Escenic Content Engine

Advanced Developer Guide

5.6.13.183224

escenic

A CCI COMPANY

Table of Contents

1 Introduction

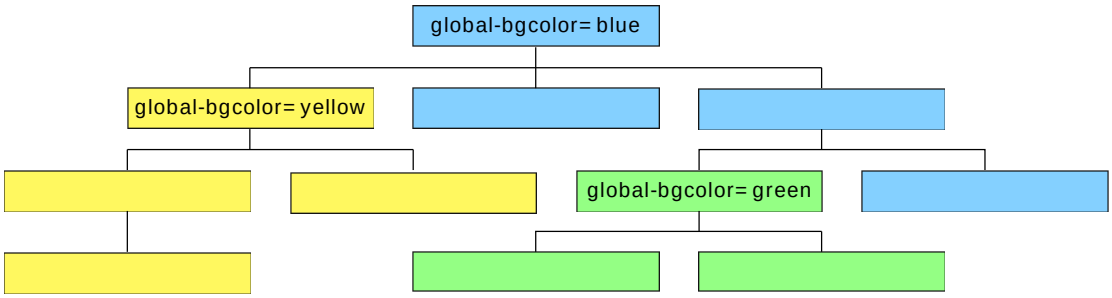
This guide is intended to describe various advanced development techniques for Escenic developers who are already familiar with the content of the **Escenic Content Engine Template Developer Guide**.

2 Section Parameters

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:



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 in Web Studio

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 Setting Section Parameters using a Web Service

escenic-admin includes a web service for manipulating section parameters. The section parameters for a specific section of a publication can be accessed via a URL like this:

```
http://server-name:port/escenic-admin/section-parameters-declared/publication/section-path
```

where:

publication
is the name of the publication

section-path
is the path of the required section, for example **/some/path/to/my/section**.

2.2.1 Retrieving Section Parameters

To retrieve a section's parameters, send an HTTP **GET** request to the appropriate web service URL:

```
curl http://server-name:port/escenic-admin/section-parameters-declared/publication/section-path
```

This returns a list of all of the section's declared section parameters. The parameters are returned as plain text, not HTML or XML. For example:

```
#Declared section parameters for section Escenic Times(22)
#Thu Mar 01 12:18:43 CET 2012
news.uniqueName=news
groupprofile.uniqueName=profile
wireFrame=default
templateVersion=v0
pageTitle=Glace
imagefile.uniqueName=images
userprofile.uniqueName=profile
forumId=162
```

2.2.2 Adding/Replacing Section Parameters

To add parameters to a section or change current settings:

1. Create a text file containing the parameters you want to add/modify. Each parameter setting must be specified on a separate line and be specified in the form:

   ```
   name=value
   ```

2. Save the file under any name you choose (for example **news.parameters**)
3. Submit the file to the appropriate web service URL using either **POST** or **PUT**. For example:

   ```
   curl -X POST http://server-name:port/escenic-admin/section-parameters-declared/publication/section-path \
   --upload-file news.parameters
   ```

   If you use **POST** then the contents of your file are merged with the section's existing parameters (that is, new parameters are added, existing parameters are overwritten with new values and any existing parameters not in the submitted file are left unchanged). If you use **PUT** then the contents of the submitted file completely replace all existing parameters.

2.2.3 Removing All Section Parameters

To remove all of a section's parameters , send an HTTP **`DELETE`** request to the appropriate web service URL:

```
curl -X DELETE http://server-name:port/escenic-admin/section-parameters-
declared/publication/section-path
```

2.3 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.notrim** parameter as follows:

```
<field name="formatted" type="basic">
  ...
  <parameter name="neo.xredsys.service.article.attributeField" value="priority"/>
  <parameter name="neo.xredsys.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 a 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. Note, however, that editorial staff can never **create** content in anything other than their home publication, whatever access rights they have in other publications.

> Cross-publishing does **not** require identical templates, only identical **`content-type`**, **`image-version`** and **`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, you must:

- Add the following line to *configuration-root*`/com/escenic/storage/metadata/MetadataInjectionTransactionFilter.properties` in one of your configuration layers:

  ```
  serviceEnabled=true
  ```

  You might need to create the file and directories.

- 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.ESCENIC.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.ESCENIC.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.xredsys.api.ArticleTransaction;
import neo.xredsys.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>
 <dispatcher>FORWARD</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` file 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 is 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>
  <init-param>
    <param-name>allowNonCachedUrls</param-name>
    <param-value>false</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.

`allowNonCachedUrls`
If set to **`true`**, this parameter will allow the CacheFilter to serve dynamic image versions even if the multimedia directory is read-only and the image version is not already available. This will mean a significant performance overhead, so the filter will log an error whenever the image version cannot be cached. By default this is not allowed, and the request will result in an exception.

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 8.1.4) 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 Article decorators

8.1.1 Creating A Decorator

An Escenic content item decorator is a Java class that:

- Extends **`neo.xredsys.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.xredsys.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`**'s **`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 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 web application's classpath

To compile a decorator you need the following Escenic JAR files in your classpath:

- **`engine-core-5.6.13.183224.jar`**
- **`engine-presentation-5.6.13.183224.jar`**
- **`common-util-`***n.n.n.n*`.jar`** (replace *n.n.n.n* with the version number of the **`common-util`** library included with the current version of the Content Engine).

8.1.2 Creating a Decorator .properties File

The article decorators executed by the system are defined in a configuration file called **`PresentationArticleManager.properties`**. To register an article decorator called **`TitleToUpperCase`**, you must create a file called **`WEB-INF/localconfig/neo/xredsys/presentation/PresentationArticleManager.properties`** in your web application folderand add the following definition to it:

```
decoratorFactory.titleToUpperCase=/com/mycompany/TitleToUpperCase
```

You must then create a properties file for the decorator (called **`WEB-INF/localconfig/com/mycompany/TitleToUpperCase.properties`** in this case) and add an entry defining your class:

```
$class=neo.xredsys.presentation.ReflectionPresentationArticleDecoratorFactory
className=com.mycompany.TitleToUpperCase
```

8.1.3 Declaring a Decorator

In order for an article decorator to be actually used, you must declare it by adding a **`ui:decorator`** element to your publication's **`content-type`** resource. For example:

```
<content-type name="xyz">
  ...
  <ui:decorator name="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 name="titleToUpperCase" />
  <ui:decorator name="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-type`** resource file. For full descriptions of the **`content-type`** and **`decorator`** elements, see the **Escenic Content Engine Resource Reference**.
- Multiple decorators are executed in the order they appear in the group definition. This may sometimes be significant. Request decorators, however, are always executed last.

8.1.4 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.xredsys.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;
  }
}
```

8.2 Pool decorators

An Escenic pool decorator is a Java class that:

- Extends **`neo.xredsys.presentation.PresentationPoolDecorator`**
- Contains a **`get`** method for each property it is to modify

Escenic supports two different types of pool decorators:

Standard pool decorator
A standard pool decorator is created when the pool is loaded from the database. It is cached and lives until the pool is removed from the pool presentation cache.

Request pool decorator
A request pool decorator is created the first time someone uses a given instance of a pool in an HTTP request. The decorator is cached in the request and flushed when the request is completed. You should only use a request pool decorator if you need the decorator to behaves differently in different requests. For example, a time-controlled decorator that enables/disables groups based on the time of the day should be implemented using a request pool decorator.

A standard pool decorator must extend **`neo.xredsys.presentation.PresentationPoolDecorator`**. A request pool decorator can also extend this class, but if it needs access to **`javax.servlet.ServletRequest`** then it should extend **`neo.xredsys.presentation.RequestPresentationPoolDecorator`** instead.

Before a decorator can be used it must be:

- Compiled
- Added to the web application's classpath

To compile a decorator you need the Escenic JAR file **`engine-presentation-5.6.13.183224.jar`** in your classpath.

Once you have a created a pool decorator class, you have to:

- Create a set of **`.properties`** files that declare the decorator and registers it in the system.
- Declare the decorator in the **`layout-group`** publication resource
- Package the decorator in a JAR file
- Deploy the JAR file in your publication

8.2.1 Create the Decorator .properties Files

For a standard pool decorator

The standard pool decorators executed by the system are defined in a configuration file called **`PresentationPoolManager.properties`**. To register a pool decorator called **`MyDecorator`**, you must create a file called *configuration-root*`/neo/xredsys/presentation/PresentationPoolManager.properties` in one of your configuration layers and add the following definition to it:

