Escenic Content Engine

Template Developer Guide

6.3.2-1

escenic

A CCI COMPANY

Table of Contents

1 Introduction

The Escenic Content Engine is a platform for building large, sophisticated web sites. It provides editorial staff with a streamlined production environment in which they can concentrate on the production, editing and publishing of content within a predefined web site structure.

The definition and maintenance of the web site structure (and all the layout issues associated with it) is regarded as a completely separate concern, and is the subject of this manual.

The structure and layout of an Escenic web site is defined by a set of Escenic **templates**. The whole web site is generated "on demand" by merging a small number of templates with content retrieved from a database. An Escenic template is a **JavaServer Pages** (**JSP**) document, a file containing a mixture of HTML and JSP tags. The HTML tags define fixed layout elements, while the JSP tags represent dynamic elements that change according to the specific content requested.

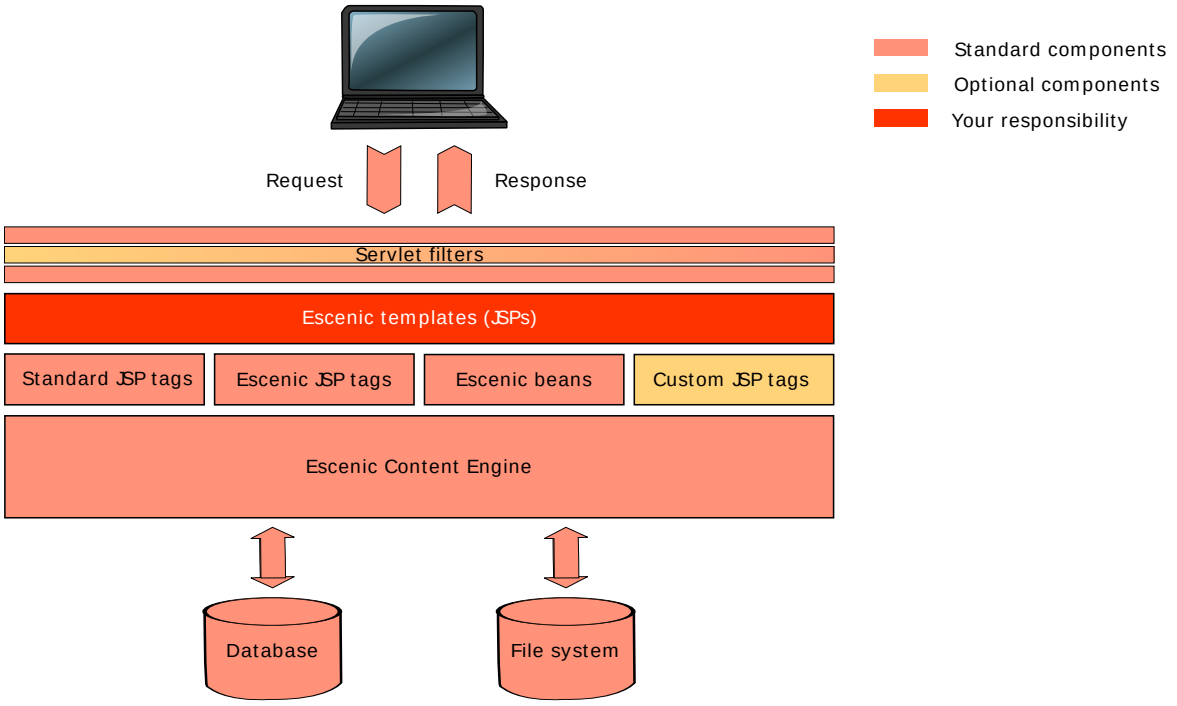
In order to be a successful Escenic template developer, you need a general understanding of:

- The Content Engine's architecture and object model
- The web technologies on which the Content Engine relies

1.1 About the Escenic Content Engine

1.1.1 Architecture

The following diagram shows a very simplified view of the Escenic Content Engine:



The Content Engine stores all textual website content in a relational database, in the form of documents called **content items**. It also stores a lot of **metadata** about these content items in the database: author names, editing and publishing history, subject matter keywords, current workflow status and so on. Finally, it also uses the database to hold information about the structure of the web site as a whole and where individual content items belong in that structure. Images and multimedia objects are stored as files in the file system, but the information required to locate the files and all metadata about them is stored in the database along with the content items.

The Content Engine manages all storage and retrieval of this website content, so that you as a template developer don't need to know anything about where or how the data is stored. All of the web site's content and structure is presented to you in the form of an **object model**, which you can access using **JSP tags**.

In the diagram above, the orange layer, **Escenic templates**, is the one you are responsible for. The pink elements are standard components of the delivered system, while the yellow elements are custom items that may or may not be present in your system.

An incoming HTTP request from a website user is passed through a series of **servlet filters** called a **filter chain**. These filters are Java classes that prepare the request for processing by:

- Analyzing the address in the request (the URI)
- Creating the Escenic objects or **beans** your templates will need to be able to handle the request
- Adding references to these beans to a bean that represents the request

The templates are files containing a mixture of HTML tags, JSP expressions and JSP tags. The JSP expressions and tags let you access information stored in the Escenic beans that have been created by the filter chain. In this way, templates are able to generate a response by combining fixed HTML layout elements with dynamic information retrieved from the database via the beans. The response is returned via the same filter chain which can, if necessary, modify it in some way - reformat it to fit a small-screen device, for example.

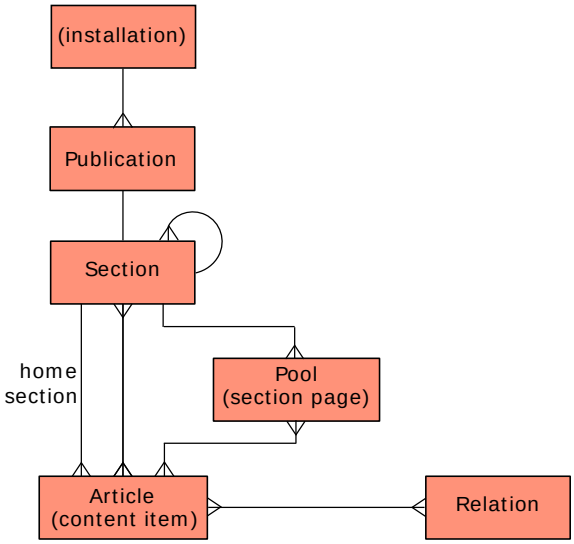
This section has introduced a lot of new terms and concepts. For more information about the Escenic object model, see section 1.1.2. For more information about templates, servlets, filters, JSP tags, beans and so on, see section 1.2.

1.1.2 Object Model

You will be able to work more effectively with the Content Engine if you understand the object model on which it is based.

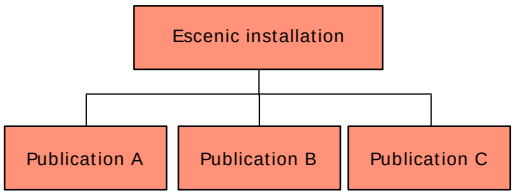
1.1.2.1 Basic Object Model

The following illustration provides a simplified view of the object model provided by the Escenic Java Application Programming Interface (API):

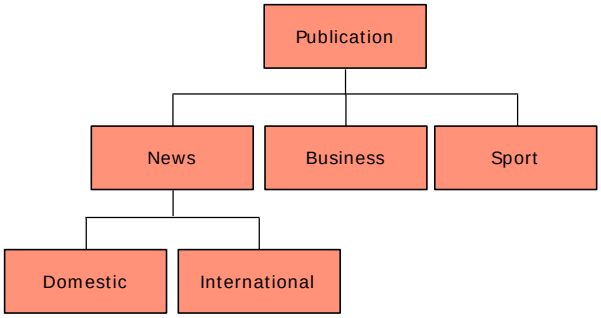


What the model tells you is this:

- An Escenic **installation** can be used to manage one or more web sites called **publications**. A magazine publisher, for example, might use a single Escenic installation to maintain one publication for each of it's print magazines:

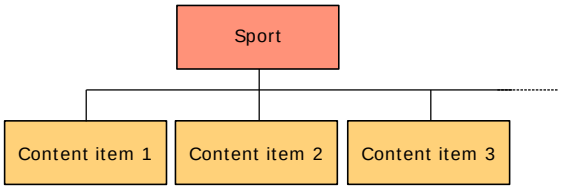


- Each publication may contain **one** root **section**. This root section may contain any number of subsections, each of which may contain further subsections. This allows you to create a **tree** or **hierarchy** of sections, which may be as large and deep as you want it to be. It determines the structure of a publication and usually provides the basis for the primary menus used to navigate the web site. Here, for example, is a very simple section hiearchy:

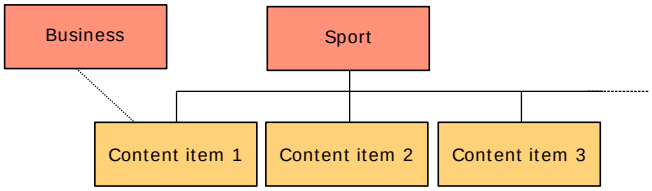


- A section may contain any number of **content items** (represented by `Article` objects). Content items are the basic text, graphical and multimedia components from which a publication is

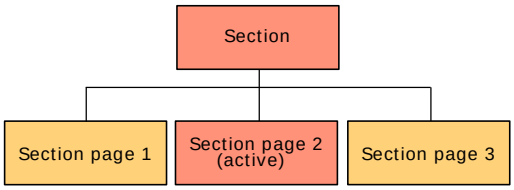
constructed: news articles, opinion pieces, images, video clips, sound files, attachments such as PDF files and so on:



- A content item may **appear** in many sections, although it only **belongs** to one of them, called its **home section**. A news story about soccer transfer fees may belong in the sports section of a newspaper, but also appear in the business section, for example:



- A section may own any number of **section pages** (represented by `Pool` objects). A section page is a "front page" or index for a section - it is what the reader of a publication sees when they click on a section link. It contains a number of links to content items in the section. Although a section may have several section pages, only one of them can be **active** (that is published in the publication):



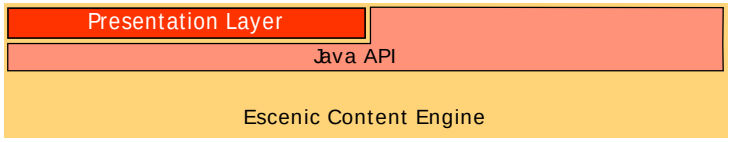
  Section pages have an internal structure involving further objects, which we will look at more closely in section 1.1.2.2.

- A content item can have relations to other content items. A content item containing a news article, for example, might have relations to images that appear in the articles, other articles on the same subject (that appear in the published article as links), and background video clips.

The package name of the Escenic API classes is **`neo.xredsys.api`**, so the fully qualified class names are **`neo.xredsys.api.Publication`**, **`neo.xredsys.api.Section`**, **`neo.xredsys.api.Article`**, etc.

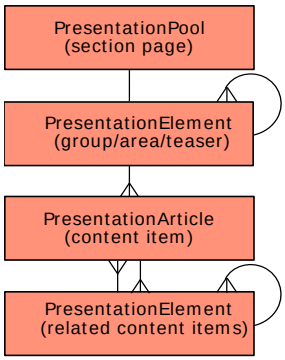
1.1.2.2 Presentation Object Model

As a template developer you need to know the overall structure of the Java object model described above. However, you will primarily work with a simpler object model called the **presentation object model**. The presentation object model is provided by a layer of software in the Content Engine called the presentation layer:

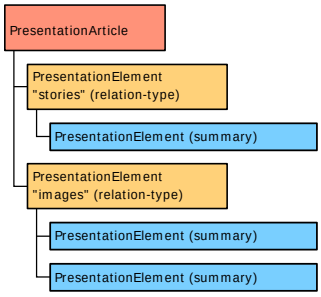


The purpose of the presentation layer is to provide template developers with a set of objects that are both simpler to use and more efficient than the underlying API objects. The presentation layer only contains presentation objects for the most important and frequently-used API objects however, so you will need to deal with both presentation objects and API objects. This is why the presentation layer is shown as an incomplete layer in the above diagram.

The main objects in the presentation object model are:

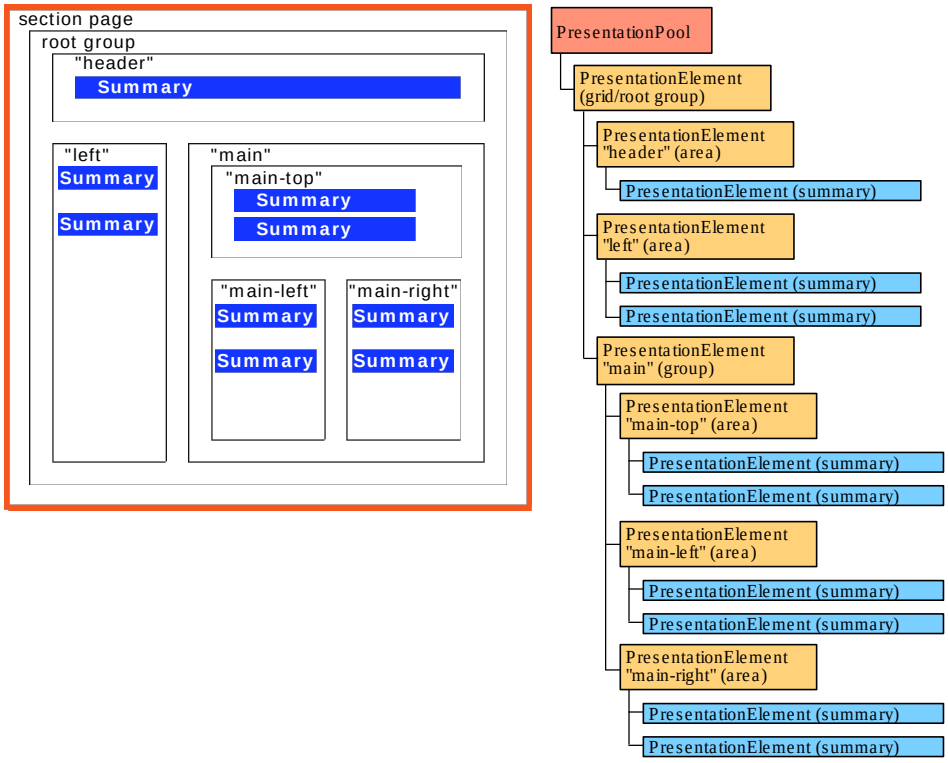


The **`PresentationArticle`** object has a direct one-to-one relationship with the API **`Article`** object, and represents a content item. The **`PresentationElement`** object represents other content items. A **`PresentationArticle`** contains one **`PresentationElement`** (referred to as **summary**) for each related content item. **`PresentationElements`** are grouped by **relation type** (see chapter 6).



The **`PresentationPool`** object has a one-to-one relationship with the API pool object, and represents a section page. It contains one **`PresentationElement`** object which represents the grid used to organize the layout of links on the section page. A **`PresentationElement`** object can contain other **`PresentationElement`** objects to form a tree of **groups** and **areas** that can represent the

logical structure of a complicated multi-column layout. PresentationElement objects can also represent the content item "teasers" (called **summaries**) displayed on section pages.



The package name of the presentation classes is **`neo.xredsys.presentation`**, so the fully qualified class names are **`neo.xredsys.presentation.PresentationPool`**, **`neo.xredsys.presentation.PresentationArticle`**, etc.

When you are designing your templates you will need to access both information held in presentation objects and information held in API objects. In both cases the objects are made available to you as **JavaBeans**. For more information about this and other Java-based web technologies, see section 1.2.

1.1.3 Content Items and Content Item Types

Content items are the central objects in the Escenic object model. All the other objects are containers for organizing content items. Content items are generic containers for all the kinds of content you might want on your web site: news stories; magazine articles; theater, film, book and restaurant reviews; obituaries; interviews; stock market reports; photos; video clips; audio files; attached documents such as PDF files - the list is very long.

In addition to holding all these different kinds of content, content items also contain additional information **about** the content: metadata such as the name of the author and the article's publication history. Content items can also contain **relations** to other content items. A news story, for example, might contain relations to:

- Images to be displayed with the story
- Related news stories to be displayed as links in a "More about..." box
- A background video report to be displayed alongside the story.

Moreover, content items have an internal structure (title, lead text, body, for example) that usually varies according to content type, and different organizations can have very different requirements with regard to content item structure. For this reason, articles are customer-definable objects. All the **content types** required in a particular Escenic installation are specified in an content type definition file called the `content-type` resource.

The `content-type` resource defines and names all the content types available to the Content Engine, and for each content type it specifies:

- A set of **fields**. All the information in an article is stored in fields. An article will usually have at least a **body** field for the main content and a **title** field (although different names may be used) plus a range of other fields that vary according to the content type.
- Optionally, a set of **relation types**. A relation type is simply a name used to classify an article's relations to other objects (other articles, images, multimedia objects, external links and so on). For further information about relation types, see chapter 6.

