



Escenic Content Engine
Advanced Developer Guide
5.2.7.2







Copyright © 2004-2011 Vizrt. All rights reserved.

No part of this software, documentation or publication may be reproduced, transcribed, stored in a retrieval system, translated into any language, computer language, or transmitted in any form or by any means, electronically, mechanically, magnetically, optically, chemically, photocopied, manually, or otherwise, without prior written permission from Vizrt.

Vizrt specifically retains title to all Vizrt software. This software is supplied under a license agreement and may only be installed, used or copied in accordance to that agreement.

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

21.12.2011



Table of Contents

1 Introduction	9
2 Section Parameters	11
2.1 Setting Section Parameters	11
2.2 Using Section Parameters	12
3 Content Item Field Indexes	13
3.1 Specifying Content Item Field Indexes	13
3.2 Generating Content Item Field Indexes	14
4 JSP Profiling	15
4.1 Enabling/Disabling Profiling	15
4.2 Controlling Output	16
4.2.1 Grouping by Template	16
4.2.2 Grouping Within Templates	16
4.2.3 Grouping By Section	17
4.3 Viewing The Results	17
4.4 Understanding The Results	18
4.5 Collection Strategies	19
4.5.1 Micro-collection	19
4.5.2 Macro-collection	20
4.6 Extending Statistics Collection	20
5 Cross-Publishing	21
5.1 Access Rights	21
5.2 Cross-Publishing Methods	22
5.2.1 Single Article Cross-Publishing	22
5.2.2 Shadow Section Cross-Publishing	22
6 Metadata Extraction	23
6.1 Using The Standard Facility	23
6.2 The Plug-in API	24
6.2.1 Metadata Injection Example	25
7 Servlet Filters	27
7.1 ECEProfileFilter	28
7.2 BootstrapFilter	28
7.3 TimerFilter	29
7.4 EscenicStandardFilterChain	30
7.4.1 The Filter Chain Processors	30



7.5 CacheFilter	38
7.6 Modifying The Filter Chain	39
7.6.1 Modifying The Main Filter Chain	39
7.6.2 Modifying The Escenic Standard Filter Chain	39
8 Decorators	43
8.1 Creating A Decorator	43
8.2 Using A Decorator	45
8.3 A Complex Decorator Example	45
9 Event Listeners	47
9.1 Making An Event Listener	47
9.2 Using An Event Listener	48
10 Transaction Filters	49
10.1 Making A Transaction Filter	49
10.2 Using A Transaction Filter	51
10.3 Error Handling	52
11 Post-transaction Filters	53
11.1 Making a Post-transaction Filter	53
11.2 Using a Post-Transaction Filter	54
11.3 Error Handling	55
12 The web.xml File	57
13 Publication Webapp Properties	59
13.1 Viewing Publication Webapp Properties	59
13.2 Modifying Publication Webapp Properties	59
13.3 The default.properties File	60
14 Mail a form	61
14.1 Create the form	61
14.2 edit struts-config.xml	62
15 Representations	63
15.1 Defining Image Representations	63
15.1.1 Derived Representations	64
15.2 Accessing Image Representations	65
16 Restricting Access to Content	67
16.1 Basic Password Authentication Example	68
16.1.1 Using The Example	70
16.2 Removing Access Control	70
17 Further Reading	71
17.1 Vizrt Resources	71

[17.2 Other Resources](#)..... 71





1 Introduction

This guide is intended to describe various advanced development techniques for Escenic developers who are already familiar with the content of the **Escenic Content Engine Template Developer Guide**.



2 Section Parameters

Section parameters can be used to:

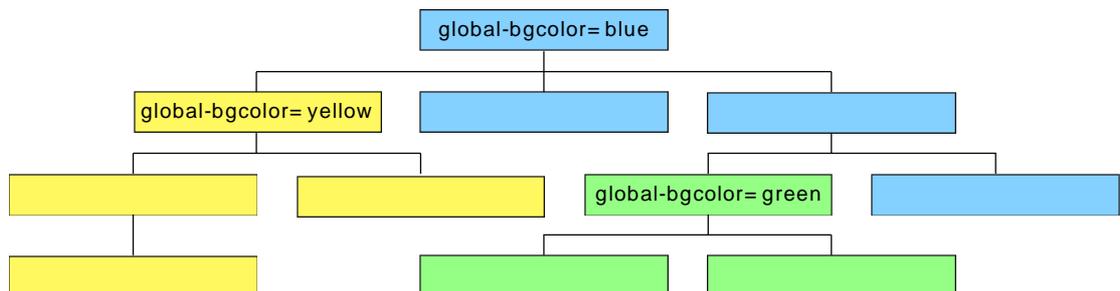
- Differentiate the sections in a publication without the need for writing many different section templates.
- Give publication administrators a simple means of control over the appearance/behaviour of the sections in a publication.

2.1 Setting Section Parameters

A section parameter is a freely-definable key-value pair that can be assigned to a section using Web Studio. For example:

```
global_bgcolor=#2661DD
```

Section parameters are inherited, so if the above parameters are assigned to the root section of a publication, then they will apply to the whole publication by default. However, you can redefine the parameters in a subsection; these new settings will override the inherited ones and apply to the subsection and all its subsections, as shown below:



To set a section parameter in Web Studio you must have section administrator rights. From the Web Studio home page:

1. Select **SECTIONS**.
2. Select the **Administer** link next to the section you are interested in.
3. Select **Edit section parameters**.
4. Enter the parameter settings you want to add in the displayed **Properties** pane.
5. Select **Save**.

When choosing names for your section parameters, you should avoid characters that have special meanings in JSTL (. and -, for example), as this complicates referencing them in your templates.

2.2 Using Section Parameters

Section parameters can be accessed in templates using the `section` bean's `parameters` property as follows:

```
${section.parameters.global_bgcolor}
```

With this example section parameter you can create templates that allow different background colors to be used in each section, for example:

```
<table
  bgcolor="${section.parameters.global_bgcolor}"
  width="800"
  cellspacing="3"
  cellpadding="3"
  border="0">
  ...
</table>
```



3 Content Item Field Indexes

The `article:list` tag returns a `java.util.List` bean containing all the content items in a publication that match various criteria. You can use it, for example, to list the last ten content items added to a specific section, or all the content items published in the last three days. The `article:list` tag also has a `field` attribute that you can use for selecting and sorting the content items to list by field value. If, for example, the content items in your publication have a `priority` field, you might want to obtain a list of all content items with a priority of 1. You can do this as follows:

```
<article:list
  id="myList"
  sectionUniqueName="mySection"
  field="priority"
  expression="1" />
```

You can, however, only use the `link` attribute on content item fields that have been **indexed**. If the `priority` field is not indexed, then this example will not work.

3.1 Specifying Content Item Field Indexes

Content item field indexes are specified in the `content-type` resource. In order to be able to index any of the fields in a content type, you must add a `neo.xredsys.service.article.attribute` parameter to the content type definition:

```
<content-type>
...
  <parameter name="neo.xredsys.service.article.attribute" value="true"/>
</content-type>
```

Once you have done this, you can specify indexing of individual fields within the content type by adding `neo.xredsys.service.article.attributeField` parameters to the field definitions:

```
<field name="priority" type="number">
...
  <parameter name="neo.xredsys.service.article.attributeField" value="priority"/>
</field>
```

Note that the `value` attribute is set to the name of the field that is to be indexed. The `neo.xredsys.service.article.attributeField` parameter actually determines the name that you will need to use to identify the indexed field in the `article:list` tag. You could set it to some other name, but you are advised not to do so.

If white space is significant in the indexed field, then you should also add a `neo.xredsys.service.article.attributeField.trim` parameter as follows:

```
<field name="formatted" type="basic">
```

```
...  
<parameter name="neo.xreditsys.service.article.attributeField" value="priority"/>  
<parameter name="neo.xreditsys.service.article.attributeField.notrim" value="true"/>  
</Field>
```

If you do not specify this parameter then for indexing purposes all leading and trailing white space will be trimmed from the field, and all internal white space will be normalized to a single space character.

3.2 Generating Content Item Field Indexes

If you add new content item field indexes to an existing publication, then you will need to explicitly generate the indexes for all existing content items. To do this:

1. Go the Escenic Web Administration interface.
2. Select **List pubs**.
3. Select the publication you have modified.
4. Select **Run field indexer**.

The new field will be indexed automatically in any content items subsequently added to the publication.

Note the following:

- Some changes to content types can cause existing indexes to get out of sync. You can correct (or avoid) such problems by running the field indexer after modifying content type definitions.
 - Indexing can be very time-consuming, and can have a significant effect on database performance.
-

4 JSP Profiling

The Content Engine incorporate a JSP profiling facility that can be useful for tuning purposes. It collects information about the time used by each of the templates in a publication, enabling you to identify the **hot spots** that you should concentrate on if you want to improve performance. The facility not only collects information about how much time is spent in each template, but also the database queries and updates performed by each template and the amount of memory allocated by each template.

You can also, if necessary, obtain more detailed information about **where** the time is being spent in a particular template by dividing the template into sections using `util:profiler` tags (see [section 4.2.2](#)). The same information will then be generated for each section of the template, enabling you to pinpoint problem code more exactly.

4.1 Enabling/Disabling Profiling

Profiling is not enabled by default, because it incurs a small (less than 1%) performance penalty. You can turn profiling on by setting the application scope attribute `neo.util.servlet.RequestInfo.StatisticsSource` as follows:

```
<%
pageContext.getServletContext().setAttribute("neo.util.servlet.RequestInfo.StatisticsSource",
neo.nursery.GlobalBus.lookupSafe("/neo/io/reports/ReportsStatisticsSource"));
%>
```

To turn profiling off again, simply remove the attribute as follows:

```
<%
pageContext.getServletContext().removeAttribute("neo.util.servlet.RequestInfo.StatisticsSource");
%>
```

You want to be able to turn profiling on and off at will during template development without exposing the option to users of the publication. A simple way of doing this is to add two JSP files containing these commands to your application: `profiling-on.jsp` and `profiling-off.jsp`, for example. You can then easily turn profiling on and off by navigating to these files in your browser. To turn profiling on, for example, you would enter something like this in your browser's address field:

```
http://server-name:port/publication/template/profiling-on.jsp
```

and something like this to turn it off again:

```
http://server-name:port/publication/template/profiling-off.jsp
```

where *publication* is the name of your publication.

4.2 Controlling Output

By default, the profiling facility will group the statistics it collects by publication and template: that is, it will generate one group of statistics for each publication on the server, with subgroups within each publication for each template. This means you can see how much time is used in a particular template for a particular document, how many database accesses each template generates and so on.

You may, however, want to break down the statistics for some templates even more, so that you can see exactly where in a template some problem is arising. You might also want to break the statistics for a publication down by section. On the other hand, you might not be interested in template-level statistics and want to simplify the output grouping.

The following sections describe how you can achieve these objectives.

4.2.1 Grouping by Template

Profiling results are automatically grouped by template. This level of grouping is carried out by a filter in the servlet filter chain. If, for any reason you do not want the results to be grouped in this way you can prevent template grouping by removing the `ECEProfileFilter` from the filter chain. For further information about this filter, see [section 7.1](#).

Note that the default template grouping provided by `ECEProfileFilter` depends on you using `jsp:include` to transfer control between templates. If you use any other means of transferring control, then you will need to explicitly redirect profiling output to a new group yourself by wrapping template content in a `util:profiler` tag as follows:

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

<util:profiler path="/wireframe/default.jsp">
  ...template contents...
</util:profiler>
```

You are recommended, however, to use `jsp:include`: this explicit grouping will then not be necessary.

4.2.2 Grouping Within Templates

You can use `util:profiler` to break down profiling output within templates as follows:

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

...[1]...
<util:profiler fragment="navigation">
  ...[2]...
  <util:profiler fragment="switch1">
    ...[3]...
  </util:profiler>
...[4]...
```



```

<util:profiler fragment="switch2">
  ...[5]...
</util:profiler>
  ...[6]...
</util:profiler>
  ...[7]...

