Chapter 4 – Implementation

In this chapter we look at the implementation of the server-side feature detection plugin for the Enonic CMS. I will present how I approached the conceptualization of the system, as well as how it was actually implemented. I demonstrate the technical considerations that were made and how the chosen technologies were used to create a functioning server-side feature detection system for the CMS.

As previously mentioned Enonic has a built-in system for doing device detection. The system only detects the very basic data that can be extracted from the UA string, though. Because it can be a potential improvement over the built-in system, we will look at the development process of the server-side feature detection plugin for Enonic CMS, from conceptualization to implementation. It is inspired by the ideas of Dave Olsen and his Detector system, which I presented in Chapter 2.

# Conceptualization

Looking at the plugin support for Enonic it is evident that a server-side feature detection 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. As mentioned in Chapter 2, 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 device families more accurately. A server-side feature detection plugin can thus be a valuable addition to Enonic.

When I considered the plugin environment API, I identified two different approaches to implementing a Server-side Feature detection 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 separate database and forward the request with the data attached to it, 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 separate 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 undocumented and not obvious. It was also not obvious if it was possible to attach any data to the request in an HttpInterceptor extension, and whether it would be sent to the device class resolver script.

I figured out from early on that the second solution should work, as making data from a database available through a datasource is what FunctionLibrary extensions are meant to do. Though it would require extra complexity in the plugin, as well as in the use of the plugin. It would need an extra extension and 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 Gathered Data

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 should 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. This is because 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 Redis or Voldemort could be useful. A relational database could also be used, but it would impose an overhead both in terms of setting up and maintaining the system, by nature of its strict structure. The plugin should nonetheless be constructed to be database-agnostic to allow for connecting to any arbitrary database system.

# 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, as mentioned in chapter 2, 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. As previously mentioned any database system can be used, but document-databases such as MongoDB suit the use-case of the plugin better than relational databases. This is especially true when considering the relatively small amount of data that needs to be stored. The amount of data and the complexity of the data structure do not warrant spending much time planning out a relational schema for it. The data that needs 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. Because Modernizr is a JavaScript tool its test results are stored in a JavaScript object, which is, as mentioned in chapter 2, represented with JSON. This makes the translation from the Modernizr test result object to a MongoDB document simple. MongoDB also has a flexible schema that allows the document structure to be changed and expanded without invalidating or corrupting older documents [[1](#_ENREF_1)]. This can prove quite useful when considering the future friendliness of the plugin; allowing it to have the set of detectable features from Modernizr expanded without having to edit any database schemas.

The plugin itself was by necessity written in Java using Spring, and Maven for dependency handling, building and deployment. It needed two extensions of the Enonic Plugin Environment API: HttpInterceptor and FunctionLibrary. The latter was 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 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, or if the GET variable indicating the lack of JS support on the UA set, indicating that client-side tests have already been run on the UA.
   1. If a cookie is present, parse the test results from the cookie and store them in a database object, go to 6.
   2. If the no JS support variable is set, store the nojs indicator and go to 6.
   3. 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. Add “noscript” element to the markup to catch UA’s lacking JS support.
      4. Send the generated markup to the client and return false to stop the request from going further up the chain. Go to 5.
5. On the client the page will look blank to the user, but will only be visible for the time it takes the JS code to execute:
   1. If the browser supports JS Modernizr will run its tests, a cookie will be generated and the page will be reloaded using JS. Go to 4.
   2. If the browser does not support JS a “noscript” element in the generated markup will add a GET variable to the end of the requesting URL before reloading the page, indicating the lacking JS support to the server. Go to 4.
6. 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.
7. 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 do 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. Manipulating the markup sent in the response should be done by the CMS based on the data and family definitions gotten from the plugin. Changing the response directly through an HttpInterceptor extension would not adhere to the RESS principles mentioned in chapter 2.

