Chapter 3 – Implementation

In this chapter we will look at the Implementation of a RESS system called Detector as a plugin for the Enonic CMS, as well as the background of this concept and the CMS it is built upon.

# Enonic CMS

## Background

Enonic CMS is a web content management system created by the Norwegian company Enonic AS and was first launched as version 1.0 in 2001. It is based on Java Enterprise Edition (JEE) and utilizes many open-source technologies such as Spring (inversion of control), Hibernate (object-relational mapping database abstraction) and Saxon (XML and XSLT processing).

Enonic is meant to function as a software platform for “medium to large organizations” and as such provides the tools for development and publishing needed to do this. As with many modern CMS’s, it is built to support content creation and publishing by users and not just developers. To do this it has a web-based portal that gives users a user-friendly way of creating, managing and publishing content. It also supports “in context editing” (ICE) of web pages – the ability to edit the content of a web page while viewing the web page itself.

Because the system is based on open-source technologies, it aims to be platform-independent. It supports all common operating systems, servlet engines and relational database servers. The Enterprise Edition also supports directory servers such as the Lightweight Directory Access Protocol (LDAP) and Microsoft Active Directory (AD) for handling enterprise-level directory information.

Enonic comes in two different editions: Enterprise Edition (EE) and Community Edition (CE), where the latter is open-source and free to use under the Afferno General Public License 3.0 (AGPL 3.0). They are mostly similar, but with the EE supporting more enterprise-oriented elements such as directory servers, load balancing and dedicated support. The EE is available under the Enonic Commercial License.

## Datasources

A datasource in Enonic CMS is a collection of one or more Java method calls used to retrieve data from a database or other sources. Methods invoked in datasources return XML or primitive types, and only accept primitive types as arguments. To supplement the native library of methods available in datasources, new ones can be added through plugins. Every call to a datasource method uses an instantiation cache that stores the data gotten from the initial call, so that subsequent calls to the same method within the page will not trigger an actual call, but rather get the return value or XML from the cache.

Datasources are defined as XML and can contain several method calls. Each call contains a name attribute for the method and a list of parameters that specify the arguments to be passed to the method.

<datasources>

<datasource name="..." [condition=" "] [result-element=" "] [cache="false"]>

    <parameter name="...">value</parameter>

    ..

</datasource>

..

</datasources>

Code Snippet : An Enonic CMS datasource. Attributes in brackets are optional.

The condition attribute may contain a condition attribute. The value is commonly a Java Unified Expression Language (Java.el) expression that dictates when the method call should be executed. The result-element attribute specifies the name of the root element for the result-set XML that is returned from the method call. The cache attribute states whether or not the result set should be stores in the instantiation cache for subsequent method invocations.

## Device Detection and Classification

Enonic CMS supports device detection on the server. It does this by allowing an XSL-based device-classification script to be referenced in the site properties of the CMS. The device-classification script gets passed data from the CMS in the form of an XML containing values from the HTTP request, and the user. This is similar to having device detection with Device Detection Repositories, as the most common resource that can be used for device classification is the UA string that is present in the HTTP request header. No information about the supported features on the UA, apart from the UA string, is passed to the script in the native XML.

The output of the XSL script is a string describing the detected device class based on the data from the XML. Detecting a device class is done by a conditional block in the script that matches data from the XML against user-defined regular expressions. Whichever regular expression result in a match decides which device class is passed to the CMS. The output is attached to a context-element in the datasource result XML of all pages and portlets in the CMS, and is thus available for tailoring the site to the detected device class.

## Plugins

Enonic supports development of plugins for extending the functionality of the CMS, this is done using Java and Spring. Plugins are packaged in OSGi bundles, which are normal JAR files with extra metadata called the Manifest that allows for the modularization that is needed for plugins in the Java system of Enonic CMS. Maven is used for building plugins. It handles all the dependencies for the plugin as well as packaging the JAR file for deployment. Enonic has also created a Maven plugin that simplifies the process of packaging the plugin into an OSGi bundle that is compatible with the CMS. To deploy a plugin the JAR is moved into the plugin directory under the Enonic installation directory.

Developing a plugin for Enonic is essentially creating a set of extensions packaged into a JAR file. In Enonic CMS this means extending Java classes that are part of the Plugin Environment API.



Figure : The Enonic Plugin Environment API

Most of the classes in the API can be extended, with two exceptions:

* Extension is an interface and cannot be extended, but is implemented by ExtensionBase; ExtensionBase and HttpProcessor are super classes that should not be extended directly.
* FunctionLibrary should not be extended directly, but should be used as a Spring bean class to define a Function Library extension. All public methods in the extended class can then be invoked from datasources in the CMS.

The classes that extend HttpProcessor are mainly meant for handling HTTP requests and responses in some way, such as filtering based on the type of request the server receives, automatically logging in a user based on data in the request header, or taking full control of the HTTP response.

The TextExtractor class is meant to extract text from various binary document types, such as PDF, MS Word and RTF, to index it for the CMS’s search engine. It can be extended to support text extraction from formats that are not natively supported by the CMS.

The TaskHandler class handles scheduling of code execution, via extensions. Scheduling when execution should occur is done in the Spring bean of the extending class using a special property called “cron”. Cron is a reference to the Unix task scheduler of the same name. The Enterprise Edition of Enonic also supports running tasks scheduled through the TaskHandler in a cluster.

The FunctionLibrary class is meant to allow for extending datasources in Enonic. They give developers the ability to create an API that gives users access to database information from the CMS itself. These functions must only take parameters of primitive types and can only return primitive types and XML. As previously mentioned a FunctionLibrary extension cannot extend the class directly, but should be defined as an extension in the FunctionLibrary Spring bean. This is because of how Enonic handles this class and gives access to function invocation from its datasources.

# Detector for Enonic