Since it defines the structure of all the articles in an Escenic installation, the `content-type` resource is obviously of central importance. It determines:

- What is stored in the database
- What is displayed in the Content Studio user interface
- The structure of the `PresentationArticle` and `PresentationRelationArticle` beans available to you when writing your templates

In order to be able to write Escenic templates you have to know what content types, fields and relation types are defined in the `content-type` resource. For further information about this, see section 5.1.

Alongside the content-type resource files is another important resource file called the `layout-group` resource. This resource defines the components of section pages (**groups**, **areas** and **summaries**, see section 1.1.2.2) and their logical relationships. For further information about the layout-group resource, see section 1.3.3.

1.2 Java Web Technologies

The Escenic Content Engine makes use of a number of web technologies that you need to know about:

- Web applications
- Application servers
- Servlets
- JavaServer Pages
- Java Beans
- JSP expression language
- Tag libraries
- Servlet filters

1.2.1 Web Applications

In general terms, a **web application** is an application that runs on a centralized **application server** and is delivered to its users over the Internet using web technology: that is, the **HyperText Transfer Protocol** (**HTTP**), the **HyperText Markup Language** (**HTML**), and **web browsers**. The term web application is generally used to describe web sites (or parts of web sites) that serve dynamic content rather than those which are simple repositories of static documents. Nevertheless, like a simple web server, the basic function of a web application is to accept HTTP requests from a client (usually a browser) and return responses to those requests, usually in the form of an HTML document.

Over the years, a number of additional web application technologies have been developed to supplement the basic WWW components, with the aim of improving the application user experience and simplifying the application development process. The web application technologies used by the Escenic Content Engine are all based on the **Java Enterprise Edition** (**JEE**) standard. This standard specifies a set of Java-based components and standard interfaces between them. It has the advantages of being very widely used and supported, and of platform-independence.

In the Java world, the term **web application** has a very specific meaning - it is used to denote a web application that conforms to the JEE standard. In order to simplify deployment, JEE web applications have a standardized structure:

- A single root folder
- A **`WEB-INF`** folder directly under the root, containing an XML file called **`web.xml`** that describes the content and structure of the application.
- Deployed as a ZIP file with the filename extension **`.WAR`** (for Web ARchive)

The Escenic Content Engine can be regarded as a framework for building a particular kind of JEE web application.

1.2.2 Application Servers

An application server is a container for web applications. Some application servers are add-ons to web servers, and just provide the additional functionality (such as database access) needed to support dynamic web applications, while others include their own web server. An application server's main function is to provide all the "plumbing" needed to connect the business logic encapsulated in a web application with an organization's back-end infrastructure.

In addition to this central function, application servers may also offer a variety of additional features such as built-in redundancy, high-performance distributed application services and support for complex database access.

Escenic web applications can run on any application server that is J2EE-compliant.

1.2.3 Servlets

A **servlet** is a Java class that implements the Java Servlet API and can therefore run as a web application in a JEE-compliant application server. Like any other web application it receives and processes HTTP requests, and returns responses to those requests, typically in the form of HTML or XML documents.

For more information about servlets, see http://java.sun.com/products/servlet/docs.html.

1.2.4 JavaServer Pages

JavaServer Pages (**JSP**) is an extension to Java servlet technology that allows web developers to build web applications without actually writing Java code. It defines a set of JSP **actions** or **tags** that can be combined with ordinary HTML or XML tags to produce dynamic web pages. A JSP tag looks rather like an HTML or XML tag (it is enclosed in angle brackets) but is actually a placeholder for some Java code that will ultimately return dynamic data.

JSP also defines an **expression language** that can be used to access values stored in **JavaBeans** (see section 1.2.5).

JSP files are called JavaServer Pages or JSPs, and are identified by the extension **.jsp**. The first time an application server receives a request for a JSP it:

1. Passes the file to a translator that translates the file to a servlet.
2. Compiles the servlet.
3. Executes the servlet, passing in the initial request as a parameter.
4. Returns the servlet output to the requesting client.
5. Caches the compiled servlet for future use.

The next time a request for the same JSP is received, the application skips the translation and compilation steps, and passes the request directly to the cached servlet.

JSPs are therefore, for all practical purposes, just an easier way to write servlets. JSPs and servlets are interoperable. A JSP can include output from a servlet or forward its own output to a servlet, and a servlet can include output from a JSP or forward its output to a JSP.

Escenic templates are JavaServer Pages. As a template developer, you will spend most of your time writing JSPs.

For more information about JavaServer Pages, see http://docs.oracle.com/javaee/5/tutorial/doc/bnagx.html.

1.2.5 JavaBeans

JavaBeans are Java language classes that have been written to comply with the JavaBeans specification. This specification defines various rules concerning how instances of a class are to be created, how properties are to be accessed and so on. Objects that conform to the JavaBeans specification are commonly referred to as **beans**, and can be easily accessed and manipulated using a variety of standard J2EE-based tools.

The application objects manipulated by a servlet are always implemented as beans. In the Escenic Content Engine, for example, **publications**, **sections** and **content items** are represented by beans.

JavaBeans are not to be confused with Enterprise JavaBeans. For more information about JavaBeans, see https://docs.oracle.com/javase/tutorial/javabeans/quick/index.html.

1.2.6 JSP Expression Language

The JSP expression language gives a JSP programmer direct access to the content of any beans available from the current context. The Escenic Content Engine, for example, makes the requested content item and section returned in response to a user request available to you as variables called

article and **section**. Using the JSP expression language, you can therefore include a content item's **title** field in a page by entering the expression:

```
${article.fields.title}
```

For more information about the JSP expression language, see http://www.oracle.com/technology/sample_code/tutorials/jsp20/simpleel.html.

1.2.7 Tag Libraries

A JSP file is usually a combination of standard HTML, JSP expressions and JSP **actions** or **tags**. A JSP tag looks like an HTML/XML tag, but it is in fact a placeholder encapsulating some Java code that gets executed when the JSP is processed by the application server.

An Escenic application includes two main types of JSP tags:

JavaServer Pages Standard Tag Library (JSTL) tags
As its name suggests, JSTL is a library of standard tags that provide a lot of generic functionality that is useful in many applications. The JSTL library provides tags for flow control (looping, if and so on), string handling, number/date formatting, and querying SQL and XML data sources. For information about tag libraries in general and the JavaServer Pages Standard Tag Library (JSTL), see http://docs.oracle.com/javaee/5/tutorial/doc/bnakc.html.

Escenic tags
The Escenic Content Engine includes some tag libraries containing specialized tags for performing functions that cannot easily be carried out using the expression language or JSTL tags. For further information, see section 1.3.1.

An Escenic application may contain other types of tags: there is nothing to prevent you using other third-party tag libraries if you find them useful, or writing your own if you have special requirements and are a Java programmer. However, most users' requirements can be met using a combination of the JSP expression language, JSTL tags and Escenic tags.

1.2.8 Servlet Filters

Servlet filters are optional components of J2EE-based webapps. They are Java classes that can be wrapped around a 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 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 Escenic Content Engine has a standard filter chain that is used to prepare inbound requests for processing. The primary functions of this chain are to:

- Parse the request URI to determine what Escenic object is being requested.
- Verify that the requested object is available.

- Get the presentation information needed to display the requested object.
- Prepare any images needed to respond to the request.

For further information about servlet filters in general, see http://java.sun.com/products/servlet/Filters.html. For detailed information about the Escenic Content Engine's servlet filters, see EscenicStandardFilterChain.

1.3 Your Toolkit

When the application server you are using receives an HTTP request for a page in an Escenic publication, it is passed to the Escenic request filter chain described in section 1.2.8. The last filter in the chain passes it to the Escenic master template file (usually called `common.jsp`). The filters in the filter chain have verified that it is a legitimate request, created the beans you will need to respond to the request and placed them in the **request scope** so that you can access them using the JSP expression language and tags.

Your job, then, is to create a `common.jsp` file that will generate appropriate responses to the requests. (Usually you will want to break the task up into a number of smaller, more specialized JSP files: `common.jsp` then just operates as a "clearing house" that examines the requests and forwards it to another JSP file.)

You have the following tools at your disposal in responding to requests:

- HTML to create the static layout
- The beans that have been created for you and the information they contain
- The JSP expression language for accessing the information in the beans
- JSTL tags for
  - creating logical constructs such as if statements and loops
  - passing control between templates
  - formatting retrieved values such as numbers and dates
- Escenic tags for accessing information that cannot be extracted from a request's beans using the expression language alone
- The `content-type` resource file, which tells you what types of content items to expect, and the fields and relations the different content types contain.
- The `layout-group` resource file, which tells you what section, grid and element templates you need to create, and also defines the logical structure of a publication's section front pages.

1.3.1 The Escenic Tag Libraries

You are recommended to use standard JSTL tags together with the JSP expression language as far as possible in your Escenic template files. The Escenic tag libraries contain specialized tags for accessing and manipulating Escenic beans and templates in ways that are difficult to achieve using standard JSP functionality.

The most important Escenic tag libraries or **taglibs** are:

publication Tag Library
: Contains tags for accessing Publication beans.

article Tag Library
: Contains tags for accessing PresentationArticle beans.

section Tag Library
: Tag LibraryContains tags for accessing Section beans.

template Tag Library
: Contains tags for transferring control between templates.

util Tag Library
: Contains a variety of useful tags.

collection Tag Library
: Contains tags for creating and manipulating standard Java collection beans.

view Tag Library
: Contains tags for accessing View beans. These are objects that can contain selections of objects from tree structures such as section hierarchies.

> The Escenic tag libraries were a more important component of the Escenic development environment in earlier versions of the Content Engine. For this reason the taglibs contain a large number of **deprecated** tags. These tags remain available, so that old applications which make use of them will still work. You should not, however, use them in new applications - the same functionality can be achieved using JSTL tags and/or JSP expressions. They are clearly marked as deprecated in the Escenic Content Engine Bean Reference.

1.3.2 The content-type Resource

The **`content-type`** resource is mainly important as a source of information for the template writer, since it defines the structure of the content items you will be dealing with. This expression:

```
<h2>${article.fields.title}<h2>
```

will only return a value if the current **`article`** bean actually has a "title" field, and the place to find out what fields your article beans contain is the **`content-type`** resource.

For more information about the content-type resource, see section 5.1.

> The extent to which you regard the **`content-type`** resource purely as a **source** of information depends upon your responsibilities. If you are a "pure" template developer, then this will be the case. In many organizations, however, the "template developer" role is combined with the "information architect" role. If you have "information architect" responsibilities, then you will also be responsible for creating this file, which plays a central role in determining the underlying structure of an Escenic publication.

1.3.3 The layout-group Resource

The **`layout-group`** resource is also mainly a source of information for the template writer. It defines the **logical structure** of the section pages. That is, it specifies the **`group`**s and **`area`**s a section page is composed of, the relationships between them, and the **`content-types`** of the summaries that may appear in each area. It does not specify the actual layout of these items, but the layout you create with your templates must match this logical structure.

As with **`content-type`**, The extent to which you regard the **`layout-group`** resource purely as a **source** of information depends upon your responsibilities. If you are a "pure" template developer, then this will be the case. In many organizations, however, the "template developer" role is combined with the "information architect" role. If you have "information architect" responsibilities, then you will also be responsible for creating this file. This file is, however, far less important than the **`content-type`** resource.

1.4 What Next?

Hopefully, after reading this introduction you now have a general understanding of:

- The general architecture of the Escenic Content Engine
- The object model used to represent the structure and content of Escenic web sites
- The technologies on which the Escenic Content Engine is based
- How you can use Escenic templates to generate HTML pages containing articles retrieved from the Content Engine

There are, however, still a few things you need to know before you are ready to start making your first templates:

- Web application structure
- The template development process

In addition, if you are completely new to JSP programming, then it would probably be a good idea to find out a bit more about:

- JavaServer Pages (JSP) - see http://docs.oracle.com/javaee/5/tutorial/doc/bnagx.html.
- JavaBeans - see https://docs.oracle.com/javase/tutorial/javabeans/quick/index.html.
- JSTL - see http://docs.oracle.com/javaee/5/tutorial/doc/bnakc.html.
- JSP expression language - see http://www.oracle.com/technology/sample_code/tutorials/jsp20/simpleel.html.

1.4.1 Web Application Structure

All J2EE web applications are stored in **WAR (Web ARchive)** files. A WAR file is in fact a ZIP file with a special internal directory structure and the extension **`.WAR`**. Escenic publications are web applications and are therefore also always stored in WAR files. In order to work on an Escenic publication you:

- Unzip the WAR file
- Edit one or more of the files in the unzipped folder tree
- Zip up the folder tree again into a web archive

The folder tree you get when you unzip a WAR file is often referred to as an **exploded** WAR file. An exploded Escenic publication has the following minimum structure:

publication-name
`META-INF`

```
  escenic
    publication-resources
      escenic
        content-type
        feature
        image-version
        layout-group
  MANIFEST.MF
WEB-INF
  template
    jsp-files
  web.xml
index.jsp
```

You should not usually need to create this structure from scratch (to create a new publication you can just copy and modify an existing one), all you need to know is where to look for the files you want to edit, and where to add new files you want to include. The files you will usually need to modify in this structure are:

- The template JSP files in *publication-name*`/WEB-INF/template`.
- The files `content-type`, `feature`, `image-version` and `layout-group` in *publication-name*`/META-INF/escenic/publication-resources/escenic`. These files are usually referred to as **publication resources**.

You may also need to add new JSP files to *publication-name*`/WEB-INF/template`. How you organize the files in *publication-name*`/WEB-INF/template` is entirely up to you: you can store them all in the same folder or create subfolders if you wish. The only requirement is that `common.jsp` (the "startup" file initially called by the Content Engine when responding to a request) is stored in *publication-name*`/WEB-INF/template`, not in a subfolder. (If you need to, you can in fact set up the Content Engine to use a different startup file name and location. For details, see The default.properties File.

The folder structure shown above is a **minimum** structure: other files and folders may be present in a publication WAR file. The **Escenic assembly tool**, a tool delivered with the Content Engine to simplify the process of assembling and deploying publications will usually add some files and folders to this minimum structure. You can ignore all of these additions.

1.4.2 Development Process

Developing Escenic templates involves editing various JSP and XML files in different locations, creating WAR archive files, uploading them to your application server, using various Escenic applications to modify settings, restarting the web application periodically and viewing the pages generated by your templates.

You will find it a lot easier to learn the details of template development if you have a clear understanding of this cycle from the very beginning.

There are two main components you have to deal with when developing your templates:

The application server
: The application server is responsible for displaying your publication. You will not see the results of any changes made to the appearance of your publication unless you update the application

server's copy of your publication. This process of copying an updated version of your publication to the server is called **deployment**.

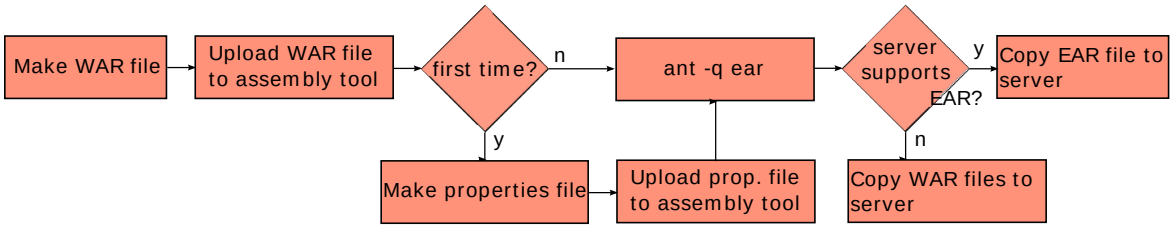
The Content Engine

The Content Engine is responsible for managing the structure of your publication, and the editing environments presented to editorial staff in Content Studio. Any changes made to the structure of your publication (such as adding a new field to an article, modifying a field creating a new content type or template) will not take effect or become visible in Content Studio until you upload your changes to the Content Engine and update the publication.

The following sections describe the procedures to follow in various cases.

1.4.2.1 Deployment

The exact deployment procedure varies slightly according to whether or not your application server supports EAR deployment. Most application servers do, but Tomcat does not.