The first approach mentioned under point 3 in the application flow proved to be impossible in the current incarnation of the HttpInterceptor, as it has no ability to change or add values to the HTTP request header in any way. This left us with the second approach, having a FunctionLibrary extension that accessed the database to retrieve and process the data stored by the interceptor extension.

## 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 and device family definition JSON.

mongodb.host = localhost

mongodb.port = 27017

mongodb.dbname = mongodetector

mongodb.collection = useragents

modernizr.uri = modernizr-2.6.2.min.js

families.uri = families.json

Code Snippet : The default.properties file

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

## The Database

The default database for the plugin is MongoDB, and it stores information to a single 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. As mentioned previously, 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, shown as JSON in Code Snippet 3.

userAgent : String

uaFamily : String

uaMajor : String

uaMinor: String

osFamily : String

osMajor : String

osMinor : String

deviceFamily : String

deviceIsMobile : Boolean

deviceIsSpider : Boolean

features : { feature : Boolean or Object, … }

Code Snippet : The database object structure

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

We have the advantage of having a database with a flexible schema, which means this object can be expanded later if new features need to be stored. This can be done without breaking the database schema or having to deal with old data being corrupted or unusable because of schema-mismatch.

## The Data Access Object

To access the database from the plugin a data access object (DAO) was created. DAO is a design pattern that is meant to separate the business logic and the persistence layer of an application [[2](#_ENREF_2)]. It consists of an abstract interface that, when implemented, exposes specific methods to the business logic that can be used to read and write data to the database. This can be done without the business logic having to concern itself with the details of the underlying database, making it database-agnostic.

Using the DAO pattern has several advantages, both for the sake of supporting several different database technologies and to enforce separation of concerns within the plugin itself. The business logic in the extensions only interacts with the DAO interface, and as such does not care about what is running underneath to store and retrieve the data. To support this it is necessary to define a domain model as a Java class that represents the objects that should be stored in the database. This allows it to be mapped to the correct database format. Having the domain model defined as a plain old Java object (POJO) means the implementations of the DAO interface can operate on the POJO and let the chosen POJO-to-database mapper do the database-specific work. It is also important to use a mapper that does not rely heavily on Java annotations, as this could potentially litter the domain model class with implementation-specific annotations that could confuse other mappers that might be needed later.

This pattern also helps in keeping the business logic clean, as all database-specific code is abstracted away behind the DAO object. The DAO object can also be instantiated through the Spring IoC container, moving the database connection and configuration into the context XML.

### The DAO implementation

Our implementation of the DAO was meant for use with our underlying MongoDB database. To be able to store the UA object in MongoDB we needed a mapper that could map from a POJO to the BSON format (Binary JSON) of MongoDB. We originally settled on using a mapper that was part of the Spring Data initiative, as this would simplify much of the process. The reason we ended up not using the Spring Data mapper was because it would cause severe compatibility issues with the version of Spring that was embedded in Enonic 4.6. We settled on a lightweight alternative called MongoJack, which is a small POJO-to-Mongo library that wraps the classes of the official MongoDB Java Driver and utilizes Jackson, a JSON and XML mapping library.

The MongoDB DAO implementation is instantiated with all the information needed to connect to a MongoDB database and construct a collection object that can be used to create, retrieve, update and delete objects from the database. The collection object is instantiated by a static method supplied by MongoJack, which creates a new collection object, but also takes the type of object that should be persisted as an argument to assist the Jackson mapper.

# The HTTP Interceptor Extension

The first extension needed for the plugin is the HTTP Interceptor extension. Its task is to handle incoming HTTP requests before they are processed by the CMS. The handling consists of checking if the UA making the request has had its featured detected, running both client- and server-side tests if it is an unknown UA. The information gathered can then be retrieved by the CMS through datasources calling methods in our FunctionLibrary extension, as per the second approach mentioned earlier.