```
decoratorFactory.myDecorator=/com/mycompany/MyDecorator
```

You must then create a properties file for the decorator (called *configuration-root*`/com/mycompany/MyDecorator.properties` in this case) and add an entry defining your class:

```
$class=neo.xredsys.presentation.ReflectionPresentationPoolDecoratorFactory
className=com.mycompany.MyDecorator
```

For a request pool decorator

The request pool decorators executed by the system are defined in a configuration file called **`RequestPresentationPoolManager.properties`**.To register a pool decorator called **`MyDecorator`**, you must you must create a file called *configuration-root*`/neo/xredsys/presentation/RequestPresentationPoolManager.properties` in one of your configuration layers and add the following definition to it:

```
decoratorFactory.myRequestDecorator=/com/mycompany/MyRequestDecorator
```

You must then create a properties file for the decorator (called *configuration-root*`/com/mycompany/MyRequestDecorator.properties` in this case) and add an entry defining your class:

```
$class=neo.xredsys.presentation.ReflectionPresentationPoolDecoratorFactory
className=com.mycompany.MyRequestDecorator
```

8.2.2 Declare the Decorator

In order for a pool decorator to be actually used, you must declare it by adding a **`ui:decorator`** element to your publication's **`layout-group`** resource. For example:

```
<group name="xyz" root="true">
  ...
  <ui:decorator name="myDecorator"/>
  ...
</group>
```

The **`ui:decorator`** element must be added as the child of a top-level **`group`** element (that is, one that is a direct child of the resource file's root **`groups`** element).

Once you have done this, the decorator automatically takes effect for all pools within the top-level group and its children. You can add the decorator to as many top-level **`group`** elements as you want.

You can add more than one **`decorator`** to a **`group`**. For example:

```
<group name="xyz" root="true">
  ...
  <ui:decorator name="myDecorator/>
```

```
    <ui:decorator name="myRequestDecorator/>
    ...
</layout>
```

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 **layout-group** resource file. For full descriptions of the **layout-group** and **decorator** elements, see the **Escenic Content Engine Resource Reference**.
- Multiple decorators are executed in the order they appear in the group definition. This may sometimes be significant. Request decorators, however, are always executed last.

8.3 Packaging Decorators

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

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

Article decorator package structure

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

```
com
+-escenic
| +-servlet
|   +-plugin-config
|     +-neo
|     | +-xredsys
|     |   +-presentation
|     |     +-ArticlePresentationManager.properties
|     +-com
|       +-mycompany
|         +-TitleToUpperCase.properties
+-mycompany
  +-TitleToUpperCase.class
```

Generic pool decorator package structure

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

```
com
+-escenic
| +-servlet
```

```
|   +-plugin-config
|     +-neo
|     | +-xredsys
|     |   +-presentation
|     |     +-PresentationPoolManager.properties
|     +-com
|       +-mycompany
|         +-MyDecorator.properties
+-mycompany
  +-MyDecorator.class
```

Request pool decorator package structure

For the request pool decorator example shown earlier, you would need to create a JAR file with the following structure:

```
com
+-escenic
| +-request
|   +-plugin-config
|     +-neo
|     | +-xredsys
|     |   +-presentation
|     |     +-RequestPresentationPoolManager.properties
|     +-com
|       +-mycompany
|         +-MyRequestDecorator.properties
+-mycompany
  +-MyRequestDecorator.class
```

8.3.1 Deploy the Decorator

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

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.xredsys.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 the abstract class **`neo.xredsys.api.services.AsyncEventListenerService`**. This is a convenience class that extends **`neo.nursery.AbstractNurseryService`**. It provides boilerplate implementations of the methods defined in the **`neo.xredsys.api.IOEventListener`** and **`neo.xredsys.api.IOEventFilter`** interfaces.

Your class must include implementations of the following methods:

`accept(event)`
: This method is defined in the **`neo.xredsys.api.services.AsyncEventListenerService`** class. Use it to determine which events your listener will respond to. Return **`true`** for all events you want the listener to respond to.

  > This method is called synchronously and blocks other event listeners from receiving events while it is executing. You should therefore only use it carry out fast initial filtering of events.

`handle(event)`
: This method is defined in the **`neo.xredsys.api.services.AsyncEventListenerService`** class. It is called every time your **`accept(event)`** method returns **`true`**. Use it to perform whatever actions you want to be carried out for accepted events.

  This method is called asynchronously and therefore does not block other event listeners. The number of events waiting to be responded to by this method can be seen in the **`backlog`** property. The execution time of this method can be recorded by the optional **`HitCollector`**.

Your class must also have a public no argument constructor. For example:

```
public class MyService extends AsyncEventListenerService {
  public MyService() {
    super(false);
  }
  //More code goes here...
```

```
}
```

This constructor creates an **`AsyncEventListenerService`** that listens to all events, both local and remote. If you want to create a service that only listens to local events, set **`super(true)`** in the constructor.

Your class may optionally implement the following methods as well:

`startEventListener()`
: This method is defined in **`neo.xredsys.api.services.AsyncEventListenerService`**. You can use it to carry out any actions that you want to be performed when the service is started. You might, for example, use it to validate the configuration of your service.

`stopEventListener()`
: This method is defined in **`neo.xredsys.api.services.AsyncEventListenerService`**. You can use it to carry out any actions that you want to be performed when the service is stopped. You might, for example, use it to clear lists that have been populated while the service was running.

For more detailed information about **`neo.xredsys.api.services.AsyncEventListenerService`**, see the javadoc.

Here is an example event listener called **`com.mycompany.events.NewArticleNotifier`** that:

- Sends an e-mail to a configured e-mail address whenever a new content item is created
- Validates its configuration on start-up

The class's constructor calls its super constructor with the parameter **`true`**:

```
public NewArticleNotifier() {
  super(true);
}
```

This ensures that the service only listens to local events and will therefore only send an e-mail when a content item is created on the local server. Setting this parameter to **`false`** would result in multiple mails being sent for every content item created (one from each server in the cluster).

Properties are defined to hold the both the address to which notifications are to be sent and the sender address to be included in the messages. It is considered good practice to inject parameters this way.

```
public void setEmail(final String pEmail) {
  this.email = pEmail;
}

public String getEmail() {
  return this.email;
}
public void setEmailSender(final EmailSender pEmailSender) {
  this.emailSender = pEmailSender;
}

public EmailSender getEmailSender() {
  return this.emailSender;
}
```

The **accept** method accepts only **OBJECT_CREATED** events, and only for objects of type article. It is important to ensure that this method returns quickly since it blocks the execution of other event listeners' **accept** methods. If your **accept** method uses several criteria to select events, It is a good idea to place the if tests that will reject most events first, as here:

```
@Override
protected boolean accept(final IOEvent pEvent) throws Exception {
  if (IOEvent.OBJECT_CREATED == pEvent.getType()) {
    if (IOAtom.OBJECTTYPE_ARTICLE == pEvent.getObjectKey().getObjectType()) {
      return true;
    }
  }
  return false;
}
```

The **handle** method is only called only for events where the **accept** method has returned **true**. It is therefore safe to assume in this case that the input event is an **OBJECT_CREATED** event for an article. The method is executed in a separate thread so it does not need to be especially fast. If it executes too slowly to keep up with the number of events being generated then the value of the **backlog** property will increase accordingly.

If you define a **hit collector** for your event listener (see **Performance monitoring** in section 9.2), then the execution time of this method will be recorded by the hit collector. You can view the information gathered by the hit collector on the **escenic-admin** web application's **Performance Summary** page. For information about **escenic-admin** see **Escenic Content Engine Server Administration Guide**, **chapter 2**.