The procedure involves the following steps:

1. Zip up the modified folder tree using any zip-capable archiving utility. If necessary, rename the resulting archive from **`.zip`** to **`.war`** (some utilities will do this for you). The archive should not contain the application's root folder. That is, if you have an application tree in which the root folder is called **`myapp`**, you should create an archive called **`myapp.zip`** with the **same** contents as the **`myapp`** folder.
2. Copy the WAR archive to the **`escenic/assembly-tool-`***version*`/publications`** folder on your application server. If you are working on a different host from the server then you will need to use a remote copying tool to do this.
3. If you are deploying the publication for the first time then you also need to create a **publication properties** file describing the publication and copy it to the same location as the WAR archive. A publication properties file is a simple text file with the same name as the publication WAR archive, the extension **.properties** and the following contents:

   ```
   name:          publication-name
   source-war:    archive-name.war
   context-root:  /publication-name
   ```

   where *publication-name* is the name to be used for the publication and *archive-name* is the name of the publication archive. *publication-name* and *archive-name* can be the same but should probably not be (you may want to create several publications based on the same publication definition).
4. Log in on the application server host, **`CD`** to the **`escenic/assembly-tool-`***version* folder and enter:

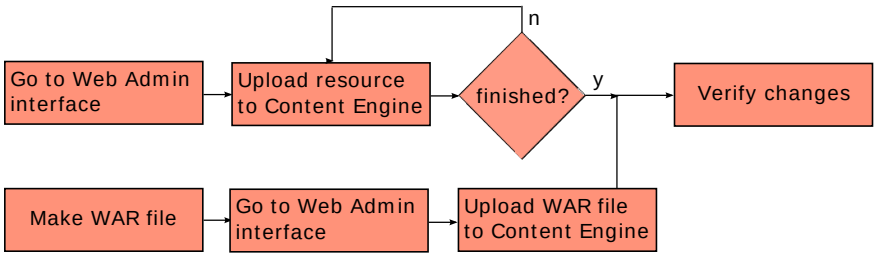
   ```
   ant -q ear
   ```

5.
   - If your application server supports EAR file deployment, copy **`escenic/assembly-tool-`***version*`/dist/engine.ear`** to your application server using the particular server's recommended deployment procedure.
   - If your application server does **not** support EAR file deployment, copy **`escenic/assembly-tool-`***version*`/dist/war/myapp.war`** to your application server using the particular server's recommended deployment procedure. If this is the first time you are deploying an Escenic publication, then you should copy all the WAR files you find in the **`escenic/assembly-tool-`***version*`/dist/war`** folder. If it is not the first time then you only need to copy the publication war file.

> You must use the assembly tool as described above to deploy publications, even if you are only redeploying a single WAR file: the assembly tool modifies the WAR file you supply to it, adding libraries, patches and other important components. But don't worry - in a normal development environment you won't need to to do it very often as you can do most of your development directly on a deployed copy of the application - see section 1.4.2.3.

Once you have completed deployment, you should visit the publication using a browser to check that it has actually been deployed/redeployed.

1.4.2.2 Creating/Updating Publication Resources



Information about the structure of a publication is supplied to the Content Engine in the form of files called **publication resources** (see chapter 5). If you change any of these resources (or create a new set of resources defining a new publication), then in order to update/create your publication, you must:

1. Start a browser and visit the Escenic Web Administration interface. This usually has the URL **`http://`***server-name*`:`*port*`/escenic-admin`** where *port* is the port number the server is listening on.
2. Upload the resources using the **New pubs** option if you are creating a new publication or **List pubs** > **Update resources** if you are updating an existing publication.

Instead of uploading the resources individually in this way, you can instead create a WAR file containing all your changes and upload the WAR file in the same way. You can use the same WAR file that you use for deployment for this purpose, and this is often the most convenient method of updating publication resources.

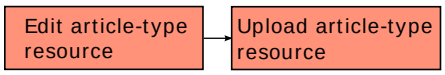
When you have uploaded the resources, you can open the publication using Content Studio to verify that the publication has been created or that your changes have taken effect.

1.4.2.3 Making Template Changes

Once you have deployed a publication on your development application server, you can work directly on the files in the application server's exploded version of the publication. This is the most effective method of development, since any changes you make to the templates are immediately visible in the application - no deployment step is necessary. You can use this method as long as all you are doing is making changes to existing templates.

If for some reason you cannot work directly on the application server, or if you need to copy your changes to another server (a test server, for example), then you will need to periodically redeploy your application as described in section 1.4.2.1.

1.4.2.4 Changing Content Item Types



In order to change a content type (for example, add, remove or change the content type's fields), add a new content type or remove one, you must:

1. Edit the **`content-type`** resource. For details of how to do this, see section 5.1.
2. Upload the changed **`content-type`** resource to the Content Engine see section 1.4.2.2).

If you have changed any templates to take advantage of the content type changes and you are not working directly on the application server then you will also need to redeploy your application as described in section 1.4.2.1.

2 Getting Started

In order to start learning how to create and modify Escenic templates you need access to:

- A working Escenic installation
- A simple Escenic publication

The Escenic installation does not have to be on your PC (usually it won't be, it will be installed on a server in your network).

For information on how to install the Escenic Content Engine, see the Escenic Content Engine Installation Guide.

The machine on which Escenic is installed must also have an Apache Ant installation. You can get Ant from http://ant.apache.org/.

On your PC you will need a Java VM installed so that you can use the Escenic editing program, Content Studio. You can find the latest version of the Java VM for your operating system here: http://www.java.com/en/download.

2.1 Installing an Example Publication

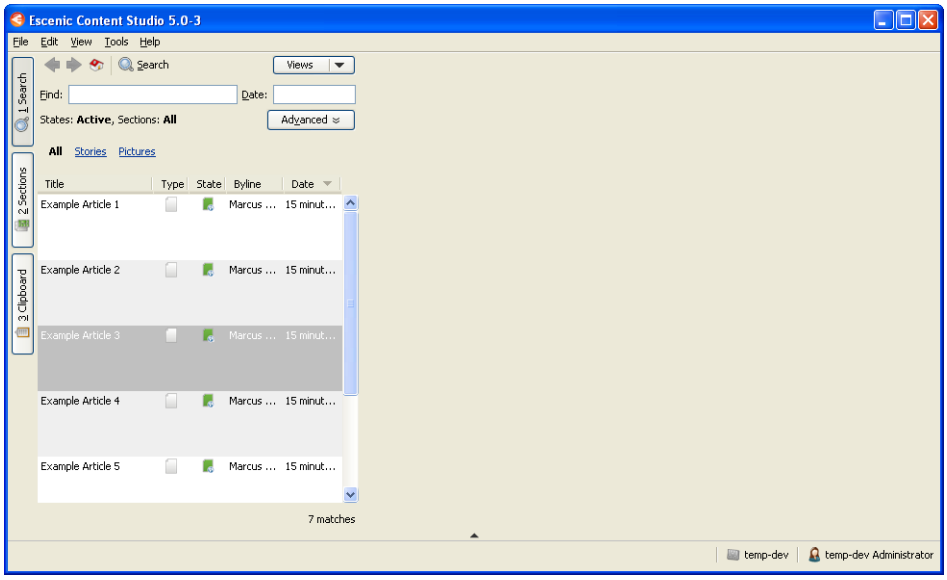
You are recommended to use the **temp-dev** publication for learning purposes. This publication is intentionally very simple, and it is used in most of the examples in this guide. You should find the publication WAR file (`temp-dev.war`) in the `<engine.root>/contrib/wars/` folder of your Escenic installation.

Get a copy of `temp-dev.war` and upload it to the Content Engine as described in section 1.4.2.2. As part of this process you will be asked to specify a **name** for the publication, and an **administrator password**, which you will need to use later. For a more detailed description of how to create new publications, see the Escenic Content Engine Installation Guide.

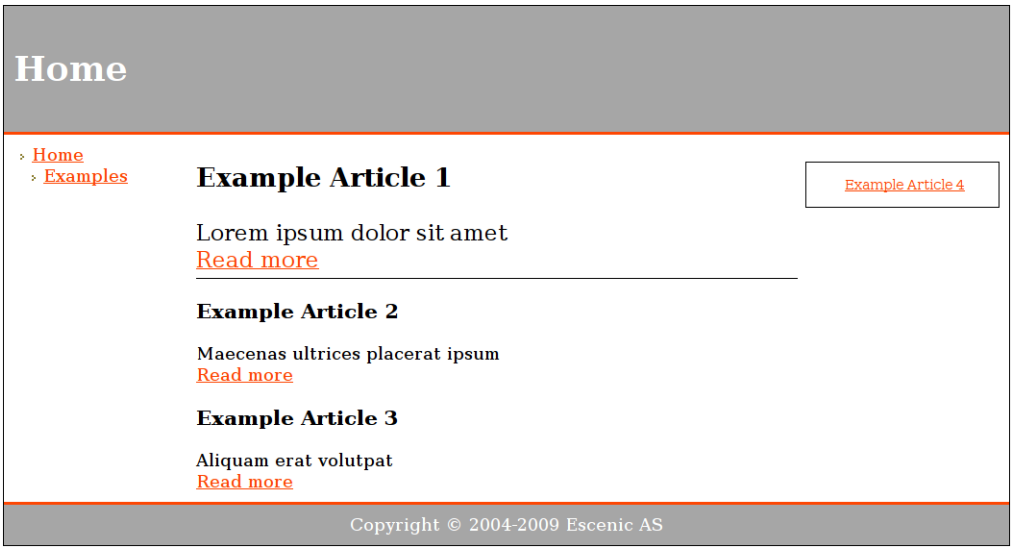
You should now verify that the publication is correctly installed in the Content Engine. To do this:

1. Point your browser to `http://`*server-name*`:`*port*`/studio`.
2. Click on the **Launch Escenic Content Studio** button displayed on this page. Escenic Content Studio is then downloaded to your PC (this can take a while the first time you do it). When downloading is complete, a login dialog is displayed.
3. Log in to the new publication. You must use a username formed by adding "_admin" after the name of the publication. If, for example, you called the publication **temp-dev**, then the admin user name is **temp-dev_admin**. The password is the one you specified when creating the publication.

You should then see something like this:



Once that is done, deploy the publication on your application server as described in section 1.4.2.1. To verify that the deployment was successful, point your browser at **`http://`***server-name*`:`*port*`/`*publication-name*, where *publication-name* is the name of your publication. If you used the WAR archive deployment method, then this will be the same as the name of the WAR archive (**temp-dev**). If you used the EAR archive deployment method, then it will be the name you specified in **`temp-dev.properties`**. You should then see something like this:



2.2 Editing and Administering Publications

Editorial staff use Escenic Content Studio to edit the content of Escenic publications, and Web Studio to manage the publications. As a template developer, you do not need to know these applications very well, but you do need some knowledge of them. You need to be able to add content to and modify the

test publications you use while developing templates, and you also need to understand how changes you make to the structure of a publication affect the editing environment presented by Content Studio.

2.2.1 Content Studio

Content Studio is a Java WebStart application, which means that you do not have to install it on your PC. As long a Java VM is installed on your PC, you can start Content Studio by pointing your browser at **`http://`***server-name*`:`*port*`/studio`** and clicking on the **Launch Escenic Content Studio** link in the displayed page as described in section 2.1.

For detailed instructions on how to use Content Studio for editing and managing publication content see the Escenic Content Studio User Guide.

2.2.2 Web Studio

Web Studio is a browser-based publication administration tool. It allows the publication administrator to modify a publication's structure (add and remove sections, for example), set access rights and carry out a variety of other administrative tasks.

To use Web Studio:

1. Point your browser at **`http://`***server-name*`:`*port*`/escenic`**.
2. Enter the administrator user name (**`temp-dev_admin`**, for example) and password of the publication you want to work on.
3. Click on the **Log in** button.

You should then see something like this:

For detailed instructions on how to use Web Studio for managing publications, see the Escenic Content Engine Publication Administrator Guide.

2.3 Examining The Example Publication

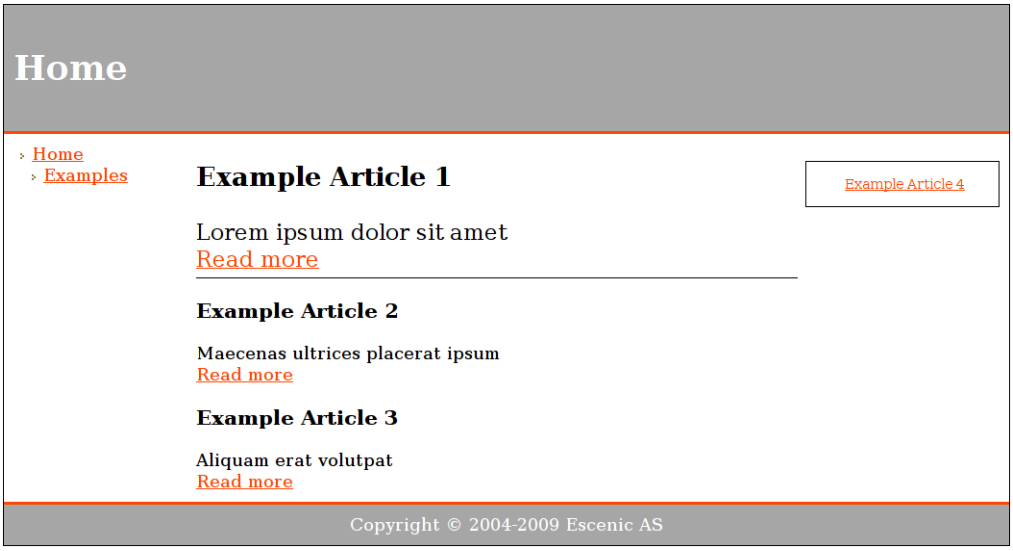
In order to understand Escenic you need to understand the relationship between:

- The published view of the publication

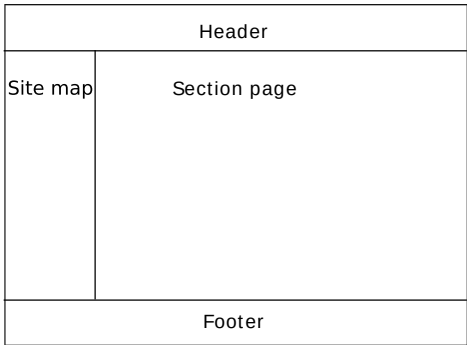
- The editorial view of the publication seen in Content Studio
- The templates and publication resources (i.e publication definition files) that govern these two views

2.3.1 The Published View

The home page of the **temp-dev** example publication you have installed (**`http://`***server-name*`:`*port*`/temp-dev`) looks like this:



Although it is extremely simple, this page has a typical Escenic page layout:



The **site map** on the left shows the structure of the publication, which consists of two **sections**:

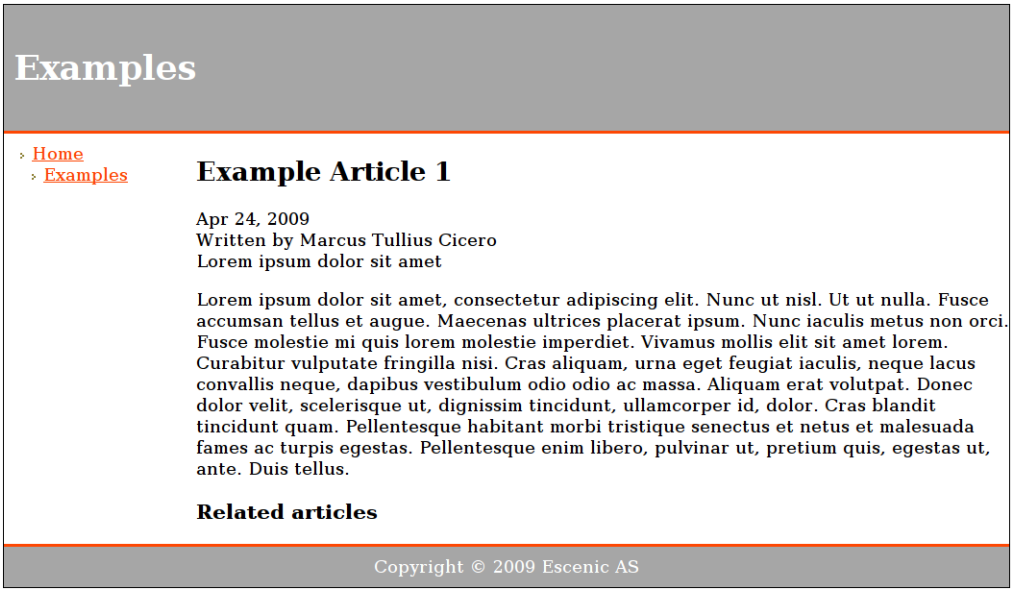
Home
Examples

Home is the **root section**, and **Examples** is a subsection of **Home** (as suggested by the indentation). The **header** contains the name of publication. The **section page** area contains **summaries** of a selection the content items in the section. The contents of this area at any particular time are determined by the section's editor, who choose what content items are to be published there using Escenic Content Studio. The **footer** contains static general information about the publication.

The section page area is often also referred to as the **grid** because it usually has a table/column structure. In this case you can see that the grid is composed of three **areas**: one at the top left above

the line, one below the line and one on the right in a box. The **summaries** are formatted slightly differently in each of these areas.

