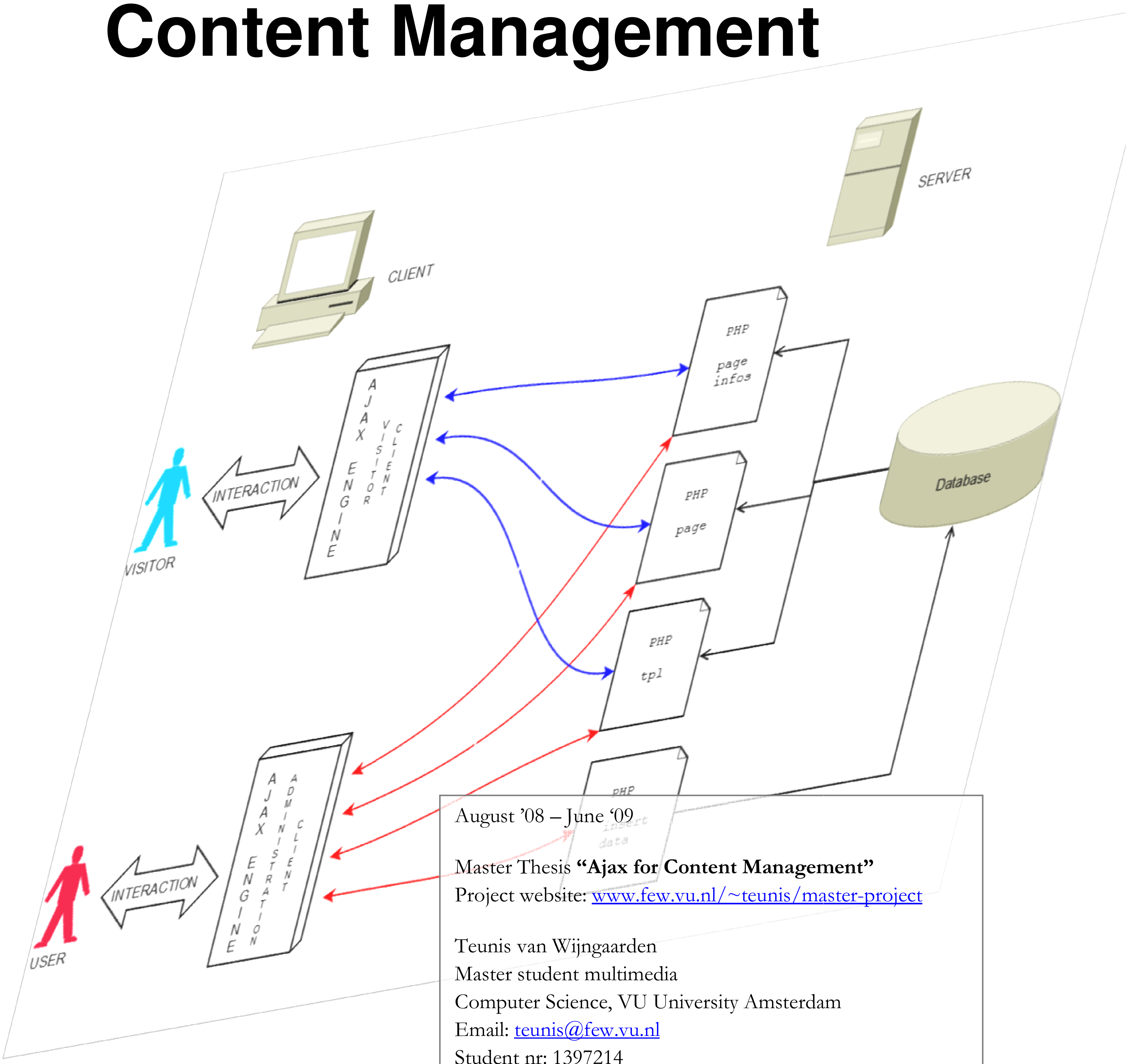
# Ajax for Content Management



August '08 – June '09

Master Thesis **"Ajax for Content Management"**
Project website: [www.few.vu.nl/~teunis/master-project](www.few.vu.nl/~teunis/master-project)

Teunis van Wijngaarden
Master student multimedia
Computer Science, VU University Amsterdam
Email: [teunis@few.vu.nl](mailto:teunis@few.vu.nl)
Student nr: 1397214

Supervisor:
Prof.dr. A. Eliëns (VU University)
Co-supervisor:
Dr. Željko Obrenović (Technical University Eindhoven)

## Abstract

Since 'Ajax' was invented in 2005, the number of applications and the interest keeps growing. Ajax offers new possibilities for websites and web application, also for Content Management Systems (CMS). Therefore, subject of this thesis is: what can the Ajax technology add to the experience of Content Management? And more hands-on: how can we, step-by-step, build an Ajax CMS?

Ajax introduces a new way of communication. In literature, useful (high level) patterns are described that can improve CMSs. Some of current CMSs administration clients are an Ajax application, but none of the visitor clients is a full single-page Ajax application.

The tutorial shows that, with the use of existing Ajax frameworks, complex but structured applications can be built. It also shows that new communication patterns for the web, like polling and prefetching, can be successfully implemented in a CMS visitor client. In this thesis, similar new functionality, submission throttling and pre-submission, is designed for a CMS administration client.

There are strong arguments to avoid heavy use of Ajax for a CMS visitor client. The browser compliance is in danger, and it does harm to Search Engine Optimization (SEO). A widget like way of implementing Ajax is less harmful. Most parts of the website then stay traditional. The cross-browser and SEO issues do not hold for the administration client and the benefits of Ajax (new way of interaction) might be even stronger than for the visitor client.

This thesis also discusses the design process of Ajax application. It shows that experts in many fields are needed (JavaScript, HTML, server side scripting, database etc.). The tutorial showed that the traditional UML Class Diagram and Sequence Diagram are still usable for designing Ajax applications.

# Table of contents

# 1 Introduction

Asynchronous JavaScript and XML, Ajax, is a combination of technologies that can change interaction with a website fundamentally. The client and server communicate without a browser reload, invisible in the back. The term Ajax was coined first in 2005. Since that day (or even before) the use of Ajax grew from small scripts that replace traditional reload script to full applications. Ajax becomes mature.

Still Ajax is not widespread in terms of percentage of websites that use Ajax. Some applications, like Google's Gmail or Microsoft's Hotmail, make heavy use of Ajax technology. But many websites (non-applications) do not. However, a fair amount of companies use, develop, research or plan to use Ajax [1]. How about Content Management Systems (CMSs)? Some of the CMSs currently available do allow developers or users to use Ajax in their website's at the visitor client. Also, some administration clients are enriched with Ajax. However, experiencing what Ajax can add more to CMSs seems an interesting topic. The reason to study CMSs, and not any other web application, is partly due to personal interest. It is in line with the Master courses on communication that the author attended.

The Ajax technology that gains ground, together with a possible more extensive use of Ajax in Content Management Systems is underneath the research question of this thesis: what can the Ajax technology add to the experience of Content Management? And more hands-on: how can we, step-by-step, build an Ajax CMS?

This study shows whether Ajax is (already) suitable for content management purposes, and what the benefits of Ajax for CMSs are. First, the history of Ajax is discussed in chapter 2, where we address the basics as well as more advanced topics. Chapter 3 is about the CMS visitor client that is built in the tutorial which can be found in the appendix. Chapter 4 discusses a CMS design with heavy Ajax usage, in which the design decisions are based on results of previous chapters. Chapter 5 evaluates the designed application but also general applications and the future trends. The conclusions are in chapter 6.

## 1.1 Terms and definitions

To understand the scope of this thesis, the term CMS is defined in this section. Also the terms visitor client versus administration client and server side versus client side are discussed.

**(Web) Content Management System**
The thesis only discusses Content Management Systems that can be used for websites. The online cooperative Encyclopedia Wikipedia describes this as a Web CMS[1]:

"A web content management system (WCMS or Web CMS) is content management system (CMS) software, usually implemented as a Web application, for creating and managing HTML content. It is used to manage and control a large, dynamic collection of Web material (HTML documents and their associated images). A WCMS facilitates content creation, content control, editing, and many essential Web maintenance functions. Usually the software provides authoring (and other) tools designed to allow users with little or no knowledge of programming languages or markup languages to create and

[1] Web content management system, Wikipedia, the free encyclopedia, http://en.wikipedia.org/wiki/Web_content_management_system

manage content with relative ease of use.
(...)
Administration is typically done through browser-based interfaces, but some systems require the use of a fat client. Unlike Web-site builders like Microsoft FrontPage or Adobe Dreamweaver, a WCMS allows non-technical users to make changes to an existing website with little or no training. A WCMS typically requires an experienced coder to set up and add features, but is primarily a Web-site maintenance tool for non-technical administrators."

This thesis uses a more basic, shorter definition. A 'Web Content Management System' is a web application that non-technical administrators can use to maintain, manage and control websites – the structure, page content and multimedia. Where the term CMS is used in this thesis, actually a WCMS is meant.

Visitor client versus administration client

Throughout the thesis the term 'visitor client' is used for the website that the visitor sees. The term 'administration client' is used for the application that the website-owner uses to manage the website. Both are part of the CMS.

**Server side and client side**
The term 'server side' indicates the web server and 'client side' means the computer of the application's or website's user (their browser). Server side technology is discussed only from an architectural perspective; details are left out because the focus is on Ajax Technology.

# 2 Background

In this chapter the history of Ajax is described. First, Ajax technologies will be introduced individually, followed by an explanation of the origin of Ajax. The context of Ajax, the changing web, is also discussed.