## 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 the 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 [[3](#_ENREF_3)]. An alternative that I considered was to send the results back to the server using HTTP POST in an asynchronous call (AJAX), and redirecting once the acknowledgement came back from the server. The reason I ended up going for the cookie-solution because it was less reliant upon getting a timely response from the server. Using AJAX can in some cases end up having the request time out because of connectivity issues or otherwise, especially in the unreliable environment of mobile devices on cellular networks. Using cookies means the result is stored on the client and can be picked up by the server the next time the client is connected and makes a request to the server.

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 because the Java cookie parser strictly 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.

flexbox--1|flexboxlegacy--1|canvas--1|canvastext--1|webgl--1|touch--0|geolocation--1|postmessage--1|websqldatabase--1|indexeddb--1|hashchange--1|history--1|draganddrop--1|websockets--1|rgba--1|hsla--1|multiplebgs--1|backgroundsize--1|borderimage--1|borderradius--1|boxshadow--1|textshadow--1|opacity--1|cssanimations--1|csscolumns--1|cssgradients--1|cssreflections--1|csstransforms--1|csstransforms3d--1|csstransitions--1|fontface--1|generatedcontent--1|video--/ogg--1/h264--1/webm--1|audio--/ogg--1/mp3--1/wav--1/m4a--1|localstorage--1|sessionstorage--1|webworkers--1|applicationcache--1|svg--1|inlinesvg--1|smil--1|svgclippaths--1|input--/autocomplete--1/autofocus--1/list--1/placeholder--1/max--1/min--1/multiple--1/pattern--1/required--1/step--1|inputtypes--/search--1/tel--1/url--1/email--1/datetime--0/date--1/month--1/week--1/time--1/datetime-local--1/number--1/range--1/color--1

Code Snippet : An example of the cookie value generated by Detector on Google Chrome.

Once on the server, the cookie is parsed by a method in the Interceptor extension class and converted into a linked hash map that represents all the UA features gotten from the client-side tests.

## 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 the Java implementation of UA Parser created by Twitter, Inc [[4](#_ENREF_4)]. 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.

The underlying parser gets its regular expressions for matching UA strings with their respective UA from YAML files that come with UA Parser. The UA strings are checked against the YAML files and a best match is found. Each of the categories, UA, OS and Device, is put into the user agent domain model object based on the results from UA Parser, along with the results from the client-side tests.

The YAML files that define how to parse the UA strings have to be maintained to be completely accurate. This means that the plugin will not be entirely maintenance-independent as long as it uses such a system. The data gathered from the UA Parser is not necessarily needed, though, and much of the functionality it provides can be implemented in custom client-side tests, which could remove the need to manually update the systems YAML files or similar due to the them being outdated in some way. Due to time constraints I elected to keep the UA Parser for this thesis, but replacing it is definitely something to look into in potential future work.

## Intercepting an HTTP request

As mentioned earlier in this chapter: when a HTTP request is made to a server running Enonic CMS with our plugin, the HttpInterceptor extension will be invoked before the CMS itself receives the request. It will ascertain if the requesting UA has had its features tested and run the tests if it is a new UA. To check this, the extension asks the DAO object to retrieve a UA object with the requesting UA string from the database. If no object is retrieved, it is a new UA that needs to be tested and stored in the database.

There are two situations in which the requesting UA will not have an entry in the database. The first is if no tests have been run yet, and the second is if tests have been run, but the results have not yet been stored in the database. In the first case the interceptor will generate HTML markup including the Modernizr tests, cookie generation JS and – for the sake of UA’s that do not support JS – a noscript element that redirects back to the requested URL. The second case is if the tests have been run, and the interceptor should parse the test results and store them in the database.

### Generating and sending the HTML markup

The HTML markup that is sent to the client is generated as a regular string containing the minimum amount of markup needed for a valid HTML document containing JS. The Modernizr JS is read from the file specified in the plugin’s properties, which can be embedded with the plugin’s JAR or defined externally by a user. The Modernizr JS and cookie generating JS are both appended to the string inside a “script” element residing in the “head” part of the HTML document. Lastly a “noscript” element is added inside the “body” part of the HTML document. The purpose of this element is to redirect back to the requested URL if the requesting UA does not support JS. Being able to redirect without the help of JS is crucial, as any site using the plugin would be stuck on the test page without it for any UA lacking JS support or that has it turned off.