```

If the example is called `/wireframe/default.jsp`, then this will give the following entries in your profiling output:

- /wireframe/default.jsp**
containing output for code blocks [1] and [7]
- /wireframe/default.jsp[navigation]**
containing output for code blocks [2], [4] and [6]
- /wireframe/default.jsp[switch1]**
containing output for code block [3]
- /wireframe/default.jsp[switch2]**
containing output for code block [5]

4.2.3 Grouping By Section

You can generate a results group for every section in each publication (rather than just a group for each publication by including something like this in your outermost template (usually `common.jsp`):

```

<%@ taglib uri="/WEB-INF/escenic-util.tld" prefix="util" %>
<%@ taglib uri="http://struts.apache.org/tags-bean" prefix="bean" %>

<bean:define id="groupName">
  ${publication.name}.${section.name}
</bean:define>

<% pageContext.getServletContext().setAttribute("neo.util.servlet.RequestInfo.StatisticsSource",
  neo.nursery.GlobalBus.lookup("/neo/io/reports/ReportsStatisticsSource"));
  request.setAttribute("neo.util.servlet.RequestInfo.group", groupName);
%>

```

This code sets the request scope variable `neo.util.servlet.RequestInfo.group`, which determines the group to which output is directed.

4.3 Viewing The Results

Once you have set the `neo.util.servlet.RequestInfo.StatisticsSource` attribute and the publication has been accessed, there should be some results available. To view them:

1. Go to the web administration interface.
2. Select **View JSP Statistics**.
3. The displayed page will at least contain a link called **other**. If you set the `neo.util.servlet.RequestInfo.group` to the name of your publication then the publication name should appear there as well.
4. Select the appropriate link. You should then see a display something like this:

These tables display results for the top 5 templates in each of four categories (DB queries, DB updates, memory and time). You can display a complete table for one of the categories by clicking on the link above the table: to display a complete **memory** table, for example, click on **memory** in the **Top 5 memory usage** heading. To display more detailed information about a particular template, click on one of the links in the **uri** columns of the tables.

To zero all the results, click on the **Reset** link at the bottom of the page.

4.4 Understanding The Results

Profiling information is collected for the following resources:

dbqueries

The number of database queries performed.

dbupdates

The number of database updates performed.

memory

The amount of memory allocated, in bytes. Note that this is an approximation since the Java virtual machine (JVM) does not provide a reliable standard method of retrieving this information.

time

Elapsed time, in milliseconds. Note that on some Windows systems, clock ticks have a length of approximately 16ms, so that all recorded times will be multiples of around 16 milliseconds.

The profiling information for all of these resources is calculated as follows:

1. Measure the resource on entry to the template.
2. Measure the resource on exit from the template.
3. Subtract one value from the other to give the template's gross consumption of that resource.
4. Subtract from this value the gross consumption of all the template's called templates in order to produce its net consumption.

This method is good enough for most purposes but has some weaknesses that you should be aware of:

- Since only elapsed time is counted, no account is taken of time-consuming "external" processes, such as garbage collection, memory contention, other threads and so on. This may occasionally cause a template to appear to use a large amount of time, when in fact the time has been used by an external process.
- Similarly, if garbage collection is carried out while a template is running, it may result in a template apparently using a negative amount of memory. You will never see this in the final results, since such values are set to 0. Nevertheless, it means that the memory usage information is not entirely accurate.



In general, this means that profiling is of little use on a machine that is heavily overloaded.

The statistics are presented in tables with the following columns:

uri

The URI of the monitored template.

total resource

The total amount of *resource* consumed by the template during the profiling period. The table is sorted by this column, so that the template with the highest consumption is listed first. It is usually this template that you will be most interested in.

invoked

The number of times the template was invoked. Some templates may be called several times per page-view; others may not be called at all.

best

The lowest consumption value recorded for the template. This column is particularly significant because gives an indication of the template's best possible performance, given optimal conditions.

avg

The template's average consumption per invocation: **total** divided by **invoked**.

worst

The highest consumption value recorded for the template. This figure should be treated with caution because it can easily be distorted by external processes such as garbage collection, memory contention or JSP compilation.

4.5 Collection Strategies

There are two basic ways you can use the statistics collection facility, called **micro-collection** and **macro-collection** here.

4.5.1 Micro-collection

This approach is suitable for detailed analysis of specific templates when you already have some idea of what your problem is and you know approximately what you are looking for. You can only really use this approach on a test or development server over which you have full control. The general procedure is as follows:

1. Prepare for the specific action you want to test. For example, start a browser and navigate to a specific location in the publication.
2. Switch statistics collection on by running your `statistics-on.jsp` file.
3. In a separate browser window or tab navigate to the web administration interface. Select the **Clear all caches** option to empty the Content Engine's caches.

4. Select **View JSP statistics** followed by the name of the publication you are interested. This displays the statistics page for that document.
5. Select the **Reset** link at the bottom of the page to clear old statistics.
6. Click **Back** to redisplay the main page.
7. In the other browser window/tab, perform the action you want to investigate - often just a single page display.
8. Switch statistics collection off by running your `statistics-off.jsp` file.
9. Back in the administration interface, redisplay the statistics for your publication. You now have a set of statistics for the action you just performed.

Statistics collected in this way give you a good basis for detailed analysis of particular templates. You might, for example, see that the action you performed has caused a particular template to be invoked many times more than you expected, or generated far more database queries than you expected.

4.5.2 Macro-collection

This approach is suitable for getting a general impression of where problems might be arising in a system, and is really the only approach that can be used on a production server. It is difficult to collect reliable statistics for a single operation on a production server because the chances of operations being affected by external processes such as competing threads and garbage collection are much higher. What you can do, though is simply gather statistics over a period of time (5 minutes say, or half an hour) on the assumption that such distortions will average out. You can then get a general impression of where most resources are used in your publication.

The procedure in this case is:

1. In the web administration interface, select the **Clear all caches** option to empty the Content Engine's caches.
2. Select **View JSP statistics** followed by the name of the publication you are interested. This displays the statistics page for that document.
3. Select the **Reset** link at the bottom of the page to clear old statistics.
4. Click **Back** to redisplay the main page.
5. Switch statistics collection on by running your `statistics-on.jsp` file.
6. Wait.
7. Switch statistics collection off by running your `statistics-off.jsp` file.
8. Redisplay the statistics for your publication.

4.6 Extending Statistics Collection

The statistic collection facility is based on a public API, so you can extend it to gather other kinds of statistics if you need to.

To be supplied.

5 Cross-Publishing

Cross-publishing means allowing content items in one publication to also appear in sister publications served by the same Content Engine.

The Content Engine fully supports cross-publishing: borrowed content items adopt the layout of the publication section in which they appear and are indistinguishable from native content items. There are, however, a few restrictions that must be observed in order to ensure problem-free cross-publishing. Publications can only borrow content items from each other if:

- They have identical content types. This means that the content types in both publications must have:
 - identical names
 - Identical fields with identical names
- They have identical image versions/representations. This means that the image versions and/or image representations in both publications must have:
 - identical names
 - Identical sizes

You can set up cross-publishing so that editorial staff not only have access rights to content items in a sister publication, but editorial rights too (that is, they will be allowed to modify content items in the sister publication). In this case the publications must have identical layout definitions.

Cross-publishing does **not** require identical templates, only identical **content-type**, **image-version** and (possibly) **layout-group** publication resources. Nor does the use of cross-publishing have any consequences for the template developer. Given identical publication resources, cross-publishing is an administration/editorial issue, and can be set up using Web Studio.

5.1 Access Rights

In order to enable cross-publishing, the administrator of a "lending" publication A must assign access rights to users of the "borrowing" publication B. The procedure is as follows:

1. Export users from publication B.
2. Import users to publication A.
3. Assign publication A access rights to the imported users. The imported users can be assigned access to the whole publication or only to restricted sections. If publication B staff are only to be allowed to borrow publication A content items, then they should be given **reader** access. If they

are to be allowed to modify content items, then they should be given **journalist/editor** access.

4. If shadow section cross-publishing (see [section 5.2.2](#)) is to be allowed, set the **shared** property of the sections that are to be shadowed.

Once access rights have been assigned, editorial staff on publication B will be able to see the parts of publication A to which they have access when using Content Studio.

5.2 Cross-Publishing Methods

There are two methods of cross-publishing:

- Single content item cross-publishing
- Shadow section cross-publishing

5.2.1 Single Article Cross-Publishing

To cross-publish a single content item, a user in the borrowing publication simply searches for the required content item in the usual way, and uses it as if it were a native content item.

5.2.2 Shadow Section Cross-Publishing

If any of the accessible sections in the lending publication are shared, then it is possible to create shadow sections in the borrowing publication that automatically reflect all the content of the source section. Cross-published shadow sections behave in exactly the same way as locally-sourced shadow sections.



6 Metadata Extraction

The Escenic Content Engine incorporates a **metadata extraction facility** that can extract the metadata embedded in some types of binary file and automatically inject it into content item fields. The standard facility supplied with the Content Engine supports the extraction of EXIF and IPTC metadata embedded in JPEG image files. The type of information typically embedded in this way includes items such as the photographer's name, copyright information, the date a photograph was taken and so on.

The metadata extraction facility incorporates a plug-in interface, which Java programmers can use to:

- Control the metadata extracted
- Control how extracted metadata is injected into content items
- Add support for other binary file formats and metadata formats.

6.1 Using The Standard Facility

The metadata extraction facility only works with content items that contain a **link** field referencing a binary object. That is, the content type definition in the **content-type** resource must contain a **field** in which the **type** attribute is set to **link**. In addition, the default facility currently only provides support for JPEG image files: other image file formats and other media objects such as audio and video files are not currently supported.

 The metadata extraction facility does not support legacy image and media content types.

To make use of the default extraction facility, all you need to do is add a field called **COM. ESCENIC. DEFAULTMETADATA** to the content type definition in the **content-type** resource. The new field must be a complex array composed of two sub fields called **KEY** and **VALUE**. Both **KEY** and **VALUE** must be **basic** fields.

The following example shows a content type definition for image files to which a default metadata field has been added (highlighted in bold).

```
<content-type name="image">
  <ui:label>Picture</ui:label>
  <ui:description>An image</ui:description>
  <ui:title-field>name</ui:title-field>
  <panel name="default">
    <ui:label>Image content</ui:label>
    <field mime-type="text/plain" type="basic" name="name">
      <ui:label>Name</ui:label>
      <ui:description>The name of the image</ui:description>
      <constraints>
        <required>true</required>
      </constraints>
    </field>
    <field mime-type="text/plain" type="basic" name="description">
      <ui:label>Description</ui:label>
    </field>
    <field mime-type="text/plain" type="basic" name="alttext">
```

```

    <ui:label>Alternative text</ui:label>
  </field>
  <field name="binary" type="link">
    <relation>com.escenic.edit-media</relation>
    <constraints>
      <mime-type>image/jpeg</mime-type>
      <mime-type>image/png</mime-type>
    </constraints>
  </field>
  <field name="COM.ESCEMIC.DEFAULTMETADATA" type="complex">
    <array default="0"/>
    <complex>
      <field name="KEY" type="basic" mime-type="text/plain"/>
      <field name="VALUE" type="basic" mime-type="text/plain"/>
    </complex>
  </field>
</panel>
<summary>
  <ui:label>Content Summary</ui:label>
  <field name="caption" type="basic" mime-type="text/plain"/>
  <field name="alttext" type="basic" mime-type="text/plain"/>
</summary>
</content-type>

```

Whenever a content item is created based on such a content type definition, and the binary content referenced is a JPEG file, then any EXIF or IPTC metadata found in the file is automatically extracted and injected into the **COM.ESCEMIC.DEFAULTMETADATA** field. An array element is created for each metadata item: the name of the item is injected into the **KEY** subfield and the content of the item is injected into the **VALUE** subfield.