In the early years of the World Wide Web, the web was little more than a huge collection of static HTML-pages, interconnected by hyperlinks. There were only a few web browsers by which to access these pages. Users were required to enter the Unified Resource Locator (URL) into the web browser, to load a page. The URL is used to invoke the server and request that specific page or file. The server then sends the requested data and the browser displays it.

**From HTML to XHTML and CSS**
The HyperText Markup Language (HTML) and its successor eXtended HTML (XHTML) provides a way to format a webpage [2] . In time, more and more attention was paid to formatting (graphical design), using (X)HTML in combination with images. Cascading Style Sheets (CSS), linked to HTML, were introduced to describe how web documents should be displayed and to separate styling code from content [3].

**Introduction of JavaScript**
Netscape was the first to introduce scripts at the client side: LiveScript [4]. This name soon changed to JavaScript. Later on the language was stabilized by the European computer Manufacturer's Association and the official name became ECMAScript. In practice, everyone calls it JavaScript, but also Jscript can be encountered, which is Microsoft's variant. This thesis will use the name JavaScript, as it is a common name. Please note that JavaScript is not a simplified version of Java, as many think [5].

According to David Flanagan, who wrote a JavaScript bible, JavaScript is a full-featured programming language that is as complex as all other programming languages and allows Object Oriented programming. JavaScript is an interpreted programming language. The core supports numbers, strings and Boolean values and there is built-in support for array, date and regular expression objects [5].

Until a few years ago, JavaScript was only used for small programs built into HTML pages, in example to perform simple tasks like form validation. Before scripting, form validation was done on the server side.

**Introduction of frames**
In the HTML4.0 specification (1998), frames were introduced [6]. The frameset HTML entity was a new element that could turn a page into a framework for other pages. This is useful to separate the static from the dynamic content: the static menu could be on a different page than the more dynamic content, but in the same framework. A click on a menu link causes the page in the content frame to change. This separation was very profitable for maintenance – and the amount of data to transfer was reduced.

**Asynchronous requests with 'old' techniques**
Frames together with JavaScript gave developers the opportunity to contact the server asynchronously [7][4]. A hidden frame – with width or height of zero pixels – was used as a communication channel. Forms on a page in this hidden frame could be completed and sent with

JavaScript. Response could be extracted from the hidden page with JavaScript too. The page in the main frame changed without reload, only the hidden page reloaded.

Microsoft was the first to introduce the Document Object Model (DOM) in a new version of the old mark-up language: Dynamic HTML [4]. Although DHTML never made it to becoming a W3C Recommendation, the DOM did in 1997 [8]. The DOM provides a way to handle structured documents, like (X)HTML documents. With the DOM it is possible to access parts of a page by their path in the HTML tree or by using DOM functions. According to W3C, the DOM is "a platform- and language-neutral interface that will allow programs and scripts to dynamically access and update the content, structure and style of documents. The document can be further processed and the results of that processing can be incorporated back into the presented page." [8] The DOM's history at W3C started with DOM level 1. Development stopped in 2004 with DOM Level 3. The DOM is now an essential part of Ajax.

In the same year that the DOM became a standard, a new element was introduced to HTML: the iframe. This element could in itself contain a webpage. The hidden frame technique could be carried out with a zero by zero iframe. Great advantage was that such an iframe could be created on-the-fly using JavaScript and the DOM. This made asynchronous calls possible without having to plan a hidden frame in the HTML design (which encourages separation of 'code' and presentation) [4].

**An easier way to perform Asynchronous requests**

Microsoft was the first to 'legalize' asynchronous requests with the introduction of an ActiveX object XMLHttp [4]. This made it possible to do HTTP requests with JavaScript, which became very popular. Mozilla introduced XMLHttpRequest as a core JavaScript object [9], with the same functionality, but without ActiveX. Other browsers followed Mozilla and in the end even Microsoft added Mozilla's implementation to Internet Explorer 7[2].

The XMLHttpRequest is at the time of writing of this study not a W3C Recommendation, but its draft is in a final state [10]. The formal description of XMLHttpRequest maintained by W3C is: "The XMLHttpRequest object can be used by scripts to programmatically connect to their originating server via HTTP." [11].

The name of the object suggests that XML is the data format used in communication.  XML, short for eXtensible Markup Language, is a text format like HTML, but allows more tags. Two of W3C's design goals for XML are that it should "support a wide variety of applications" and it should be "easy to write programs which process XML documents" [12]. Indeed many parsers for XML exist for client side and server side scripts. In fact, Ajax is not restricted to the XML format, but any format can be used in communication.

**The new approach: Ajax**

The introduction of XMLHttpRequest completed the set of technologies that together form Ajax. In 2005, Jesse James Garrett was the first to combine these technologies and he invented the term Ajax: Asynchronous JavaScript and XML. According to him it incorporates the following [13]:

- XHTML and CSS
- The DOM
- XML and XSLT (a style sheet for XML)
- XMLHttpRequest

[2] MSDN, XMLHttpRequest object: http://msdn.microsoft.com/en-us/library/ms535874(VS.85).aspx

- JavaScript

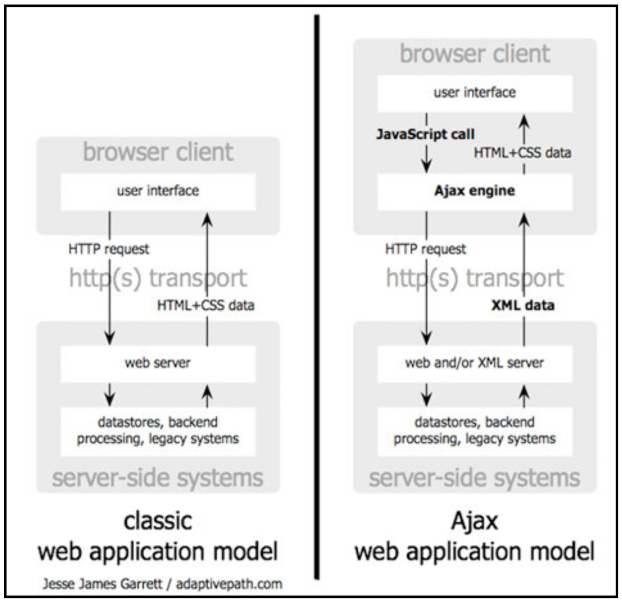


*Figure 1. The traditional and Ajax model for Web applications according to Jesse James Garett* [12].

This definition of Ajax does not include the server side, which plays a central role in handling the requests. Like Garrett's publication, this thesis emphasizes client side technology too. Server side scripts (in PHP, ASP or others) probably with a connection to a database, will only be touched upon lightly. These scripts do not use any new technologies (like XMLHttpRequest in JavaScript), except that the output format XML is relatively new.

Ajax changes the communication model for web applications [12][3]. Without Ajax the user requests a page (HTTP request by the browser), waits, and receives a page (HTML and CSS). The real work is done at server side: using a database, form validation etcetera. Now, with Ajax the user requests a page containing an Ajax application (the Ajax engine). Once the application is received, further HTTP requests are done by the Ajax engine in the background. Whereas the 'old' web application relies heavily on the server, the new web application moves functionality to the client side. Figure 1 shows the new application model.

Why should developers convert their traditional web application to Ajax based applications? One reason could be the new interaction model. The old applications cause the user to work on a page, request a new page or send a form, and wait until a new page is loaded. This repetition of working and waiting is called the work-wait model [13][7]. Ajax applications break this traditional model. They allow the user to work on client side, while the sending and loading happens most of the time behind the scenes by the Ajax engine. Ajax applications behave more like desktop applications.

Another reason to start programming Ajax is the range of new possibilities. With requests being made in the background it is possible to reload parts of page, e.g. actualize the news headlines at a page's sidebar while the user reads an article. Also, preloading pages, for instance the next chapter of an e-book, can be done by Ajax. Other patterns are: pre-submitting parts of a form (i.e. a file) while the user fills out the rest or buffering submitted information (submission throttling) [14].

**Ajax's context: Web 2.0**
Up until now Ajax has been discussed as a technology on its own. Ajax is, however, part of a technological shift. According to Tim O'Reilly, Ajax is a key component of *web2.0*. Although some companies use it as a buzzword, the term web2.0 does define the 'new internet' with a specific type of website. Unfortunately, the characteristics are not clear yet [15]. At least it can be said that web2.0 websites or applications share certain characteristics: rich user experience, user participation, dynamic content, metadata, web standards and scalability [16]. O'Reilly adds to this: lightweight programming and the end of software lifecycles.

Macromedia invented a term for web applications that accomplish a rich user experience: Rich Internet Applications (RIA). Technologies to build such applications are not only Ajax, but also Flash/Flex and Java Applets. Common driving forces are: the introduction of broadband internet, increased computing power of clients, demand for responsive applications, choice of big companies (like Google) and the emergence of web services and Service Oriented Architectures (and XML) [17].

The use of asynchronous requests is not the only thing that creates a richer experience. The visual design – that contributes most to the look-and-feel – adds to this. The techniques used to create an enhanced graphical layout have no direct link with Ajax, but they are in some way part of it: HTML, CSS and XSLT.

Another web2.0 thing is (third party) web services. Google for instance provides programming access to their applications (*Google Maps*, *Picasa Webalbums* and more) by providing an API[3]. Yahoo does the same for their *Flickr* photo album[4]. These APIs allow developers to include parts of the services in their website, embedded in their own layout. They also allow some creativity: mashups, a combination of services. Examples of mashups are photos from a Flickr album combined with locations on a Google Map or traffic information (possibly also from a third party) shown on a Google Map.