```
@Override
protected void handle(final IOEvent pEvent) throws Exception {
  Article article = (Article) pEvent.getObject();

  EmailEvent email = new EmailEvent();
  email.setRecipients(new Address[] {new InternetAddress(getEmail())});
  email.setSubject("New article with title '" + article.getTitle() + "' was
created");

  getEmailSender().sendMessage(email);
}
```

The **startEventListener** method is called after all properties have been set, but before the **EventListener** is registered with the Content Engine so that it can receive events. If **startEventListener** throws an exception then the **EventListener** is not registered and will not receive any events. The **startEventListener** method must include a **super.startEventListener()** call.

The example shown here performs some simple checks on the supplied configuration values:

```
@Override
protected void startEventListener() throws IllegalStateException,
IllegalArgumentException, Exception {
  super.startEventListener();

  Validate.notNull(getEmailSender(), "Email sender is not set, will not start the '"
+ getClass().getName() + "' service");
  Validate.notNull(getEmail(), "Email is not set, will not start the '" +
getClass().getName() + "' service");
}
```

Here is the example code again in full, including all the necessary boilerplate code:

```
package com.mycompany.events;

import javax.mail.Address;
import javax.mail.internet.InternetAddress;

import neo.xredsys.api.Article;
import neo.xredsys.api.IOAtom;
import neo.xredsys.api.IOEvent;
import neo.xredsys.api.services.AsyncEventListenerService;
import neo.xredsys.email.EmailEvent;
import neo.xredsys.email.EmailSender;

import org.apache.commons.lang.Validate;

public class NewArticleNotifier extends AsyncEventListenerService {


  public NewArticleNotifier() {
    super(true);
  }

  private String email;

  public void setEmail(final String pEmail) {
    this.email = pEmail;
  }

  public String getEmail() {
    return this.email;
  }


  private EmailSender emailSender;

  public void setEmailSender(final EmailSender pEmailSender) {
    this.emailSender = pEmailSender;
  }

  public EmailSender getEmailSender() {
    return this.emailSender;
  }


  @Override
  protected boolean accept(final IOEvent pEvent) throws Exception {
    if (IOEvent.OBJECT_CREATED == pEvent.getType()) {
      if (IOAtom.OBJECTTYPE_ARTICLE == pEvent.getObjectKey().getObjectType()) {
        return true;
      }
    }
    return false;
  }


  @Override
  protected void handle(final IOEvent pEvent) throws Exception {
    Article article = (Article) pEvent.getObject();
```

```
    EmailEvent email = new EmailEvent();
    email.setRecipients(new Address[] {new InternetAddress(getEmail())});
    email.setSubject("New article with title '" + article.getTitle() + "' was
 created");

    getEmailSender().sendMessage(email);
  }


  @Override
  protected void startEventListener() throws IllegalStateException,
 IllegalArgumentException, Exception {
    super.startEventListener();

    Validate.notNull(getEmailSender(), "Email sender is not set, will not start the '"
 + getClass().getName() + "' service");
    Validate.notNull(getEmail(), "Email is not set, will not start the '" +
 getClass().getName() + "' service");
  }
}
```

Before an event listener can be used it must be:

- Compiled
- Added to the Content Engine's classpath

To compile the example you must add the following JAR files to the classpath:

`engine-core-5.6.13.183224.jar`
: This is the Escenic jar that contain most of the classes needed.

`common-nursery-5.6.13.183224.jar`
: The Content Engine's dependency injection framework.

`commons-lang-2.3.jar`
: The Content Engine's validation framework.

`mail-1.4.jar`
: This jar contain the dependency injection framework (Nursery) we use.

9.1.1 Handling Staged Content Item Events

By default, **`neo.xredsys.api.services.AsyncEventListenerService`** does **not** handle events generated for staged content items (see chapter 21). If you want your event listener to be able to handle events for staged content items as well as ordinary content items then you need to explicitly extend it to implement the **`neo.xredsys.api.StagedEventListener`** marker interface.

To make our example event listener handle staged content items as well, therefore, all we need to do is modify the class declaration as follows:

```
public class NewArticleNotifier extends AsyncEventListenerService implements
 StagedEventListener {
  ...
}
```

This event listener will now handle events for all content items, both staged and unstaged. If you only want to handle events for staged content items (or handle staged content items differently), you can either:

- Use the new **`neo.xredsys.api.Article.isStaged()`** method to distinguish between staged and unstaged content items, or
- Check if the event has a **`Parameter`** property called **`event-type-parameter`** with the value **`staged`**

The following code, for example, will filter out all events for unstaged objects:

```
if ("staged".equals(pEvent.getParameter("event-type-parameter")) {
  //process the event
} else {
 //do nothing
}
```

9.1.2 Staged Content Items and Publishing Status

If your handler is concerned with the publishing status of content items (whether or not they are **`published`**), then you need to think carefully about what state-change events actually mean for staged and unstaged content items, and if necessary handle them differently. If, for example, you trap an **`OBJECT_STATE_CHANGED`** event and the new state of the content item is **`approved`**, then:

- If the content item is unstaged, then there is no **`published`** version.
- If the content item is staged then there is a **`published`** version.

9.2 Using An Event Listener

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

The event listeners executed by the Content Engine are defined in a configuration file called *configuration-root*`/Initial.properties`. To enable the **`NewArticleNotifier`** listener, therefore, you must add a declaration to **`Initial.properties`** in one or more of your configuration layers. For example:

```
service.60-articlenotifier=/com/mycompany/NewArticleNotifier
```

You must also create a properties file for the listener called **`NewArticleNotifier.properties`**, and save it in the location you have specified in the same configuration layer(s) - *configuration-root*`/com/mycompany/NewArticleNotifier.properties` in this case. The file must at least contain **`$class`** and **`eventManager`** entries. In the particular case of the **`NewArticleNotifier`** example you will also need to set the **`email`** and **`emailSender`** properties. For example:

```
$class=com.mycompany.events.NewArticleNotifier
eventManager=/io/api/EventManager
email=sample@mycompany.com
emailSender=/neo/io/services/MailSender
```

Finally, you must make sure that the email sender declared in *configuration-root*`/com/mycompany/NewArticleNotifier.properties`. is correctly configured. Create a configuration file called *configuration-root*`/neo/io/services/MailSender.properties` and configure it with appropriate settings. For example:

```
enabled=true
mailHost=smtp.mycompany.com
defaultSender=noreply@mycompany.com
```

If you have trouble running your event listener, try checking the **View Services** page of the `escenic-admin` web application or the Content Engine's log files.

Performance monitoring

If you want to use the Content Engine's `HitCollector` component to gather statistics about the performance of an event listener, then you can do so by adding one more line to its properties (i.e., *configuration-root*`/com/mycompany/NewArticleNotifier.properties` in the case of this example):

```
collector=./NewArticleNotiferCollector
```

You will then also need to create a configuration file for the collector you have declared (*configuration-root*`/com/mycompany/NewArticleNotifierCollector.properties` in this case):

```
$class=neo.util.stats.HitCollector
denominator=events handled
description=Events handled by 'com.mycompany.events.NewArticleNotifier'
failureDescription=failed events
```

The information gathered by the hit collector can be viewed on the `escenic-admin` web application's **Performance Summary** page. For information about `escenic-admin` see **Escenic Content Engine Server Administration Guide**, **chapter 2**.

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.xredsys.api.services.TransactionFilterService`**. This is a convenience class containing empty "do nothing" implementations of the methods defined in the **`neo.xredsys.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.

`doStagedUpdate(object)`
: which is executed immediately before changes to an existing **staged** object are saved. For an explanation of staging, see chapter 21.

> Note that you cannot change the state of a staged object to **`published`** in a transaction filter. Any attempt to do so will result in an exception.

`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.xredsys.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 (called **wordcount** - the example assumes that the content item has a field with this name)

```
package com.mycompany.transactionFilters;

import java.util.*;
import neo.xredsys.api.*;
import neo.xredsys.api.services.*;

public class WordCount extends TransactionFilterService {

  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 doStagedUpdate(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-5.6.13.183224.jar`** and the corresponding current version of **`common-nursery-`***version-number*`.jar`** in your classpath.