If you want better control over this process (if, for example, you want to select what metadata items are injected and to inject them into specific fields), then you will need to create your own plug-in. For a description of how to this, see [section 6.2](#).

6.2 The Plug-in API

The metadata extraction plug-in API allows you to create plug-ins for both metadata extraction (reading metadata from files) and metadata injection (inserting metadata into content item fields). Both processes are managed by the **MetadataService** class, which delegates the actual work to type-specific plug-ins.

Metadata extraction is performed by implementations of the **StreamMetadataExtractor** interface. Metadata injection is performed by implementations of the **ContentMetadataInjector** interface.

The plug-in API uses the Service Provider facility as described in the JAR file specification for discovery of plug-ins (see <http://java.sun.com/j2se/1.4.2/docs/guide/jar/jar.html#Service%20Provider>).

A plug-in, then, usually consists of a single JAR file containing:

- One or more implementation classes
- A service provider (**spi**) class for each implementation

Each service provider class is responsible for instantiating the corresponding implementation class.



Metadata Extraction and Storage Plug-ins

The Content Engine also supports plug-in implementations of the **Storage** interface, which allows developers to add support for different file storage back ends. Third-party implementations of **Storage** should ideally provide a storage-specific **StorageMetadataExtractor** implementation in order to enable the use of metadata stored outside the data stream itself (usually file attributes such as size, last modification date and so on).

6.2.1 Metadata Injection Example

This example metadata injection plug-in inserts the standard IPTC **Caption/Abstract** field into a content item's **caption** field. The plug-in consists of the following items:

- An implementation of the **ContentMetadataInjector** interface called **CustomMetadataInjector**.
- A corresponding **spi** class called **CustomMetadataInjectorSpi** that extends **ContentMetadataInjectorSpi**.
- A resource file to configure the plug-in.

6.2.1.1 The Implementation Class

The following listing shows the content of **example/CustomMetadataInjector.java**, which does the actual injection. It is a very simple example but could easily be extended to inject more fields, or even automatically look up the fields in content items, and inject data into fields with names that match metadata field names.

 Note that the class extends **AbstractMetadataBase**. This saves some work implementing the provider-specific details.

```
package example;

import neo.xreditsys.api.ArticleTransaction;
import neo.xreditsys.content.type.Field;

class CustomMetadataInjector extends AbstractMetadataBase implements ContentMetadataInjector {
    static final String CAPTION_FIELD_NAME = "CAPTION";

    CustomMetadataInjector(final DefaultMetadataInjectorSpi pProvider) {
        super(pProvider);
    }

    public void inject(ArticleTransaction pContent, Map<String, List<Object>> pMetadata) {
        Field defaultMetadataField = pContent.getArticleType().getField(CAPTION_FIELD_NAME);
        List<Object> captions = pMetadata.get("Caption/Abstract"); // Standard IPTC field
        if (defaultMetadataField != null && captions != null) {
            pContent.setFieldValue(METADATA_FIELD_NAME, toNLSV(captions));
        }
    }

    private static String toNLSV(final List<Object> pValues) {
        StringBuilder buffer = new StringBuilder();
        for (Object value : pValues) {
            if (buffer.length() > 0) {
                buffer.append("\n ");
            }
            buffer.append(value);
        }
    }
}
```

```
    }  
    return buffer.toString();  
}
```

6.2.1.2 The Spi Class

The following listing shows the content of `example/CustomMetadataInjectorSpi.java`.

```
package example;  
  
import com.escenic.storage.metadata.spi.ContentMetadataInjectorSpi;  
  
public class CustomMetadataInjectorSpi extends ContentMetadataInjectorSpi {  
    public CustomMetadataInjectorSpi() {  
        super("Example Inc.", "1.0");  
    }  
  
    public ContentMetadataInjector createMetadataInjector() {  
        return new CustomMetadataInjector(this);  
    }  
}
```

6.2.1.3 The Resource File

The following listing shows the content of `/META-INF/services/com.escenic.storage.metadata.spi.ContentMetadataInjectorSpi`.

```
example.CustomMetadataInjectorSpi
```

6.2.1.4 Deploying The Plug-in

To deploy the plug-in:

1. Compile the Java source files.
2. Package the resulting class files and the resource file together in a JAR file.
3. Add the JAR file to the classpath.
4. Restart the Content Engine.

7 Servlet Filters

Servlet filters are Java programs that can be wrapped around a J2EE servlet in order to modify:

- Inbound requests
- Outbound responses

Servlet filters are easy to write and provide a simple mechanism by which common functionality can be encapsulated for re-use in different contexts. They have standardized input/output interfaces which allows them to be assembled into chains. An inbound request can be passed through a chain of filters to prepare it for processing by a servlet. The response generated by the servlet is then passed back as a return value through the same filter chain. This means that each filter in the chain can be used to modify the inbound request or the outbound response or both.

The servlet filters used by a particular web application and the order in which they are called is defined in the application's **WEB.XML**.

Servlet filters play an important role in Escenic applications. Most importantly, servlet filters are used to:

- Parse incoming request URLs
- Create the section and article beans needed to generate appropriate responses
- Add the created beans to the request as request scope attributes

A standard set of filters is supplied with the Content Engine, along with a default **WEB.XML** file that specifies:

- The filters to be used
- The parameters required by the filters
- The order in which the filters are to be called

You can modify the default filter chain by inserting filters of your own, either in addition to the standard filters or as replacements for them. Some possible reasons for modifying the filter chain might be:

- You want to use a custom algorithm to derive the publication name from the content of the request URI
- You want to transform responses returned to certain types of devices (reformat them for small screens, for example)
- You want to carry out cache filtering

In order to be able to do this successfully, you need to know what the standard filters delivered with the Content Engine do, and how they are organized.

If you look in the default **WEB.XML** supplied with the Content Engine, you will see that the Escenic filter chain contains the following filters:

- **ECEProfileFilter**
- **BootstrapFilter**
- **TimerFilter**
- **EscenicStandardFilterChain**
- **CacheFilter**

They are described in the following sections.

7.1 ECEProfileFilter

The **ECEProfileFilter** filter ensures that profiling output is grouped by template. Note, however, that it only works when `jsp:include` is used to transfer control between templates. For general information about Escenic profiling, see [chapter 4](#).

The **ECEProfileFilter** is configured in **WEB.XML** as follows:

```
<filter>
  <filter-name>ECEProfileFilter</filter-name>
  <filter-class>com.escenic.servlet.ECEProfileFilter</filter-class>
</filter>

<filter-mapping>
  <filter-name>ECEProfileFilter</filter-name>
  <url-pattern>/*</url-pattern>
  <dispatcher>INCLUDE</dispatcher>
</filter-mapping>
```

 Note that the `filter` element must be followed by a `filter-mapping` element defining how the filter is to be applied.

Configuration Parameters

This filter has no configuration parameters.

Request Scope Attributes

This filter sets no request scope attributes.

7.2 BootstrapFilter

The **BootstrapFilter** is intended to protect the Content Engine from traffic during startup. When the Content Engine is started or restarted, its bootstrap service primes the system by sending fake requests to itself. These requests force the Content Engine to initialize various subsystems, load content from the database and so on, until the system is fully operational and ready to respond to requests at full speed.

While this process is underway, the **BootstrapFilter** only passes through requests from the bootstrap service. For all other requests it returns an HTTP 503 response (Service Unavailable).

The **BootstrapFilter** is configured in **WEB.XML** as follows:



```
<filter>
  <filter-name>BootstrapFilter</filter-name>
  <filter-class>
    com.escenic.presentation.servlet.BootstrapFilter
  </filter-class>
  <init-param>
    <param-name>oncePerRequest</param-name>
    <param-value>true</param-value>
  </init-param>
</filter>
```

Configuration Parameters

oncePerRequest

If set to **true**, then the filter is only executed for the initial request. If set to **false**, then the filter is executed for every request, which on some servers may mean that it is re-applied each time an include operation calls a new JSP file.

Request Scope Attributes

This filter sets no request scope attributes.

7.3 TimerFilter

The **TimerFilter** has the following functions:

- Measuring template performance
- Throttling requests should the server be overloaded
- Recording all requests along with their start and end time

The throttling function is carried out by the Escenic throttling service, /**neo/io/services/JspThrottleService**. If the number of concurrent requests exceeds a specified maximum, then the throttling service informs **TimerFilter**, which rejects any excess requests by returning HTTP 503 (Service Unavailable) responses.

The **TimerFilter** is configured in **WEB.XML** as follows:

```
<filter>
  <filter-name>TimerFilter</filter-name>
  <filter-class>neo.servlet.TimerFilter</filter-class>
  <init-param>
    <param-name>collector</param-name>
    <param-value>/neo/io/reports/HitCollector</param-value>
  </init-param>
</filter>
```

Configuration Parameters

collector

The component path of the hit collector that records all requests to the publication. The default value is **/neo/io/reports/HitCollector**.

Request Scope Attributes

This filter sets no request scope attributes.

7.4 EscenicStandardFilterChain

The `EscenicStandardFilterChain` is different from all the other filters specified in `WEB.XML`. It is not a single filter, but a chain of "subfilters" called **processors**. Most of the important filter functionality is actually performed by the processors in this chain. The reason that the filters have been packed together like this is as follows:

- New releases of the Content Engine sometimes involve changes to this part of the filter chain, needed to support new functionality.
- `WEB.XML` is part of the publication application, not the Content Engine.
- If all the processors were not packed into this single "master filter", then upgrading to new versions of the Content Engine would often require customers to make changes to their `WEB.XML` file(s).

The `EscenicStandardFilterChain` is configured in `WEB.XML` as follows:

```
<filter>
  <filter-name>EscenicStandardFilterChain</filter-name>
  <filter-class>com.escenic.presentation.servlet.CompositeFilter</filter-class>
  <init-param>
    <param-name>oncePerRequest</param-name>
    <param-value>true</param-value>
  </init-param>
  <init-param>
    <param-name>chain</param-name>
    <param-value>/com/escenic/servlet/StandardFilter</param-value>
  </init-param>
</filter>
```

Configuration

`oncePerRequest`

If set to `true`, then the filter is only executed for the initial request. If set to `false`, then the filter is executed for every request, which on some servers may mean that it is re-applied each time an include operation calls a new JSP file.

`chain`

The path of the component that manages the chain of processors. There is no default value for this parameter: it **must** be specified, and should normally be set to `/com/escenic/servlet/StandardFilter`.

Request Scope Attributes

This filter sets no request scope attributes itself. However, the filter chain processors it executes do set request scope attributes. See the descriptions of the individual processors for details.

7.4.1 The Filter Chain Processors

The main practical difference between the processors called by `EscenicStandardFilterChain` and ordinary servlet filters is that they are not configured in `WEB.XML`.

Configuring The Filter Chain Sequence



The sequence of processors executed by `EscenicStandardFilterChain` is defined in a system file called `StandardFilter.properties`. For information on modifying the standard processor sequence, see [section 7.6.2](#)

Configuring The Processors

The standard way to configure a processor is to create a file called `component-name.properties` in your publication's `WEB-INF/localconfig` folder and add a `name=value` property setting to it. A few of the most commonly-used properties can also be set more simply by creating a `default.properties` file in `WEB-INF/localconfig` and adding a similar name/value pair to this file. For a detailed description of how to configure these processors and other publication web application components, see [chapter 13](#).

The following processor descriptions list all the properties that can be set for each processor.

Processor Descriptions

The processors are described in the following sections. The descriptions contain the following sections:

Component Name

The name of the component that implements the processor functionality.

Description

A general description of the processor.

Configuration

Describes properties that you can set to can modify the behavior of the processor (see **Configuring Processors** above). If there is a `default.properties` parameter for setting a property, then this is described as well.

Input Request Scope Attributes

The request scope attributes read by the filter.