## 2.1 Ajax fundamentals

This chapter handles how to program Ajax: the basics of an HTTP Request, basic DOM manipulation and some practical information on data formats. The basics of XHTML and CSS are considered to be known. If not, W3 Schools might be helpful[5].

**The Document Object Model**
We will use the DOM to access and manipulate XHTML documents. The DOM views such a document as a tree, with the XHTML elements as nodes[6]. A way to access a node is by navigating through the tree. A node's attributes to do this are: `parentNode`, `childNodes` (which returns a list of nodes), `firstChild`, `lastChild`, `previousSibling` and `nextSibling`. Another way of navigating is by retrieving a list of nodes with a certain tag from the `document` object by using its function `getElementsByTagName()`. In addition, an easy way to access a node with a known id attribute is `getElementById()`.

JavaScript programmers are probably not satisfied with read-only access and want to manipulate the document or nodes. Therefore, the DOM provides ways to add and remove nodes, for instance with the document functions `appendChild()` or `removeChild()`. In addition there is a way to

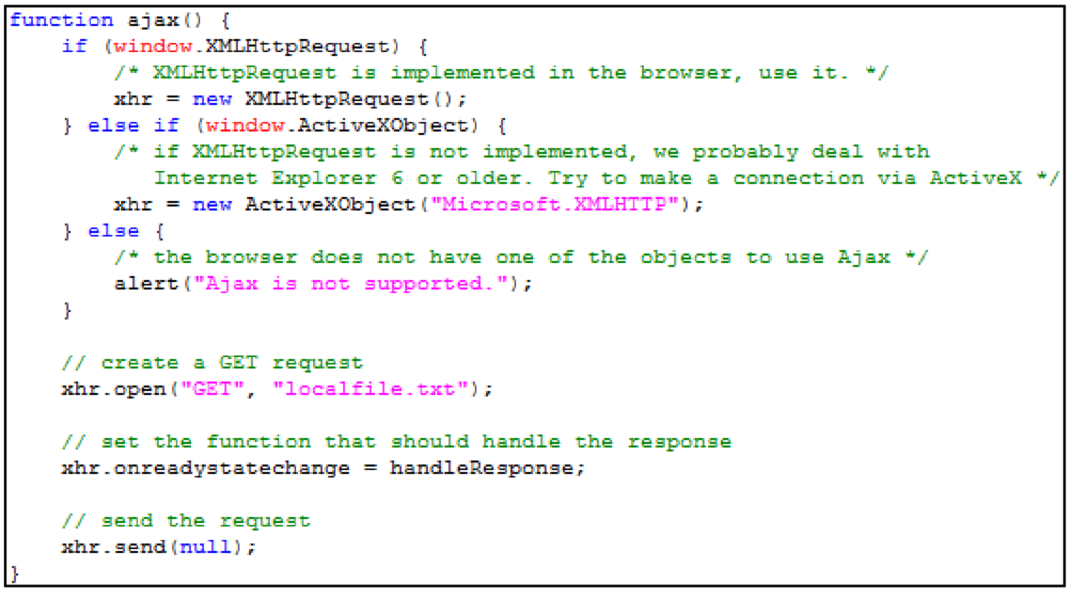
[3] http://code.google.com/
[4] http://www.flickr.com/services/api/
[5] W3 Schools, http://www.w3schools.com
[6] http://www.w3.org/TR/2004/REC-DOM-Level-3-Core-20040407/core.html

manipulate the nodes itself by changing their inner HTML or attributes. W3Schools[7] offers an overview of functionality to use the DOM in JavaScript to access (XHTML) pages.

**XMLHttpRequest**

Most popular browsers have implemented XMLHttpRequest in their newest versions[8], but for older versions Microsoft's implementation differs. In Internet Explorer 6 and before, the object was not a native part of the browser. Instead ActiveX was used. This is why developers should always include a check for the XMLHttpRequest object to exist and they should add the ActiveX constructor too.

Once the object is created, requests can be send by using the `open()` and `send()` function. Any HTTP command can be used, but `GET` and `POST` are the most useful for Ajax purposes. The `open()` function needs the HTTP command and the file. For a `POST` request the variables should be added as a parameter too. Before sending the request, be sure to set an `onreadystatechange()` function that handles the response. An example request is shown in Figure 2.

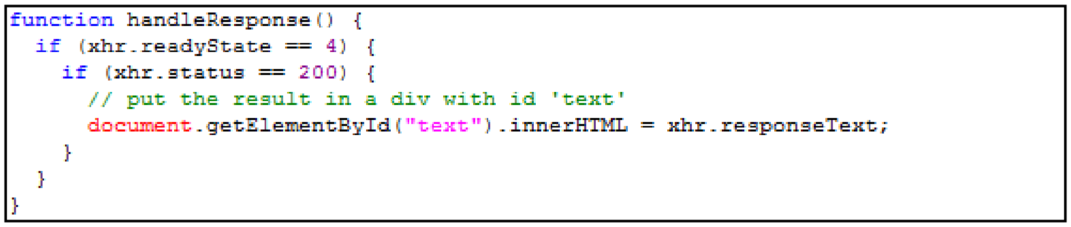


```
function ajax() {
    if (window.XMLHttpRequest) {
        /* XMLHttpRequest is implemented in the browser, use it. */
        xhr = new XMLHttpRequest();
    } else if (window.ActiveXObject) {
        /* if XMLHttpRequest is not implemented, we probably deal with
           Internet Explorer 6 or older. Try to make a connection via ActiveX */
        xhr = new ActiveXObject("Microsoft.XMLHTTP");
    } else {
        /* the browser does not have one of the objects to use Ajax */
        alert("Ajax is not supported.");
    }

    // create a GET request
    xhr.open("GET", "localfile.txt");

    // set the function that should handle the response
    xhr.onreadystatechange = handleResponse;

    // send the request
    xhr.send(null);
}
```

*Figure 2. An Ajax request in JavaScript.*

The variables of a `POST` request are probably send to a server side script that might consult a database and return an answer. Just assume that the response is there. The `onreadystatechange`-function will not only be invoked when the request is finished, but also in intermediate states: when open has not been called (readystate is 0), when open has been executed (1), when send has been executed (2), when the server has returned a chunk of data (3) and when the request is complete and the server is finished sending data (4) [6]. Furthermore, when finished, the status property of the XMLHttpRequest object contains the HTTP status, i.e. 200 for success or 404 in case the file is not found. An example of response handling can be found in Figure 3.

[7] http://www.w3schools.com/js/js_obj_htmldom.asp
[8] Firefox, http://developer.mozilla.org/en/XMLHttpRequest
Internet Explorer, http://msdn.microsoft.com/en-us/library/ms535874(VS.85).aspx
Opera, http://www.opera.com/docs/specs/opera9/xhr/
Safari, http://developer.apple.com/internet/safari/faq.html#anchor19



```
function handleResponse() {
  if (xhr.readyState == 4) {
    if (xhr.status == 200) {
      // put the result in a div with id 'text'
      document.getElementById("text").innerHTML = xhr.responseText;
    }
  }
}
```

*Figure 3. An Ajax callback function that outputs the response to a div element with id "text".*

There are some more functions and variables in the XMLHttpRequest object, having to do with headers, status and MIME types. Please consult the W3C website for more information on this[9].

Important to remark is the cross-domain issue. Browsers do not allow a script that is acquired from one server, to communicate with another server. So, the developer is bound to his own server with its own services. There are some ways to overcome this, for instance when a (third party) service 'knows' the developer's server and allows its requests. Another solution is using a proxy.

**Data formats: XML and JSON**

Although the 'x' in Ajax suggests that only XML is used for communication, every format is possible. XML is indeed a good choice, because it is a W3C Recommendation [12] and programming languages on both client and server side contain (third-party) XML parsers. In practice, XML is often replaced by the lightweight format JavaScript Object Notation (JSON), which is also covered by server side and client side languages.

- **eXtensible Markup Language (XML)** like HTML, is derived from SGML, which is a system for defining markup languages. Each SGML application contains a reference to a Document Type Definition (DTD[10]) that defines the syntax of markup constructs, like explained by W3C[18]. A root element of an example XML file could be `<news>`. This element contains for instance one or more `<news-items>`. Those items have some properties, saved in child nodes. The definition of what elements `<news> and <news-item>` can contain, is stated in the DTD.

- **JavaScript Object Notation (JSON)** is a new lightweight data-interchange format [19], proposed by David Crockford [4]. JSON exists as long as JavaScript does, but it was not used before Ajax. Not much can be found about JSON in literature, but the technique is much used in practice. JSON is based on a subset of JavaScript, containing object and array. Objects contain pairs of property and value. Arrays contain values. A value could be a string, number, object array, true, false or null [18]. On average, JSON requires less characters, because the delimiters of objects and arrays are `{}` or `[]` instead of `<tag></tag>`. The JSON format therefore requires less bytes than the same data in XML. Because JSON uses JavaScript syntax, it requires less parsing than XML when used in Ajax Applications [4]. Unfortunately, the syntax makes JSON harder to read for programmers. JSON is also covered by server side technology like PHP.

[9] http://www.w3.org/TR/2008/WD-XMLHttpRequest2-20080930/#references
[10] The syntax of a DTD is not covered in this study, please consult w3C, http://www.w3.org/TR/REC-html40/sgml/dtd.html.

## 2.2 Advanced topics

It is hard to program advanced applications by hand with the basic techniques given in the previous chapter. Therefore, frameworks have been developed to overcome this. In addition some patterns were developed that describe how to overcome some well known (communication) issues. Both are discussed in this chapter.