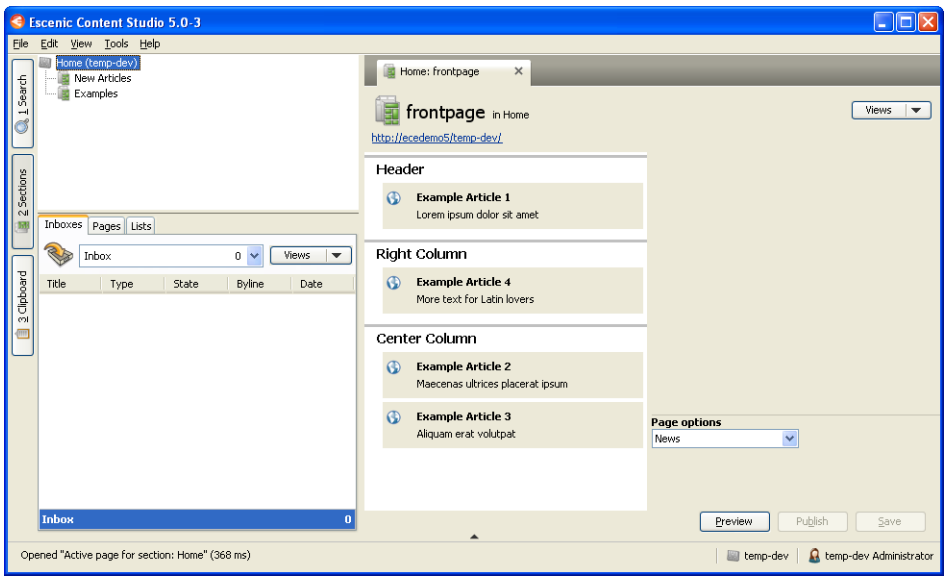
Now click on one of the summary links:



The section page grid is now replaced by the selected article (content item), but the header, footer and sitemap (often collectively called the publication's **wireframe**) remain unchanged.

2.3.2 The Editorial View

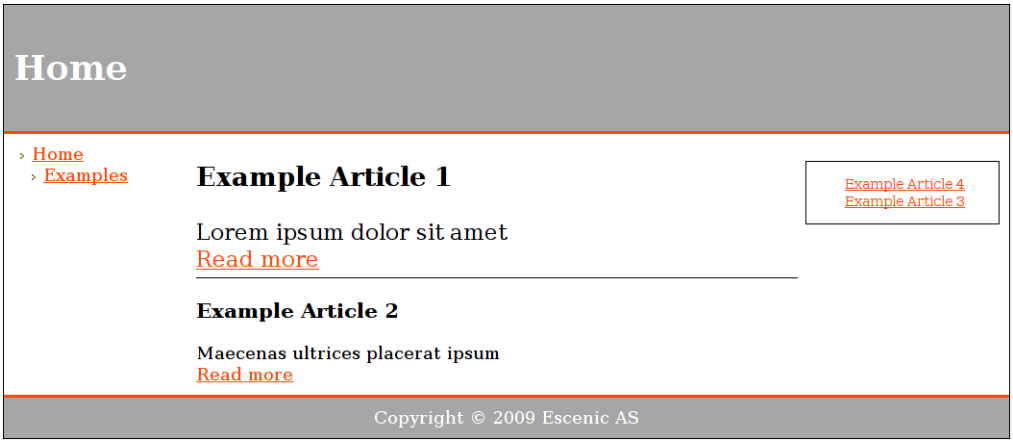
Start Content Studio again and log in to the **temp-dev** publication. Click on the **Section** tab (down the left side of the window), then find **Home** in the section list (at the top left) and double click on it to open the **Home** section in an editor tab:



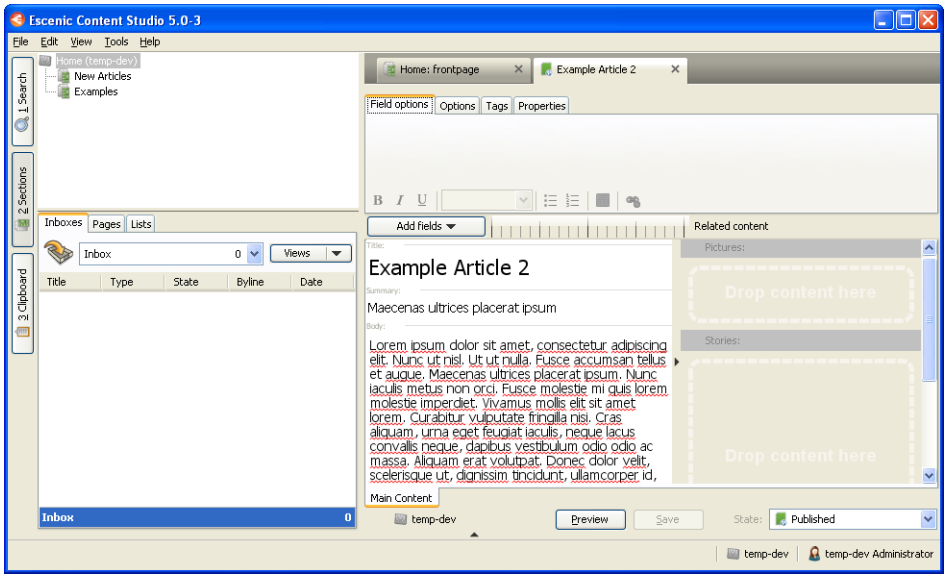
The section contains three content areas called **Header**, **Right Column** and **Center Column**. These areas directly correspond to the three layout blocks in the **Home** section page of the publication. To verify this, try using Content Studio to move a content item:

1. Select **Example 3** and drag it from the **Center Column** element group to **Right Column**.
2. Click on the **Publish** button in the bottom right corner of the editor tab to save and publish your changes.
3. Look at the **Home** page in your browser (refresh/redraw the view if necessary).

Example 3 has now moved to the boxed **Right Column** element group in the published view:



Now go back to Content Studio and double-click on **Example 2** to open the content item itself in an editor tab:



The content item consists of three fields:

- **Title**, which contains the title of the item
- **Summary**, which contains a brief summary of/extract from the content
- **Body**, which contains the body of the item

Try changing the content of the **Title** field, save your changes and look at the results in the browser. The title should now be changed both on the **Examples** front page and in the article itself.

3 The Template System

An Escenic template set is a set of JSP files that:

- Have access to a number of **beans** representing Escenic objects such as articles and sections
- Have access to a number of Escenic **tag libraries** providing functionality that cannot easily be achieved using the available beans, JSP expressions and standard JSTL tags

You can organize these JSP files anyway you like. The only requirement is that there is a single "master template" - one JSP file that is called by the Content Engine in response to every user request. Normally this file must be called **`common.jsp`**, and located in the *publication*`/WEB-INF/template` folder. It is possible to change this default name and location (see The default.properties File), but should not normally be necessary.

In theory you could implement the whole template system in **`common.jsp`**, but normal practice is to break the system down into a number of smaller, simpler files. The **`temp-dev`** application templates described in this section are broken down in this way. The structure of this template set is, however, just an example: precisely how you organize your templates is up to you.

CSS usage

You will notice that the template examples listed in the following sections contain **no** CSS or other HTML formatting information. Many elements, on the other hand, are enclosed in HTML **`div`** elements with **`class`** or **`id`** attributes. The **`common.jsp`** template contains a reference to a CSS file which contains all formatting instructions required to render the pages. The **`class`**/**`id`** attributes in the final HTML output are used by the browser to look up the appropriate styles in the referenced CSS file.

In this way, the templates deal only with content and structure, and all layout issues are dealt with in the CSS file. It is possible to completely change the appearance of a site by simply changing the CSS file that is used.

3.1 The Common Template

All HTTP requests handled by an Escenic publication are either requests for content items or requests for section pages. Each incoming request is processed by Escenic servlet filters (see section 1.2.8), which add the appropriate beans to the request scope and finally pass it to the "master" template (usually called **`common.jsp`**).

`common.jsp` usually has very little content. Here is **`temp-dev`**'s **`common.jsp`**:

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

<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="no" lang="no">
  <head>
    <meta http-equiv="Content-type" content="text/html; charset=iso-8859-1" />
    <c:url var="css" value="/css/main.css" scope="request"/>
    <link rel="stylesheet" type="text/css" href="${css}">
```

```
    <title>
      ${publication.name}
      <c:if test="${requestScope['com.escenic.context']=='art'}">
        - ${article.fields.title}
      </c:if>
    </title>
  </head>
  <body>
    <jsp:include page="wireframe/default.jsp"/>
  </body>
</html>
```

The first line identifies the file as a JSP file, and the second line is a taglib declaration identifying the JSTL **`core`** tag library that is to be used in the file.

The remainder of the file contains a template for generating the outer shell of an XHTML document: the **`html`**, **`head`** and **`body`** elements.

The first things to take note of here are:

- The JSTL **`url`** tag is used to create a variable (**`contextUrl`**) containing a context URL (the current folder, "/") that can be used as a base for constructing further URLs. Since it is created with **request scope**, it can be used in any of the other JSP files called while processing this request.
- The variable is immediately used to create a link to the CSS file that will determine the appearance of all generated output. The JSP expression language is used to reference the variable: **`${contextUrl}`**.

Three other variables are referenced to create the content of the HTML **`title`** element:

`requestScope`
: Is a JSP **implicit object**, a standard variable available in all JSP requests. It is used to hold **request scope attributes**: named values that are available for the duration of a request. In this case we are testing the value of the request scope attribute **`com.escenic.context`**, which is always present in Escenic requests. It is added to the request by the servlet filters, and it contains either "art" if the current request is a content item (article) request or "sec" if the current request is a section request. (For a fuller explanation of the term **request scope attribute**, see section 4.1).

`publication`
: is an Escenic **request bean**. The Escenic request beans are, like **`com.escenic.context`**, request scope attributes that are added to the **`requestScope`** by the servlet filters. The **`publication`** request bean is a Publication bean that is available in all requests. Here, the bean's **`name`** property is retrieved.

`article`
: is also an Escenic request bean (PresentationArticle). However, the **`article`** bean is only available in content item requests: if the current request is a content item request, then the content of the article's **`title`** field is retrieved for inclusion in the HTML **`title`** element.

`common.jsp` then passes responsibility for generating the body of the HTML document to another template, **`wireframe/default.jsp`**.

3.2 The Wireframe Template

wireframe/default.jsp has an overall structure that matches the page layout diagram in section 2.3.1: it consists of four HTML **div** elements representing the main page layout areas: **header**, **footer**, **menu** (or site map) and **content** (i.e, page content - a content item or a section page).

```
<%@ 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="http://struts.apache.org/tags-bean" prefix="bean" %>
<%@ taglib uri="http://www.escenic.com/taglib/escenic-section" prefix="section" %>
<%@ taglib uri="http://www.escenic.com/taglib/escenic-view" prefix="view" %>
<div id="body">
  <div id="header">
    <h1>${section.name}</h1>
   </div>
  <div id="menu">
    <bean:define id="rootSection" name="publication" property="rootSection"
 type="neo.xredsys.api.Section"/>
    <section:recursiveView id="secView" name="rootSection" depth="2"/>
    <c:url var="arrow" value="/gfx/arrow.gif" scope="request"/>
    <view:iterate view="<%=secView%>" id="item" type="neo.xredsys.api.Section">
     <section:use section="<%=item%>">
            <view:forEachLevel> </view:forEachLevel>
             <img src="${arrow}" alt="escenic"/> <section:link
 styleClass="menu_item"><section:name/></section:link><br/>
     </section:use>
    </view:iterate>
  </div>
  <div id="content">
    <c:choose>
      <c:when test="${requestScope['com.escenic.context']=='art'}">
        <jsp:include page="../article/ats.jsp"/>
      </c:when>
      <c:otherwise>
        <jsp:include page="../group/${pool.rootElement.type}.jsp" />
      </c:otherwise>
    </c:choose>
  </div>
  <div id="footer">Copyright © 2009 Escenic AS</div>
</div>
```

The **header div** consists of an **h1** element containing an expression that returns the name of the current section. **section**, like **publication** is an Escenic **request bean** that is added to all requests by the Escenic servlet filters. It is a Section bean representing either the section requested by the user or the owning section of the content item requested by the user.

The **footer div** simply contains a logo image. Note the use of the **contextUrl** variable created in **common.jsp**.

The **menu div** makes use of tags from the Escenic section Tag Library and view Tag Library taglibs to generate a link menu or site map.

The **content div** contains a JSTL **choose** element that tests the content of **com.escenic.context** to find out whether this is a content item request ("art") or a section request ("sec"), and then passes control to the appropriate template.

For content item requests, control is passed to **article/ats.jsp** (article type selector). This template simply calls a new template that depends on the content of the **article** bean's **articleTypeName** property:

```
<jsp:include page="markup/${article.articleTypeName}.jsp"/>
```

In the context of our very simple publication, this means that if the selected content item was defined in Content Studio as being of type **News**, then **article/markup/news.jsp** will be called, and if it was created as being of type **Picture**, then **article/markup/image.jsp** will be called. For a description of these templates, see section 3.3.

For section requests, the template to which control is now passed depends upon the content of another Escenic request bean created by the servlet filters: **pool**.

```
<jsp:include page="../group/${pool.rootElement.type}.jsp" />
```

pool is a PresentationPool bean that represents the active section page. Its **rootElement** property is a PresentationElement bean that represents one of several possible section page layouts defined by **group** elements in the **layout-group** resource file (see group). The name of the particular layout required for this request is held in this **PresentationElement** bean's **type** property. The content of this **type** property is therefore used to select the template to call. The **temp-dev** publication has only one section page layout defined in the layout-group resource, called **news**. For a description of the section page template that is therefore called (**group/news.jsp**), see section 3.4.

3.3 Content Item Templates

Here is the **temp-dev** publication's **article/markup/news.jsp** template:

```
<%@ 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="http://java.sun.com/jsp/jstl/fmt" prefix="fmt" %>
<h2>
  ${article.fields.title}
</h2>
<div class="date">
  <fmt:formatDate value="${article.lastModifiedDateAsDate}"/>
</div>
<div class="byline">
  Written by ${article.author.name}
</div>
<div class="summary">
  ${article.fields.summary}
</div>
<div class="body">
  ${article.fields.body}
  <c:forEach items="${article.relatedElements.images.items}" var="related">
    <img src="${related.content.fields.alternates.value.SmallSquare.href}"
 alt="${related.fields.caption}"
       width="${article.fields.alternates.value.SmallSquare.width}"
 height="${article.fields.alternates.value.SmallSquare.height}"/>
  </c:forEach>
</div>

<div class="stories">
```

```
    <h3>Related articles</h3>
    <ul>
      <c:forEach items="${article.relatedElements.stories.items}" var="related">
        <li>
          <a href="${related.content.url}">${related.fields.title}</a>
        </li>
      </c:forEach>
    </ul>
  </div>
```

This file contains the HTML needed to present the content item, and uses JSP expressions to retrieve the desired fields from the **`article`** bean at the appropriate points in the document. Note in particular:

- The **`article`** bean's **`author`** property is a PresentationPerson bean, which in turn has a **`name`** property that holds the author's name. This property can be directly accessed using the expression language, as shown in the "byline" **`div`** above.
- The "body" **`div`** contains the **`body`** field, followed by a JSTL **`forEach`** element that displays all the content item's related **`images`** (**`images`** is the name of a **`relation-type`** defined in the **`content-type`** resource). The **`article`** bean's **`articles`** property is an array of PresentationRelationArticle elements.
- Similarly, the "stories" **`div`** contains a JSTL **`forEach`** element that is used to output an HTML link to all the content item's related **`stories`** (**`stories`** is the name of a **`relation-type`** defined in the **`content-type`** resource).

3.4 Section Page Templates

Here is the **`temp-dev`** publication's only section page template, **`group/news.jsp`**:

```
<%@ page language="java" pageEncoding="UTF-8" contentType="text/html; charset=UTF-8"
 %>
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
<div class="news">
  <div class="header" style="background-color:
 ${pool.rootElement.areas.header.options['news-area-background']}">
    <c:if test="${fn:length(pool.rootElement.areas.header.items) > 0}">
      <div style="${pool.rootElement.areas.header.items[0].options.border}">
        <c:set var="element" value="${pool.rootElement.areas.header.items[0]}"
 scope="request"/>
        <jsp:include page="ats.jsp"/>
      </div>
    </c:if>
  </div>
  <div class="center">
    <c:forEach items="${pool.rootElement.areas.center.items}" var="item">
      <c:set var="element" value="${item}" scope="request"/>
      <c:choose>
        <c:when test="${item.type eq 'two-col'}">
          <jsp:include page="twoCol.jsp"/>
        </c:when>
        <c:when test="${item.type eq 'three-col'}">
          <jsp:include page="threeCol.jsp"/>
        </c:when>
        <c:otherwise>
```