Output Request Scope Attributes

The request scope attributes written by the filter, and what they are used for.

7.4.1.1 PublicationResolverProcessor

Component Name

`/com/escenic/servlet/PublicationResolverProcessor`

Description

This processor attempts to find out the name of the publication the request should be directed to. It does this by applying the following rules:

1. If the request scope attribute `com.escenic.publication.name` is already set, then this value is kept. This means that you

can override this processor by inserting your own filter before the `EscenicStandardFilterChain` and setting `com.escenic.publication.name`. You might, for example, choose to set the publication name based on the host name in the request URL.

2. If the processor's `publicationName` property has been set, then this value is used.
3. If the `publicationName` property is not defined, then the web application context path is used (minus the initial slash).
4. If the application server's context root is empty, (in other words, the web application is mounted on the root context), then `com.escenic.publication.name` is not set.

The usual case is that the publication name is used as the web application context path so that the `publicationName` property does not need to be set and rule 3 sets `com.escenic.publication.name`.

 If no publication name can be determined then `com.escenic.publication.name` is not set. In this case the remaining processors in the chain do nothing, and the request is handled by the servlet container.

Configuration

The following property can be defined in `WEB-INF/localconfig/com/escenic/servlet/PublicationResolverProcessor.properties`:

`publicationName`

The name of the publication. If not set, the web application's context path is used as the publication name. This property can also be set by adding a `publication-name` parameter to `/WEB-INF/localconfig/defaults.properties` (see [section 13.3](#) for details).

Input Request Scope Attributes

None

Output Request Scope Attributes

`com.escenic.publication.name`

`PublicationResolverProcessor` writes the name of the publication to this attribute, unless it is already set.

`com.escenic.context.path`

`PublicationResolverProcessor` writes the path of the requested item to this attribute, unless it is already set.

"Path" here means the relative path from the publication root to the requested item. The path for the soccer section of a newspaper, for example, might be `/sports/soccer/` while the path of a news content item might be `/news/article123.ece`.



7.4.1.2 SectionResolverProcessor

Component Name

`/com/escenic/servlet/SectionResolverProcessor`

Description

This processor attempts to find out:

- The id of the section the request should be directed to
- Whether or not the request is a section request

It does this by analyzing the content of the request scope attribute `com.escenic.context.path`.

This filter takes the path (stored in `com.escenic.context.path`) and decodes it as far as possible as the section tree, based on the publication's section hierarchy. It stores the section ID in the `com.escenic.context.section.id` request attribute.

Configuration

The following property can be defined in `WEB-INF/localconfig/com/escenic/servlet/SectionResolverProcessor.properties`:

APIResolver

The resolver to use. The resolver must belong to the class `com.escenic.servlet.APIResolver`. If not set, the Content Engine's default `APIResolver` is used.

Input Request Scope Attributes

`com.escenic.publication.name`

`com.escenic.context.path`

Output Request Scope Attributes

`com.escenic.context`

If the request is a section request, then the `SectionResolverProcessor` sets this attribute to `sec`.

`com.escenic.context.section.id`

The id of the section the request is to be directed to.

`com.escenic.context.path`

The `SectionResolverProcessor` overwrites this attribute with a shorter path from which the section components have been removed. For example:

- If the original value was `/news/article123.ece` and `/news/` is found to identify a section in the publication, then it is now set to `article123.ece`.

- If the original value was `/sports/soccer/` and this is found to identify a section in the publication, then it is now set to an empty string.

7.4.1.3 ArticleResolverProcessor

Component Name

`/com/escenic/servlet/ArticleResolverProcessor`

Description

This processor attempts to find out:

- Whether or not the request is a content item request
- The id of the content item being requested

It does this by analyzing the content of the request scope attribute `com.escenic.context.path`.

Configuration

The following property can be defined in `WEB-INF/localconfig/com/escenic/servlet/ArticleResolverProcessor.properties`:

`serverConfig`
To be supplied.

Input Request Scope Attributes

`com.escenic.context.path`
To be supplied.

Output Request Scope Attributes

`com.escenic.context`
If the request is a content item request, then the `ArticleResolverProcessor` sets this attribute to `art`. If not, then the attribute is not set.

`com.escenic.context.article.id`
The content item id extracted from `com.escenic.context.path`. If no content item id could be extracted then the attribute is not set.

`com.escenic.context.path`
If the `SectionResolverProcessor` was able to extract a content item id from the original value of this attribute, then it is now set to an empty string. If no content item ID could be extracted, then the attribute is left unmodified.

7.4.1.4 PresentationProcessor

Component Name

`/com/escenic/servlet/PresentationProcessor`



Description

This processor attempts to create the implicit beans needed by the templates that will handle the request, using the information in the request scope attributes created by preceding processors.

- `com.escenic.publication.name` (the publication name)
- `com.escenic.context` (set to either `sec` or `art`)
- `com.escenic.context.section.id` - the section id
- `com.escenic.context.article.id` - the content item id (for content item requests only)

Configuration

The following properties can be defined in `WEB-INF/localconfig/com/escenic/servlet/PresentationProcessor.properties`:

`APIResolver`

The resolver to use. The resolver must belong to the class `com.escenic.servlet.APIResolver`. If not set, the Content Engine's default `APIResolver` is used.

`presentationLoader`

The loader to use. The presentation loader must be of class `neo.xredsys.presentation.PresentationLoader`. If not set, the Content Engine's default `PresentationLoader` is used.

`useRequestHeaderForRemoteAddr`

To be supplied.

Input Request Scope Attributes

`com.escenic.publication.name`

To be supplied.

`com.escenic.context`

To be supplied.

`com.escenic.context.section.id`

To be supplied.

`com.escenic.context.article.id`

To be supplied.

Output Request Scope Attributes

`com.escenic.context.publication`

A `neo.xredsys.api.Publication` bean, loaded using the publication name in `com.escenic.publication.name`.

`com.escenic.context.section`

A `neo.xredsys.api.Section` bean, loaded using the section id in `com.escenic.context.section.id`.

`com.escenic.context.article`

A `neo.xredsys.presentation.PresentationArticle` bean, loaded using the content item id in `com.escenic.context.article.id`. This request attribute is only set for requests in which `com.escenic.context` is set to `art`.

7.4.1.5 PreviewProcessor

Component Name

`/com/escenic/servlet/PreviewProcessor`

Description

This processor is currently a placeholder and does nothing.

Configuration

There are no properties to set for this processor.

Input Request Scope Attributes

None.

Output Request Scope Attributes

None.

7.4.1.6 AgreementProcessor

Component Name

`/com/escenic/servlet/AgreementProcessor`

Description

This processor checks whether or not the Escenic objects in the request (section, content item and possibly any related images or multimedia objects) require a valid agreement to be viewed. If an agreement is required, the appropriate `AgreementProvider` is called. The processor then sets the `com.escenic.agreement.accessDenied` request attribute to indicate the result of the check.

Configuration

The following property can be defined in `WEB-INF/localconfig/com/escenic/servlet/AgreementProcessor.properties`:

`alwaysAllowPreview`

Set this parameter to `false` if you want to enforce agreements for previews. If not set, a default value of `true` is used.

Input Request Scope Attributes



`com.escenic.context.section`
To be supplied.

`com.escenic.context.article`
To be supplied.

Output Request Scope Attributes

`com.escenic.agreement.accessDenied`
Is set to `true` if the requirements for displaying the Escenic objects in the request have not been satisfied.

7.4.1.7 TemplateDispatchResolver

Component Name

`/com/escenic/servlet/TemplateDispatchResolver`

Description

This processor inspects the request scope attribute `com.escenic.context.path` to see whether or not the request has been successfully parsed by the preceding processors:

- If `com.escenic.context.path` is empty, then the request was successfully parsed, so the processor then checks whether or not the requested object exists/is accessible and:
 - If it exists and is available, dispatches the request to `/template/common.jsp` or the handler defined with the `page` property.
 - If it does not exist, returns an HTTP 404 response (Not Found).
 - If it exists but is unavailable, returns an appropriate HTTP response.
- If `com.escenic.context.path` contains anything at all, then the request was not parsed successfully. However, the request will still be dispatched to the web application, allowing it to determine how to deal with the invalid request.

Configuration

The following property can be defined in `WEB-INF/localconfig/com/escenic/servlet/TemplateDispatchResolver.properties`:

`page`

The JSP template or other handler to which the request is to be forwarded. If not set, a default value of `/template/common.jsp` is used. This property can also be set by adding a `forward-to-page` parameter to `/WEB-INF/localconfig/defaults.properties` (see [section 13.3](#) for details).

Input Request Scope Attributes

`com.escenic.context.path`

Request Scope Attributes

This processor sets no request scope attributes.

7.5 CacheFilter

The `CacheFilter` manages a simple disk cache for images, and ensures that:

- Cached images are used if available
- All requested and currently un-cached images are cached

If an incoming request has not already been identified as a section or content item request, then the `CacheFilter` checks the publication's image cache for the requested file. If it finds the requested image, then it returns the image file, terminating the filter chain.

If it does not find the requested file in the image cache, then it calls the next filter in the filter chain. If that filter returns an image file, then it:

- Adds the image to the cache
- Returns the image

Cached images are never invalidated. This means that if an image is updated (a new version is uploaded or the original image is modified), the cached copy will not be updated. The only way to fix this problem is to physically remove the outdated copy from the cache.

The `CacheFilter` needs to be configured **before** the filters/servlets that it should filter see the [Web XML Chapter \(chapter 12\)](#) for an example.

The `CacheFilter` is configured in `WEB.XML` as follows:

```
<filter>
  <filter-name>CacheFilter</filter-name>
  <filter-class>com.escenic.presentation.servlet.multimedia.CacheFilter</filter-class>
  <init-param>
    <param-name>oncePerRequest</param-name>
    <param-value>true</param-value>
  </init-param>
  <init-param>
    <param-name>path</param-name>
    <param-value>/var/www/images</param-value>
  </init-param>
  <init-param>
    <param-name>cache-control</param-name>
    <param-value>private;max-age=2</param-value>
  </init-param>
</filter>
```

Configuration

`oncePerRequest`

If set to `true`, then the filter is only executed for the initial request. If set to `false`, then the filter is executed for every request, which on some servers may mean that it is re-applied each time an include operation calls a new JSP file.

**path**

The path of the image cache folder. If not specified, then the cache location is determined by `filePublicationRoot`. `filePublicationRoot` is a server property, defined in the `ServerConfig.properties` file. The cache will by default be located under `filePublicationRoot` in a folder with the name `publication-name/multimedia/dynamic`.

cache-control

The HTTP Cache-Control header to included with images served by the Content Engine. If this parameter is not specified, then a default Cache-Control header of `public;max-age=60` is included. This default value will cause browsers to cache the images for 60 seconds. The Cache-Control header format is specified in <http://www.w3.org/Protocols/rfc2616/rfc2616-sec14.html#sec14.9>.

7.6 Modifying The Filter Chain

Some publications may have special requirements that are best satisfied by modifying the filter chain: usually by adding an extra filter, or replacing one of the standard filters. The way you do this depends upon whether you need to modify the main filter chain, or the processor chain in the `EscenicStandardFilterChain`.

7.6.1 Modifying The Main Filter Chain

The main filter chain is a standard servlet filter chain, and can be modified in the normal way. To insert a filter of your own, for example, you would need to:

- Write your own filter class that implements the interface `javax.servlet.Filter`
- Declare and configure it by adding a filter element to `WEB.XML`
- Insert it into the filter chain by modifying the filter chain mapping in `WEB.XML`

For further information, see <http://java.sun.com/products/servlet/Filters.html>.

7.6.2 Modifying The Escenic Standard Filter Chain

In some cases, modifying the main filter chain may not solve your problem - you might need, for example, to insert a filter between two of the processors called by the `EscenicStandardFilterChain`. In this case you will need to:

- Create an Escenic processor class (not a filter) that performs the functions you require
- Create a set of `.properties` files that declare the processor and insert it into the processor chain at the required point
- Package these items in a JAR file
- Deploy the JAR file in your publication

7.6.2.1 Creating an Escenic Processor

An Escenic processor is a class that implements the `com.escenic.presentation.servlet.GenericProcessor` interface.