**Frameworks**
The browsers that are currently available use different ways to perform a HTTP request. The DOM and some CSS-styling interpretation also differs for every browser[11]. A good way to handle these cross-browser discrepancies is by writing a function for every browser dependant task that performs a browser check, and base the code on that. An even better way is by retrieving these functions from third party libraries.

The Prototype JavaScript framework is such a library that overcomes browser compatibility issues. This JavaScript class provides functions for handling Ajax Requests, but also for using arrays, strings, classes, events and other high level programming structures [20]. Similar libraries are Script.aculo.us[12], jQuery[13], but many more are available [21].

The Google Web Toolkit (GWT)[14] also attempts to make Ajax programming easier. According to Google, GWT can be used to build "highly performant JavaScript front-end applications in the Java programming language". It consists of a library with all kinds of structures and graphical interface objects defined in the API[15], together with a Java to JavaScript compiler [22]. GWT solves cross-browser issues in an efficient way, because it only loads the code that is needed for a specific browser (ActiveX function calls are not sent when using Firefox). Solutions like this increase performance of applications made with GWT [23]. From a developer's viewpoint, programming Java might be more familiar and thus easier (quicker) than programming JavaScript.

**Patterns**
Ajax programmers do not only profit from the libraries and frameworks that hide some low level issues; there are also some (communication) patterns that bring Ajax to an even higher level. Zakas et al. mention patterns that have to with synchronization with the server or increasing perceived speed of the internet [4]:

- *Predictive Fetch*; the Ajax application guesses what the user is going to do next and retrieves the appropriate data.
- *Submission throttling*; the application buffers data that should be sent to the server and thereby decreases the number of requests. Zakas et al. mention Google Suggest as a fine tuned example.
- *Periodic Refresh*; this pattern is also known as 'polling': checking the server from time to time for new information.

[11] In example, changing an objects opacity in Firefox van be done by using the CSS declaration 'opacity:x', while in Internet Explorer the syntax is 'filter:alpha(opacity=x)'. Source: W3Schools, http://www.w3schools.com/css/css_image_transparency.asp
[12] Script.aculo.us, http://script.aculo.us
[13] jQuery, http://jquery.com
[14] http://code.google.com/intl/nl/webtoolkit/
[15] Google Web Toolkit Javadoc, http://google-web-toolkit.googlecode.com/svn/javadoc/1.5/index.html?overview-summary.html

- *Try Again*; this pattern provides a way of dealing with errors. As the name suggests, in this pattern the request is resent after failure.

Combinations of the patterns above and others create the new feeling and new interaction model that comes with Ajax and which is part of web2.0.

# 3 Ajax in current Content Management Systems

Ajax involves many techniques and can be used for many purposes, therefore it is hard to define what a real 'Ajax CMS' is. For the CMS in this thesis, Ajax is a native part of the visitor client as well as the administration client, but other developers use the term Ajax CMS for administration clients with an enhanced interface or for visitor clients that allow web designers to use Ajax. In this chapter a selection of current CMSs that mention Ajax on their website are explained and compared.

## 3.1 Current systems

Below is a list of CMSs that advertise with supporting Ajax or being an "Ajax CMS".

- The open source PHP Application Framework MODx[16] is an Ajax CMS, according to the creators. The commercial text on their website says: "It makes child's play of building content managed sites with validating, accessible CSS layouts using any Ajax library or techniques you wish, hence why we refer to it as an Ajax CMS." This CMS only allows web developers to incorporate Ajax in their visitor client: "Our philosophy all along has been to impose no restrictions on the Ajax libraries you use." No off-the-shelf Ajax functionality is described.
- The 'Ajax CMS'[17], by Sosign Interactive, has an Ajax powered administration client. A video on the company's website shows What You See Is What You Get (WYSIWYG) editing with drag-and-drop support. From a visitor client perspective Ajax is not explicitly mentioned on the website.
- Skeletonz[18] CMS, by Amir Salihefendic, is said to be simple, powerful and extensible. Ajax is mentioned under the header 'Responsive interface': "You and your users won't reload that often! Most things in Skeletonz are Ajax based." Again this is administration client only.
- NEOCMS[19] is another CMS that uses Ajax mainly for the interface. The demo on their website shows some fancy WYSIWYG editing, online image editing and drag-and-drop. However most functionality requires a new window to open, and changes to the structure for example are not visible immediately, which means that Ajax is not used for this functionality. Ajax in the visitor client is not discussed on the website. Though, some parts inside the company's website reload separately from other content, which indicates Ajax usage. So, probably it is possible to use Ajax in the visitor client.
- The open source 'CMS from scratch'[20] has a "Quick & simple AJAX-powered interface." The demo shows indeed a one-page administration client that seems to upload and download on the background and therefore behaves desktop-like. The system does not explicitly support Ajax in the visitor client , because there are only three content types: text, HTML and sites (flexible tables).
- Enano[21], by the Enano team, is an open source project in development (now released in beta). Like the others, this system uses Ajax primarily for a desktop like interface at administration client, but they take it to the visitor client in some aspects: "Users can leave

[16] The MODx CMS project, http://modxcms.com/
[17] Ajax CMS, Sosign Interactive, http://www.sosign.com/Ajax-CMS,65.html
[18] Skeletonz, http://orangoo.com/skeletonz/
[19] NEOCMS, http://www.neocms.be
[20] CMS from scratch, http://cmsfromscratch.com/
[21] Enano CMS, http://enanocms.org

comments on articles using an extremely streamlined AJAX-based flat style discussion system."

- The Pligg[22] CMS allows using Ajax in visitor client plug-ins. There is for instance an Ajax Chat module for Pligg available.

The list shows that there are many ways to incorporate Ajax in a CMS. CMSs that are not on the list, like commercial ones or the popular/downloaded systems, might also use or support Ajax in one of these ways.

## 3.2 Comparison

The CMSs that are listed in the previous section all use Ajax in the administration client. Some only use widgets, like WYSIWYG-editors or drag-and-drop for suitable functionality, others are full single-page Ajax applications. The visitor clients show less Ajax. Either Ajax is supported, which means that the user (web designer) is allowed to program Ajax in the in the website that is in the visitor client, or Ajax is not supported, which means that the web designer can try, but the developers of the CMS do not explicitly support Ajax. There are also systems for which there are plug-ins already available that use Ajax. An overview of Ajax support in the CMSs visitor client and administration client is listed in Table 1.

| CMS | Ajax in administration client | Ajax in the visitor client |
|---|---|---|
| MODx | Partly | Supported |
| Ajax CMS | Full | No |
| Skeletonz | Full | No |
| NEO-CMS | Partly | Plug-in |
| CMS from scratch | Partly | No |
| Enano | Partly | Plug-in |
| Pligg | - | Plug-in |
| **Ajax Based CMS** | **Full** | **Full + plug-in** |

*Table 1 Ajax support in visitor client (FE) and administration client of the systems listed in the previous section.*
*Partly: means integrated on widget base. Full: means a single-page application. Supported: means that the developers explicitly allow visitor client designers to use Ajax. Plug-in: means that plug-ins for the visitor client using Ajax are available.*

The next chapters of this thesis show that the Ajax Based CMS that is designed in the tutorial differs most from current systems in visitor client. None of the listed CMS visitor clients are full single-page applications, but the one designed in this thesis is. In the current CMSs, Ajax is only used in special pages (plug-ins), but our Ajax Based CMS uses Ajax for every (simple) page. Using the Ajax Based CMS means using Ajax. So, not only on self-made or plug-in base. The administration client that is designed in this thesis is less innovative, because some administration clients are already single-page Ajax applications. However in this thesis, some Ajax patterns are included.

[22] Pligg CMS, http://www.pligg.com

# 4 Ajax Experience: prototype CMS Visitor client

The tutorial (see appendix) is the result of an explorative study on Ajax programming. It shows an application setup with Google Web Toolkit (GWT) and explains a CMS visitor client example. Architectural issues, together with technical challenges are part of the Ajax tutorial.

## 4.1 Visitor client design

The visitor client example is used to explore Ajax's typical design issues. The main characteristic is the asynchronous calls. That means that requests to the server do not receive an answer immediately, like with a normal function return value. Instead, a callback function is called when the response arrives at the client.

**Separation inside the application**
The Object Oriented programming language Java allows GWT-programmers to build highly structured applications. In the tutorial, objects with specific tasks are created. Most important for this thesis is the one object that does the Ajax requests and handles the result. Doing it this way has the advantage that one object 'knows' what requests are pending and what responses are expected. In complex applications, this can avoid that multiple objects fire Ajax requests not knowing that the limit of two pending request was already reached.

The same is done for the Graphical User Interface (GUI). One object manages this interface to the user. It keeps communication with the user simple. It also separates styling from code, because if there is any styling information in the code at all, it could only be in the object that handles the GUI.

**Sequence Diagram**
The tutorial shows that modelling communication with UML Sequence Diagrams clears things up. The UML Sequence diagram is used in this thesis to depict interaction. UML Sequence diagram can be used to depict the messages that have to be exchanged between objects to carry out a certain task [24]. The objects are represented horizontally and their lifeline is shown vertically. Synchronous and asynchronous calls are distinguished by straight or oblique lines and different tips [25].