When the generated markup is sent from the server, one of two things will happen: either the UA supports JS and the tests are run normally, a cookie containing the test results is generated and the UA is redirected back to the requested URL. Or the UA does not support JS or has support for it turned off, in which case the “noscript” element will redirect directly back to the requested URL. In this case it will attach a “nojs=true” HTTP GET parameter to the end of the URL. This is to inform the interceptor of the lack of JS when it intercepts the redirected request.

### Intercepting the redirected request

Once the client redirects back to the requested URL, the interceptor will once again query the DAO object for a database entry containing the requesting UA string. Because the test results have not yet been stored, no entry will be found. The interceptor will thus look for the presence of the “nojs” HTTP GET parameter that might be present in the URL, indicating that JS is unsupported or turned off on the client. If it is present the interceptor will add the key-value pair “nojs : true” to the features map that is added to the UA object. This will be the only entry in the features map for this UA object, as no feature tests can be run on the UA as long as JS is unsupported or turned off. Because of the possibility that the requesting UA might support JS, but has it turned off, some kind of mechanism must be in place to handle this special case. For example a timeout value could be set so that the UA might be tested again later, we will discuss this in later chapters.

If the HTTP GET parameter has not been set the interceptor will check for the presence of the cookie that should have been generated by the cookie generating JS that was sent to the UA. If it is present it means that tests have been run and the test results should be stored in the features map mentioned earlier. The cookie is turned into a linked hash map by the parser mentioned earlier in the “client-side tests” section. Once the client-side test results have been stored the UA Parser is invoked to extract the useful information the requesting UA string. The results from the UA Parser are stored in the main fields in the user agent object that is illustrated in the “database” section earlier.

Once the client-side and server-side test results have been stored in a user agent object the object is passed to the DAO, which maps the object to the format of the underlying database and saves it. Once the object has been saved the job the interceptor is complete. The data can then be accessed by function library extensions utilizing the DAO.

# The Function Library Extension

For the CMS to access the data gathered by the HttpInterceptor extension we created a FunctionLibrary extension to work as an API for Enonic datasources to retrieve UA data from our database. The idea was to expose two methods. One method that returns an XML representation of the stored data users can utilize within the CMS. The other method should simply return a string representing the device family defined by a separate JSON file specified by the user. The second method is supposed to be akin to Dave Olsens family system that we looked at in Chapter 2.

## Getting an XML representation of user agent features

Calling the method getUAFeaturesXML in the plugin’s FunctionLibrary extension retrieves an XML representation of the UA object. The reason for needing XML is because, as we saw in Chapter 3, Enonic’s templates are based on XSLT parsing of XML data that is provided by its datasources. The UA object in the database is a series of fields containing strings and Booleans, as well as a map containing either a Boolean or a subsequent map. These are simple data structures that can be mapped quite easily by standard POJO-to-XML mappers. For our purposes we used JAXB, which is a part of the standard Java extended library (javax).

Figure : The program flow of a datasource call to getUAFeaturesXML.

When getUAFeaturesXML is called from a datasource in Enonic, as seen in Figure 2, it gets the requesting UA from the database and subsequently calls a method that maps the UA object to the XML format using JAXB. The result from the mapper is a string-representation of the resulting XML document. Because Enonic supports JDOM XML Document objects in its datasources we then convert this string representation to an actual Document object by using the JDOM SAXBuilder class. The result from the SAXBuilder is then returned to getUAFeaturesXML, which in turn returns the Document object to the Enonic datasource that invoked it.

## Resolving a user agent family

As defined earlier in the thesis: UA families are classifications of UA’s that are based on which features they support. In our case this means the features our HttpInterceptor extension has detected that they support. UA families are defined in a JSON file. One is packaged with the plugin and is used if no other file is defined in an external plugin properties file. If an external JSON file is referenced in a user-specified properties file, that one will take precedence.