10.2 Using A Transaction Filter

For detailed information about configuration files, configuration layers and an explanation of the *configuration-root* 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 *configuration-root*`/Initial.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:

```
service.60-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) - *configuration-root*`/com/mycompany/WordCountFilter.properties` in this case. The file must at least contain **`$class`**, **`filterName`** and **`objectUpdater`** entries. These specify the class that implements the filter, a name, and the component to register the **`TransactionFilter`** against:

```
$class=com.mycompany.transactionFilters.WordCount
filterName=MyWordCountFilter
objectUpdater=/io/api/ObjectUpdater
```

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

```
$class=com.mycompany.transactionFilters.WordCount
filterName=MyWordCountFilter
objectUpdater=/io/api/ObjectUpdater

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 **`TransactionFilterService`** class has a **`serviceEnabled`** property that can be used to enable/disable filters. You can therefore disable any filter by adding

```
serviceEnabled=false
```

to the properties file.

> If no other transaction filters or post-transaction filters have been defined, then there may be no *configuration-root*`/Initial.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.xredsys.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 an event listener 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.xredsys.api.services.PostTransactionFilterService`**. This is a convenience class containing empty "do nothing" implementations of the methods defined in the **`neo.xredsys.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.

`doStagedPostUpdate(object)`
: which is executed immediately after changes to an existing **staged** object are saved. For an explanation of staging, see chapter 21.

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-5.6.13.183224.jar** in your classpath.

11.2 Using a Post-Transaction Filter

> For detailed information about configuration files, configuration layers and an explanation of the *configuration-root* 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 *configuration-root*/**Initial.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:

```
service.60-myPostFilter=/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) - *configuration-root*/**com/mycompany/MyPostTransFilter.properties** in this case. The file must at least contain **$class**, **filterName** and **objectUpdater** entries. These specify the class that implements the filter, a name, and the component to register the **PostTransactionFilter** against:

```
$class=com.mycompany.MyPostTransFilter
filterName=MyWordCountFilter
objectUpdater=/io/api/ObjectUpdater
```

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

```
$class=com.mycompany.MyPostTransFilter
filterName=MyWordCountFilter
objectUpdater=/io/api/ObjectUpdater

myFilterParameter=99
```

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

```
serviceEnabled=false
```

to the properties file.

If no other transaction or post-transaction filters have been defined, then there may be no *configuration-root*`/Initial.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.xredsys.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 Escenic.

The **`filter-mapping`** elements define the sequence of the filters, which is significant. The Escenic-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 2.1.15**, 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.

   > The name is the **`context path`** of the web application. This can be changed by adding
   >
   > ```
   > <display-name>my web application</display-name>
   > ```
   >
   > in **`web.xml`** in your web application.

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 2.1.15**.

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=mypub
```

> 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
`com/escenic/servlet/TemplateDispatchResolver`	`page`	`forward-to-page`
`com/escenic/servlet/PublicationResolverProcessor`	`publicationName`	`publication-name`

> 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 CAPTCHA Support

The Content Engine provides built-in support for adding CAPTCHA challenges to publication pages that need protection from robot visitors. The CAPTCHA functionality can be configured to use challenges provided by different CAPTCHA providers. By default, the Content Engine is configured to use Jcaptcha. You are, however, strongly recommended to reconfigure it to use ReCaptcha instead.

14.1 Configuring a CAPTCHA Provider

The following sections describe how to configure the Content Engine to use various CAPTCHA providers.

14.1.1 ReCaptcha

ReCaptcha is an on-line service that provides CAPTCHA challenge and verifies user responses. To use this service in one of your publications:

1. Create an account at http://recaptcha.net/
2. Add a new site to the account, specifying the domain name of the publication containing the pages you wish to protect. ReCaptcha will then generate a private and public key pair for the domain. You will need to use these keys in step 3 below.
3. Add a file called **`com/escenic/captcha/ReCaptchaProvider.properties`** to your publication web application's **`WEB-INF/localconfig/`** folder. The file must contain the following property settings:

   ```
   privateKey=your-private-key
   publicKey=your-public-key
   ```

   where *your-private-key* and *your-public-key* are the keys assigned to your publication domain by the ReCaptcha service in step 2.
4. Add a file called **`com/escenic/captcha/CaptchaProviderFactory.properties`** to your publication web application's **`WEB-INF/localconfig/`** folder. The file must contain the following property setting:

   ```
   captchaProvider=./ReCaptchaProvider
   ```

14.1.2 Jcaptcha

The **`/com/escenic/captcha/CaptchaProviderFactory`** component is configured to use **`/com/escenic/servlet/captcha/JCaptchaProvider`** by default, so no Content Engine configuration is required.

To use **`JCaptcha`** from Struts, however, you need to add the following Struts configuration:

```
<action path="/jcaptcha"
 type="com.octo.captcha.module.struts.image.RenderImageCaptchaAction" />
<plug-in className="com.octo.captcha.module.struts.CaptchaServicePlugin"/>
```

14.1.3 Custom CAPTCHA Provider

As an alternative to using either Jcaptcha or ReCaptcha you can write your own CAPTCHA provider. To do this you need to:

1. Create a provider class that implements the **`com.escenic.servlet.captcha.CaptchaProvider`** interface.
2. Create a configuration **`.properties`** file for your provider class and add it to your publication web application's **`WEB-INF/local-config`** folder.
3. Add a file called **`com/escenic/captcha/CaptchaProviderFactory.properties`** to your publication web application's **`/localconfig/WEB-INF`** folder. The file must contain the following property setting:

   ```
   captchaProvider=component-path
   ```

   where *component-path* is the path of your CAPTCHA component.

14.2 Displaying Your CAPTCHA Challenge

Once you have correctly configured CAPTCHA support, the JSP code shown below can be used to add a CAPTCHA challenge to a Struts form in your publication templates (the **`html:`** prefix in some of the tag names identifies them as members of the standard Struts HTML tag library). The code examples assume that you want to add the CAPTCHA challenge to a form that users will use to create blog entries.

ReCaptcha

```
<html:form action="/blog/add">
  <!-- JSP code to render the form properties related to creating a blog -->

  <div class="captcha">
    <captcha:recaptchaHTML theme="red" lang="en"/>
  </div>

  <!-- The rest of the JSP code to render the form properties related to
       creating a blog goes here. -->
</html:form>
```

In this example, the **`<captcha:recaptchaHTML/>`** tag generates the HTML code required to display the ReCaptcha challenge.

JCaptcha

```
<html:form action="/blog/add">
  <!-- JSP code to render the form properties related to creating a blog -->

  <div class="captcha">
    <label for="jcaptcha_response">Verification code: </label>
    <input id="jcaptcha_response" type="text" name="jcaptcha_response" /><br/>
    <html:image page="/jcaptcha.do"/>
  </div>

  <!-- The rest of the JSP code to render the form properties related to
       creating a blog goes here. -->
```

```
</html:form>
```

The **input** element called **jcaptcha_response** is the field in which the user is required to enter the characters displayed in the CAPTCHA image. The tag **<html:image page="/jcaptcha.do"/>** generates an ordinary HTML **img** element containing the CAPTCHA challenge image.