Standard servlet filters can create wrappers around the request and response objects they handle in order to add information to them. It is **not** possible to do this in Escenic processor classes.

7.6.2.2 Creating Processor .properties Files

The processors executed by the default `EscenicStandardFilterChain` are defined in a system file called `StandardFilter.properties`, which looks something like this:

```
$class com.escenic.presentation.servlet.LooseFilterChain

## config.1xx -- reserved for third parties

## config.2xx -- reserved for escenic core resolver filters
config.210 = ./PublicationResolverConfig
config.220 = ./SectionResolverConfig
config.230 = ./ArticleResolverConfig

## config.3xx -- reserved for third parties

## config.4xx -- reserved for escenic core presentation filters
config.400 = ./PresentationConfig

## config.5xx -- reserved for third parties

## config.6xx -- reserved for escenic core authentication and authorization filters
config.600 = ./PreviewConfig
config.601 = ./AgreementConfig

# config.7xx -- reserved for third parties

# config.8xx -- reserved for escenic core dispatcher filters
config.800 = ./TemplateDispatchConfig

# config.9xx -- reserved for third parties after
#             dispatching in case we didn't dispatch!
```

This file specifies the names of a series of configuration files, which in turn identify and configure the classes implementing the processors in the chain. The file `PublicationResolverConfig.properties`, for example, looks like this:

```
$class = com.escenic.presentation.servlet.LooseFilterChain$FilterChainConfigBuilder

filterName = PublicationResolver

filter = ./PublicationResolverFilter

# must be an absolute path
initParameter.processorName = /com/escenic/servlet/PublicationResolverProcessor
```

The other files that are involved in configuring the `PublicationResolverProcessor` are:

PublicationResolverFilter.properties
which contains the following:



```
$class = com.escenic.presentation.servlet.ProcessorFilter
```

PublicationResolverProcessor.properties

which contains the following:

```
$class = com.escenic.presentation.servlet.PublicationResolverProcessor
publicationName = ${/defaults.publication-name}
```

As you can see, the last of these files contains a reference to the class which actually implements the processor, **com.escenic.presentation.servlet.PublicationResolverProcessor**.

In order to insert your own processor into the default chain, you need to create a corresponding set of files of your own. If, for example, you have created a processor class called **com.mycompany.mysterious.MysteriousProcessor**, and you want to insert it into the processor chain immediately after the standard **ArticleResolver** processor, then you would need to create the following files:

StandardFilter.properties

which should contain something like:

```
config.310 = /com/mycompany/mysterious/MysteriousProcessorConfig
```

This file is merged with the system **StandardFilter.properties** file by the Content Engine. The parameter name **config.310** ensures that the processor is executed after the **ArticleResolver** processor (at slot 230) and before the **PresentationProcessor** (at slot 400). It is also within the 300-399, which is reserved for use by third-party developers. You should only use slots in these third-party ranges, because then you are sure that your processor will not come into conflict with any standard processor added in later versions of the Content Engine.

MysteriousProcessorConfig.properties

which should contain something like:

```
$class = com.escenic.presentation.servlet.LooseFilterChain$FilterChainConfigBuilder
filterName = Mysterious
filter = ./MysteriousFilter
# must be an absolute path
initParameter.processorName = /com/mycompany/mysterious/MysteriousProcessor
```

This file has a standard structure and contents. The parts that change from processor to processor are highlighted in bold.

MysteriousFilter.properties

which must contain the following line:

```
$class = com.escenic.presentation.servlet.ProcessorFilter
```

This file always has exactly the same contents. Its name must match the name specified with the filter parameter in the **MysteriousProcessorConfig.properties** file.

MysteriousProcessor.properties

which must at least contain the following line:

```
$class = com.mycompany.mysterious.MysteriousProcessor
```

If your processor has configuration parameters, then the file can also contain settings for these parameters, for example:

```
$class = com.mycompany.mysterious.MysteriousProcessor
theAnswer = 42
```

7.6.2.3 Packaging The Processor

The processor class and all the properties files that configure it must be correctly packaged in a JAR file before you can deploy the processor. To package it you must copy the files into a directory structure that matches the package name of your processor class and the package naming conventions required by the Content Engine's plugin architecture.

For the example in the previous section, you need to create the following structure:

```
com
+-escenic
| +-servlet
|   +-plugin-config
|     +-com
|       +-escenic
|         +-servlet
|           +-StandardFilter.properties
|             +-mycompany
|               +-mysterious
|                 +-MysteriousProcessorConfig.properties
|                 +-MysteriousFilter.properties
|                 +-MysteriousProcessor.properties
+-mycompany
  +-mysterious
    +-MysteriousProcessor.class
```

Once you have created the structure and you are sure that:

- The Escenic parts of it match the structure above
- Your own parts of the structure match the package name you have used

pack it in a JAR file. (To do this, use an archiving utility that is capable of creating JAR files.)

7.6.2.4 Deploying the processor

To deploy the processor, copy the JAR file into the `WEB-INF/lib` folder of your publication and redeploy the publication.

8 Decorators

A **decorator** is a commonly-used Java programming pattern that enables programmers to extend the functionality of an object in a way that is transparent to users of the object. Decorators are used in the Content Engine to add functionality to content items and section pages. You need Java programming skills to create a decorator, but not to make use of one in your templates.

Decorators can be used in Escenic applications to modify the properties or field values returned from content items in various ways. The code examples used in this section, for example, can be used to:

- Automatically convert a content item's title to upper case
- Automatically expand any **collection content items** (see [section 8.3](#)) in a related content item's relation

All the functions that can be performed with decorators can also be performed in other ways. You could, for example, either include in-line Java code in your templates or write your own JSP tags. However, using decorators has a number of advantages:

- No in-line code ensures simpler, more legible templates
- The template developer does not need to remember to use special tags
- Decorators only need to be declared in the **content-type** resource: once this has been done they are fully automatic and completely transparent
- They can be re-used simply by adding declarations to the **content-type** resource

Obviously, decorators are not always the right solution. If, for example, you want content item titles to be upper case in some contexts but not in others, then you would need to use in-line code or a custom tag rather than a decorator. And in other cases it may be better to use **transaction filters** (see [chapter 10](#)).

Note that if you are using memcached, the decorator must be serializable.

8.1 Creating A Decorator

An Escenic content item decorator is a Java class that:

- Extends `neo.xreditsys.presentation.PresentationArticleDecorator`
- Contains a `get` method for each content item property it is to modify

The following example shows a content item decorator that converts a content item's `title` property to upper case:

```
package com.mycompany.decorators;

import neo.xreditsys.presentation.*;

public class TitleToUpperCase extends PresentationArticleDecorator {
```

```
public TitleToUpperCase(PresentationArticle pa) {
    super(pa);
}

public String getTitle() {
    String title = super.getTitle().toUpperCase();
    return title;
}
}
```

The `PresentationArticleDecorator` class implements the `PresentationArticle` interface. Since the `TitleToUpperCase` class extends `PresentationArticleDecorator`, it can be used to override methods in the standard implementation of `PresentationArticle`. In this case, the `getTitle()` method will override the standard `PresentationArticle.getTitle()` method.

The `TitleToUpperCase` constructor accepts a `PresentationArticle` bean as input parameter and stores it in the property `super` (i.e, superclass). When a bean for a content type that is configured to use the `TitleToUpperCase` decorator is created, the Content Engine:

- Creates a standard `PresentationArticle` bean
- Creates a `TitleToUpperCase` bean, passing in the `PresentationArticle` bean as its constructor parameter

When the article's `getTitle()` method is called, `TitleToUpperCase.getTitle()`:

1. Calls `super.getTitle()` method to get the `title` property
2. Calls `String.toUpperCase()` to convert the title to upper case
3. Returns the result

Note the following points:

- A decorator changes the behavior of the content item bean itself. It does not matter whether you access the `title` property via a JSP tag or in-line java - it will always be returned in upper case.
- Only the specified property is affected. If you access the title field directly via the `article:field` tag or the Java `getFieldElement()` method, then it will not be converted to upper case.

Before a decorator can be used it must be:

- Compiled
- Added to the Content Engine's classpath

To compile a decorator you need the Escenic JAR file `engine-core-5.2.7.2.jar` in your classpath.



8.2 Using A Decorator

Once you have created a decorator class, all you need to do to use it is add it to a content type definition in the `content-type` resource using the `decorator` element. For example:

```
<content-type name="xyz">
  ...
  <ui:decorator class="com.mycompany.decorator.TitleToUpperCase"/>
  ...
</content-type>
```

Once you have done this, the decorator will automatically take effect for all content items of that type. You can add the decorator to as many content type definitions as you want.

You can add more than one decorator to a content type. For example:

```
<content-type name="xyz">
  ...
  <ui:decorator class="com.mycompany.decorator.TitleToUpperCase" />
  <ui:decorator class="com.mycompany.decorator.TitleTrim" />
  ...
</content-type>
```

Note the following:

- The `decorator` element belongs to the `interface-hints` namespace, which means that its name will usually be preceded by a prefix (`ui` in the examples above) declared at the start of the `content-types` resource file. For full descriptions of the `content-type` and `decorator` elements, see the **Esenic Content Engine Resource Reference**.
 - The `decorator` element has an optional `name` attribute that you can use for documentation purposes if you wish.
 - Multiple decorators are executed in the order they appear in the content type definition. This may sometimes be significant.
-

8.3 A Complex Decorator Example

In order to understand the point of this example, you need to know what **collection content items** are.

A collection content item is a content type (defined in the `content-type` resource) that has one or more relations, but no ordinary fields. It is not intended for display in a publication, but is created for internal use by editorial staff as a container for relations. Imagine, for example a group of background articles on a particular subject that all need to be added as related content items to any new article written on the same subject. Instead of requiring editorial staff to individually add each related content item, a manager can create a collection content item and add all the related content items to this one content item. Editorial staff can then add all the relations in one go by simply adding the collection content item to their new content items.

The catch, of course, is that the template programmer must then "unpack" these collection content items, converting the single related collection content item back into a series of ordinary related content items in order to display them properly. This could be done using custom tags or in-line Java, but this example shows how it can be done using a decorator class.

```

package com.mycompany.decorators;

import neo.xreditsys.presentation.*;
import java.util.*;

public class UnpackCollectionArticles
    extends PresentationArticleDecorator {

    /* to hold the names of all "collection" content types */
    private Set articleTypeNames;

    public UnpackCollectionArticles(PresentationArticle pa) {
        super(pa);
        articleTypeNames = new HashSet();

        /* add the content type names (only one in this case) */
        articleTypeNames.add("collection");
    }

    public List getArticles() {
        /* get all of this article's related articles */
        List relatedArticles = new ArrayList(super.getArticles());

        /* get this article's related collection articles WHY IS THIS DIFFERENT */
        List collectionArticles = super.getArticles(articleTypeNames);

        /* remove the collection articles from the list of related articles */
        relatedArticles.removeAll(collectionArticles);

        /* for each related collection article... */
        Iterator i = collectionArticles.iterator();
        while(i.hasNext()) {
            PresentationArticle a = (PresentationArticle)i.next();

            /* ...add all the collection article's related articles to
               THIS article's related articles list */
            relatedArticles.addAll(a.getArticles());
        }
        return relatedArticles;
    }
}

```

You can use a single decorator class to modify more than one method. It might be the case, for example, that your collection content items are used to hold lists of related images as well as lists of related content items. In this case you would want to override the `getImages()` method as well as `getArticles()`:

```

public List getImages() {
    List relatedImages = new ArrayList(super.getImages());
    List collectionArticles=super.getArticles(articleTypeNames);
    Iterator i = collectionArticles.iterator();
    while(i.hasNext()) {
        PresentationArticle a = (PresentationArticle)i.next();
        relatedImages.addAll(a.getImages());
    }
    return relatedImages;
}
}

```



9 Event Listeners

An **event listener** is a Java object that listens for Content Engine **events** and responds to them. Events in this context are messages that are broadcast by the Content Engine whenever certain things occur. An event is broadcast, for example, whenever an Escenic object is created or modified. You can use event listeners to extend and customize the Content Engine in various ways. You might, for example, create an event listener that listens for the creation of new content items and automatically exports the newly-created content items to an external system.