```
          <jsp:include page="ats.jsp"/>
        </c:otherwise>
      </c:choose>
    </c:forEach>
  </div>
  <div class="right">
    <ul>
      <c:forEach items="${pool.rootElement.areas.rightcolumn.items}" var="item">
        <li><a href="${item.content.url}">${item.fields.TITLE}</a></li>
      </c:forEach>
    </ul>
  </div>
</div>
```

In order to fully understand this template you need to understand the section page structure it is operating on. This structure is defined in the publication's **layout-group** resource (and reflected in the section page editor displayed in Content Studio). For more about the **layout-group** resource, see section 5.2.

The section page structure defined in the **layout-group** resource consists of three main areas: **header**, **center** and **rightcolumn**. **header** is intended as a headline area to highlight the main story, **center** is the main area in which a number of different summaries are displayed, while **rightcolumn** is for lower priority stories, displayed as a simple list of links.

The code for the **header** area:

1. Checks that the area contains an item.
2. If it does, makes an HTML **div** to format the item and then creates a request scope variable called **element** that references the item. (For more detailed information about the **div** element's **style** attribute, see section 5.2.3.)
3. Passes control to **article/ats.jsp** (article type selector). This template calls a new template that depends on the content of the **element** variable's **article.articleTypeName** property:

   ```
   <jsp:include page="../element/${element.article.articleTypeName}.jsp"/>
   ```

The definition of the **center** area in the **layout-group** resource includes references to two **groups** called **two-col** (which is defined as containing two sub areas, **left** and **right**) and **three-col** (which is defined as containing three sub areas, **left center** and **right**). What this means is that the Content Studio user can choose to insert two-column and three-column sub-areas into the center area instead of (or as well as) content items. The code for the center area therefore has to be able to deal with these variations.

The code for the **center** area therefore uses a **forEach** element to cycle through all the area's items, find out whether it is a content item or a group, and take appropriate action. Note that the variable created for each item has request scope and has the same name (**element**) as the variable created in the **header** area code. For content items, control is passed to the same article type selector as in the **header** area code.

The code for the **rightcolumn** area simply cycles through all the area's items and creates a link to each content item.

3.5 Summary Templates

Here is the **temp-dev** publication's **element/news.jsp** template:

```
<%@ page language="java" pageEncoding="UTF-8" contentType="text/html; charset=UTF-8"
 %>
<h3>
  ${element.article.fields.title}
</h3>
<div class="summary">
  ${element.article.fields.summary}
</div>
<a href="${element.article.url}">Read more</a>
```

It is called by the **article/ats.jsp** template and uses the request scope variable **element** created in **group/news.jsp**. Both of these templates are described in section 3.4. It simply creates an HTML snippet that displays the **title** and **summary** fields of the content item referenced by the **element** variable.

4 Accessing The Escenic Beans

The Escenic basic object model and presentation layer object model were introduced in section 1.1.2.1 and section 1.1.2.2. Learning to write Escenic templates is largely a matter of learning to understand these models and learning how to access the objects (or beans) in them from JSP.

This chapter will describe this aspect of template writing in more detail.

4.1 Bean Scope

Beans in JSP applications always have a clearly defined **scope** which determines the context in which they are available. There are four scopes, listed here in order of "size" (largest first):

application
Beans with application scope are always available within an application.

session
Beans with session scope are available for the duration of a particular user's application session. Sessions normally expire after an application-defined period of inactivity. Use of session scope should be avoided in Escenic applications.

request
Beans with request scope are available within all the JSP files involved in responding to a particular HTTP request.

page
Beans with page scope are only available within a single JSP file. When a JSP file contains a call to another JSP file, any page scope beans created in the calling file are **not** available in the called file.

The current application, session, request and page are represented internally in JSP by Java objects called **implicit objects** because they are always available. Any application-created beans are stored as special properties of these objects called **attributes**. The scope of an application bean is determined by which of these implicit objects it belongs to.

4.2 Request Scope Attributes

Every request you will need to handle in your templates is preprocessed by the Escenic filter chain (see section 1.2.8), which determines whether it is an **article request** (a request to view a content item) or a **section request** (a request to view a section page), and creates a corresponding set of **request scope attributes**: that is, attributes of the request scope bean, which is called `requestScope`.

The request scope attributes create by the Escenic filter chain are:

Name/type/full reference	Article request	Section request

`com.escenic.context`
`String`
`requestScope['com.escenic.context']`
	`'art'`	`'sec'`

Name/type/full reference	Article request	Section request
`article` **`PresentationArticle`** **`requestScope['article']`**	Requested content item	Not present
`section` **`Section`** **`requestScope['section']`**	Section containing requested content item[1]	Requested section
`pool` **`PresentationPool`** **`requestScope['pool']`**	Section page of section containing requested content item[1]	Section page of requested section
`publication` **`PresentationPublication`** **`requestScope['publication']`**	Publication containing requested content item	Publication containing requested section

[1] "Section containing requested content item" means one of the sections in which the content item appears (usually, the section from which it has been requested), but not necessarily its home section.

The "full references" in the above table show how these attributes can be unambiguously referenced using the JSP expression language. The **`article`** request scope attribute, for example, can be referenced as follows:

```
${requestScope['article']}
```

and its **`title`** field can be accessed as follows:

```
${requestScope['article'].fields.title}
```

You can, however, user a more compact notation to access them in most cases. The **`article`** request scope attribute can usually be accessed as follows:

```
${article}
```

and its **`title`** field can be accessed as follows:

```
${article.fields.title}
```

This shorter form is the recommended way of accessing the Escenic request scope attributes (referred to in the rest of this manual as the Escenic **`request beans`**). Note, however, that you must then make sure you do not create any page scope attributes with the same name as one of the request beans. If you create a page scope attribute with the name **`article`**, then

```
${article}
```

will return the page scope attribute, and you will have to use the full reference notation to access the request bean.

You should not use the short notation described here to access the **`escenic.com.context`** request scope attribute. (You are not allowed to use the "." character in JSP expression language names.) Always use the full reference shown in the table above for this attribute.

4.3 Accessing Bean Properties

Beans are objects: structured variables that instead of holding a single value, hold a set of related values called properties. A PresentationArticle bean, for example, has more than 40 properties describing various aspects of a content item. Its **articleId** property contains a unique ID for the content item, its **createdDateAsDate** property contains the date on which it was created, its **author** property is another bean containing information about the author of the content item.

You can access these properties with the JSP expression language by using **dot notation**. You simply add the name of the property you are interested in to the end of the bean name, separated by a dot. To access the **article** request bean's **articleId** property, for example, you can use:

```
${article.articleId}
```

A bean property may be a reference to another bean rather than a simple string or number. **PresentationArticle**'s **author** property, for example, is a reference to a **Person** bean. In this case, you can use the same notation to access that bean's properties. You can, for example access the name of the author of a content item as follows:

```
${article.author.name}
```

4.3.1 Indexed Properties

Some beans have properties that are Java arrays or lists. You can therefore add an index subscript to a property name in order to specify which element in the list you want to access. The following expression, for example, returns the name of the first author of an article:

```
${article.authors[0].name}
```

(Note that this returns exactly the same value as the second example in section 4.3. The **authors** property returns an array containing **all** of a content item's authors; the **author** property returns the first member of the array, just like this example.)

4.3.2 Mapped Properties

Some beans have properties that are Java **maps**. A map is like an array except that each member of the array is identified by a **key**. You can therefore retrieve a specific member of such a property by specifying the key that identifies it. You can, for example, access a section page area called "header" area as follows:

```
${pool.rootElement.areas['header']}
```

In this example **pool** is a PresentationPool bean representing the section page requested by a user. Its **rootElement** property is a PresentationElement bean representing the section page's **root group** or **grid**. The grid's **areas** property is a mapped property containing all the areas in the grid, each of which can be accessed by specifying its key (in this case, **header**).

The JSP expression language allows you replace the key specification syntax shown above with simple dot notation. You can therefore also access the "header" area as follows:

```
${pool.rootElement.areas.header}
```

All of the examples in this manual use this simpler method of addressing mapped properties.

5 The Publication Resources

The **publication resources** are files that define the underlying structure of a publication. In many media organizations, the job of designing this structure will be carried out by a publication designer or information architect rather than by a template developer, so you may never need to modify these resources yourself. Nevertheless you need to know about them and understand how they determine both the editorial interface displayed in Content Studio and the data structures available to you in your templates.

There are four publication resources:

`content-type`
An XML file that defines the types of content items present in a publication.

`layout-group`
An XML file that defines the logical structure of the section pages in a publication.

`image-version`
An XML file that defines the different versions of images that are to be used in a publication (for example thumbnails, small, large and so on).

`feature`
A plain text file containing property settings that govern the behavior of the Escenic Content Engine.

The publication resources are stored in the following location in a publication WAR file or folder tree:

`META-INF/escenic/publication-resources/escenic`

The publication resources are fully described in the Escenic Content Engine Resource Reference.

5.1 content-type

The **`content-type`** resource determines (among other things):

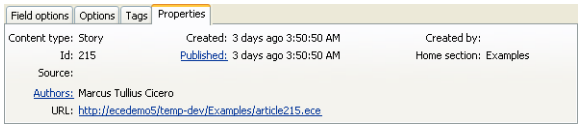
- What kinds of content-items a publication can contain
- What fields each kind of content-item contains
- The type of each field
- Constraints on what you can enter in fields (maximum length, maximum and minimum values and so on)
- How the fields will be displayed in Content Studio
- What relations each kind of article can have

The following sections provide an introduction to the **`content-type`** resource, and some of the things it is used for. For a full, formal description of the **`content-type`** resource format and all the things you can do with it, see here.

5.1.1 Defining Content Types

The **temp-dev** content items discussed earlier have three fields: **title**, **summary** and **body**. These fields are defined in the content item's **content-type** definition.

Look at one of the **temp-dev** content items in Content Studio. Click on the **Properties** tab in the article editor to display general information about the article. You will see that its **Content type** is set to **Story**:



If you select **File** > **New** from the menu bar you will see that the displayed submenu contains a **Story** option. If you select this option, a new content item is created and opened in an editor tab. The editor tab contains the same three fields: **Title**, **Summary** and **Body**.

Now go to your application server's web applications folder, then find and open **temp-dev/META-INF/escenic/publication-resources/escenic/content-type** in a text editor. Search for the string "Story" and you should find the following definition:

```
<content-type name="news">
  <ui:label>Story</ui:label>
  <ui:description>A news story</ui:description>
  <ui:title-field>title</ui:title-field>
  <panel name="default">
    <ui:label>Main Content</ui:label>
    <ui:description>The main content fields</ui:description>
    <ref-field-group name="title"/>
    <ref-field-group name="summary"/>
    <ref-field-group name="body"/>
  </panel>
  <ref-relation-type-group name="attachments"/>
   <summary>
    <ui:label>Content Summary</ui:label>
    <field name="title" type="basic" mime-type="text/plain"/>
    <field name="summary" type="basic" mime-type="text/plain"/>
  </summary>
</content-type>
```

This code defines a content type called "news" with the following characteristics:

- The **label** "Story". This is the name that is used wherever it appears in Content Studio.
- A **Panel** with the **name** "default" and the **label** "Main Content". If you look in Content Studio you will see that the article fields are all displayed on a tab card called **Main Content**.
- References to three field groups called **title**, **summary** and **body**.
- Other characteristics we will look at later.

Immediately above the "news" element, you should find the definitions of the referenced field groups:

```
<field-group name="title">
  <field mime-type="text/plain" type="basic" name="title">
    <ui:label>Title</ui:label>
    <ui:description>The title of the article</ui:description>
    <constraints>
```

```
        <required>true</required>
      </constraints>
    </field>
  </field-group>

  <field-group name="summary">
    <field mime-type="text/plain" type="basic" name="summary">
      <ui:label>Summary</ui:label>
      <ui:description>The summary text of the article.</ui:description>
    </field>
  </field-group>

  <field-group name="body">
    <field mime-type="application/xhtml+xml" type="basic" name="body">
      <ui:label>Body</ui:label>
      <ui:description>The body text of the article.</ui:description>
    </field>
  </field-group>
```

As you can see, each field group contains the definition of a single field with the same name as the group, and they define the fields displayed on the **Main Content** tab in Content Studio.

The following example shows the definition of a new content type called "review", intended to be used for review articles. Try adding it to the **`content-type`** resource after the existing "news" content type.

```
  <content-type name="review">
    <ui:label>Review</ui:label>
    <ui:description>A product review</ui:description>
    <ui:title-field>title</ui:title-field>
    <panel name="default">
      <ui:label>Main Content</ui:label>
      <ui:description>The main content fields</ui:description>
      <ref-field-group name="title"/>
      <ref-field-group name="summary"/>
      <ref-field-group name="body"/>
      <ref-field-group name="review"/>
    </panel>
    <ref-relation-type-group name="attachments"/>
     <summary>
      <ui:label>Content Summary</ui:label>
      <field name="title" type="basic" mime-type="text/plain"/>
      <field name="summary" type="basic" mime-type="text/plain"/>
    </summary>
  </content-type>
```

It is, as you will see, very similar to the original "news" content type definition. Apart from the name and label, the only significant difference is the addition of a fourth field group called "review".

In order for this new content type to work, therefore, we must also add a definition for the new field group it refers to:

```
  <field-group name="review">
    <field type="enumeration" name="review-type">
      <ui:label>Review Type</ui:label>
      <ui:description>Select the required type</ui:description>
      <enumeration value="film"/>
      <enumeration value="play"/>
      <enumeration value="book"/>
      <enumeration value="game"/>
```

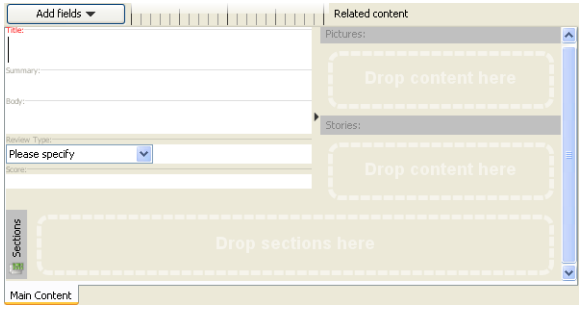
```
    </field>

    <field type="number" name="score">
      <ui:label>Score</ui:label>
      <ui:description>Enter your rating</ui:description>
      <constraints>
        <minimum>1</minimum>
        <maximum>6</maximum>
      </constraints>
    </field>
  </field-group>
```

This field group, unlike the other field groups in the **`content-type`** file, contains more than one field. Field groups, however, have no effect on the layout of the fields in Content Studio or on how fields are accessed from templates. You can use them to group fields in whatever way is most convenient.

The fields in this group are of a different type from the fields we have seen so far:

- The **`review-type`** field is of type **`enumeration`**, and contains a list of alternative values.
- The **`score`** field is of type **`number`** and contains a **`constraints`** element that specifies the allowed range for the field.

After making these changes to **`content-type`**, upload the new resource as described in section 1.4.2.2. Then restart Content Studio. Select **File** > **New** from the men bar: you will see that the displayed submenu now contains a **Review** option, which you can select to create a **`review`** content item. The editor displayed contains the new fields you have defined:



- **Review Type** is displayed as a combo box containing the options you specified.
- **Score** will only let you enter a value in the range 1-6. If you enter a value outside this range, then the field title turns red to indicate that the value is invalid.

There are many other field types you can select from. For more information about field types, see section 5.1.8. For a complete list of available field types, see here.

5.1.2 Defining Editor Panels

It is sometimes useful to be able to group a content type's fields and display the groups on separate tabs or **panels** in Content Studio. You can, for example, move the "review" group of fields you created to a separate **Review Content** panel as follows:

```
  <content-type name="review">
    <ui:label>Review</ui:label>
    <ui:description>A product review</ui:description>
    <ui:title-field>title</ui:title-field>
```

```
    <panel name="default">
      <ui:label>Main Content</ui:label>
      <ui:description>The main content fields</ui:description>
      <ref-field-group name="title"/>
      <ref-field-group name="summary"/>
      <ref-field-group name="body"/>
    </panel>
    <panel name="review-panel">
      <ui:label>Review Content</ui:label>
      <ui:description>Additional review fields</ui:description>
      <ref-field-group name="review-fields"/>
    </panel>
    <ref-relation-type-group name="attachments"/>
     <summary>
      <ui:label>Content Summary</ui:label>
      <field name="title" type="basic" mime-type="text/plain"/>
      <field name="summary" type="basic" mime-type="text/plain"/>
    </summary>
  </content-type>
```

This will have the following effect in Content Studio:

5.1.3 Defining Summaries

The article links displayed on section pages are called **`summaries`**. A summary usually consists of a subset of the fields in a content item, which are displayed along with a link to the full content item. The fields that are to be used in a summary are defined using the **`summary`** element. In the following example:

```
  <content-type name="review">
    <ui:label>Review</ui:label>
    <ui:description>A product review</ui:description>
    <ui:title-field>title</ui:title-field>
    <panel name="default">
      <ui:label>Main Content</ui:label>
      <ui:description>The main content fields</ui:description>
      <ref-field-group name="title"/>
      <ref-field-group name="summary"/>
      <ref-field-group name="body"/>
      <ref-field-group name="review"/>
    </panel>
    <ref-relation-type-group name="attachments"/>
     <summary>
      <ui:label>Content Summary</ui:label>
      <field name="title" type="basic" mime-type="text/plain"/>
      <field name="summary" type="basic" mime-type="text/plain"/>
    </summary>
```