14.3 Verifying the CAPTCHA Response

The following example contains the Java code you need to add to a custom Struts action or servlet class in order to verify a CAPTCHA response:

```
Bus bus = ApplicationBus.getApplicationBus(servletContext);
CaptchaProviderFactory captchaProviderFactory =
    (CaptchaProviderFactory) bus.lookupSafe("/com/escenic/captcha/
CaptchaProviderFactory");

CaptchaProvider captchaProvider = captchaProviderFactory.getCaptchaProvider();
boolean isValidCaptchaResponse =
 captchaProvider.checkCaptchaResponse(httpServletRequest);
```

An alternative approach is to create a generic servlet filter that you can then use to filter all requests that require CAPTCHA verification.

If you are using either **Viz Community Expansion** or the **Forum plug-in** then you do not need to write your own verification code, since both of these plug-ins include Struts action classes that perform CAPTCHA verification. For details see the documentation of these plug-ins.

15 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

15.1 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://thisisaurl.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>
```

15.2 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.

16 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).

16.1 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: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.

For using crop mask binary field constraints for **mime-type** only supports **image/jpeg**, **image/png** and **image/gif** types. As for example,

```
<constraints>
  <mime-type>image/jpeg</mime-type>
  <mime-type>image/png</mime-type>
  <mime-type>image/gif</mime-type>
</constraints>
```

16.1.1 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>
```

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.

16.2 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:

```
<img
   src="${image.fields.representations.value.thumbnail.href}"
   width="${image.fields.representations.value.thumbnail.width}"
   height="${image.fields.representations.value.thumbnail.height}"/>
```

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.

17 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.xredsys.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.xredsys.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.

17.1 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.xredsys.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 th 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 **Escenic 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.

17.1.1 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-5.6.13.183224.jar`** in your classpath.
2. Create a **`.properties`** file like the one listed in section 17.1 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.

17.2 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.

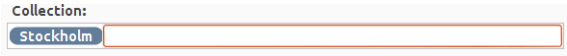
18 Collection Fields

A **collection field** is a content item field type that provides a simple means of integrating content published via Atom feeds into your content items. You can therefore use collection fields for two purposes:

- To integrate external Atom-published content (from a public feed, for example)
- To integrate internal data by publishing it as an Atom feed that you can then access from Content Studio via a collection field

A collection field is initially displayed as a "search as you type" text field in Content Studio. When the user starts to type in the field, a list of Atom entries with matching titles is displayed below the field. When the user selects one of the listed entries:

- A value from one of the entry's fields is stored as the collection field's value
- The value is displayed in the field as a **token**: a user interface component that looks like this:



18.1 Defining and Using a Collection Field

A collection field is defined in the `content-type` resource as a `field` element with the `type` attribute `collection`. It also has a `src` attribute for specifying the data source, which you can use in two different ways:

Only using an Atom feed
In this case, you set the `src` attribute to point directly to an Atom feed. When the user types in the collection field, Content Studio searches through the entries in this feed and displays the results in a list below the field.

Using OpenSearch
In this case, you set the collection field's `src` attribute to point an Atom feed that contains a `search link` referencing an OpenSearch document. If the specified feed contains a link to an OpenSearch document, then the entries in the feed are not used (it can, in fact, be an empty feed). Instead, Content Studio retrieves the OpenSearch document and composes a query URL by combining the search template it contains with whatever the user types in the collection field. It displays the results of this search in the list below the field. In order for this method to work, the composed search query must return an Atom feed.

The text items displayed in the "search as you type" field are the `title` elements of entries that match the string the user has typed. When the selects one of the options that are offered, a value is extracted from the corresponding Atom entry and stored as the collection field's value. The extracted value is taken from one of the Atom entry's sub elements, as specified by the `field` element's `select` attribute. This attribute can have one of the following values:

content
The field value is taken from the Atom entry's `content` element.

title
The field value is taken from the Atom entry's **`title`** element (that is, it will be the same value as is actually displayed in the field).

locator
The field value is taken from a proprietary **`viz:locator`** element in the Atom entry. (Not currently used.)

link
The field value is taken from one of the Atom entry's **`link`** elements. An additional **`linkrel`** attribute specifies which link attribute is to be used.

For a full description of the collection **`field`** element, see **Escenic Content Engine Resource Reference**, **section 2.7.5**.

The following example shows a collection field definition that allows the Content Studio user to select images from a Flickr public feed:

```
<field name="pwImage"
  type="collection"
  src="http://api.flickr.com/services/feeds/photos_public.gne?
tags=creativecommons,norway"
  select="link"
  linkrel="enclosure"
  mime-type="text/plain">
  <ui:label>Images of Norway</ui:label>
</field>
```

This is a simple test you can use to see how the field works. If you add a field like this to a content type and then try using it in Content Studio, you will see that image descriptions taken from the entry **`title`** elements are displayed under the input field like this:

If you select one of the displayed options, then a value is taken from the related Atom entry, and stored as the field's value. Exactly which entry element the value is copied from is determined by the field definition's **`select`** element. In this case it was set to **`"link"`**, which means the value will be copied from one the Atom entry's **`link`** elements. Since an Atom entry can contain many **`link`** elements, an additional **`linkrel`** element is required to specify which link element to use: in this case, the one with a **`rel`** attribute set to **`"enclosure"`**.

You can retrieve collection field values in your publication templates using JSTL as follows:

```
<img src="${article.fields.pwImage.value.value}"/>
```

> Note the extra **`value`** component in the above example. This is always required when retrieving values from collection fields.

18.2 Using Your Own Feeds

You don't have to use public Atom feeds, you can create your own. The collection field provides you with a simple standards-based means of integrating data from other systems into your Escenic publications. All you have to do is generate a correctly-formatted Atom feed in which:

- The **`title`** element of each **`entry`** contains the label that you want to appear in the field.
- Some other element of each feed contains the data item you want to store in the field.

You can make the drop-down displayed when the user starts typing more informative by including the following additional elements in your feed entries:

`summary` elements
: If your entries contains **`summary`** elements, then their content will be displayed below entry titles in the collection field drop down.

"top" `link` elements
: If you include **`link`** elements with the Vizrt proprietary relation **`http://www.vizrt.com/types/relation/top`**, in your entries, then content of the **`link`** elements' **`title`** attributes will be displayed before entry titles in the collection field drop down. Link elements with this relation appear in some of the feeds returned by the Content Engine web service. In section feeds, for example, the "top" link references the publication and in tag feed it references the tag collection).

Here, for example, is a very simple feed:

```
<?xml version="1.0"?>
<feed xmlns="http://www.w3.org/2005/Atom">
  <author>
    <name>my-site.com</name>
  </author>
  <id>http://my-host/my/first/feed</id>
  <link rel="self" href="http://my-host/my-first-atom-feed" type="application/atom
+xml"/>
  <updated>2012-07-26T12:34:25.323Z</updated>
  <title type="text">My first feed</title>
  <entry>
    <id>http://my-host/my/first/entry</id>
    <title type="text">One</title>
    <content type="text">1</content>
    <summary type="text">The first</summary>
    <updated>2012-07-26T12:34:25.323Z</updated>
  </entry>
  <entry>
    <id>http://my-host/my/second/entry</id>
    <title type="text">Two</title>
    <content type="text">2</content>
    <summary type="text">The second</summary>
    <updated>2012-07-26T12:34:25.323Z</updated>
  </entry>
  <entry>
    <id>http://my-host/my/third/entry</id>
    <title type="text">Three</title>
    <content type="text">3</content>
    <summary type="text">The Third</summary>
    <updated>2012-07-26T12:34:25.323Z</updated>
  </entry>
```

```
</feed>
```

If this is referenced by the following collection field definition:

```
<field name="feedTest"
  type="collection"
  src="http://my-host/my-first-atom-feed"
  select="content"
  mime-type="text/plain">
  <ui:label>My first feed</ui:label>
</field>
```

then Content Studio users will be able to choose between the values "One", "Two" and "Three" when filling in the field. Since the entries contain summary elements, each option displayed in the collection field drop-down will consist of two lines ("One" and "The first", for example).