An Escenic event is an object of the class `neo.xreditsys.api.IOEvent`. An `IOEvent` object has a `type` property that indicates what kind of event it represents. An event generated when an object is created has a type of `OBJECT_CREATED`, for example. An `IOEvent` object also has a number of other properties that are used to hold information about the event that has occurred. An event listener can therefore interrogate an event for the information it needs to respond to the event.

9.1 Making An Event Listener

To make an event listener, create a Java class that extends one of the following abstract classes:

`neo.xreditsys.api.EventListenerService`

Use this class if you are interested in both local events and remote events .

`neo.xreditsys.api.LocalEventListenerService`

Use this class if you are only interested in local events (remote events are filtered out by this class).

Whichever base class you use, you will need to implement the following methods:

`acceptEvent(event)`

This method is defined in the `neo.xreditsys.api.IOEventFilter` interface. Use it to determine what event types your listener will respond to. Return `true` for all events you want your listener to respond to.

`handleEvent(event)`

This method is defined in the `neo.xreditsys.api.IOEventListener` interface. It will be called every time your `acceptEvent()` method returns `true`. Use it to perform whatever action you want to be carried out for accepted events.

Before an event listener can be used it must be:

- Compiled
- Added to the Content Engine's classpath

To compile an event listener you need the Escenic JAR file `engine-core-5.2.7.2.jar` in your classpath.

9.2 Using An Event Listener

To be supplied.

10 Transaction Filters

A **transaction filter** is a user-defined function that gets executed whenever certain Content Engine operations (or **transactions**) are performed, and can thereby modify the outcome of those operations. The transactions that can be modified in this way are:

- Object creation
- Object update
- Object deletion

A transaction filter is implemented as a Java class. It is in many ways similar to an event listener (see [chapter 9](#)). However, where an event listener performs an operation immediately **after** some Content Engine operation has been carried out, a transaction filter is executed during the operation itself. A transaction filter can, for example:

- Modify the content of a content item as it is being saved
- Prevent a content item from being deleted unless certain criteria are met

10.1 Making A Transaction Filter

To make a transaction filter, create a Java class that extends the abstract class `neo.xreditsys.api.TransactionFilterAdapter`. This is a convenience class containing empty "do nothing" implementations of the methods defined in the `neo.xreditsys.api.TransactionFilter` interface:

doCreate(object)

which is executed immediately before a new object is saved.

doUpdate(object)

which is executed immediately before changes to an existing object are saved.

doDelete(object)

which is executed immediately before an existing object is deleted.

isEnabled()

which is called by the Content Engine to determine whether or not the filter is currently enabled.

All you need to do in your class is re-implement the "do" method(s) that you are interested in. The `object` parameter passed in to these methods is a `neo.xreditsys.api.IOTransaction` object and represents the object being created, updated or deleted. You can both query this object and modify it. In the case of create and update transactions, any changes you make to it will be reflected in the saved object.

Here is an example transaction filter called `com.mycompany.transactionFilters.WordCount` that:

- Counts the words in the body of a newly-created or updated content item
- Writes the word count to one of the content item's fields

```

package com.mycompany.transactionFilters;

import neo.xredsys.api.*;

public class WordCount extends TransactionFilterAdapter {

    public WordCount() {
    }

    public void doCreate(IOTransaction pObject) throws FilterException {
        if (pObject instanceof ArticleTransaction) {
            countWords((ArticleTransaction)pObject);
        }
    }

    public void doUpdate(IOTransaction pObject) throws FilterException {
        if (pObject instanceof ArticleTransaction) {
            countWords((ArticleTransaction)pObject);
        }
    }

    public void countWords(ArticleTransaction pArticle) {
        String original = pArticle.getElementText("body");
        int count = new StringTokenizer(original).countTokens();
        pArticle.setElementText("wordcount", Integer.toString(count));
    }
}

```

Only update and create operations are of interest here, so the class does not contain an implementation of `doDelete()`. The `doCreate()` and `doUpdate()` methods are identical: they simply check if the transaction object is a content item and if it is, call `countWords()`. `countWords()` counts the words in the content item's `body` field and writes the result to the `wordcount` field.

This version of the filter cannot easily be re-used because the field names `body` and `wordcount` are hard-coded. It is better to create properties for these field names:

```

private String mFieldToCount = "body";
private String mFieldToUpdate = "wordcount";

public void setFieldToCount(String pFieldToCount) {
    mFieldToCount = pFieldToCount;
}

public String getFieldToCount() {
    return mFieldToCount;
}

public void setFieldToUpdate(String pFieldToUpdate) {
    mFieldToUpdate = pFieldToUpdate;
}

public String getFieldToUpdate() {
    return mFieldToUpdate;
}

```

and modify the `countWords()` method accordingly:

```

public void countWords(ArticleTransaction pArticle) {
    String original = pArticle.getElementText(getFieldToCount());
    int count = new StringTokenizer(original).countTokens();
    pArticle.setElementText(getFieldToUpdate(), Integer.toString(count));
}

```



If you do this, then the field names can be configured externally by setting parameters - see [section 10.2](#) for further information.

The `countWords()` method is a very rudimentary word counter and not recommended for real use.

Before a transaction filter can be used it must be:

- Compiled
- Added to the Content Engine's classpath

To compile a transaction filter you need the Escenic JAR file `engine-core-5.2.7.2.jar` in your classpath.

10.2 Using A Transaction Filter

For detailed information about configuration files, configuration layers and an explanation of the `engine-config` placeholder used in the file paths in this section, see the **Escenic Content Engine Server Administration Guide**.

The transaction filters executed by the Content Engine are defined in a configuration file called `engine-config/io/api/ObjectUpdater.properties`. To enable the `WordCount` filter, therefore, you must add a declaration to this file in one or more of your configuration layers. For example:

```
filter.wordcount=/com/mycompany/WordCountFilter
```

You must also create a properties file for the filter called `WordCountFilter.properties`, and save it in the location you have specified in the same configuration layer(s) - `engine-config/com/mycompany/WordCountFilter.properties` in this case. The file must at least contain a `$class` entry specifying the class that implements the filter:

```
$class=com.mycompany.WordCount
```

You can also set class properties in this file. For example:

```
$class=com.mycompany.WordCount  
  
fieldToCount=main  
fieldToUpdate=articleLength
```

Once you have added these configuration files, word counts will be added to any content items that are created or saved. The `TransactionFilterAdapter` class has an `enabled` property that can be used to enable/disable filters. You can therefore disable any filter by adding

```
enabled=false
```

to the properties file.

If no other transaction filters or post-transaction filters have been defined, then there may be no *engine-config/io/api/ObjectUpdater.properties* file. In this case you must create one.

10.3 Error Handling

You can deal with errors in your transaction filters by raising `neo.xreditsys.api.FilterException` exceptions.

11 Post-transaction Filters

The previous two chapters described two types of user-defined functions called **event listeners** and **transaction filters**. Post transaction filters provide a third way to implement your own functions. The differences between the three types can be summarized as follows:

- An **event listener** is executed **after** a specified type of object transaction (create, update or delete) has completed. It is triggered by the event system.
- A **transaction filter** is executed **during** a specified type of object transaction (create, update or delete), before it is completed. Specifically, it is executed immediately **before** the commit that terminates the transaction, giving it the opportunity to modify the data that is committed. It is not triggered by the event system, but directly called by the Content Engine module that performs the transaction.
- A **post-transaction filter** is similar to a transaction filter in the following respects:
 - It is executed **during** an object transaction.
 - It is not event-dependent, but called directly by the Content Engine module that performs the transaction.

On the other hand, it is similar to an event listener in that it is executed right at the end of the transaction, after the object has been committed, and can therefore not affect the outcome of the transaction.

These differences mean that a **post-transaction filter** can be used to carry out the same types of function as an event listener. It is, however, likely to be somewhat more reliable than a post-transaction filter because it does not depend on the timely delivery of an event. It is possible, for example, for some other operation to be executed between the end of a transaction and the execution of a event listener triggered by it, thus affecting the operation of the event listener. On the other hand, an event listener does not need to run on the same host as the Content Engine to which it is responding, making it more flexible. In a multi-host installation, you may want an event listener running on one of your hosts to be able to respond to events triggered by any of the Content Engines in the cluster.

11.1 Making a Post-transaction Filter

To make a post-transaction filter, create a Java class that extends the abstract class `neo.xreditsys.api.PostTransactionFilterAdapter`. This is a convenience class containing empty "do nothing" implementations of the methods defined in the `neo.xreditsys.api.PostTransactionFilter` interface:

```
doPostCreate(object)
```

which is executed immediately after a new object is saved.

doPostUpdate(object)

which is executed immediately after changes to an existing object are saved.

doPostDelete(object)

which is executed immediately after an existing object is deleted.

isEnabled()

which is called by the Content Engine to determine whether or not the filter is currently enabled.

Before a post-transaction filter can be used it must be:

- Compiled
- Added to the Content Engine's classpath

To compile a post-transaction filter you need the Escenic JAR file `engine-core-5.2.7.2.jar` in your classpath.

11.2 Using a Post-Transaction Filter

For detailed information about configuration files, configuration layers and an explanation of the `engine-config` placeholder used in the file paths in this section, see the **Escenic Content Engine Server Administration Guide**.

The post-transaction filters executed by the Content Engine are defined in a configuration file called `engine-config/io/api/ObjectUpdater.properties`. To enable a filter called `MyPostTransFilter` filter, therefore, you must add a declaration to this file in one or more of your configuration layers. For example:

```
postFilter.myFilter=/com/mycompany/MyPostTransFilter
```

You must also create a properties file for the filter called `MyPostTransFilter.properties`, and save it in the location you have specified in the same configuration layer(s) - `engine-config/com/mycompany/MyPostTransFilter.properties` in this case. The file must at least contain a `$class` entry specifying the class that implements the filter:

```
$class=com.mycompany.MyPostTransFilter
```

You can also set class properties in this file. For example:

```
$class=com.mycompany.WordCount
```

```
myFilterParameter=99
```

Once you have added these configuration files the filter will be called every time a create/delete/update operation is performed. The `PostTransactionFilterAdapter` class has an `enabled` property that can be used to enable/disable filters. You can therefore disable any filter by adding



```
enabled=false
```

to the properties file.

If no other transaction or post-transaction filters have been defined, then there may be no *engine-config/io/api/ObjectUpdater.properties* file. In this case you must create one.

11.3 Error Handling

You can deal with errors in your post-transaction filters by raising `neo.xreditsys.api.FilterException` exceptions. Note that, unlike transaction filters, setting the `FilterException`'s `errorFatal` property to true will **not** cause the calling create/delete/update operation to fail.