```
    </content-type>
```

the summary contains **`title`** and **`summary`** fields.

Note that the **`summary`** element has no effect on how summaries are created in your JSP templates: the summary template example listed in section 3.5 would work with the above content type even if it did not contain a **`summary`** element. The **`summary`** element is used by Content Studio to determine the content of the summaries displayed in section page editors.

The **`summary`** element is also used by the Content Engine, which supports **local overwriting** of summary fields. What this means is that when a section editor adds a content item to a section page, she can also make local changes to the content of the **`summary`** fields: a content item title that reads "England Loses Again" can be overwritten with "Humiliated!" on the section page without affecting the main title. If the same content item appears in a second section, it can have yet another local title there.

This functionality is "built-in" to the Content Engine: you do not need to do anything in your templates to enable it.

Note that:

- You cannot use **`ref-field-group`** inside **`summary`** elements: you must directly specify the fields to be included in the summary.
- You cannot use rich text fields (that is, **`basic`** fields with the MIME type **`application/xhtml+xml`**) in summaries.

5.1.4 Content Item Relations

Content items can have relations to other content items. What specific kinds of relations a content item is allowed to have depends upon content type, and is defined in the **`content-type`** resource: you may already have noticed the

```
<ref-relation-type-group name="attachments"/>
```

element in the content type definitions we have looked at. For a detailed discussion of content item relations, including their definition in the **`content-type`** resource, see chapter 6.

5.1.5 Dealing With Media Content

The **`news`** and **`review`** content types we have looked at so far only have textual content. How do you deal with media content such as images, video clips and sound files? There is a **`link`** field type that can be used to contain references to media objects, so you can use this to create content types specifically designed for media objects. A **`content-type`** element is only allowed to contain one **`field`** of type **`link`**, so you need to define a number of different **`content-type`**s for handling different types of media objects.

The **`temp-dev content-type`** resource contains the following content type definition for image objects:

```
    <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>
    <panel name="crop">
      <ui:label>Crop</ui:label>
      <field name="alternates" type="basic" mime-type="application/json">
        <representations xmlns="http://xmlns.escenic.com/2009/representations"
type="image-versions">
          <representation name="WideBig">
            <output width="572" height="204"/>
            <crop/>
            <resize/>
          </representation>
          <representation name="SmallSquare">
            <output width="150" height="150"/>
            <crop/>
            <resize/>
          </representation>
        </representations>
      </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>
```

This content type's "default" panel contains a **`link`** field for the URL of an image file plus three other fields for storing information about the image: **`name`**, **`description`** and **`alttext`**.

The "crop" panel contains a field that can be used to set up alternative image versions called representations.

In a typical "newspaper-like" Escenic publication, media objects such as images are usually "secondary" content items that appear embedded in text-based content items. This "embedding" is achieved by means of **relations**. For more information about this, see chapter 6.

5.1.6 Hidden Content Types

Any content type can be hidden by adding a **ui:hidden** element to it; it will then not appear in Content Studio's **File** > **New** menu.

The following **content-type** element defines a hidden content type with the name **news-old**:

```
<content-type name="news-old">
  <ui:hidden/>
  ...
</content-type>
```

Hidden content types are accessed from application code in exactly the same way as ordinary visible content types.

5.1.7 Controlling Content Item URLs

By default, the URL assigned to a published content item is generated from the content item's id, prefixed by the string **article** and followed by the suffix **.ece** - for example:

article1234.ece

You can, however, replace these with more informative (or "pretty") URLs by adding **url** elements to your publication's **content-type** elements. This element allows you to specify a template from which more meaningful or "pretty" URLs are generated. The template is composed from any text you want, plus a set of standard place-holders representing parts of a content item's publication date (**{dd}**, **{MM}**, **{YY}**), the content of selected content item fields (typically the title field) and so on.

The **url** element must be specified as the child of a **content-type** element, and determines the URLs assigned to content items of thahrt type. It affects:

- Newly-created content items belonging to its parent **content-type**.
- Any existing content items belonging to its parent **content-type** that are updated after the addition of the **url** element. The original URL of an updated content item is retained as an alternative URL, and attempts to access the original URL are redirected to the new URL (the Content Engine returns an HTTP **301 Moved Permanently** response).

For detailed information and examples, see here.

5.1.8 More About Defining Fields

The **content-type** element you will use most frequently is almost certainly the **field** element. It is also one of the more complicated elements in the content-type resource, so it is probably worth looking at it more closely.

The **field** element has a **type** attribute that determines what type of data the field will accept. The main **type** values are:

basic
: Accepts **any** string data, including XHTML fragments. This is the default field type.

number
: Accepts only numbers.

`boolean`
Accepts only Boolean (true/false) values.

`enumeration`
Accepts only values from a predefined list.

`uri`
Accepts only URIs.

`date`
Accepts only date/time values.

`schedule`
Accepts only schedule definitions.

`collection`
Accepts only values from an atom feed.

The type of a field determines not only what kind of data it is capable of accepting, but also:

- What kind of constraints can be placed on the input data
- What kind of child elements the **`field`** element may contain
- What kind of data the field returns when accessed from a template, and therefore how the field is best accessed.

These types are described in more detail in the following sections. Also discussed are **field arrays**, **complex fields** and **hidden fields**.

> Changing field type for any given field that is currently in use, is not supported. The behaviour if done is undefined.

For a complete list of all available field types, see here.

5.1.8.1 Basic Fields

The following **`field`** element defines a **`basic`** field with the name **`title`**:

```
<field mime-type="text/plain" type="basic" name="title">
  <constraints>
    <required>true</required>
  </constraints>
</field>
```

Note the following points:

- The **`name`** attribute may not contain spaces and must start with a letter (not a number).
- The **`mime-type`** attribute specifies more precisely the type of data allowed in the field. Currently, the following **`mime-type`** values are supported:

  `text/plain` **(default)**
  Any text. A simple text editing field is displayed in Content Studio.

  `application/xhmtl+xml`
  XHTML. An XHTML editing field is displayed in Content Studio. (Whenever this field is selected, then XHTML formatting controls are displayed in the editor's **Field Options** tab.)
- The optional **`constraints`** child element specifies that a value is required.

- The optional **representations** child element causes a basic **field** element to be rendered in Content Studio as a set of image representations rather than as a text input field. It only has any meaning to assign a **representations** element to a basic **field** element in a content-type that also has a **link** field that references an image. For further information about this see section 5.3.

Accessing a Basic Field

To retrieve the content of a **basic** field in your templates:

```
${article.fields.title}
```

5.1.8.2 Number Fields

The following **field** element defines a **number** field with the name **age**:

```
<field type="number" name="age">
  <constraints>
    <minimum>18</minimum>
    <maximum>99</maximum>
  </constraints>
  <format>##</format>
</field>
```

Note the following points:

- The optional **constraints** child element specifies the allowed value range for the field.
- The optional **format** child element controls formatting in the input field in Content Studio. **format** syntax is based on the Java **DecimalFormat** class. See https://docs.oracle.com/javase/8/docs/api/java/text/DecimalFormat.html for details.

Accessing a Number Field

To retrieve the content of a **number** field in your templates:

```
${article.fields.age}
```

5.1.8.3 Boolean Fields

The following **field** element defines a **boolean** field with the name **debug**:

```
<field type="boolean" name="debug"/>
```

Note the following points:

- A **debug** field is displayed as a checkbox in Content Studio.
- It can contain only two values, **true** (checked) or **false** (not checked).

Accessing a Boolean Field

A **boolean** field returns a true boolean value, not a string. You can therefore simply test it directly in your templates (using the JSTL **if** element, for example):

```
<c:if test="${article.fields.debug}">
  ...
</c:if>
```

5.1.8.4 Enumeration Fields

The following **field** element defines an **enumeration** field with the name **review-type**:

```
<field type="enumeration" name="review-type">
  <enumeration value="film"/>
  <enumeration value="play"/>
  <enumeration value="book"/>
  <enumeration value="game"/>
</field>
```

Note the following points:

- in Content Studio an **enumeration** field is displayed either as a drop-down list (combo box) from which the user can make a single selection or as multi-select list from which the user can make multiple selections.
- The example above displays a **drop-down** list. To display a multi-select list you must add a **multiple="true"** attribute to the field element.

Accessing an Enumeration Field

To retrieve the content of a single-select **enumeration** field in your templates:

```
${article.fields.review-type}
```

If **multiple** is set to **true**, the selections made by the user are stored in an array, so you can cycle through the contents using a JSTL **forEach** element, for example:

```
<ul>
  <c:forEach var="cinema" items="${article.fields.showing-in}">
    <li>${cinema}</li>
  </c:forEach>
</ul>
```

5.1.8.5 URI Fields

The following **field** element defines a **uri** field with the name **homepage**:

```
<field type="uri" name="homepage"/>
```

A **uri** field will only accept a valid URI (Uniform Resource Identifier) as input in Content Studio. URI syntax is defined in the IETF's RFC 2396 (http://www.ietf.org/rfc/rfc2396.txt) and RFC 2732 (http://www.ietf.org/rfc/rfc2732.txt).

Accessing a URI Field

To retrieve the content of a **basic** field in your templates:

```
${article.fields.homepage}
```

5.1.8.6 Link Fields

Link fields are mainly used to contain references to binary objects such as images, video and audio files and so on. The following element defines a **link** field with the name **binary**.

```
<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>
```

Note the following points:

- The child **relation** element defines the relationship between the field and the objects it used to reference. In this case, the value **com.escenic.edit-media** (currently the only supported **relation** value) indicates that the field is to be used to reference binary media objects.
- The child constraints element lists the types of media objects the field is allowed to reference (in this case, JPEG and PNG image files)

Accessing a Link Field

To retrieve the content of a **link** field in your templates:

```
${article.fields.binary.value.href}
```

The above example will always return a value (the URL of the referenced object). In the specific case of **link** fields that reference images, however, there may be additional options. It is normally the case that images are available in multiple versions, which are specified in the **image-version** resource (see section 5.3). In the case of images, therefore, the expression shown above will return the URL of the **default version** of the referenced image. If you want to retrieve a particular version, then you can replace **href** with the name of the version you want. To retrieve an image version called **thumbnail**, for example, you would need to enter:

```
${article.fields.binary.value.thumbnail}
```

If, on the other hand, the image versions have been defined using representation elements in the **content-type** resource, then you would access them via the field containing the representations: for further information about this see the Escenic Content Engine Advanced Developer Guide.

To get the MIME type of the object referenced in a link field, enter:

```
${article.fields.binary.value.mime-type}
```

5.1.8.7 Date Fields

The following **field** element defines a **date** field with the name **startdate**:

```
<field type="date" name="startdate"/>
```

A **date** field is displayed in Content Studio as two specialized fields, one for the date and one for time of day. The content of a date field is stored as a UTC time in ISO-8601 format (that is, *YYYY-MM-DD*`'T'`*HH:mm:ss*`'Z'`) and is indexed as a date.

Accessing a `date` Field

To retrieve the content of a **date** field in your templates:

```
<fmt:formatDate value="${article.fields.startdate.value}" type="both"/>
```

5.1.8.8 Schedule Fields

A schedule field is a specialized date/time field that contains a schedule start and end date, an event start and end time and an optional set of recurrence rules. Together, they define a sequence of date/time values. Schedule fields are typically used in articles describing events such as concerts, meetings etc.

A schedule field is defined as follows:

```
<field name="when" type="schedule">
  <ui:label>Schedule</ui:label>
</field>
```

A schedule is defined by:

- A schedule start date
- Either a schedule end date or a specified number of occurrences
- An event start and end time
- A recurrence specification (daily, weekly on Fridays, etc.)

When a content item containing a schedule field is stored, all of the event occurrences defined by the schedule are indexed, and can be searched by their start date. You can search for a list of content items contain scheduled events that start within a certain start date range.

You can limit the maximum number of occurrences users are allowed to specify by setting the **`maxOccurrences`** property in **`/com/escenic/schedule/OccurrenceHelper.properties`** file in one of you installation's **configuration layers**. If you do not specify this, then a default occurrence limit of **`100`** is used. For information about configuration layers, see Configuring The Content Engine.

Accessing a Schedule Field

To retrieve the recurrence state of a **`schedule`** field in your templates:

```
${article.fields.when.value.recurring}
```

This returns **`true`** if recurrence is specified in the schedule field, or **`false`** if recurrence is not specified.

To retrieve the first or only event occurrence defined by a **`schedule`** field:

```
${article.fields.when.value.instance}
```

To retrieve all the event occurrences defined by a **`schedule`** field:

```
${article.fields.when.value.instances}
```

This will return a **`java.util.SortedSet`** object containing one **`neo.xredsys.presentation.PresentationScheduleInstance`** object for each event occurrence. A **`neo.xredsys.presentation.PresentationScheduleInstance`** object contains the start and end date/time of an event.

To display the start and end date/time of the first or only event occurrence defined by a **`schedule`** field:

```
From: <fmt:formatDate value="${article.fields.when.value.instance.startDateTime}"
 type="both"/>
To: <fmt:formatDate value="${article.fields.when.value.instance.endDateTime}"
 type="both"/>
```

To display the start and end date/time of all the event occurrences defined by a **schedule** field:

```
<ul>
  <c:forEach items="${article.fields.when.value.instances}" var="instance">
    <li>
      From: <fmt:formatDate value="${instance.startDateTime}" type="both"/>
      To: <fmt:formatDate value="${instance.endDateTime}" type="both"/>
    </li>
  </c:forEach>
</ul>
```

5.1.8.9 Collection Fields

A collection field is special field that can contain value from an atom feed.

A collection field is defined as follows:

```
<field name="collection" type="collection" mime-type="text/plain" src="http://j.mp/
cwaXJM" select="title">
  <ui:label>Collection</ui:label>
</field>
```

Accessing a Collection Field

To retrieve the origin of a **collection** field in your templates:

```
${article.fields.collection.value.origin}
```

To retrieve the value of a **collection** field:

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

5.1.8.10 Field Arrays

The following **field** element defines an array of **basic** fields with the name **cities**:

```
<field type="basic" name="cities">
  <array default="3" max="10"/>
</field>
```

Note the following points:

- You can make arrays of any field type.
- The **array** element's **default** attribute specifies the number of fields that are displayed by default, and **max** specifies the maximum number of fields that can be displayed.

Accessing a Field Array

The values input by the user are stored in an array, so you can cycle through the contents using a JSTL **forEach** element, for example:

```
<ul>
```

```
    <c:forEach var="city" items="${article.fields.cities}">
      <li>${city}</li>
    </c:forEach>
  </ul>
```

5.1.8.11 Complex Fields

A **complex field** is an array of related **field**s that are displayed as a group in Content Studio. The component **field**s can be of any type except **complex**.

The following code defines a complex **field** with the name **details**:

```
<field type="complex" name="details">
  <complex>
    <field type="basic" name="availability"/>
    <field type="basic" name="colors"/>
    <field type="number" name="price"/>
  </complex>
</field>
```

You can create arrays of complex fields.

Accessing a Complex Field

The values input into the component fields are stored in a map, using the field names as keys. This means you can access them as follows:

```
<p>Available from: ${article.fields.details.value.availability}</p>
<p>Colors: ${article.fields.details.value.colors}</p>
<p>Price: ${article.fields.details.value.price}</p>
```

5.1.8.12 Hidden Fields

Any **field** can be hidden by adding a **ui:hidden** element to it.

The following **field** element defines a hidden field with the name **result**:

```
<field type="basic" name="result">
  <ui:hidden/>
</field>
```

Note the following points:

- A hidden field is not displayed in Content Studio.
- Hidden fields are intended to be filled by application code.

Accessing a Hidden Field

Hidden fields are accessed in exactly the same way as ordinary visible fields.

5.1.9 Changing Content Types That Are in Use

Most changes you make to content type definitions will be made during the publication design phase before it is in active use. Occasionally, however, you may need to update content types that are in active use in a live system. For publications that are actively updated 24/7 there may be little or no opportunity to stop all Content Studio editing activity while the change is made.

The precise consequences of changing a content type that is in use depends on which of the following three classes the change belongs to:

- Compatible change
- Incompatible change
- Unsupported change

5.1.9.1 Compatible Changes

A **compatible change** to a content type causes no problems or side effects in Content Studio. The following changes are compatible changes:

- Changes to a field's `ui:label`.
- Changes to a field's `ui:description`.

Compatible changes do not necessarily become visible in Content Studio immediately after the changed `content-type` resource is uploaded to the Content Engine: they become visible the next time Content Studio refreshes the affected content type for some reason or a content item of the affected type is opened for editing.

5.1.9.2 Incompatible Change

All the following types of change are classed as **incompatible changes:**

- Adding new optional field
- Making a mandatory field optional
- Adding a new option to an enumeration field
- Removing any constraint
- Adding a new field to a complex field

If a content item is open for editing in Content Studio when an incompatible change is made to its content type, then when the user saves his changes, a message is displayed stating that the content type has changed in an incompatible way. When the user clicks **OK**, the change is saved in the usual way but the editor tab is then immediately closed and reopened, incorporating the content type change. The user can then continue editing with the content type change in effect.

5.1.9.3 Unsupported Change

All other types of change to content types are **unsupported changes**. This means Content Studio cannot save a content item that is affected by such a change.

If you need to make an unsupported change to a content type, you should inform all Content Studio users so that they can avoid editing content items of the affected type while the updated `content-type` resource is uploaded.

5.2 layout-group

The `layout-group` publication resource defines the logical structure of the layouts available for use on a publication's section pages.

The following sections provide an introduction to the **`layout-group`** resource, and some of the things it is used for. For a full, formal description of the **`layout-group`** resource format and all the things you can do with it, see here.

5.2.1 Defining Section Page Layouts

Go to your application server's web applications folder, then find and open **`temp-dev/META-INF/escenic/publication-resources/escenic/layout-group`** in a text editor.

You should see the following code:

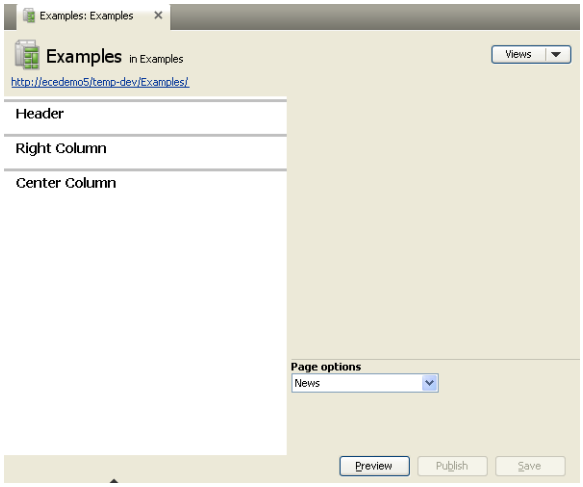
```
<groups xmlns="http://xmlns.escenic.com/2008/layout-group"
        xmlns:ct="http://xmlns.escenic.com/2008/content-type"
        xmlns:ui="http://xmlns.escenic.com/2008/interface-hints">
  <group name="news" root="true">
    <ui:label>News</ui:label>
    <area name="header">
      <ui:label>Header</ui:label>
      <ui:description>Content placed here will appear at top of page</ui:description>
      <ct:options scope="current">
        <ct:field type="enumeration" name="news-area-background">
          <ui:label>Area background</ui:label>
          <ui:description>Changes the area background</ui:description>
          <ct:enumeration value="white">
            <ui:label>white</ui:label>
          </ct:enumeration>
          <ct:enumeration value="blue">
            <ui:label>blue</ui:label>
          </ct:enumeration>
          <ct:enumeration value="red">
            <ui:label>red</ui:label>
          </ct:enumeration>
        </ct:field>
      </ct:options>
      <ct:options>
        <ct:field type="enumeration" name="border">
          <ui:label>Style</ui:label>
          <ui:description>Sets the style of the header</ui:description>
          <ct:enumeration value="border: 1px solid black;">
            <ui:label>Border</ui:label>
          </ct:enumeration>
          <ct:enumeration value="border: 5px solid black;">
            <ui:label>Fat Border</ui:label>
          </ct:enumeration>
          <ct:enumeration value="background: #F55;">
            <ui:label>Red Background</ui:label>
          </ct:enumeration>
        </ct:field>
      </ct:options>
    </area>
    <area name="rightcolumn">
      <ui:label>Right Column</ui:label>
      <ui:description>Content placed here will appear in the right column</
ui:description>
    </area>
    <area name="center">
      <ui:label>Center Column</ui:label>
```

```
      <ui:description>Content placed here will appear in the Center column</
ui:description>
      <ref-group name="two-col"/>
      <ref-group name="three-col"/>
    </area>
  </group>
  <group name="two-col">
    <area name="left"/>
    <area name="right"/>
  </group>
  <group name="three-col">
    <area name="left"/>
    <area name="center"/>
    <area name="right"/>
  </group>
</groups>
```

This specifies that:

- There is one layout **group** available for structuring section pages, called **news**.
- It contains three **areas** called **header**, **rightcolumn** and **center**.
- The **header** area has two sets of options that can be set in Content Studio. The "Area background" option applies to the whole area, while the "Style" options applies to the individual content items placed in the area. For more information about this, see options.
- The **center** area in section pages that use the **news** layout **can be** subdivided by inserting **two-col** and/or **three-col** groups using Content Studio.

If you start Content Studio, log in to the **temp-dev** publication and open the **New Articles** section's active section page, then you will see this structure reflected in the displayed editor:



The section page contains three areas called **Header**, **Right Column** and **Center Column**. If you right-click on **Center Column**, then the displayed menu's **Insert** option will show a submenu with the options **Insert new Two-col group** and **Insert new Three-col group**. At the top of the section page editor is a combo box field with the name **Section page descriptor** containing the name of this layout group (**News**). If you click on the combo box, you will see that **News** is the only possible value for the field - no other layout group is available.

To make an alternative layout, copy the news group in the layout group resource, rename it and simplify it by deleting a few elements as follows:

```
  <group name="simple" root="true">
    <ui:label>Simple</ui:label>
    <area name="header"/>
    <area name="center">
      <ui:label>Center Column</ui:label>
      <ui:description>Content placed here will appear in the Center column</
ui:description>
      <ref-group name="two-col"/>
    </area>
  </group>
```

Upload the modified **`layout-group`** resource as described in section 1.4.2.2. Then restart Content Studio and open the **New Articles** section's section page again. Select the **Section page descriptor** combo box again. This time it should contain a second option, **Simple**. If you select **Simple** then you will see that the section page structure displayed in the section page editor is simplified accordingly:

It is the **`group`** element's **`root`** attribute that determines whether or not a group can be used as a section page layout. The groups **`two-col`** and **`three-col`** do not appear in the **Section page descriptor** combo box's list because their **`root`** attributes are not set to **`true`**.

5.2.2 Rendering Section Page Layouts

Note that what is defined in the **`layout-groups`** resource is the **logical structure** of the section page layouts, not their graphical appearance. It specifies the containment rules for groups and areas, but it does not say anything about where they are located on the page (although the names in the example hint at locations), nor does it say anything about the appearance of the groups and areas. These things are all defined in your JSP templates.

In order to write a template that renders section pages correctly, you have to know the section page layout's logical structure. For a template that renders the section page layout discussed in this section, see section 3.4. In a more complex publication with multiple section page layouts (root groups) it would be necessary to first test the section page's root group to find out which layout is in use. The following code uses the JSTL **`choose`** element to distinguish between our **`news`** and **`simple`** layouts:

```
<c:choose>
  <c:when test='${pool.rootElement.type == "news"}'>
    ...generate news layout...
  </c:when>
  <c:when test='${pool.rootElement.type == "simple"}'>
    ...generate simple layout...
  </c:when>
  <c:otherwise>
    ...handle error...
  </c:otherwise>
</c:choose>
```

5.2.3 Area and Group Options

The general objective of the Escenic system is to separate content creation and editing from layout: creating and editing is the responsibility of editing staff, layout is the responsibility of designers and template developers. When editing a section page, the Content Studio can choose what to place on the

page and where to place it, but cannot in general modify the appearance of the teasers placed on the page.

Area and group options provide a means of softening this division of responsibilities and providing editorial staff with some additional control over section page layout. The **news** group's **header** area is defined as follows in the **layout-group** resource:

```
<area name="header">
  <ui:label>Header</ui:label>
  <ui:description>Content placed here will appear at top of page</ui:description>
  <ct:options scope="current">
    <ct:field type="enumeration" name="news-area-background">
      <ui:label>Area background</ui:label>
      <ui:description>Changes the area background</ui:description>
      <ct:enumeration value="white">
        <ui:label>white</ui:label>
      </ct:enumeration>
      <ct:enumeration value="blue">
        <ui:label>blue</ui:label>
      </ct:enumeration>
      <ct:enumeration value="red">
        <ui:label>red</ui:label>
      </ct:enumeration>
    </ct:field>
  </ct:options>
  <ct:options>
    <ct:field type="enumeration" name="border">
      <ui:label>Style</ui:label>
      <ui:description>Sets the style of the header</ui:description>
      <ct:enumeration value="border: 1px solid black;">
        <ui:label>Border</ui:label>
      </ct:enumeration>
      <ct:enumeration value="border: 5px solid black;">
        <ui:label>Fat Border</ui:label>
      </ct:enumeration>
      <ct:enumeration value="background: #F55;">
        <ui:label>Red Background</ui:label>
      </ct:enumeration>
    </ct:field>
  </ct:options>
</area>
```

As you can see there are two sets of options defined in the **header** area:

- The first **ct:options** element has a **scope** attribute set to **"current"**. This means that the options it defines apply to the **header** area itself. It defines an option called **news-area-background**, an enumeration containing three CSS settings. The options defined here are displayed in Content Studio whenever the user selects the **header** area itself.
- The second **ct:options** element has no **scope** attribute. This means that it uses the default **scope** setting, **"items"**, and the options it defines apply to the **content** of the **header** area. It is an enumeration containing three CSS settings labelled **Border**, **Fat Border** and **Red Background**. The options defined here are displayed in Content Studio whenever the user selects a content item summary placed in the **header** area.

Note that the **ct:options** element is not actually a member of the **layout-group** namespace, it is "borrowed" from the **content-type** namespace. If you use this element, therefore, you must include a declaration of the **content-type** namespace in the **layout-group** resource, as follows:

```
<groups xmlns="http://xmlns.escenic.com/2008/layout-group"
        xmlns:ct="http://xmlns.escenic.com/2008/content-type"
        xmlns:ui="http://xmlns.escenic.com/2008/interface-hints">
...
</groups>
```

Any option values defined in this way are made available to the template developer in **PresentationElement** beans that represent content item summaries. The options are made available in a **Map** field called **options** so that they can be addressed by name. The options for the area itself can be retrieved from the area using JSTL expression like this:

```
${pool.rootElement.areas.header.options['news-area-background']}
```

The options for the individual content items in the area can retrieved as shown in this extract from the section page template **group/news.jsp**:

```
      <div style="${pool.rootElement.areas.header.items[0].options.border}">
        <c:set var="element" value="${pool.rootElement.areas.header.items[0]}"
 scope="request"/>
        <jsp:include page="ats.jsp"/>
      </div>
```

To see the whole of the above JSP file, look in section 3.4.

You can add **ct:option** elements to **group** elements as well as to **area** elements, and use them in a similar way. It is particularly useful to add **ct:option** elements to root **group**s. Adding a **ct:option** element to a root **group** makes it possible to change the layout of entire pages from within Content Studio.

It is important to be clear about how the various different kinds of options are applied:

- Group options apply to the group itself.
- Area options (**scope="current"**) apply to the area itself.
- Area item options apply to the content of the area.

You can see this difference in the way Content Studio displays options. Group and area options are displayed when the group/area itself are selected, whereas area item options are displayed whenever one of the elements in the area is selected. In the example shown above, for example, each teaser added to the header area can have a different **border** option setting.

A group placed inside an area can therefore have both area item options defined in the enclosing area and its own group options.

5.3 image-version

Use of the **image-version** resource is deprecated. You should use **representation** elements in the **content-type** resource instead, where possible. (In other words, you should **only** use

image-version if you require functionality that cannot be provided using **representation** elements.)

The Content Engine can generate different versions of images for use in different contexts: large versions for use in articles and thumbnails for use for front page teasers, for example. The **image-version** resource predefines the different image versions that may be generated. The usual case is that all the versions used in a publication are down-sampled and possibly compressed from the original image. The image-version resource therefore has:

- A single **originalVersion** element that specifies an **id** and a **label** for the original images
- A **version** element for each down-sampled image version. These elements also specify an **id** and a **label**, plus information regarding the down-sampling operation to be performed, such as the required resolution (**maxWidth** and **maxHeight** in pixels), the required format and so on.

Here is an example **image-version** resource that defines two image versions, **thumbnail** and **big**.

```
<imageDef>
  <originalVersion id="original">
    <label>Original</label>
  </originalVersion>

  <version id="thumbnail">
    <label>Thumbnail</label>
    <maxWidth pix="75"/>
    <maxHeight pix="75"/>
    <fallback operation="skip"/>
    <format name="jpg" />
  </version>

  <version id="big">
    <label>Big</label>
    <maxWidth pix="200"/>
    <maxHeight pix="200"/>
    <fallback operation="skip"/>
    <format name="jpg" compression="0.75"/>
  </version>
</imageDef>
```

This resource is now partially superseded by the **representation** element in the **content-type** resource. The representation element allows you to define variants of an image that have a specified height and width. Image variants defined using the **representation** element do not currently offer all the functionality provided by the **image-version** resource, but they give increased "editor control". Representations appear as crop windows superimposed over the base image in Content Studio, which an editor can move around and resize in order to select a specific area of the image for publication.

For a full, formal description of the **image-version** resource format and all the things you can do with it, see image-versions.

5.4 feature

Unlike the other publication resources, the **feature** resource is not an XML file. It is a plain text file containing a series of simple property settings like this:

```
allowFrontPageAsHomeSection=true
```

All the property settings consist of a single *keyword=value* pair like the one above, and modify the behavior of the publication in some way. For a full, formal description of the **`feature`** resource format and all the things you can do with it, see feature.

5.5 User Interface Hints

You have probably noticed that the resource file examples in section 5.1 and section 5.2 contain some elements with the prefix **`ui:`**. These elements are user interface hints, and the **`ui:`** prefix identifies them as belonging to the **`http://xmlns.escenic.com/2008/interface-hints`** namespace.

If you look at the syntax diagrams for the content-type and content-type resources you will see that many of them include the placeholder *ANY-FOREIGN-ELEMENT*. This placeholder is used to indicate that an element can contain elements from any foreign namespace, but is primarily intended to indicate that you can insert elements from the **`interface-hints`** namespace.

Use of the **`interface-hints`** elements is entirely optional - you can create a working **`content-type`** or **`layout-group`** resource without using them. By using them, however, you can create a more user-friendly interface for your publication in Content Studio.

The following sections discuss the use of some of the most frequently used **`interface-hints`** elements. For full details about all elements in this namespace, see interface-hints.

5.5.1 label

The **`interface-hints`** element you will probably make most use of is **`label`**. By default, Content Studio generates labels for user interface components from the **`name`** attribute of the resource file elements they are based on, by simply capitalizing the first letter of the name. A **`field`** element called **`title`** in the **`content-type`** resource, for example, will result in the field label **Title** in Content Studio. If, however, you want the field to be called **Headline** in Content Studio, then you can achieve this by adding a **`ui:label`** element as follows:

```
<field type="basic" name="title">
  <ui:label>Headline</ui:label>
</field>
```

The other important function of the label element is to enable multilingual user interfaces. An element can have several child **`label`** elements, each with a different **`xml:lang`** attribute identifying its language. For example:

```
<field type="basic" name="title">
  <ui:label xml:lang="fr">Titre</ui:label>
  <ui:label xml:lang="de">Titel</ui:label>
</field>
```

5.5.2 description

You can add more explanatory information to resource file elements with child **`description`** elements, which will be used where appropriate in Content Studio, for example as help bubbles displayed when the mouse is held over a user interface component:

```
<field type="basic" name="title">
  <ui:description>Enter the title of your article in this field.</ui:description>
</field>
```

5.5.3 value-if-unset

This is a very useful element that you can use to specify default values for fields:

```
<field type="uri" name="homepage">
  <ui:value-if-unset>http://www.escenic.com/</ui:value-if-unset>
</field>
```

5.5.4 group

You can use the **group** element in your **content-type** resource to define groups of content types that will be used for filtering search results in Content Studio. The following entries in **temp-dev**'s **content-type** resource:

```
<ui:group name="articles">
  <ui:label>Stories</ui:label>
  <ui:ref-content-type name="news"/>
</ui:group>

<ui:group name="image">
  <ui:label>Pictures</ui:label>
  <ui:ref-content-type name="image"/>
</ui:group>
```

result in two buttons being displayed on the Search panel in Content Studio:

Clicking on the **Pictures** button will cause the current search results to be filtered to contain only content items of type **image**. A **group** can of course contain more than one **ref-content-type** element, so you can create filters that select several content types.

5.5.5 style

You can use the **style** element to control the appearance of content in Content Studio's rich text fields in (that is, **basic** fields where **mime-type** is set to **application/xhtml+xml**) using CSS. You can put any standard CSS in the body of the element, giving you detailed control over the appearance and layout of rich text field content in Content Studio. To set the color of **h1** and **h2** headings in a field, for example, you could specify:

```
<field mime-type="application/xhtml+xml" type="basic" name="body">
  <ui:style>
    h1 {color:red}
    h2 {color:green}
  </ui:style>
</field>
```

You can also use this element to style in-line relations so that Content Studio users can easily distinguish between relations to different content types. To do this, you must create CSS classes with names of the form:

escenic-*content-type-name*

To make in-line links to **news** content items green and in-line links to **blog** content items red, for example, you could specify:

```
<field mime-type="application/xhtml+xml" type="basic" name="body">
  <ui:style>
    .escenic-news { color: green;}
    .escenic-blog {color: red;}
  </ui:style>
</field>
```

> The **style** element can **only** be used with **basic** fields where **mime-type** is set to **application/xhtml+xml**. It has no effect if used with any other elements.

For hints and examples about more advanced uses of the **style** element, see style.

5.5.6 style-class

You can use the **style-class** element together with the **style** element to add style buttons to the editing toolbar Content Studio displays with rich text fields. You could, for example, create a style button for marking text green as follows:

```
<field mime-type="application/xhtml+xml" type="basic" name="body">
  <ui:style>span.green { color: green; }</ui:style>
  <ui:style-class name="green">
    <ui:icon>http://my.server/myicon.png</ui:icon>
    <ui:description>Mark text green</ui:description>
  </ui:style-class>
</field>
```

For hints and examples about more advanced uses of the **style-class** element, see style-class.

5.5.7 icon

This element lets you set the icons used in Content Studio to represent content items (for example, in search result lists, inboxes, on editor tabs and so on) and CSS styling buttons for rich text field editors. You can only specify this element as the child of a **content-type** element or **style-class** element: it is ignored in all other contexts. You can either specify the name of one of Content Engine's predefined icons, for example:

```
<ui:icon>audio</ui:icon>
```

or the absolute URI of an image that you want to use as an icon. The image must be accessible from all the machines on which Content Studio will run. For example:

```
<ui:icon>http://my-company-server/icons/custom-audio.png</ui:icon>
```

For a complete list of all the predefined icons, see icon.

5.5.8 inline

The **inline** element can be used to define a set of options hence allowing for a better control over positioning of images within a content field (that is, **basic** field where **mime-type** is set to **application/xhtml+xml**)

You could, for example, enable/disable the image **alignment** option on a field by setting the **value** attribute to **on** or **off**. In addition a default alignment can be defined using **value-if-unset** as follows:

```
<field mime-type="application/xhtml+xml" type="basic" name="body">
 <ui:inline>
   <ui:alignment value="on">
    <ui:value-if-unset>top</ui:value-if-unset>
   </ui:alignment>
  </ui:inline>
</field>
```

6 Relations

A content item can be **related to** a number of other content items. A text content item such as a news article, for example, might have relations to:

- Images to be displayed with the article
- An image to be displayed with the article summary on section pages
- Other articles on the same subject, to be displayed as a list of links
- Related media objects, such as audio and video files
- Links to resources such as external web pages

6.1 Defining Relations

The Content Engine's **relation** concept allows these related items to be managed in an organized and standardized way.

The **news** content type defined in **temp-dev**'s **content-type** resource file contains the following reference:

```
<ref-relation-type-group name="attachments"/>
```

Elsewhere in the **content-type** resource the relation type group **attachments** is defined as follows:

```
<relation-type-group name="attachments">
  <relation-type name="images">
    <ui:label>Pictures</ui:label>
  </relation-type>
  <relation-type name="stories">
    <ui:label>Stories</ui:label>
  </relation-type>
</relation-type-group>
```

In other words, the **news** content type supports two relation types: **articles** and **image**.

6.2 Creating Relations

These relation types appear in Content Studio editors as **drop zones**, areas into which users can drag and drop items they want to relate to the content item they are editing. If you start Content Studio and open a news content item for editing, you should see the drop zones displayed to the right of the content item's fields:

If you cannot see the drop zones, select **View** > **Show content relations** from the main menu to display them.

To add relations to a content item, therefore, a Content Studio user simply has to drag the required items into one of these drop zones. The user needs to know the purpose of each relation, however, since Content Studio exercises no control over what the user drops in the zones. In this case, it will not

prevent you from dropping an image content item in the **Stories** drop zone, or from dropping a text content item in the **Pictures** zone.

Normally, relation drop zones are displayed in a relatively small area to the right of a content item's fields in Content Studio. You can, however, force them to be displayed in the main editing area along with the content item fields if required. See section 6.4 for further information.

6.3 Rendering Relations

Related content items are made available in JSP templates as PresentationRelationArticle beans. The PresentationArticle bean has a relatedElements property that returns a map from relation type to **`PresentationElement`** beans representing all the content item's related items.

`PresentationElement` beans that represent summaries hold the extra information about the relationship as well as a method to obtain the **`PresentationArticle`** bean of the related object. Therefore, it is up to the template developer to choose between information particular to this relationship and information that applies to all relations with this content item.

The **`temp-dev`** publication's **`article/markup/news.jsp`** template contains the following code to display links to all related items:

```
<div class="stories">
  <h3>Related articles</h3>
  <ul>
    <c:forEach items="${article.relatedItems}" var="relationType">
      <c:forEach items="${relationType.value.items}" var="related">
        <li>
          <a href="${related.content.url}">${related.fields.title}</a>
        </li>
      </c:forEach>
    </c:forEach>
  </ul>
</div>
```

As it stands, however, this list may well include links to images dropped onto the content item's **Pictures** zone in Content Studio. Given the title "Related articles", this is probably not the intention - more than likely what is wanted is a list that only contains other **`news`** items that have been dropped into the **Stories** drop zone.

You can achieve this by using **`relation types`** - in other words the drop zone onto which the Content Studio user dropped it. To select only items that were dropped into the **Stories** zone, therefore, you can modify this code as follows:

```
<div class="stories">
  <h3>Related articles</h3>
  <ul>
    <c:forEach items="${article.relatedElements.stories.items}" var="related">
      <li>
        <a href="${related.content.url}">${related.fields.title}</a>
      </li>
    </c:forEach>
  </ul>
</div>
```

This, however, ignores the images dropped onto the **Pictures** zone. Remedy this as follows:

```
<div class="images">
  <h3>Related pictures</h3>
  <ul>
    <c:forEach items="${article.relatedElements.images.items}" var="related">
      <li>
        <a href="${related.content.url}">
          <img title="${related.fields.caption}"
 src="${related.content.fields.binary.value.thumbnail}"
 alt="${related.fields.alttext}"/>
        </a>
      </li>
    </c:forEach>
  </ul>
</div>
```

Note how the code mixes summary fields (**`related.fields.caption`**) with content item fields (**`related.content.fields.binary.value.thumbnail`**).

6.4 Gallery Relations

In most cases, relations represent secondary content in a content item: they represent images or video that accompany an article, or links to other relates articles. It therefore makes sense for them to be displayed on the far right of the editor pane in Content Studio. Occasionally, however, this is not the case. In some kinds of content items, relations represent the primary content. The image galleries seen in many web publications are an example of this kind of content item. In an image gallery, images are the primary content, so they should be presented as primary content in Content Studio.

You can cause relations to be displayed as primary content in Content Studio by adding a ui:editor-style element to a **`relation-type`** definition in the **`content-type`** resource.

The following example shows how it is used:

```
<relation-type name="gallery">
  <ui:editor-style>gallery</ui:editor-style>
</relation-type>
```

`ui:editor-style` only has any effect if it is the child of a **`relation-type`** element and if it contains the value **`gallery`**. It causes relations of the specified type to be displayed like ordinary fields in Content Studio. It only affects display in Content Studio and has no other effects. Despite the use of the keyword "gallery", it is not restricted to use with image relations: you can use it for any relation type that you want to appear in the main part of the editor pane in Content Studio.

7 Tagging

Content Studio enables journalists and editors to **tag** content by attaching keywords called **tags** to content items. Throughout this chapter, the word **tag** is used to refer to a tagging keyword, (whereas elsewhere in this manual it is used to refer to an HTML or JSP element name).

Tags provide a simple and flexible way of categorizing and grouping content items for search and retrieval purposes. A journalist might, for example tag a travel article about Thailand with the tags **`Travel`** and **`Thailand`**. It would then be possible to find the article by searching for "Travel" or "Thailand" even if neither of those words appear in the article itself.

Tags can be organized in hierarchies, in order to be able to represent logical associations between the concepts they represent. If the tag **`Thailand`**, for example, is organized under another tag called **`Asia`**, then a search for content using the tags **`Travel`** and **`Asia`** would return the **`Thailand`** article (possibly along with other travel articles about other Asian countries). Structuring tags in this way is optional: it is also possible to just have a collection of unrelated tags.

In order for tagging to be enabled, the system administrator must first create at least one "container" for tags, called a **tag structure**. How many tag structures are maintained at a site is a design decision. You might choose to have a single tag structure for all tags, or a set of thematically organized tag structures such as **`geography`**, **`sport`**, **`culture`**, **`politics`** and so on.

Tag structures are created using the **`escenic-admin`** web application. You can also use **`escenic-admin`** to import predefined sets of tags to tag structures. For details of how to:

- Create tag structures
- Create tag syndication files
- Import tag syndication files

see Create a Tag Structure.

Once at least one tag structure has been created, tagging functions can then be made available to Content Studio users, as described in section 7.1. Subject to various controls, Content Studio users will then be able to:

- Attach existing tags to content items
- Set the **relevance** of attached tags (indicating how relevant the tag is to this particular content item)
- Change the order of the tags attached to a content item
- Create and delete tags
- Re-organize tag structures

7.1 Controlling Tag Usage

You can control the use of tags by specifying:

- Which content types in a publication may be tagged

- Which tag structures may be used to tag each content type
- Who is allowed to create/delete and reorganize tags

7.1.1 Controlling Tagging

By default, no content items may be tagged. You can enable tagging on a per-content type basis by adding **`ui:tag-scheme`** elements to a publication's **`content-type`** resource. Adding a **`ui:tag-scheme`** element to a **`content-type`** element enables access to one tag structure. You can enable access to several tag structures by adding multiple **`ui:tag-scheme`** elements.

For example:

```
<content-type name="news">
  <ui:tag-scheme>tag:iptc-topics.escenic.com,2002</ui:tag-scheme>
  <ui:tag-scheme>tag:folksonomy.escenic.com,2002</ui:tag-scheme>
...
</content-type>
```

Note that the content of a **`ui:tag-scheme`** element must be the **scheme** of one of the site's tag structures. A tag structure scheme is a URI that uniquely identifies the tag structure. The schemes of all tag structures defined on a Content Engine site are listed on the **`escenic-admin`** application's tag management page (see Manage Tag Structures).

The tagging functionality in Content Studio is exposed in a content editor **Tags** tab. If you edit a content item whose content type definition contains no **`ui:tag-scheme`** elements, then no **Tags** tab is displayed and tagging is not possible. If the content item's type definition does contain one or more **`ui:tag-scheme`**, elements, then a **Tags** tab is displayed, and you will have access to tags belonging to the specified tag structures.

7.1.2 Controlling Tag Creation and Deletion

All users with **`journalist`** rights are allowed to add tags to content items and remove them. Special access rights, however, are required to create, delete and reorganize tags. These access rights can be assigned to users by the publication administrator using Web Studio (see Editing Users and Persons).

7.2 Rendering tags

If any tags have been attached to a content item, then they are available via the **`PresentationArticle`** bean's **`tags`** property. This property is a collection of **`PresentationTag`** beans, each of which has the following properties:

`name`
: This tag's name.

`parent`
: This tag's parent tag (if any).

`children`
: This tag's child tags (if any).

`relevance`
: A number between `0` (not relevant) and `1` (highly relevant) indicating how relevant this tag is to the content item to which it is attached.

Tags can therefore easily be accessed using JSTL.

7.2.1 Accessing Content Item Tags

The following example lists all of a content item's tags, plus the relevance of each tag:

```
<ul>
  <c:forEach var="tag" items="${article.tags}">
    <li>${tag.name} ${tag.relevance}</li>
  </c:forEach>
</ul>
```

The **`relevance`** value would not normally be displayed like this, but you might use it to determine:

- Whether or not to display the tag name
- How to display the tag name

Note that tags are returned by **`${article.tags}`** in the same order that they are displayed in Content Studio. The Content Studio user is allowed to rearrange the order of the tags, so there may be significance in the order.

7.3 Accessing Parent Tags

The following example lists the parents (if any) of a content item's tags:

```
<ul>
  <c:forEach var="tag" items="${article.tags}">
    <li>${tag.parent.name}</li>
  </c:forEach>
</ul>
```

7.4 Accessing Child Tags

The following example lists the children (if any) of a content item's tags:

```
<ul>
  <c:forEach var="tag" items="${article.tags}">
    <c:forEach var="childTag" items="${tag.children}">
      <li>${childTag.name}</li>
    </c:forEach>
  </c:forEach>
</ul>
```

7.5 Tags and Search

Tags are usually indexed along with other content, so that tags can be searched for. If Solr's faceting functionality is enabled at your site, then it is also possible to make more sophisticated use of tags to provide "drill-down" links in search results, and tag clouds. For further information about this, see Using Solr.

8 The Tag Libraries

The Escenic tag libraries played a central role in earlier versions of the Escenic Content Engine: most of the functionality in Escenic templates were provided by Escenic tag library tags. This is no longer the case: the central tools for Escenic template designers are now JSTL and the JSP expression language, which provide a simpler, more standard and more intuitive means of accessing the content of the Escenic beans. Some of the Escenic tags are still useful, but many are no longer needed.

All of the Escenic tag libraries are retained in order to ensure backwards compatibility with older applications. They are:

template Tag Library
: This library contains tags for manipulating templates.

publication Tag Library
: This library contains only one tag, **`use`** which can be used to change the current publication.

article Tag Library
: This library contains tags for retrieving information from **`PresentationArticle`** beans.

section Tag Library
: This library contains tags for retrieving information from **`Section`** beans.

util Tag Library
: This library contains a variety of general-purpose tags.

collection Tag Library
: This library contains tags for manipulating collections.

view Tag Library
: This library contains tags for manipulating tree structures stored in **`View`** beans.

profile Tag Library
: This library contains tags for managing user profiles.

tag Tag Library
: This library contains tags for accessing tags (**`PresentationTag`** beans).

The tag libraries are described in detail in the Escenic Tag Library Reference.

The tag descriptions in the **Tag Library Reference** clearly indicate which tags are **deprecated** (no longer recommended for use). You should not use deprecated tags in new applications (and you should avoid their use if possible in new additions to old applications).

8.1 Common Attributes

Some Escenic tag attributes are used in the same way everywhere they appear:

id Attributes

In contrast to some other tag libraries (including the Struts tag libraries), **`id`** attributes are often **not** required. Whether or not an **`id`** attribute is required, it always has the same purpose. The **`id`** attribute is used to specify both the name of a scripting variable and the key of a page attribute that

can be used to access the value returned by the tag. For example, the template in section 3.3 shows the **article:field** tag used to retrieve article fields as follows:

```
<h1><article:field field="headline" /></h1>
```

However, **article:field** has an **id** attribute, so the same effect could be achieved as follows:

```
<article:field field="headline" id="myHeadline"/>
<h1><util:valueof param="myHeadline" /></h1>
```

If an **id** is optional and is not specified, then the result returned by the tag is printed (that is, replaces the tag in the final output).

name Attributes

The name of a bean that is to be used by the tag. In the following example, the **switch** tag will contain **case** tags that test the value of the **article** bean's **articleTypeName** property.

```
<util:switch name="article" property="articleTypeName">
...
</util:switch>
```

property Attributes

The name of a property (of the bean identified by the **name** attribute) that is to be used by the tag. If not specified, the bean identified by the **name** attribute itself will be used as the value. See above for an example.

9 What Next?

This guide has hopefully provided you with enough information for you to be able to set up a working environment for yourself and write the templates for a simple Escenic publication. There is, however, plenty more to learn, and this section contains a list of resources for further reading.

9.1 Escenic Resources

Reference Material

Escenic Content Engine Bean Reference

Escenic Content Engine Resource Reference

Escenic Content Engine Syndication Reference

Escenic Tag Library Reference

Other Developer Guides

Escenic Content Engine Advanced Developer Guide

Escenic Content Engine Integration Guide

Escenic Content Studio Plug-in Guide

System Administration

Escenic Content Engine Installation Guide

Escenic Content Engine Publication Administrator Guide

Escenic Content Engine Server Administration Guide

User Guides

Escenic Content Studio User Guide

9.2 Other Resources

Servlets
http://java.sun.com/products/servlet/docs.html

JavaServer Pages
http://docs.oracle.com/javaee/5/tutorial/doc/bnagx.html

JavaBeans
https://docs.oracle.com/javase/tutorial/javabeans/quick/index.html

JavaServer Pages Standard Tag Library
http://docs.oracle.com/javaee/5/tutorial/doc/bnakc.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