Selecting "One" will cause the value `1` to be stored in the field.

You can retrieve this field's value in your publication templates as follows:

```
${article.fields.feedTest.value}
```

18.2.1 Making an OpenSearch-based Feed

The basic Atom feed mechanism described above works fine if the Atom feed returned by the **`src`** attribute is not too large. If the data set you want to provide access to is very large, however, then it will soon become impractical. Using an OpenSearch-based feed solves this problem. An example of this kind of feed is actually included with the Content Engine: the Content Engine web service includes a special section feed that provides information about all the sections to which the current user has access.

This feed is available at the following URL:

```
http://host-ip-address/webservice/escenic/section
```

If you follow this link at your installation you will be required to log in, and then get an empty feed like this returned:

```
<?xml version="1.0"?>
<feed>
  <author>
    <name>Escenic Content Engine</name>
  </author>
  <id>http://host-ip-address/webservice/escenic/section</id>
  <link rel="self"
        href="http://host-ip-address/webservice/escenic/section"
        type="application/atom+xml"/>
  <updated>2012-09-25T10:51:02.419Z</updated>
  <title type="text">Dummy Atom Feed</title>
  <link rel="search"
        href="http://host-ip-address/webservice/open-search/section-search-
description.xml"
        type="application/opensearchdescription+xml"/>
</feed>
```

If you follow the (highlighted) **search** link, then an OpenSearch document like this is returned:

```
<?xml version="1.0" encoding="UTF-8"?>
<OpenSearchDescription xmlns="http://a9.com/-/spec/opensearch/1.1/">
  <ShortName>Section Search</ShortName>
  <Description>Search for sections</Description>
  <Url type="application/atom+xml" template="http://host-ip-address/webservice/
escenic/section/search/{searchTerms}"/>
  <LongName/>
  <Developer/>
  <Attribution/>
  <SyndicationRight/>
  <AdultContent>false</AdultContent>
  <OutputEncoding>UTF-8</OutputEncoding>
  <InputEncoding>UTF-8</InputEncoding>
</OpenSearchDescription>
```

The important part of this document is the highlighted URL template. You can search for sections by simply replacing the **{searchTerms}** placeholder with a search string and submitting the resulting URL. The results of the search are returned in an Atom feed that looks something like this:

```
<feed xmlns:ece="http://www.escenic.com/2007/content-engine" xmlns:dcterms="http://
purl.org/dc/terms/"
      xmlns:opensearch="http://a9.com/-/spec/opensearch/1.1/" xmlns="http://
www.w3.org/2005/Atom">
  <title>+(acl:"section\:4225" acl:"publication\:723") +(+a*
 +contenttype:com.escenic.section) -state:deleted</title>
  <author>
    <name>Escenic Content Engine</name>
  </author>
  <link rel="self" type="application/atom+xml" href="http://host-ip-address/
webservice/escenic/section/search/a?pw=1&amp;c=2"/>
  <link rel="first" type="application/atom+xml" href="http://host-ip-address/
webservice/escenic/section/search/a?pw=1&amp;c=2"/>
  <link rel="last" type="application/atom+xml" href="http://host-ip-address/
webservice/escenic/section/search/a?pw=1&amp;c=2"/>
  <updated/>
  <id>urn:uuid:60a76c80-d399-11d9-b93C-0003939e0af6</id>
  <opensearch:totalResults>2</opensearch:totalResults>
  <opensearch:startIndex>0</opensearch:startIndex>
  <opensearch:itemsPerPage>2</opensearch:itemsPerPage>
  <opensearch:Query role="request"
    searchTerms="+(acl:"section\:4225" acl:"publication\:723") +(+a*
 +contenttype:com.escenic.section) -state:deleted" startPage="0"/>
...
  <entry>
    <title>News</title>
    <link rel="edit" href="http://host-ip-address/webservice/escenic/section/3984"/>
    <link rel="self" href="http://host-ip-address/webservice/escenic/section/3984"/>
    <link href="http://host-ip-address/webservice/publication/dev.nightly/"
 rel="http://www.vizrt.com/types/relation/top" type="application/atom+xml; type=entry"
 title="dev.nightly"/>
    <id>urn:com.escenic.section:3984</id>
    <dcterms:created>2008-05-15T06:55:45Z</dcterms:created>
    <published>2012-02-02T13:16:43Z</published>
    <updated>2012-02-02T13:16:43Z</updated>
    <summary ui:align="right">tps / News</summary>
    <ece:state>published</ece:state>
    <ece:type>com.escenic.section</ece:type>
    <ece:home>
```

```
      <ece:uri>http://host-ip-address/webservice/content/com.escenic.section/</
ece:uri>
      <ece:name/>
    </ece:home>
    <ece:last_edited_by/>
  </entry>
...
</feed>
```

You can define a collection field to use this service as follows:

```
<field name="sectionSearch"
  type="collection"
  src="escenic/section"
  select="link"
  linkrel="self"
  mime-type="text/plain">
  <ui:label>Section</ui:label>
</field>
```

Note that the **src** attribute is set to just **escenic/section**: if you enter a relative URL here, then it is resolved relative to the URL of the Content Engine web service.

When a content item containing this kind of collection field is opened in Content Studio, Content Studio follows the **src** attribute URL, and then follows the search link in the empty Atom feed in order to retrieve the OpenSearch document. As soon as the user starts to type in the **sectionSearch** field, Content Studio:

1. Combines the typed character(s) with the search template.
2. Submits the resulting search URL.
3. Displays the titles from the returned Atom feed entries in a list below the field.

A new search is submitted each time a new character is entered in the field, giving the same "search as you type" behavior as with a simple Atom feed set up.

You can retrieve this field's value in your publication templates as follows:

```
${article.fields.sectionSearch.value.value.name}
```

Since **${article.fields.sectionSearch.value.value}** in this case retrieves a complete section object, you can in fact substitute **name** in the above example with the name of any other section property you are interested in, for example:

```
${article.fields.sectionSearch.value.value.lastModified}
```

18.2.2 Feed and OpenSearch MIME Types

If you are generating your own Atom feeds or OpenSearch documents for use with collection fields, then you must ensure that they are served with the correct MIME types. These are:

- **application/atom+xml** for an Atom feed.
- **application/opensearchdescription+xml** for an OpenSearch document.

If a feed or OpenSearch document is served with the wrong MIME type then when any content item containing a collection field that references it is opened, Content Studio logs an exception and the field will not work.

18.3 Using a Content Engine Proxy Service

It is a good idea to set up a Content Engine proxy service for accessing external Atom feeds rather accessing them directly. The advantages of doing this are:

- It simplifies firewall configuration, since Content Studio and other Content Engine clients can always access the Content Engine web service. You then only need to make sure that the Content Engine itself has access to the external Atom feeds.
- You can ensure that any sensitive information such as domain credentials are removed from requests before they are forwarded.
- You can insert any authentication credentials that may be required by target hosts before requests are forwarded.

For detailed information about Content Engine proxy services and how to create them, see chapter 19.