Figure : Program flow when a datasource invokes getUAFamily in the function library extension.

To resolve a family for the requesting UA, datasources can invoke the method called getUAFamily in the plugin’s FunctionLibrary extension; the flow of this call is shown in Figure 3. This method will, in short, parse the UA family JSON file and compare its values to that of the UA object to establish a best matching family. The best matching family is defined as the family with the most matching fields and no mismatches compared to the UA object. If no family is found it will fall back to the default family, represented by the string “default”. To parse the JSON file and compare the fields to those in the UA object, two methods are used, one to traverse the nested objects of the JSON file and one to actually compare the fields to those of the UA object. These methods are not publicly accessible from datasources as they are meant to be helper functions for getUAFamily. To parse the JSON file it is necessary to map the JSON objects into Java objects. This is done by using Jackson, which is the same library used to translate the UA objects into MongoDB objects in the DAO implementation.

Using Jackson we get a JsonNode object, which includes an iterator that can be utilized to traverse the JSON structure so we can compare its fields to those in the UA object. Since the top-level fields in the family JSON object are the families themselves, traversing them is done using a simple loop. The internal structure of each family object is more complex and may contain several levels of nesting. This arbitrary number of nested objects suggests that recursion is both the most efficient, and the most practical way to traverse them.

**private** **int** traverseJSONAndCountMatches(JsonNode json, String parent, UserAgent userAgent) {

**int** matchedFields = 0;

Iterator<Entry<String, JsonNode>> jsonIterator = json.fields();

**while** (jsonIterator.hasNext()) {

Entry<String, JsonNode> jsonEntry = jsonIterator.next();

String key = jsonEntry.getKey();

JsonNode value = jsonEntry.getValue();

**if** (!value.isObject()) {

**if** (testUAFeature(key, value.asText(), parent, userAgent)) {

matchedFields++;

} **else** {

**return** 0;

}

} **else** {

**int** recursivelyMatched = traverseJSONAndCountMatches(value, key, userAgent);

**if** (recursivelyMatched == 0) {

**return** 0;

} **else** {

matchedFields += recursivelyMatched;

}

}

}

**return** matchedFields;

}

Code Snippet : The recursive method used for calculating the number of matches in a given UA family definition.

The traversing algorithm, shown in Code Snippet 5, also has to keep track of the number of matches found so that a best matching family can be found, as well as giving an indicator of a mismatch if it occurs. The algorithm will count the number of matches found, but will return 0 if it encounters a mismatch. The return value from each recursive call is checked, and if it is 0 the caller will also return 0. This means that if any mismatch is found somewhere in the call-stack, the whole call will return 0, giving the algorithm a Boolean-like return value where 0 indicates a mismatch while any positive integer signifies a match of n elements. The return value from the recursive algorithm can thus be used to find not just a match, but the best match by resolving which family has the most matching fields.

Once the algorithm resolves a best matching family, the method traversing the JSON object returns the name of the resulting family as a string. This is then returned to the invoking datasource as an XML element, to make it easy to look up in XSL templates.

# Summary

In this chapter I have described the implementation of my server-side feature detection system as a plugin for Enonic. In addition I presented how the plugin intercepts HTTP requests to do its device- and feature-detection, how this data is persisted, as well as how I implemented a function library for the CMS to retrieve it.

# Bibliography

[1] 10gen. (2013). *Data Modeling Considerations for MongoDB Applications*. Available: <http://docs.mongodb.org/manual/core/data-modeling/>

[2] Oracle. (2002). *Core J2EE Patterns - Data Access Object*. Available: <http://www.oracle.com/technetwork/java/dataaccessobject-138824.html>

[3] A. Barth. (2011). *HTTP State Management Mechanism*. Available: <http://tools.ietf.org/html/rfc6265>

[4] S. Jiang, "ua\_parser Java Library," ed. GitHub, 2013.