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

In this part we will look at the parts of Enonic CMS that are relevant for the development of a Detector-like plugin.

## 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 1: 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 1: 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

In this part we will look at the development process of the Detector plugin for Enonic CMS, from conceptualization to implementation.

## Conceptualization

Looking at the plugin support for Enonic it is evident that a Detector-like system is feasible to implement as a plugin for the CMS. The native device detection system is also lacking in the amount of data it makes available to the user. Being able to identify and catalogue UA-specific features on the client and store it on the server can help developers and users tailor their web pages to specific devices more accurately. A Detector plugin can thus be a valuable addition to Enonic.

When we considered the plugin environment API, there were two different approaches to implementing a Detector system. They would both extend HttpInterceptor to intercept requests and detect the requester’s UA features on both the client- and server-side, and then either:

1. Store and/or retrieve the data from a database and forward the request with the data attached to the header, so it is available in the CMS’s device class resolver script.
2. Store the data if the device’s features are not already in the database, and then use an extension of FunctionLibrary to make the device features available in a datasource.

The first solution would be the most optimal; as it would not require users to radically change the way they handle device classes on their web pages. This would make the plugin more practical to use and integrate into already existing web pages built in Enonic. While this solution was conceptually sound, it was not necessarily feasible. This is because the way Enonic builds its device class resolver XML data was not transparent, and it was not obvious if custom headers attached to the request in an HttpInterceptor extension would carry on through to the device class resolver script.

The second solution would work, as making data from a database available through a datasource is what FunctionLibrary extensions are meant to do. It would require extra complexity in the plugin, as well as in the use of the plugin. It would need an extra extension as well as an alternative to the XSL-driven device class resolver script system that is built into Enonic. This approach would need a device-family classification system that would make the plugin more akin to the original Detector system of Dave Olsen.

Storing the data gathered by the plugin should be done in its own database. The reason for this is that the device feature data gathered is not a part of the native system and can not be stored in the same database as the contents of the website. A low-maintenance, lightweight database system should be used to minimize the amount of overhead caused by adding another persistence-system to the CMS, as well as simplifying the setup and configuration of the plugin for users. This solution will probably function as a better alternative to storing the data as a content-type through the Enonic Java API as storing and retrieving data directly from a database will normally be orders of magnitude faster than going through the whole call-stack that is invoked when retrieving content through the Enonic Java API. For a simple database some form of NoSQL system, either document-based such as MongoDB or CouchDB, or a key-value store such as Voldemort or Dynomite could be useful.

## Implementation

For the implementation we decided to attempt both approaches to see if they both worked, and potentially which one was the most efficient in terms of speed and ease of use.

### Technologies

For the client-side test suite we used Modernizr with all available tests, licensed under the MIT License. For the server-side UA string parsing we used a version of UA Parser created by Twitter, Inc. and licensed under the Apache License 2.0. For the database we chose MongoDB, a document-based NoSQL database system, available under the GNU Afferno General Public License 3.0.

The choice of using Modernizr for client-side tests and UA Parser for UA string parsing was made firstly because they are both used in the original Detector system. Secondly because Modernizr is the industry-leading feature detection system, and UA Parser is very lightweight while retaining all the functionality needed to gather the necessary information from UA strings.

MongoDB was chosen because it is lightweight, easy to set up and removes all object-relational-mapping work and schema planning needed when using relational databases. The amount of data and the complexity of the data structure do not warrant spending much time planning out a relational schema for the data. The data to be stored is essentially a single object containing key-value pairs in which the values are either objects themselves or Booleans. JSON, which is the format MongoDB stores its documents, is well suited for this kind of use case.

The plugin itself was by necessity written in Java using Spring, and Maven for dependency handling, building and deployment. It will have two extensions of the Enonic Plugin Environment API: HttpInterceptor and FunctionLibrary. The latter is only necessary for the second approach mentioned under Conceptualization.

### Application flow

The HttpInterceptor extension is what intercepts and handles the HTTP request that comes from the client before it is passed to the CMS. It contains two methods that must be overridden: preHandle and postHandle. As their names suggest they are invoked before and after the CMS has handled the request, and they accept an HttpServletRequest object and an HttpServletResponse object as arguments. These objects are passed to them by the CMS servlet that receives the request. All of our logic must thus be rooted in the preHandle method, as we need to intercept the request before any HTML is served by the CMS, i.e. before the HTTP response has been generated.