12 The web.xml File

The **web.xml** file provides configuration and deployment information for web components in a web application. It must reside in the web application's **WEB-INF** directory. The following listing shows the content of the **web.xml** file for a standard Escenic publication.

```
<?xml version="1.0" encoding="utf-8"?>
<web-app
  xmlns="http://java.sun.com/xml/ns/j2ee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee http://java.sun.com/xml/ns/j2ee/web-
app_2_4.xsd"
  version="2.4">
  <filter>
    <filter-name>BootstrapFilter</filter-name>
    <filter-class>com.escenic.presentation.servlet.BootstrapFilter</filter-class>
    <init-param>
      <param-name>oncePerRequest</param-name>
      <param-value>true</param-value>
    </init-param>
  </filter>

  <filter>
    <filter-name>TimerFilter</filter-name>
    <filter-class>neo.servlet.TimerFilter</filter-class>
    <init-param>
      <param-name>collector</param-name>
      <param-value>/neo/io/reports/HitCollector</param-value>
    </init-param>
  </filter>

  <filter>
    <filter-name>EscenicStandardFilterChain</filter-name>
    <filter-class>
      com.escenic.presentation.servlet.CompositeFilter
    </filter-class>
    <init-param>
      <param-name>oncePerRequest</param-name>
      <param-value>true</param-value>
    </init-param>
    <init-param>
      <param-name>chain</param-name>
      <param-value>
        /com/escenic/servlet/StandardFilter
      </param-value>
    </init-param>
  </filter>

  <filter>
    <filter-name>imageVersionFilter</filter-name>
    <filter-class>com.escenic.presentation.servlet.ImageVersionFilter</filter-class>
  </filter>

  <filter>
    <filter-name>cache</filter-name>
    <filter-class>
      com.escenic.presentation.servlet.multimedia.CacheFilter
    </filter-class>
    <init-param>
      <param-name>oncePerRequest</param-name>
      <param-value>true</param-value>
    </init-param>
  </filter>

  <filter-mapping>
    <filter-name>BootstrapFilter</filter-name>
    <url-pattern>/*</url-pattern>
  </filter-mapping>

  <filter-mapping>
    <filter-name>TimerFilter</filter-name>
```

```

    <url-pattern>/*</url-pattern>
  </filter-mapping>

  <filter-mapping>
    <filter-name>EscenicStandardFilterChain</filter-name>
    <url-pattern>/*</url-pattern>
  </filter-mapping>

  <filter-mapping>
    <filter-name>cache</filter-name>
    <url-pattern>/multimedia/dynamic/*</url-pattern>
  </filter-mapping>

  <filter-mapping>
    <filter-name>cache</filter-name>
    <servlet-name>binaryFieldRetriever</servlet-name>
    <dispatcher>FORWARD</dispatcher>
  </filter-mapping>

  <filter-mapping>
    <filter-name>imageVersionFilter</filter-name>
    <servlet-name>binaryFieldRetriever</servlet-name>
    <dispatcher>FORWARD</dispatcher>
  </filter-mapping>

  <listener>
    <description>Escenic Presentation layer bootstrap listener</description>
    <listener-class>com.escenic.presentation.servlet.PresentationBootstrapper</listener-class>
  </listener>

  <servlet>
    <servlet-name>binaryFieldRetriever</servlet-name>
    <servlet-class>com.escenic.presentation.servlet.BinaryFieldRetrieverServlet</servlet-class>
    <init-param>
      <param-name>storage</param-name>
      <param-value>nursery://global/com/escenic/storage/Storage</param-value>
    </init-param>
  </servlet>

  <welcome-file-list>
    <welcome-file>index.html</welcome-file>
    <welcome-file>index.jsp</welcome-file>
  </welcome-file-list>
</web-app>

```

Lines 1 through 5 make up the standard **web.xml** header.

The **filter** elements identify the servlet filters used by the publication. You can add filter elements referencing your own custom filters here. See [chapter 7](#) for more information about servlet filters in general and descriptions of the filters supplied by Vizrt.

The **filter-mapping** elements define the sequence of the filters, which is significant. The Vizrt-supplied filters should always appear in the order shown. You can insert your own filters anywhere in the sequence so long as you do not change the sequence of the standard filters.

The **welcome-file-list** element contains a list of file names that will be assumed to represent "welcome" files.

This standard **web.xml** file can be found in the **template/WEB-INF** folder of the Content Engine distribution.

13 Publication Webapp Properties

The components of a publication web application have properties that can be modified in much the same way as Content Engine component properties. Content Engine component properties can be:

- Viewed and temporarily modified via the `escenic-admin` application's **Component Browser** option described in **Escenic Content Engine Server Administration Guide, section 2.1.13**, or
- Permanently modified by editing `.properties` files in one of the configuration layers described in **Escenic Content Engine Server Administration Guide, section 4.1**.

You can modify the behavior of the components of a publication web application in exactly the same way. The only difference is that web application configuration files are not layered: only one set of configuration files is allowed, and it must be located in the web application's `WEB-INF/localconfig` folder.

13.1 Viewing Publication Webapp Properties

To view the current properties of a publication web application component:

1. Start a browser and go to:

```
http://your-server:8080/escenic-admin/
```

where *your-server* is the domain name or IP address of the server on which the Content Engine is running.

2. Click on the **Component Browser** option (In the **Field Support** section at the bottom of the page).
3. Click on **Browse other scope** and then select the name of the publication you are interested in from the **Select a scope to browse** list.

You will then see a **Directory Listing** section containing links that you can use to navigate the component hierarchy, allowing you to view and temporarily modify component properties. For general instructions on how to use the component browser, see **Escenic Content Engine Server Administration Guide, section 2.1.13**.

The components that are most likely to be of interest are the filter chain processors described in [section 7.4.1](#). These components are all located under `com/escenic/servlet`.

13.2 Modifying Publication Webapp Properties

To modify the properties of a publication web application component you must create a `.properties` file in the publication's `WEB-INF/localconfig` folder,

and add a *name=value* property setting to it. The file must have the same name and path as the component it is modifying.

For example, if you look for the `PublicationResolverProcessor` component of a publication in the `escenic-admin` component browser, you will see that it has the path `com/escenic/servlet`. To set this component's `publicationName` property, therefore, you would need to create a file called `WEB-INF/localconfig/com/escenic/servlet/PublicationResolverProcessor.properties` containing an entry like this:

```
publicationName=mysub
```

 See [section 13.3](#) for a simpler way of setting this particular property.

In order to see the results of changing one of these properties you must redeploy the web application and restart the Content Engine.

13.3 The default.properties File

The `default.properties` file provides an easy way of setting the most commonly-used publication web application properties. Instead of creating a `.properties` file for each component to be modified, you can just create a file called `default.properties` in a publication's `WEB-INF/localconfig` folder and add *name=value* property settings to it.

You can only set the following properties in this way:

Component	Property	Name in default.properties
<code>com/escenic/servlet/TemplateDispatchResolver</code>	<code>page</code>	<code>forward-to-page</code>
<code>com/escenic/servlet/PublicationResolverProcessor</code>	<code>publicationName</code>	<code>publication-name</code>

 Note that the property names used in `default.properties` are not the same as the component properties they set.

In order to see the results of changing one of these properties you must redeploy the web application and restart the Content Engine.

14 Mail a form

This component is dependent upon the 'neo/io/sevices/MailSender' component. This must be configured first

- A html form with a certain set of mandatory and optional form elements
- A specification of a struts DynamicActionForm containing these elements
- A mapping between the form, the DynamicActionForm and a struts action

The DynamicActionForm and the action mapping is set up in struts-config.xml

14.1 Create the form

The form must contain the following elements:

- recipients. Comma-separated list of email-adresses
- sender. Can also be specified in the mailSender component
- mailType. defines what content to send. Chose between these types: text_plain, text_html, html_url, html_mixed.
- Depending on the mailType the folowing elements must be included
- text_plain: plainContent. Will be sent with content-type 'text/plain'
- text_html: plainContent and htmlContent. Will send a 'multipart/alternative' message with plainContent as 'text/plain' and htmlContent as 'text/html' The client mailreader will choose what content to display
- html_url: plainContent and url. Works the same way as text_html, but the 'text/html' part will contain the content of the specified url.
- html_mixed: plainContent, url and attachments Works the same way as html_url, but attachments can be included (comma-separated list of filenames)
- The following elements are optional
- -subject
- -ccRecipients
- -bccrecipients

```
<%@ page language="java" %>
<%@ taglib uri="/WEB-INF/struts-html.tld" prefix="html" %>

<html:form action="sendform?method=sendFormAsMail">
  <html:hidden property="mailType" value="text_plain|text_html|html_url|html_mixed"/>
  <html:hidden property="plainContent" value="This is plain text"/>
  <html:hidden property="htmlContent" value="this is html"/>
  <html:hidden property="url" value="http://thisisaur1.com/" />
  <html:hidden property="attachments" value="c:\attachment.txt,c:\attachment2.txt"/>
  recipients: <html:text property="recipients" size="20" value="" /><br/>
  cc recipients: <html:text property="ccRecipients" size="20" value="" /><br/>
  bcc recipients: <html:text property="bccRecipients" size="20" value="" /><br/>
  Subject: <html:text property="subject" size="20" /><br/>
  <p/>
  <html:submit/></html:submit>
</html:form>
```

14.2 edit struts-config.xml

Add the following to the struts-config.xml in the 'form-beans' section.

```
<form-bean name="mailForm"
  type="org.apache.struts.action.DynaActionForm">
  <form-property name="subject" type="java.lang.String"/>
  <form-property name="sender" type="java.lang.String"/>
  <form-property name="recipients" type="java.lang.String"/>
  <form-property name="ccRecipients" type="java.lang.String"/>
  <form-property name="bccRecipients" type="java.lang.String"/>
  <form-property name="mailType" type="java.lang.String"/>
  <form-property name="url" type="java.lang.String"/>
  <form-property name="plainContent" type="java.lang.String"/>
  <form-property name="htmlContent" type="java.lang.String"/>
  <form-property name="attachments" type="java.lang.String"/>
</form-bean>
```

and the following in the 'action-mappings' section

```
<action path="/sendform"
  input="/sendform.jsp"
  parameter="method"
  name="mailForm"
  type="neo.servlet.MailFormSender"
  validate="false"
  scope="request">
  <forward name="success" path="/success.jsp" redirect="true"/>
  <forward name="failure" path="/failure.jsp" redirect="true"/>
</action>
```

Edit the paths to reflect your form.

15 Representations

The Content Engine allows you to define different representations of some types of field content, so that the field content can be re-used in different ways. Currently, representations are used in only way: to define crop masks for images (a cropped image being regarded as one of many possible representations of the original image).

15.1 Defining Image Representations

Crop masks can be added to an image content type by adding a basic `field` containing `representation` elements to the content type definition in the `content-type` resource.

Here is a simple image content type definition:

```
<content-type name="image">
  <ui:label>Picture</ui:label>
  <ui:description>An image</ui:description>
  <ui:title-field>name</ui:title-field>
  <panel name="default">
    <ui:label>Image content</ui:label>
    <field mime-type="text/plain" type="basic" name="name">
      <ui:label>Name</ui:label>
      <ui:description>The name of the image</ui:description>
      <constraints>
        <required>true</required>
      </constraints>
    </field>
    <field mime-type="text/plain" type="basic" name="description">
      <ui:label>Description</ui:label>
    </field>
    <field mime-type="text/plain" type="basic" name="alttext">
      <ui:label>Alternative text</ui:label>
    </field>
    <field name="binary" type="link">
      <relation>com.escenic.edit-media</relation>
      <constraints>
        <mime-type>image/jpeg</mime-type>
        <mime-type>image/png</mime-type>
      </constraints>
    </field>
  </panel>
  <summary>
    <ui:label>Content Summary</ui:label>
    <field name="caption" type="basic" mime-type="text/plain"/>
    <field name="alttext" type="basic" mime-type="text/plain"/>
  </summary>
</content-type>
```

You can add crop mask support by adding a second panel with the following content:

```
<panel name="crop-masks">
  <ui:label>Cropped Versions</ui:label>
  <field mime-type="application/json" type="basic" name="representations">
    <ui:label>Image Versions</ui:label>
    <rep:representations type="image-versions">
      <rep:representation name="thumbnail">
        <rep:output width="100" height="100"/>
        <rep:crop/>
        <rep:resize/>
      </rep:representation>
      <rep:representation name="narrow">
        <rep:output width="500" height="400"/>
      </rep:representation>
    </rep:representations>
  </field>
</panel>
```

```

    <rep:crop/>
    <rep:resize/>
  </rep:representation>
  <rep:representation name="wide">
    <rep:output width="1000" height="800"/>
    <rep:crop/>
    <rep:resize/>
  </rep:representation>
</rep:representations>
</field>
</panel>

```

Note that the field's **mime-type** attribute must be set to **application/json**.

