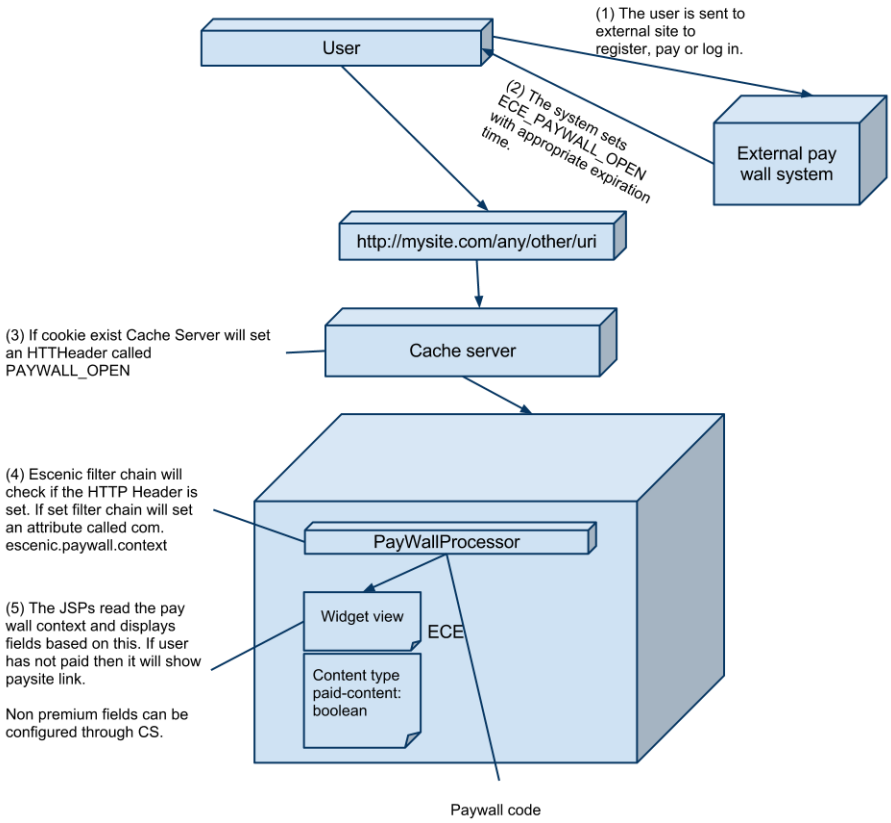
6.4 To Use Paywall Feature

Now editor can mark news type content as premium content. If user wants to read the full story then he/she have to pay for that. Currently premium feature is implemented only for news type item. If you mark any news as premium content then only non-premium fields will be visible to user. If user clicks on the news system will check whether the user is paid or unpaid. If he has paid then he can read the full news otherwise not. Payment should be made to a third party who is responsible for managing users premium account.

The full solution for payment system involved communication with the third party system. The steps are depicted below-

- When user clicks on premium news he will be promted to go to payment system if he has not paid.
- Then third party payment systems will set cookie for that user with appropriate expiration date.
- User will be redirected to the news site. There should be a **Cache Server** (e.g Varnish). **Cache Server** will check the existance of the cookie. If cookie is present then it will add a value `PAYWALL_OPEN` in HTTP header.
- Then Escenic filter chain will check if the `PAYWALL_OPEN` is set or not. If it is set then filter chain will set a request scope attibute called `com.escenic.paywall.context`

- Then the template JSP read the paywall context and displays fields based on this



6.5 Filter duplicate stories

It is possible to filter duplicate stories in a page. As for example, two widgets which fetches latest articles from a publication can fetch duplicate article. So, if those two widgets are placed in a section they can display same article twice. A client side javascript solution has been implemented so that widgets those fetch latest articles will not show same article more than one. **blog navigation, filmstrip, list, stories, trailers and video widget** will support filtering of duplicate stories. But source must be **latest articles** and filtering must be enabled from the widget.

In the **`<article:list/>`** we set the **`max`** attribute to twice the number of size specified in the widget. If this tag returns articles more than specified in the widget then we add a CSS class with each article and resulted articles will be hidden. The CSS class is adding by the decorator **`CssClassAdderDecorator`**. We also add a CSS class in widget's main div. Pattern of this CSS class is **`wf-filtered-list-size-xyz`** where xyz is the number of articles to fetch.