19 Content Engine Proxy Services

The Content Engine can provide Content Studio and other Content Engine clients with proxied access to specified resources outside the Content Engine's own domain. Such a proxy service can manipulate HTTP requests before they are forwarded to the target host. The service can, for example:

- Remove header fields
- Add header fields
- Modify the request body

Using such proxy services provides centralized control over access to external resources and has a number of advantages:

- It simplifies firewall configuration, since Content Studio and other Content Engine clients can always access the Content Engine web service. You then only need to make sure that the Content Engine itself has access to the external resources.
- You can ensure that any sensitive information such as domain credentials are removed from requests before they are forwarded.
- You can insert any authentication credentials that may be required by target hosts before requests are forwarded.

A proxy service is defined by associating a **service name** with a URL. The service is then accessible to Content Studio or any other Content Engine web service client at a local URL formed by appending the service name to the URL **`http://`***content-engine-domain***`/webservice/escenic/proxy/`**. If, for example, you define a proxy service with the name **`vizrt`** that points to **`http://www.vizrt.com`**, then:

- A request sent to **`http://`***content-engine-domain***`/webservice/escenic/proxy/vizrt`** will return **`http://www.vizrt.com`**.
- A request sent to **`http://`***content-engine-domain***`/webservice/escenic/proxy/vizrt/products`** will return **`http://www.vizrt.com/products`**.

> A content engine proxy service is intended to provide centrally controlled access to external resources, not to support browser-like activity. So, for example, following a link in an HTML resource accessed via a proxy service would result in a direct, unproxied request being sent.

19.1 Defining a Proxy Service

To define a proxy service you need to add a configuration file called *configuration-root***`/com/escenic/webservice/proxy/ProxyResourceConfig.properties`** to one of your configuration layers and add the following settings:

`filterHeaders=`*`list`*
: where *list* is a comma-separated list of header field names that you want to be removed from the request before it is forwarded. The list should normally include at least one name - **`Authorization`**. This ensures that Content Engine authentication data is removed from the header.

`serviceMapping.`*`name=url`*
where:

- *name* is the name of the proxy service you want to define.
- *url* is the URL of the target resource represented by the service.

The file can contain any number of **`serviceMapping.`***name* entries, each representing a different proxy service.

A **`ProxyResourceConfig.properties`** file containing two service definitions might look like this:

```
filterHeaders=Authorization,Header0
serviceMapping.myservice1=http://www.myservice1.com/
serviceMapping.myservice2=http://www.myservice2.com/
```

19.2 Defining a Proxy Service Filter

By defining a proxy service filter you can modify the request forwarded by a proxy service in the following ways:

- Remove specified header fields (in addition to any header fields removed by the filter defined in **`ProxyResourceConfig.properties`**).
- Add specified header fields.
- Replace specified strings in the body of the request.

To define a proxy service filter you need to add a configuration file called *configuration-root*`/com/escenic/webservice/proxy/ProxyResourceFilterConfig.properties` to one of your configuration layers and add one setting to it:

`serviceConfigMapping.`*`name=path`*
where:

- *name* is the name of the proxy service for which you want to a filter.
- *path* is the relative path of the configuration file in which the filter is defined.

You can add several such entries, one for each proxy service you want to filter. For example:

```
serviceConfigMapping.myservice1=./MyService1Filter
serviceConfigMapping.myservice2=./MyService2Filter
```

You then need to create the actual filter configuration files you have referenced. In the case of the above example, you would need to create two files in the same folder as your **`ProxyResourceConfig.properties`** file:

configuration-root`/com/escenic/webservice/proxy/MyService1Filter.properties`
configuration-root`/com/escenic/webservice/proxy/MyService2Filter.properties`

A filter configuration file can contain the following settings:

`filterHeaders=`*`list`*
where *list* is a comma-separated list of header field names that you want to be removed from the request before it is forwarded. This filter is applied in addition to any filter specified in your **`ProxyResourceConfig.properties`** file.

`addHeader.`*`name=value`*
where:

- *name* is the name a header field you want to add to the request before it is forwarded
- *value* is the value to be written to the header field

You can specify several such settings in order to add more than one header field to the request.

`replaceMapping.`*`pattern=replacement`*
where:

- *pattern* is a sequence of characters in the request body that are to be replaced.
- *replacement* is the sequence of characters that is to replace *pattern*.

Only the first occurrence of the specified pattern is replaced.??

Here is an example filter configuration file that:

- Removes two headers called **`BadHeader`** and **`WorseHeader`**
- Adds a header called **`GoodHeader`**
- Replaces the string **`credentials-here`** with the string **`verysecretpassword`**

```
filterHeaders==BadHeader,WorseHeader
addHeader.GoodHeader=very.useful.value
replaceMapping.credentials-here=verysecretpassword
```

20 Using Solr

All content items added to the Content Engine are submitted to Solr for indexing. You can therefore use Solr to add search functionality and other related functionality to your publications.

This chapter contains basic information about how the Content Engine indexes content items (that is, what information it submits to Solr for indexing) and a few examples of how you can use Solr for various purposes. It is not intended to be any kind of general guide to using Solr. You should have a general understanding of Solr before reading this chapter, and refer to Solr documentation for detailed descriptions of Solr functionality. A good place to start is here:

http://lucene.apache.org/solr/tutorial.html

20.1 What Gets Indexed

Content items are indexed when they are first created, whenever they are modified, and whenever re-indexing is explicitly requested by the administrator. When a content item is indexed, the Content Engine submits a specially-formatted XML document to Solr. This document contains a set of named **`field`** elements, that Solr can then index. The fields in these documents are described in the following sections.

20.1.1 Standard Fields

The documents submitted for indexing by the Content Engine always contain the following fields:

`objectid`
The content item's database ID.

`publication`
The name of the publication to which the content item belongs.

`author`
The name of the content item author.

`creator`
The name of the content item creator.

`last_edited_by`
The name of the last user to edit the content item.

`state`
The current state of the content engine (**`draft`**, **`published`**, **`deleted`** or **`approved`**)

`source`
The content item's **`source`** property.

`sourceid`
The content item's **`sourceid`** property.

`contenttype`
The name of the content item's content type.

section
The database ID?? of one of the sections to which the content item belongs. This field may appear several times, once for each section to which the content item belongs.

home_section
The database ID?? of the content item's home section.

home_section_name
The name of the content item's home section.

creationdate
The date and time at which the content item was created.

lastmodifieddate
The date and time at which the content item was last modified.

activatedate
The date and time at which the content item was/will be activated.

expiredate
The date and time at which the content item expired/will expire.

20.1.2 Content Type-dependent Fields

In addition to the standard fields, an index document contains one field for each field defined in a content item's **content-type** definition. These fields' names are composed from the field names and types defined in the **content-type** resource, as follows:

field-name_type-name

where:

field-name
is the name of the field as defined in the **content-type** resource. In the special case of the title field (the field designated as the title field with the **ui:title-field** element), **title** is always used, irrespective of the the field's actual name.

type-name
is derived from the field type defined in the **content-type** resource as follows:

content-type Field Type	***type-name***
basic	**text**
number	**double**
boolean	**b**
enumeration	**enum**
date	**date**
link	**link**
schedule	**start_tdate** and **end_tdate**

For example, a content type with the following field definitions in the **content-type** resource:

```
<ui:title-field>headline</ui:title-field>
<panel name="main">
  <field mime-type="text/plain" type="basic" name="headline"/>
  <field mime-type="text/plain" type="basic" name="summary"/>
  <field mime-type="application/xhtml+xml" type="basic" name="body"/>
  <field type="number" name="score"/>
</panel>
```

will result in the following Solr field names:

`title_text` (this is actually the **`headline`** field, which has been designated the title field)
`summary_text`
`body_text`
`score_double`

20.1.3 Tag-related Fields

If a content item has been tagged, then its index document will contain two additional fields for each tag:

- The first field has a name of the form:

  structure`_facet`

  where *structure* is the name of the tag structure to which the tag belongs, **converted to lower case**. The suffix **`facet`** reflects the fact that the content of the field is not a tag name such as **`bangkok`** but a tag URI. All Content Engine tags are identified by URIs internally, not by their external labels (which are not necessarily unique).
- The second field is called **`classification_parent_path`** and contains the tag's path. If, for example, the tag **`bangkok`** belongs to a hierarchical tag structure, then it's **`classification_parent_path`** field would contain the URIs of its ancestor tags **`asia`** and **`thailand`**, separated by spaces.