The program flow in our overridden preHandle method is as follows:

1. Get the UA string from the header of the request and look it up in the database.
2. If the UA string is present in the database, go to 3, else go to 4.
3. Do either, depending on approach:
   1. Attach the resulting object from the database to the HTTP request header and pass the request up the chain by returning true.
   2. Do nothing – the data will be fetched by a method invoked from a datasource later, pass the request up the chain by returning true.
4. Check if a cookie with the correct ID is present in the request, indicating that client-side tests have already been run on the UA.
   1. If present, parse the test results from the cookie and store them in a database object, go to 5.
   2. If not present, generate the correct HTML markup and JavaScript code to send Modernizr tests to the client.
      1. Generate HTML markup.
      2. Add Modernizr code and cookie-generator code to the markup.
      3. Send the generated markup to the client and return false to stop the request from going further up the chain.
5. Get information from the UA string.
   1. Parse the UA string using UA Parser.
   2. Store the collected data in a database object and go to 6.
6. Put all both the client-side data and server-side data into a common database object and store it in the database. Go to 3.

The postHandle method does not need to anything, as the goal of the plugin is to intercept the request and get the necessary data from it, not to manipulate the response on the way out.

### Plugin configuration

Enonic supports configuration of plugins using property files. These files define key-value pairs that can be referenced within the plugin. A default property file is present in the JAR file itself, with the possibility of having external property files overwriting the default values. Our plugin has several values set in the default property file to give users the ability to configure their database and reference external files such as their own customized Modernizr JavaScript file.

mongodb.uri = localhost

mongodb.port = 27017

mongodb.dbname = mongodetector

mongodb.collection = useragents

modernizr.uri = modernizr-2.6.2.min.js

Code Snippet 2: The default.properties file

An external property file can here be used to define a custom URI or port for the MongoDB instance being run on the server. It can also define the URI for the Modernizr file if the user wants to use one that is not bundled with the JAR file.

### Client-side tests

Modernizr handles the client-side tests. What tests are present in each Modernizr file can be customized on the Modernizr website. The default test-suite in our plugin contains all available tests from the Modernizr website, this includes tests for all HTML 5 and CSS 3 functionality, as well as miscellaneous web functionality such as WebGL and Geolocation. All the Modernizr tests are situated in a separate file and can be switched to suit each user by referencing an external Modernizr file in the plugin-properties file.

To send the results of the client-side feature tests to the server, they are put into a cookie with a special ID and format that only uses special characters allowed by RFC 6265. The results of the Modernizr tests are stored in a JSON object on the client, so the intuitive solution for the cookie value would be to use a similar format. We settled on replacing each delimiter with an RFC 6265-approved character. The reason this is important is that the Java cookie parser adheres to this standard, and will stop parsing a cookie value if it encounters an illegal character.

Colons (:) are not allowed in cookie values, so to split key-value pairs we used “double dash” (--). The outer delimiters of JSON objects are “curly brackets” ({}) and are not allowed in cookie values, they were switched for pipes (|). This is not ideal, as having similar opening and closing delimiters hinders detecting which level of nesting our parser is in. Since the result object only has one level of nesting, this problem was resolved by adding an extra delimiter to denote nested objects, we settled on using “forward slash” (/). This way the parser on the server can know which level of nesting its in by which character is delimiting each key-value pair.

Once on the server the cookie is parsed by a method the Interceptor extension class and converted into a MongoDB basic database object.

### Server-side tests

The server has access to a small, but useful, set of data about the UA through the UA string in the HTTP request header. To extract this information we use UA Parser created by Twitter, Inc. It takes the UA string as an argument and returns an object containing data about the UA family and version, operating system family and version, and the device family, as well as if the device is mobile or a search engine spider. Each of these categories, UA, OS and device, is put into their own MongoDB basic database object.

### The database

The database for the plugin is MongoDB and stores information to one collection of objects. Each of these objects contains the gathered data on the features of a single UA. The unique identifier for these objects is the UA string. MongoDB has a flexible schema, which means that collections do not enforce the structure of objects stored within it. This means that two objects within the same collection can have a completely different structure and set of fields. An object the plugin stores in the database does have a predefined structure, though, here shown as JSON:

userAgent : String

ua : {

family : String

major : String

minor: String

}

os : {

family : String

major : String

minor : String

}

device : {

family : String

isMobile : Boolean

isSpider : Boolean

}

features : { feature : Boolean or Object, … }

Code Snippet : The database object structure

The “userAgent” field contains the UA string. The “ua” field contains the UA data from UA Parser. The “os” field contains the OS data from UP Parser. The “device” field contains device data from UA Parser. Lastly the “features” field contains the result object from the clients-side tests.

The advantage of having a database with a flexible schema in our case is that this object can be expanded later if new features need to be stored, without breaking the database schema or having to deal with old data being corrupted or unusable because of schema-mismatch.