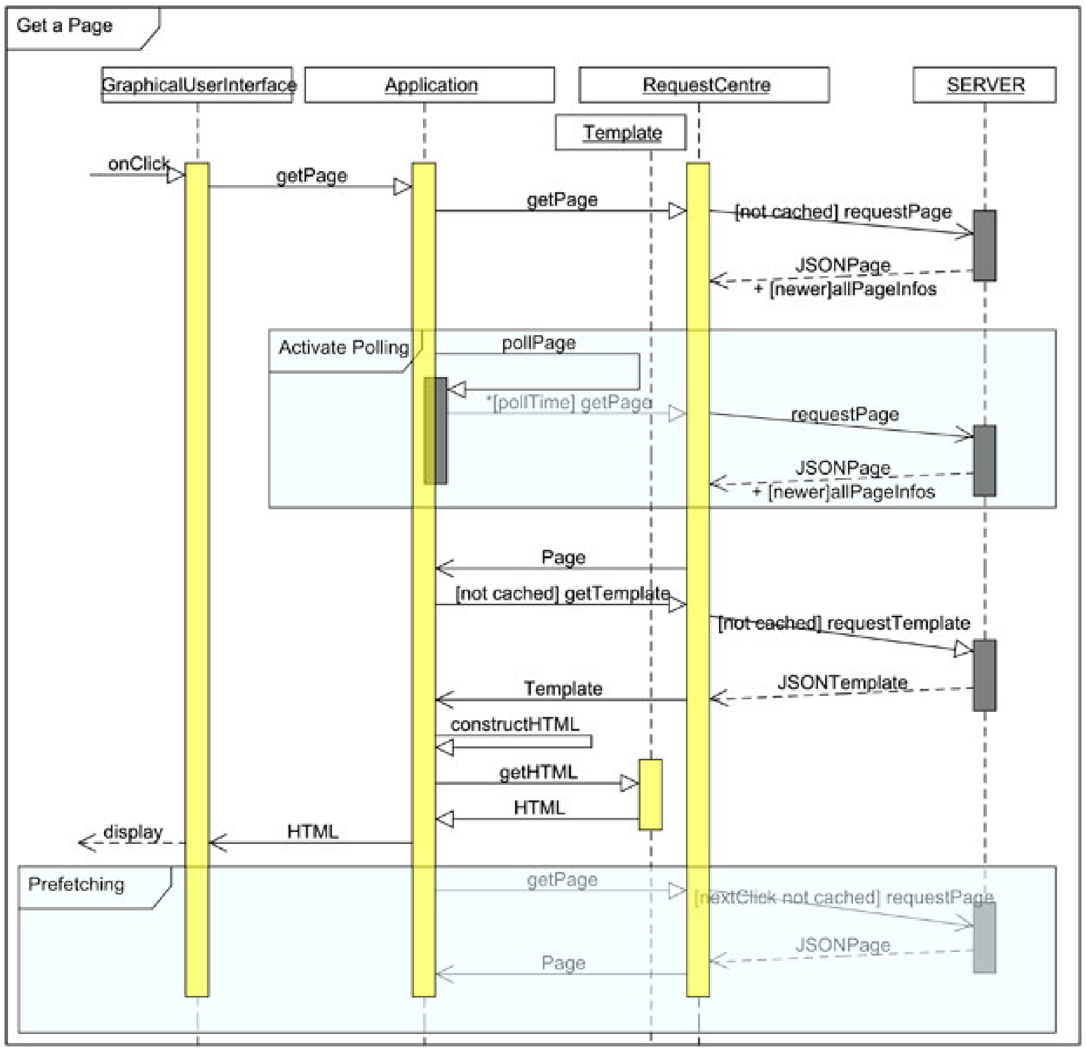


*Figure 4. Sequence diagram of the communication at a page request as implemented in the tutorial's application.*

The sequence diagram that depicts the interaction of objects in the tutorial's application is in Figure 4. It shows the communication that happens with a page request. The initialization of polling and prefetching is included.

The sequence diagram shows that an HTTP request is like a synchronous function call. But should not be treated like that. How many times the webserver is invoked, what to do with errors and the format of parameters and results should be assessed by the software designer. That is what makes it different from synchronous calls.

The lifelines in the sequence diagram show that the objects that are in the Ajax engine are alive through the whole process, while the server comes alive for every request. This is what really happens: every call is treated independently. There are no sessions or other mechanisms to keep a state on server side. This would require extra overload and is unnecessary.

**Callbacks and error handling**
When a server side error occurs, the callback function could be called, just like when no errors occur. It displays (or otherwise handles) the error on client side. But what if the server gets useless because

of a connection failure or fatal error? Therefore the client should know what to do when nothing is received. Resending the request or not is one of the considerations. A timeout mechanism could prevent the client from waiting endlessly on a response that will never arrive.

**Performance**
The design focus of this study is on the client's structure and communication between server and client. The number of HTTP requests and the payload of one request are of importance. Therefore, attention is paid to the balance between them. With Ajax the demands to the client side increase. The (CPU) load and the performance need to have some attention too.

While building the Ajax visitor client these issues popped up for some CMS functionality:

- Front-end templating. The server sends over the template once. For new requests only the page content needs to be send over. This decreases payload, because no layout needs to be send for every page request. It increases load on the client side, because the template needs to be parsed.

- Polling. The client requests the current page repeatedly. This increases the number of requests and increases the load on client side.

- Prefetching. The 'next' page is loaded directly after the current one is received. What page is 'next' should be decided by the server. It can be for instance the page that most visitors request next, based on statistics (predictive fetch). If the user never uses the page, the number of HTTP requests increases unnecessarily. The same holds for the work done on client site.

## 4.2 Possible improvements

The visitor client that was build in the tutorial is far from perfect, but it reveals architectural issues that can be included in CMS design. The experiment showed that although the server and client are closely related, their development can be done separately, as long as there is a standard communication format.

From a server perspective the client is dumb. The visitor client doesn't know what page has been changed on the server or what page is the most visited. Therefore, the server decides on what question to answer and with what data. For instance: the client re-requests a page, but the server knows that nothing has changed. The server then doesn't send the full page to the client, but only a short message ('no recent changes'). Another example: when prefetching the client does not say 'send me page 4', but it does say 'send me the most visited page'. The server knows best what page that is.

Server side caching could decrease the number of database queries. The JSON representation of very static pages could be stored in a file. Instead of connecting to the database for every visitor, the server can simply send the file. The number of requests increases server load and decreases performance. Bundling requests can solve this problem.

The usage of callback functions requires some extra care for errors. The server should let the client know that an error happened. This is not included completely on the tutorial's client. The client should also check what question was answered by the server, because multiple open questions could be waiting for a response. Did the server indeed send the right page? Check the page id.

# 5 Ajax CMS Design

The experience gathered in the tutorial is used in this chapter to design an Ajax Bases CMS, visitor client and administration client. The architecture shows less detail, but is designed with care for Ajax specific constraints.

## 5.1 Constraints

This section shows constraints that come with web programming and Ajax programming. Constraints can be used to induce architectural properties [21].

**Cross-browser**
Websites should allow multi-browser access. Especially the visitor client may expect visitors with various (versions of) browsers. This constraint is less important for the administration client with a group of users you probably know. You can force them to use a specific browser (describe the system requirements).

**Stateless**
A stateless server treats every request as an independent transaction [21]. For an Ajax application statefull transactions could be simulated by sending over IDs and setting cookies. This means extra overhead on communication between client and server. The architecture shows a stateless server, because there is no need for keeping track of states on the visitor client and also not on the administration client. A multi-user administration client should probably use states.

**Synchronity**
Ajax is asynchronous. HTTP requests are not answered immediately, if they are answered at all. That means most functions require a callback function. This holds not only for the Ajax request functions but also for functions that make use of the Ajax functions. The asynchronous nature thus is deeply embedded in the application.

**Cross domain issue**
By default it is not possible to connect an Ajax application to a different server than it was fetched from. Though, there are workarounds available. Those workaround are not needed for the Ajax CMS.

**Client side processing**
Processing at client side improves interactivity and user perceived latency through round-trip reduction [21]. However, it also increases load on the client's webbrowser, which for old browsers may cause delays. The task balance between client and server should therefore be carefully designed.

**Web standards**
All pages should admit to the web standards stated by the World Wide Web Consortium (W3C). For the CMS visitor client it means that the template developer must be allowed to create a template that meets those standards.

**Search Engine Optimization**
Search engine optimization is an important topic for many organizations [1]. This means that the website that is rendered by the visitor client should be readable for search engine robots. Ajax content is dynamically loaded and thus not readable for robots. We ignore this constraint throughout the next paragraph and give priority to client side templating.

## 5.2 CMS Architecture

The CMS architecture describes both client side and server side for the visitor client and the administration client. The Ajax Based CMS makes heavy use of Ajax. The visitor client as well as the administration client is one HTML page, with JavaScripts that we together call the Ajax Engine. The HTML page with Ajax Engine is loaded from server side and initialized at client side once. All subsequent requests will not be fired by the client's browser, but by the Ajax Engine.