20.2 How It Is Indexed

Which of the various fields submitted for indexing are actually indexed, and how they are indexed determines what kinds of searches are possible. You can only do date range searches on fields that have been expressly indexed as dates, for example, and you can only search for individual words in fields that have been indexed as text (rather than string) data.

How your content items are indexed is controlled by the Solr **schema**. A default Solr schema, **`schema.xml`** is supplied with the Content Engine, and is used to provide the search functionality required by Content Studio. When you implement search functionality for your publications you can use this schema as a starting point, but are recommended to modify it, since it is unlikely to be well suited to general-purpose search. For further information about this, see **Escenic Content Engine Server Administration Guide**, **section 5.2.2**.

20.3 Example Searches

This section simply provides some examples of things you can do with Solr on a Content Engine web site. The examples are based on the default **`schema.xml`** supplied with the Content Engine. The

examples are shown as HTTP requests that can be submitted directly to your Solr server. The basic format of such a request is:

```
http://your-host/solr/select?query-parameters
```

where:

your-host
is the host name or IP address of the host on which your Solr server is running

query-parameters
is an ampersand-separated list of query parameters. There are many query parameters you can use to control Solr. The parameters used in these examples include:

`q=`*query*
The query itself, for example:

`q=london`
Return all documents containing "london" in any indexed field

`q=city:london`
Return all documents containing "london" in the **`city`** field

`q=*:*`
Return all documents

`wt=`*writer-type*
Determines the format of the returned data. For example:

`q=xml`
Return XML data

`q=json`
Return JSON data

`rows=`*integer*
Determines the number of rows to return.

`facet=`*boolean*
Specifying **`facet=true`** switches faceting on for this query.

`facet.field=`*field-nam*
Specifies a field on which faceting is to be performed. This parameter can be specified several times in the same query.

20.3.1 Faceted Searching

Solr supports faceted searching. Faceted searching is a very useful extension to basic search functionality that allows you group search results in useful ways and perform simple analyses of database content. For an introduction to Solr's faceted search functionality and how to use it, see http://wiki.apache.org/solr/SolrFacetingOverview.

In order to use faceted searching with the Content Engine you must enable it as follows:

1. Add the following line to *configuration-root*`/com/escenic/classification/IndexerPlugin`

   ```
   enableFacets=true
   ```

2. Restart the application server.

3. Re-index the search engine in order to enable faceted search for existing content items.

   Note that re-indexing may take a long time.

20.3.1.1 Content Items per Publication

The following query will return XML data containing the numbers of content items in each publication at a site:

```
http://your-host/solr/select?q=*:*&wt=xml&rows=0&facet=true&facet.field=publication
```

20.3.1.2 Tags Related to a Search

At an installation with a tag structure called **`Topics`**, the following query will return:

- Information about the first 20 content items containing the word "christmas" in the **`title`** field
- A list of the **`Topics`** tags attached to all content items containing the word "christmas" (not just the first 20), and how many content items are tagged with each tag.

```
http://your-host/solr/select?
q=christmas&wt=xml&rows=20&facet=true&facet.field=topics_facet
```

The tags related to a search can be used to:

- Provide links in the search result that allow the user to narrow the search results. A user who is interested in the history of Christmas, for example might click on a "History" link to see a list of items that both match the search and are tagged with the keyword "History".
- Generate a tag cloud for the search. The faceting results include the number of content items tagged with each tag, which can be used to determine the relative sizes of the links in the cloud.

The index fields generated for tag structures always have lower case names. So even though the tag structure is called **`Topics`**, the corresponding index field is called **`topics_facet`**. The above query will not work if the field name **`Topics_facet`** is used.

21 Content Item Staging

Content item staging simplifies the process of working with published content items. It is enabled by default and you are recommended to use it.

Without content item staging, any changes made to a published content item are published as soon as they are saved. Users who want to make changes "in private" before publishing are forced to work around this limitation in some way - for example by working on a copy and then copying the final changes into the published content item. Such workarounds are both time-consuming and error-prone.

Content item staging makes such workarounds unnecessary. When a user saves any changes to a published content item, the content item is automatically duplicated: a new draft version of the content item is created and the changes are saved in this draft version. The published version remains unmodified. From this point forward, whenever the content item is opened in Content Studio, it is this revised draft that users see. Visitors to the site, however, still see the unmodified published version.

Working on a revised draft in Content Studio is just the same as working on a draft that has never been published. It can be moved through the **submitted** and **approved** states in the same way as an unpublished content item, and published in exactly the same way as the original version. When a revised version is published, it replaces the original published version and site visitors see the revised version.

21.1 Disabling Content Item Staging

Content item staging is enabled by default and you are recommended to use it. There are, however, some situations in which you may need to disable it. Some of the Content Engine plug-ins, for example, have content item types for which staging must be disabled.

You can disable/enable staging in 3 different ways:

Global
To disable content item staging completely, add `articleStaging=false` to `ServerConfig.properties` in one of your configuration layers. This sets the system property `com.escenic.article.staging` to `false`.

Per publication
To disable content item staging for a specific publication, set the publication feature `com.escenic.article.staging` to `false` (see **Escenic Content Engine Resource Reference**, **chapter 6**). If staging is globally disabled, then you can enable it for a specific publication by setting `com.escenic.article.staging` to `true` instead.

Per content type
To disable content item staging for a specific content type, add a `parameter` element (see **Escenic Content Engine Resource Reference**, **section 2.16**) to the content type definition in the `content-type` resource. The `parameter` element must have the name `com.escenic.article.staging` and the value `false`. If staging is globally disabled, then you can enable it for a specific content type by setting `com.escenic.article.staging` to `true` instead.

21.2 Requirements

An installation where content item staging is enabled has a few additional requirements:

- Presentation hosts **must** use a separate `solr` engine from editorial hosts. This is already the recommended configuration for production installations, but if content item staging is enabled then it is an absolute requirement. The Content Engine provides two indexer web services for logging updates to content items: one of them (`index`) logs all changes and the other (`presentation-index`) excludes changes made to staged content items. One `solr` engine must be set up to use the `index` web service in order to generate an internal index for use by Content Studio, while the other must be set up to use `presentation-index` and generate an index for use by the presentation layer. For further information, see **Escenic Content Engine Server Administration Guide**, **chapter 5**.
- If your application includes any event listeners, you will probably need to extend them to accept events for staged content items. A new marker interface, `neo.xredsys.api.StagedEventListener`, has been added for this purpose. For detailed instructions, see section 9.1.1.
- If your application includes any transaction filters then you will probably need to modify them. The `TransactionFilter` interface has a new method called `doStagedUpdate()`. The abstract class `neo.xredsys.api.services.TransactionFilterService` has also been updated with a "do nothing" implementation of this method. This means that if your transaction filter is based on `TransactionFilterService`, it will ignore updates to staged content items unless you add an implementation of `doStagedUpdate()` to it. If your transaction filter is **not** based on `TransactionFilterService` then failure to add a `doStagedUpdate()` implementation will result in many `NoSuchMethod` exceptions. For further information see chapter 10.
- If your application includes any post-transaction filters, then they will probably need the same kinds of modifications as transaction filters.

22 Further Reading

This chapter lists various resources for further reading.

22.1 Escenic 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

22.2 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