We have a script that runs on page load. The scripts starts filtering from the bottom of the result list. The script performs the following operation -

- will look for html element having class with the pattern **`wf-filtered-list-size-`**
- will extract number of articles to display from class name
- will look for article's having class pattern **`wf-article-`**
- will check whether this article has been displayed or not. If not displayed then it will display otherwise it will hide this article

To apply filtering to a new content type **`CssClassAdderDecorator`** must be added in the content type definition. This decorator add css class **`wf-article-xyz`** to the **`article.fields`** map. Here xyz is the article id. In the template article's parent should have the class **`wf-filtered-list-size-<listSize>`** and article must use the class **`${article.fields.wfClass}`**

In the widget we specify how many latest articles to fetch. Therefore, we expect that if this number of article returns by the tag then in the publication all of them will be displayed. In the script after filtering is done it will check the number of displayed article for each widget. If it is less than the required number then it will display some hidden articles from this widget to make the number equal.

6.6 Override Core Content Type

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

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

7 Configuring Components

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

7.1 reCAPTCHA

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

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

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

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

8 Writing New Controllers

In version 2.0.3.136893 of the Widget Framework, it is possible to write a custom Java controller for a particular widget. In this chapter, we will outline the steps needed to write and deploy such a custom controller.

8.1 Writing the Java Class

The custom controller should extend the class **`com.escenic.framework.controller.AbstractController`** and override appropriate methods to customize the controller functionality. It is also possible to implement the interface **`com.escenic.framework.controller.Controller`** directly, but in that case it is not possible to reuse the common logic that is part of **`AbstractController`**

For information on the sequence of steps performed by **`AbstractController`** or the methods that can be overridden, please consult the javadoc for this class.

8.2 Using the Custom Controller

Let us suppose that we have written the class **`com.escenic.framework.controller.impl.CustomController`** which extends the class **`com.escenic.framework.controller.AbstractController`**. Let us also assume that this class is in a separate maven module and packaged in a jar file. We now want to use this class as the controller for one of the standard widgets. In the following paragraphs, we will describe how to do so in the context of the **Widget Framework Core** distribution.

- Please ensure that the maven module for the custom controller contains a properties file named **CustomController.properties** in the folder **/src/main/resources/com/escenic/servlet/default-config/com/escenic/framework/controller/impl**. The content of the file will be as follows :

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

- Run **`mvn clean install`** command in the custom maven module so that the jar file is created and installed in your local repository.
- Download and extract **`Widget Framework Core`** distribution. Go to **`misc/widgets`** folder.
- Choose one of the core widgets for modification. For the sake of discussion, let us suppose that we have chosen the 'ad' widget.
- Create a properties file named **`ControllerFactory.properties`** in the **`widget-core-ad/src/main/resources/com/escenic/servlet/default-config/com/escenic/framework/controller/factory`** folder. The contents of the file will be as follows :

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

- Run **`mvn clean install`** command in **`misc/widgets`** folder.
- Then go to **`misc/demo`** folder. Edit the pom.xml file in that folder and add the dependency to the jar file that contains the custom controller and relevant configuration files. The dependency should be added in **compile** scope to ensure that the jar file containing the custom controller is present in **`WEB-INF/lib`** folder of the demo war after the build.
- Run **`mvn clean install`** command in **`misc/demo`** folder. During the build, all the **`ControllerFactory.properties`** files in various modules will be merged. The merged file will contain the line that we placed in the **`ControllerFactory.properties`** file specific to the 'ad' widget.
- Deploy the demo webapp war file that is created in **`misc/demo/target`** folder.

9 Modifying the Functionality of an Existing Controller

In version 2.0.3.136893 of the Widget Framework, most of the controller functionality has been moved to java space. However, the framework still supports JSP controllers. These controllers are invoked after the java controller has read the general and view-specific fields from the relevant panels in the widget and put them in the map (the map, in turn, is available in request scope). So, the JSP controllers can be used to execute any custom logic.

10 Modifying Resource Bundles and Merging Javascripts

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

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

10.1 Overriding Messages in Standard Resource Bundles

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

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

10.2 Adding New Resource Bundles

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

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

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

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

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

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

10.3 Merging javascript from various widgets

From Widget Framework 2.0.0 it is possible to merge javascript files from various widgets. Purpose is to reduce lots of javascript file inclusion in the header. Browser has to open HTTP connection for each individual file present in the header which hampers the performance of site. So, merging javascripts files to a single file will improve the performance. Merging is done by maven shade plugin and ant. Condition is that if you have javascript plugin or code for widgets you have to put that in **<widget>/src/main/webapp/jscripts** directory. You can put as many javascript file in the jscripts directory. The merged javascript file will be created in **<war>/resources/js/wf-widget.js**.

```
widget1
 src
   main
     webapp
       jscripts
         somejs1.js
         somejs2.js
         ...


widget2
 src
   main
     webapp
       jscripts
         somejs3.js
         somejs4.js
         ...
```

Be careful about javascript files having the same name. If there is more than one file with same name then maven shade plugin will consider only the first one.