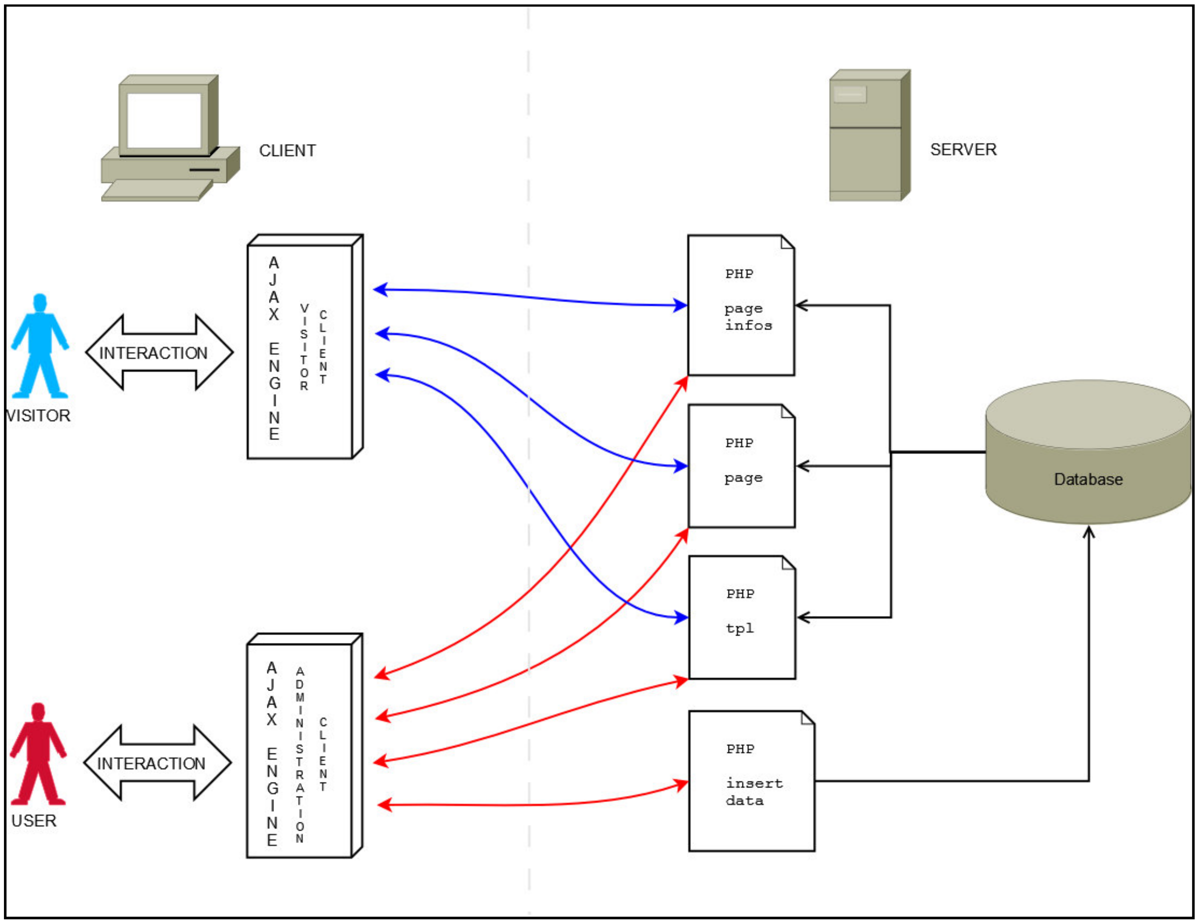


*Figure 5. Ajax Based CMS architecture. The coloured arrows in the centre depict two-way JSON communication.*

The model in Figure 5 shows the system's components and its interactions. In short, the difference between visitor client and administration client is that they require different server side scripts and that the Ajax Engine is fundamentally different.

**Ajax Engine**
The visitor client's engine serves mainly as a template parser. It does some background tricks to prefetch and poll. The administration client's engine however consists of a rich Graphical User Interface (GUI) to not only display, but also modify the information that is received from the server. The visitor client never connects to the 'insert data' script, but the administration client will.

**Server side scripts**
The server side scripts each have a specific task. The 'page' scripts for example receives a page id in the POST variables, queries the database and sends the page content back. Different scripts are used

for different tasks in order to reduce the size of the JSON messages. Now, these messages do not include the type of question ('give me a page'). The number of bytes sent over decreases.

The 'pageinfos', 'page' and 'tpl' scripts are used by both visitor client and administration client. But these scripts do not send the same data to visitor client and administration client. The administration client should for instance receive hidden page, but the front-end should not. All secret information should not be send to the visitor client. It is not secure enough to do so, because the user could intercept the data between server and Ajax engine. Some authentication mechanism should be included to ensure that the scripts are invoked by an Ajax Engine, not by a hacker's browser.

**Client-server interaction**
There are multiple formats for data exchange. Because the CMS is designed to exchange short messages, we want to reduce the overload. We use JSON for this. JSON is native for the Ajax Engine and PHP scripts can also handle it.

## 5.3 Visitor client design

The CMS visitor client is what the visitor of the public websites sees. The word 'public' is crucial, because there are some accessibility and portability issues that determine requirements that must be carefully implemented in the design. This chapter sums up the requirements and further refines the architecture.

**Requirements**
The visitor client should behave like a 'normal' website, enriched with some extra typical Ajax functionality. In order to get displayed as a 'normal' page, the W3C Recommendation should be met.

- The W3C XHTML 1.0 Recommendation is met.

The requirements below are more experimental. They consider specific Ajax features.

- Separation of data. These items are requested separately in order to cache them at front-end: site structure, page content, page template.
- There are no restrictions in the usage of HTML tags.
- For every page polling can be switched on or off and polling time can be set.
- For every page one or more pages can be set that need to be prefetched. These pages can be defined by number, but also by statistics (e.g. 'Prefetch the website's most visited page').

**Architecture**
The architecture of the visitor client is modelled in Figure 5. The visitor client Ajax Engine and everything that connects to it is part of the visitor client. We walk through this from database to end user.

The visitor client requires only read actions from the database. So, the three PHP scripts contain select queries of which the outcome is represented in JSON format. They should also contain some error handling code, because the script may not output undefined text. Because there are only select queries, caching could be implemented as part of the server side scripts. The JSON code of very static pages could for instance be stored as a file, which for subsequent requests saves a (possible complex) query. Whenever visitor client plug-in like a guestbook does require data to be written into the database, attention should be paid to validation.

For every page request all three scripts are used: site structure, page content and template. Though, when one of those elements is already available in the Ajax engine, it is not requested again. Only when the age reaches a limit, a re-request is fired. It is also possible for the server to decide if the client needs new information. The server attaches it to another request. For example, if the server side 'get page' script finds out that the page structure has changed, it sends the page structure together with the requested page. More of these 'smart' actions are performed by the server: in example the client can request 'the most visited page' for prefetching.

The Ajax engine, which is a black box in the model, has the templating mechanism as its main task. Furthermore it just translates link clicks or URL changes into (multiple) requests. It should also requests pages by itself when polling or prefetching is activated.

Users may browse the site by clicking links or changing the URL. Every page sets a history token (after the '#' in the URL) which can be used for direct reference or the use of back and forward browser functionality. Visitor client users (visitors) typically are a diverse audience. They use different technological environments. Where GWT lacks cross-browser compatibility, the programmer should take care of this.

**Ajax Engine (class diagram)**
The class diagram in Figure 6 shows the internal structure of the visitor client Ajax Engine. Inside the engine there is a kind of front-end-back-end construction. The `GraphicalUserInterface` communicates with the user (front) and the `RequestCentre` communicates with the server (back). The application is in between and keeps an instance of both. The rest are helper objects.

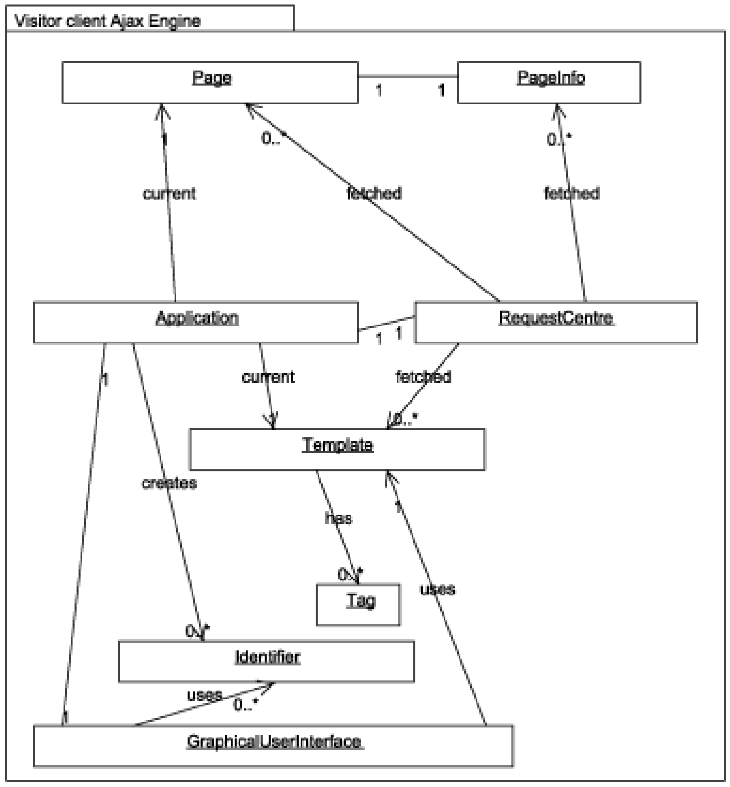


*Figure 6. UML Class Diagram (without details) of the Ajax Engine in the CMS visitor client.*

The application holds one object of type `RequestCentre`. This object does HTTP requests and receives their result. It is the only class that handles JSON and converts it into objects. All others do not operate on this 'low' level. The RequestCentre keeps an internal list of fetched objects. The HTTP requests in the RequestCentre are asynchronous, with callback functions. Therefore, communication between application and RequestCentre also uses callbacks. So, the RequestCentre keeps track of listeners. By centralizing the communication this way the situation is avoided that several object fire requests not knowing from each other, which may overload the server.