The effect of this addition will be that a "Cropped Versions" panel will now appear in Content Studio when a content item of this type is edited. The panel will display the original image three times, with a red crop mask superimposed on each copy. The Content Studio will be able to move the crop masks around and resize them in order to select the precise image content required. Although the crop masks are sizable, they will retain their defined aspect ratio.

The **representation** elements in the above example belong to the **http://xmlns.escenic.com/2009/representations** namespace. The conventional prefix for this namespace is **rep**, as used in the examples above. In order for the example to work, the **http://xmlns.escenic.com/2009/representations** must be declared and assigned to the **rep** prefix. You can do this by adding the attribute **xmlns:rep="http://xmlns.escenic.com/2009/representations"** to the root element of the **content-type** resource.

15.1.1 Derived Representations

Sometimes you want Content Engine users to be able to create a set of representations that have identical contents (that is, use the same crop mask), but have different output sizes for use in different contexts. You can do this by defining **derived representations**. A derived representation is a representation that is based on another representation. It gets its crop mask from the representation it is based on, but can have a different output size.

The crop mask example shown in the previous section can be simplified by replacing the **wide** representation with a derived presentation as follows:

```

<panel name="crop-masks">
  <ui:label>Cropped Versions</ui:label>
  <field mime-type="application/json" type="basic" name="representations">
    <ui:label>Image Versions</ui:label>
    <rep:representations type="image-versions">
      <rep:representation name="thumbnail">
        <rep:output width="100" height="100"/>
        <rep:crop/>
        <rep:resize/>
      </rep:representation>
      <rep:representation name="narrow">
        <rep:output width="500" height="400"/>
        <rep:crop/>
        <rep:resize/>
      </rep:representation>
      <rep:representation name="wide" based-on="narrow">
        <rep:output width="1000"/>
      </rep:representation>
    </rep:representations>
  </field>
</panel>

```

```
</field>
</panel>
```

With this set-up, Content Studio users will no longer need to (or be able to) define a crop mask for the **wide** representation. This representation will always have exactly the same contents as the **narrow** representation: it will just be output at twice the width and height.

15.2 Accessing Image Representations

A field defined in this way in the content-type resource will be presented as a complex field (that is, a map) in **PresentationArticle** beans. Each item in the map holds a representation, indexed by its name. To access the representations in a **PresentationArticle** bean containing a content item of the type defined above, for example, you could use the following expressions:

```
${image.fields.representations.value.thumbnail}
${image.fields.representations.value.narrow}
${image.fields.representations.value.wide}
```

Each representation is itself a map containing the following fields:

width

The width of the crop mask in pixels.

height

The height of the crop mask in pixels.

sourcewidth

The width of the original image in pixels (only present in derived representations).

sourceheight

The height of the original image in pixels (only present in derived representations).

href

The URL of this representation.

crop

Information about the crop mask.

To include an image representation in a template, therefore, you would do something like this:

```

```

The **sourcewidth** and **sourceheight** values included in derived representations makes it possible to calculate how much of the original image is included in the representation. This information might be used to estimate the probable quality of the cropped image and determine how it is presented in the output.



16 Restricting Access to Content

You may sometimes want to be able to restrict access to all or part of a publication. It may be a simple case of restricting access to registered users, or you may wish charge for access to certain content. The Content Engine's `neo.xreditsys.content.agreement` classes provide support for this kind of functionality. They do not presuppose any particular method of access restriction, but simply provide an interface for associating an access control method with all or part of a publication.

Access control can be switched on or off per section in Web Studio, by setting the following section properties:

Is agreement required

Must be set to **Yes**.

Agreement information

Must specify the name of a correctly configured `AgreementPartner` (see below).

If these properties are correctly set, then access to the section is denied to unauthorized readers. For a description of Web Studio and how to set these properties, see **Escenic Content Engine Publication Administrator Guide, section 3.4.1.1**.

Access control is configured with the `/neo/io/managers/AgreementManager` component. To set up access control, you need to:

- Create a Java class that implements the interface `neo.xreditsys.content.agreement.AgreementPartner` (see Javadoc for details). The class must have a `service()` method that performs the required access control check and either allows or denies access.
- Create a component from your class by adding a `.properties` file to one of your configuration layers (most likely the common configuration layer located in `/etc/escenic/engine/common`).
- In the same configuration layer, edit the file `/neo/io/managers/AgreementManager.properties` and add the line:

```
agreementPartner.component-name=component-path
```

where:

- `component-name` is the name specified in the **Agreement information** property of the protected sections in Web Studio.
- `component-path` is the path of the component you added to the configuration layer.

This line tells the `AgreementManager` to create a component called `component-name` based on the information it finds in `component-path.properties`.

Your Java class must of course also have been compiled, packaged in a JAR file and added to the Content Engine's classpath.

16.1 Basic Password Authentication Example

The `AgreementPartner` class you implement can exercise any kind of access control you choose. It can perform straightforward password protection, require payment or provide an interface to an external micro-payment system.

This example shows a very simple implementation of `AgreementPartner` that provides basic password authentication.

```
package com.mycompany.escenic.agreements;
import neo.xreditsys.content.agreement.*;
public class PasswordAgreement implements AgreementPartner {
    AgreementConfig config;
    String realm = "Undefined";
    java.util.Map users = new java.util.HashMap();
    public PasswordAgreement() {
        config = new AgreementConfig();
        config.setAuthentication(true);
    }
    public AgreementConfig getAgreementConfig() {
        return config;
    }
    public void setRealmName(String newRealm) {
        realm = newRealm;
    }
    public String getRealmName() {
        return realm;
    }
    public void addUser(String user, String password) {
        users.put(user, password);
    }
    public java.util.Set getUsers() {
        return users.keySet();
    }
    public void service(AgreementRequest request, AgreementResponse response) {
        String username = request.getUserName();
        if (username == null || username.equals("")) {
            response.setBasicAuthenticationRealm(realm);
            return;
        }
        String password = (String) users.get(username);
        if (password == null || request.getCredentials() == null) {
            response.setBasicAuthenticationRealm(realm);
            return;
        }
        if (!password.equals(request.getCredentials())) {
            response.setBasicAuthenticationRealm(realm);
        }
    }
}
```

And here is the content of a `.properties` file that can be used to configure a `PasswordAgreement` component:

```
$class=com.mycompany.escenic.agreements.PasswordAgreement

realmName=TestRealm
user.johndoe=johnspassword
user.someone=secret
```

The first line specifies the class that is to be instantiated, and the following lines contain the values of properties that are to be set. After instantiating the class, the Content Engine automatically searches the rest of the file for properties that it can set using the class's methods. In this case it sets



realmName by calling **PasswordAgreement**'s **setRealmName()** method, and fills the **users** **HashMap** by calling **addUser()** for every element of the mapped property **user**.

For detailed information about the **.properties** file format, see **Esenic Content Engine Server Administration Guide, section 4.2**.

In addition to these methods that allow instances to be automatically configured by the Content Engine, the class contains two other important components:

- The **getAgreementConfig()** method, which returns an **AgreementConfig** instance to the caller. This method is required by the **AgreementPartner** interface. The **AgreementConfig** instance is used by the Content Engine to determine what items of information the **AgreementPartner** requires in order to perform authorization. In this example, the **AgreementConfig**'s **authentication** property is set to **true**.

```
public PasswordAgreement() {
    config = new AgreementConfig();
    config.setAuthentication(true);
}
```

authentication is defined here as meaning basic password authentication, so this setting indicates that the **PasswordAgreement** requires a **realm name, user name** and **password** in order to carry out authentication. **AgreementConfig** has other methods that you can use to add details of other information required for authorization. If, for example, successful authorization depends on the presence of one or more cookies on the user's computer, you must add this information using the **AddCookieName()** method - otherwise the **service()** method won't have access to the cookies.

- The **service()** method, which is also required by the **AgreementPartner** interface. This is the method that carries out the actual authorization:

```
public void service(AgreementRequest request, AgreementResponse response) {
    String username = request.getUserName();
    if (username == null || username.equals("")) {
        response.setBasicAuthenticationRealm(realm);
        return;
    }
    String password = (String) users.get(username);
    if (password == null || request.getCredentials() == null) {
        response.setBasicAuthenticationRealm(realm);
        return;
    }
    if (!password.equals(request.getCredentials())) {
        response.setBasicAuthenticationRealm(realm);
    }
}
```

The user's authorization data is passed in as an **AgreementRequest** object and compared with the user names and passwords in the **users** property. If no match is found, then the authentication request is rejected by setting the **realm** property of the **AgreementResponse** object that was supplied in the **response** parameter. If this property is **not** set, then authentication succeeds and the user will be granted access to the protected content. If it is set, then authentication fails and the application will carry out an appropriate action such as displaying a login page.

16.1.1 Using The Example

To actually make use of this example, you need to do the following:

1. Write and compile the `PasswordAgreement` class. In order to compile the class you need `engine-core-5.2.7.2.jar` in your classpath.
2. Create a `.properties` file like the one listed in [section 16.1](#) and save it somewhere in a configuration layer. Lets say you save it as `/agreements/PasswordAgreement.properties` in your common layer (that, is `/etc/escenic/engine/common/agreements/PasswordAgreement.properties` in a standard installation).
3. Edit `/neo/io/managers/AgreementManager.properties` in the same configuration layer and add the line:

```
agreementPartner.test=/agreements/PasswordAgreement
```

4. In Web Studio, select the publication sections you want to be password-protected and set:
 - **Is agreement required** to **Yes**.
 - **Agreement information** to **test**.

It should now not be possible to access these sections without entering one of the username/password combinations specified in `/agreements/PasswordAgreement.properties`.

 This example is provided purely to illustrate how the Content Engine's agreement system works. The `PasswordAgreement` class is deliberately simplified and not considered suitable for production use.

16.2 Removing Access Control

Note that you cannot "switch off" access control by simply removing the `agreementPartner` line from `/neo/io/managers/AgreementManager.properties`. If you do that you will find that all the sections that were previously protected by this agreement partner are now completely inaccessible. The only way to remove the access control is to remove it explicitly from each protected section in Web Studio.



17 Further Reading

This chapter lists various resources for further reading.

17.1 Vizrt Resources

Reference Material

Escenic Content Engine Bean Reference

Escenic Content Engine Resource Reference

Escenic Tag Library Reference

Other Developer Guides

Escenic Content Engine Template Developer Guide

System Administration

To be supplied.

User Guides

Escenic Content Studio User Guide

17.2 Other Resources

Servlets

<http://java.sun.com/products/servlet/docs.html>

JavaServer Pages

<http://java.sun.com/products/jsp/docs.html>

JavaBeans

<http://java.sun.com/javase/technologies/desktop/javabeans/docs/index.html>

JavaServer Pages Standard Tag Library

<http://java.sun.com/products/jsp/jstl/reference/docs/index.html>

JSP Expression Language

http://www.oracle.com/technology/sample_code/tutorials/jsp20/simpleel.html

Servlet filters

<http://java.sun.com/products/servlet/Filters.html>

XML

XML 1.0 specification: <http://www.w3.org/TR/REC-xml/>

XML base tutorial: <http://www.zvon.org/xxl/XMLBaseTutorial/Output/>

XSL

XSL home: <http://www.w3.org/Style/XSL/>

XSLT specification: <http://www.w3.org/TR/xslt>

XPath specification: <http://www.w3.org/TR/xpath>

XSLT tutorial: <http://www.zvon.org/xxl/XSLTutorial/Output/>

XPath tutorial: <http://www.zvon.org/xxl/XPathTutorial/General/examples.html>