The applications also hold a gui object of type `GraphicalUserInterface`. The interface of the visitor client is the website itself. The application passes the HTML from the template and page to the gui, which sets the HTML on a panel and displays it. Some low level functionality handles the header tags. Because the Ajax Engine writes to the body of the HTML page on which it is included, the header tags are set separately. The gui is the first to receive events from gui objects and passes them to whatever object is registered at the gui (probably the application).

The `Page` and `PageInfo` objects carry information about pages. The PageInfo represents all information about a page, which means the information that should be on the variable places in the template. The PageInfo hold more descriptive data, like name, menu name, level in the menu, modification dates. These PageInfos are ordered in a hierarchy – a PageInfo could carry child PageInfos – and together form the site structure. Because PageInfos are separated from the Pages, these small objects can be sent over together at once.

The Template object holds the body HTML, together with some descriptive data. The id property of HTML tags is used to determine what spots should be filled with dynamic content. The Template also provides information about where the menu and submenu (a special content element) should be placed and how it should behave (should the submenu collapse?). The Template keeps some general header tags.

**Interaction (sequence diagram)**
The use of asynchronous calls is the main characteristic of Ajax. This type of function calls affects the full application design. Figure 7 shows the communication between objects in the visitor client that happens when a page is requested. The server is included like a normal Java object, but in fact those calls are calls to a webservice (PHP script).

The diagram in Figure 7 describes the actions that happen when a user requests a page. It starts in the upper left, where the user clicks a link (or invokes an URL). The event starts at the gui, which is told to send the event to the Application. The Application requests the page at the RequestCentre object. The ServerSide object checks if the page is requested before. If that is the case, it can immediately call the callback function and pass the Page. If not, a request to the server is done. After a (hopefully short) while the server returns the page in JSON. The ServerSide object turns the JSON into a Java object and passes it to the Application.

The Application activates polling. A timer is set that fires every x seconds, based on the polltime set for the current Page. When the timer is fired the request process is re-run.

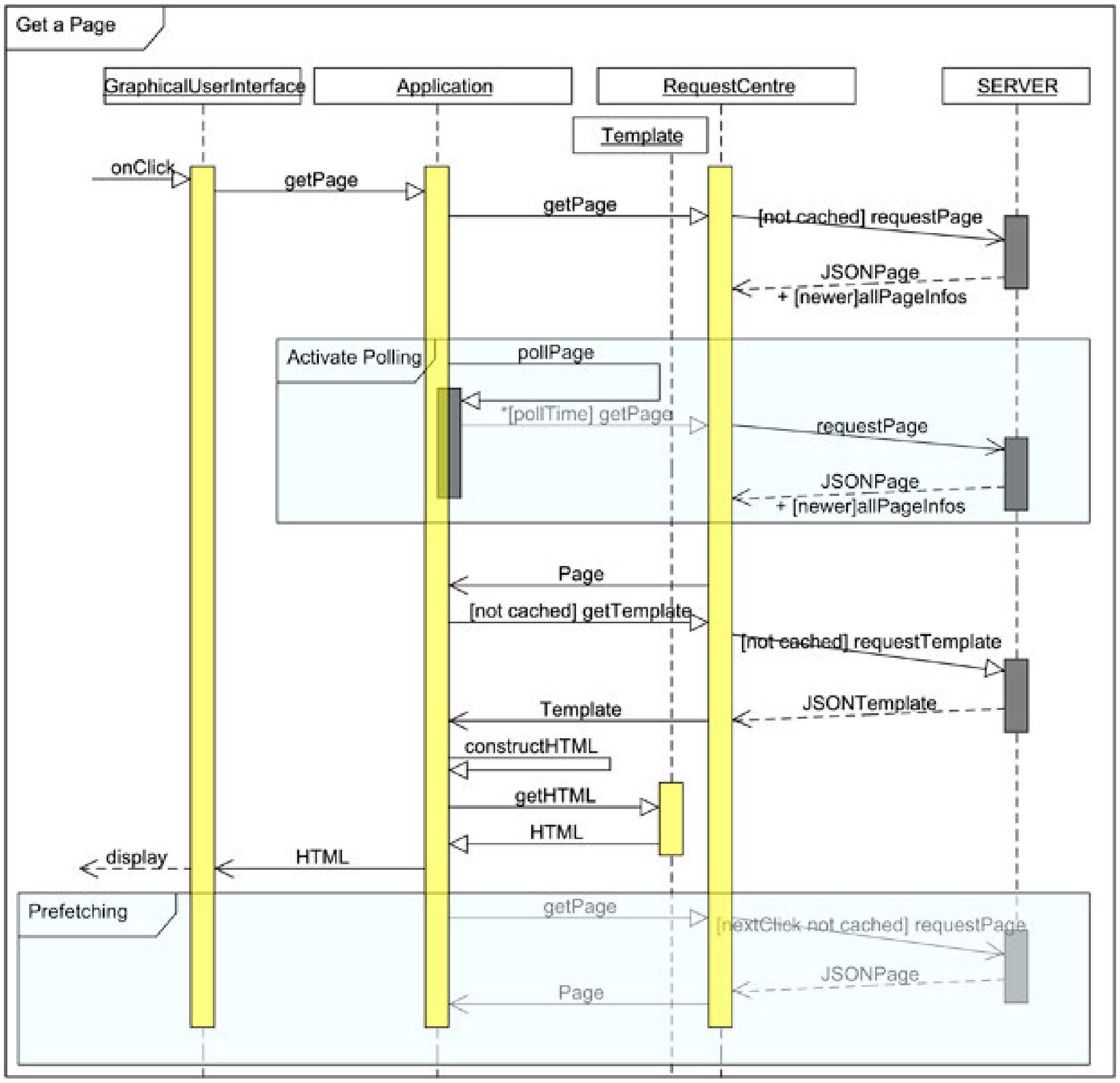


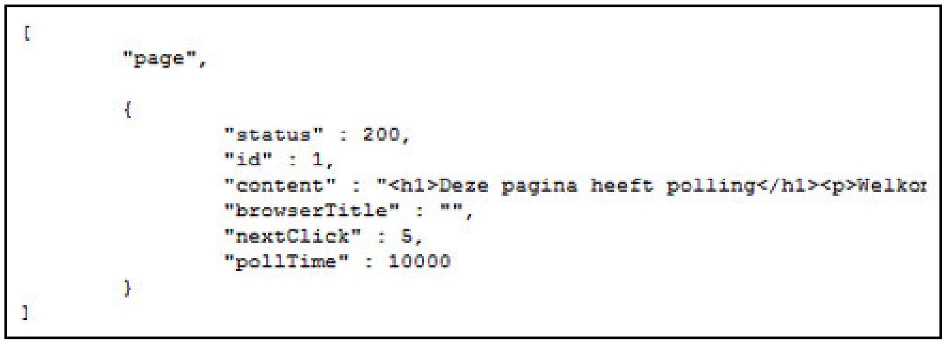
*Figure 7. UML Sequence diagram of the communication at a page request. (Reprint of Figure 4)*

When the Page is received by the Application, it checks what Template it should use (this can be different for every Page). If the template is not the same as the last requested page, it is re-requested. The ServerSide does the same trick as for the Page, but now for the template. If it is not requested before, it requests the template, and delivers it as a Java object to the Application.

With the Page and the Template the applications hold enough information to output the page to the user. Therefore it creates the menu HTML, defines the identifiers and their HTML, and passes that information, together with the Template to the gui. The gui displays the page to the user.

The application's work is not finished when the page is delivered to the user. It activates prefetching (if needed for this page). Prefetching is like a normal page request, except that the actions end at the Application, not at the gui. The application adds the page to a hidden frame, which causes the images on the page to load.

Figure 8 shows the JSON representation of a page. The list of pageinfos and the template are similar. The messages exchanged should stay lightweight. This means: no unnecessary details. For instance, the page request is send to the server side page script, not a general script, so there is no need to include the type of request ('give me a page') into the message.



```
[
        "page",

        {
                "status" : 200,
                "id" : 1,
                "content" : "<h1>Deze pagina heeft polling</h1><p>Welko
                "browserTitle" : "",
                "nextClick" : 5,
                "pollTime" : 10000
        }
]
```

*Figure 8. A page in JSON format. This format is used in the tutorial.*

Other unnecessary information is for instance the information about who changes the page last. This is of interest for the administration client users, but not for the visitor client users. You might also want to hide this for security reasons.

## 5.4 Administration client design

The CMS administration client is what the owner or author of the website uses to update the website. This chapter sums up the requirements and further refines the architecture.

**Requirements**
Because the administration client had a controlled group of users, the accessibility issues are not part of the requirements. When the application does not need to be commercially exploited, the developer only needs to ensure that the application works in the customer's environment. Even when the application is meant to sell to a wider public, certain system requirements can be stated.

- Works in the customer's environment
- Single user
- Adding pages
- Editing pages (title, metadata).
- Editing templates
- Editing page content
- Inserting and uploading images and downloads
- A structure that allows plug-ins
- Pre-submitting
- Intuitive layout that behaves like desktop applications (including typical Ajax interactivity like drag-and-drop)

**Architectural refinements**
The architecture of the administration client is depicted in Figure 5. The administration client shares scripts and the database with the visitor client.

The administration client should display the same information as the visitor client does, supplemented with some extra information. The scripts that are used by the administration client to read from the database are the same as the ones used for the visitor client. In addition, the administration client uses a script to update or delete information in the database. This update script should also validate the input and on failure send an error back. This not likely to happen, because the validation also happens at the front-end. The data interchange format is JSON again.

The Ajax engine at the administration client is powered by GWT widgets that enrich the graphical user interface (gui). Those are: rich text fields, trees to manage the site structure and panes that hold all the widgets in an organized and customizable way. Furthermore, the engine uses two Ajax patterns: submission throttling to bundle small updates (and reduce the number of requests) and pre-submitting to fasten large updates (and reduce user perceived latency).

**Ajax Engine (class diagram)**
The administration client Ajax Engine is similar to that of the visitor client. The objects used are the same, as depicted in

Figure 9, but those objects differ in functionality. The main differences are in this paragraph.

The RequestCentre does not only retrieve data, but also send data. It should turn objects into JSON and add them to a queue. The object-to-JSON transformation should be part of every object that could be send. In example, the RequestCentre could ask Page object for its JSON representation. The RequestCentre sends the object queue when the application wants.

The role of the GraphicalUserInterface changed fundamentally. Instead of just outputting HTML, it should now show an advanced user interface. Therefore, it uses the rich library of widgets provided by GWT. The GraphicalUserInterface does not need the `Identifier` class anymore.

The helper objects, like `Page` and `PageInfo`, are the same as for the visitor client, but enriched with some extra data. For instance the 'last modified' date is now stored in the PageInfo objects. Also, hidden pages are part of the system. The `Page`, `PageInfo` and `Template` object include a `toJSON()` function.

Where the visitor client application's extra task is polling and prefetching, the administration client's extra task is validation of the user's input.

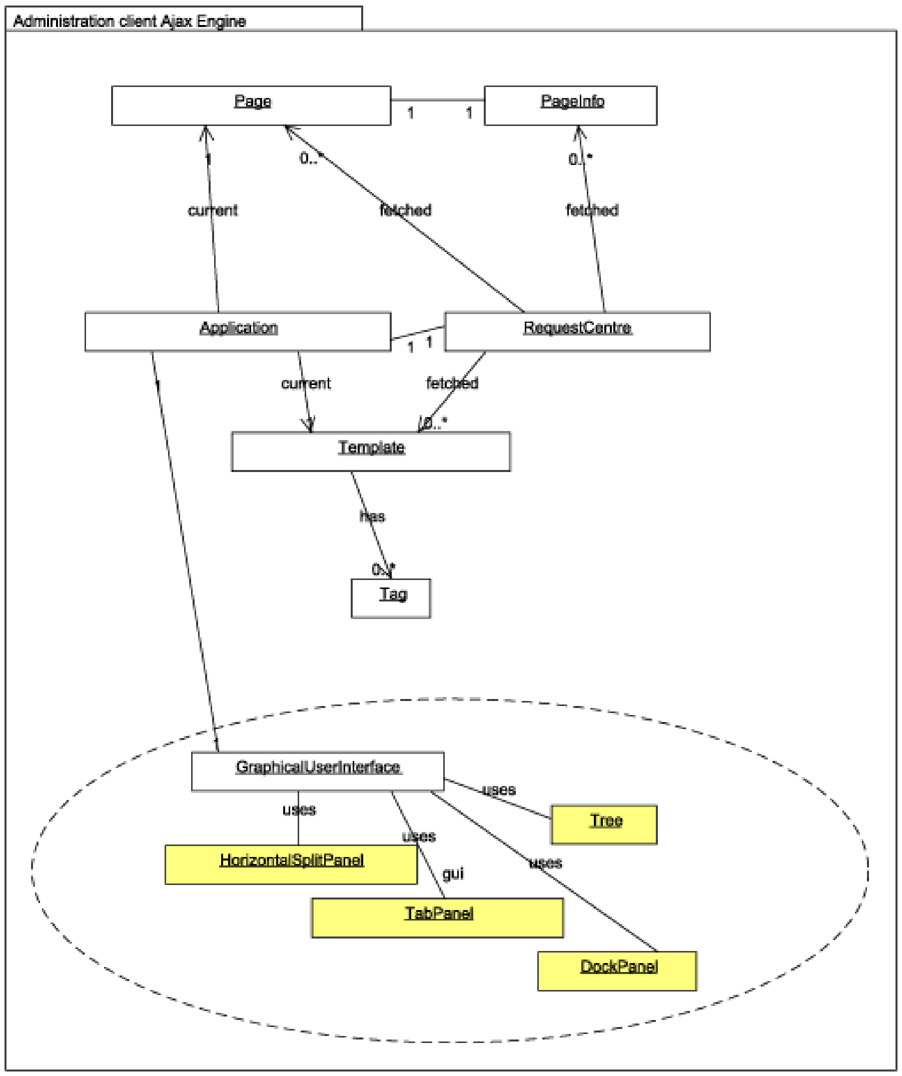


*Figure 9. UML Class diagram of the administration client. The Identifier object that was present in the visitor client is not needed in the administration client. Instead, the administration client's gui uses a lot of GWT widgets. A few of them, which will be used for sure, are included (yellow).*

**Interactions (sequence diagram)**

The sequence diagram in Figure 10 shows the process of a page change being queued (submission throttling). It starts at the gui, where the user clicks a save button. The application object plays a central role and applies the change to the instance of the Page object. Afterwards, the application notifies the RequestCentre object, which queues the page. The RequestCentre requests the page's JSON format, adds some instructions and puts it on a list.

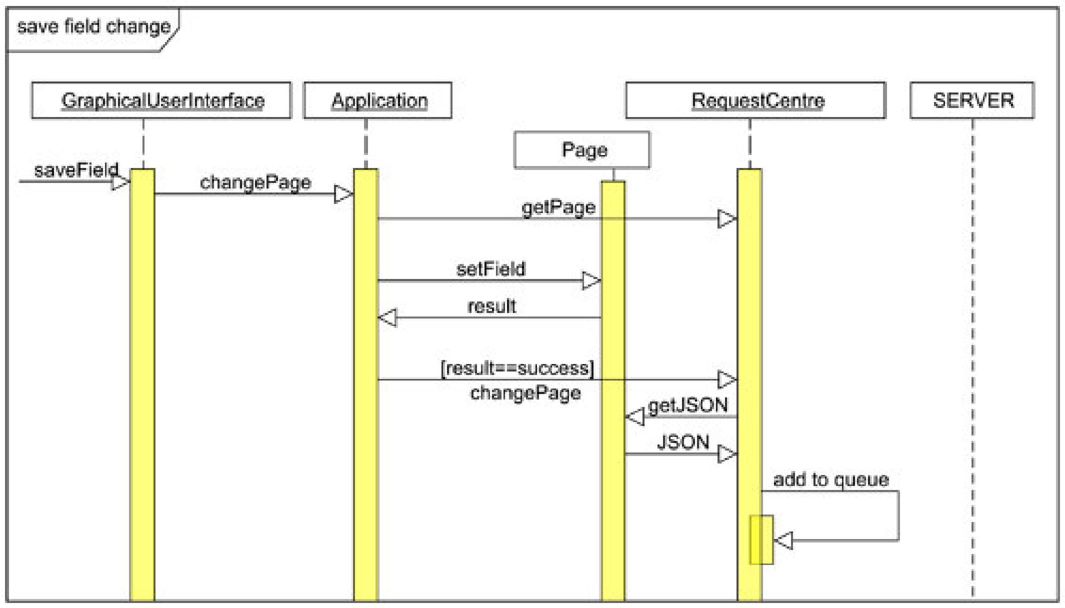


*Figure 10. A save action. As you can see the server is not yet invoked (submission throttling).*

Figure 11 shows what happens when a user decides to save all changes that have been made up until that moment. This process comes down to just sending the queue. Figure 12 shows an example queue in JSON.

Figure 13 shows the process of pre-submitting a large image (or file). Three (sub-) scenarios are sketched: the user decides to upload a different file. Then the previous file should be deleted when the new file is uploaded (this requires a unique id or other type of reference. The second scenario is the user that saves the form. Then the rest of the information (the text fields) is send to the server. This is not part of the submission throttling, because that would really create a huge gap between server and client. The third scenario is that of the user pressing cancel. In that case the uploaded file should be deleted.

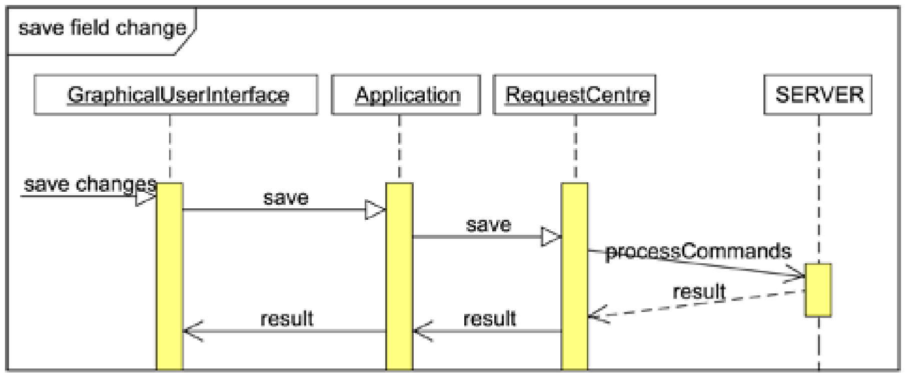


*Figure 11. The save action. Here the action starts at the gui, but it is possible that the Application itself starts this as 'auto-save' action.*

```
[
        { "command" : "addpage",
          "object" :
                {
                        "id" : 10,
                        "name" : "New Page",
                        "menuname" : "New Page",
                        "show" : true,
                        "showinmenu" : true,
                        "content" : "",
                        "location" : [2,3]
                }
        },
        { "command" : "changecontent",
          "object" :
                {
                        "id" : 8,
                        "content" : "New page content...",
                }
        }

]
```

*Figure 12. An example JSON format of the command queue that is sent in Figure 11*

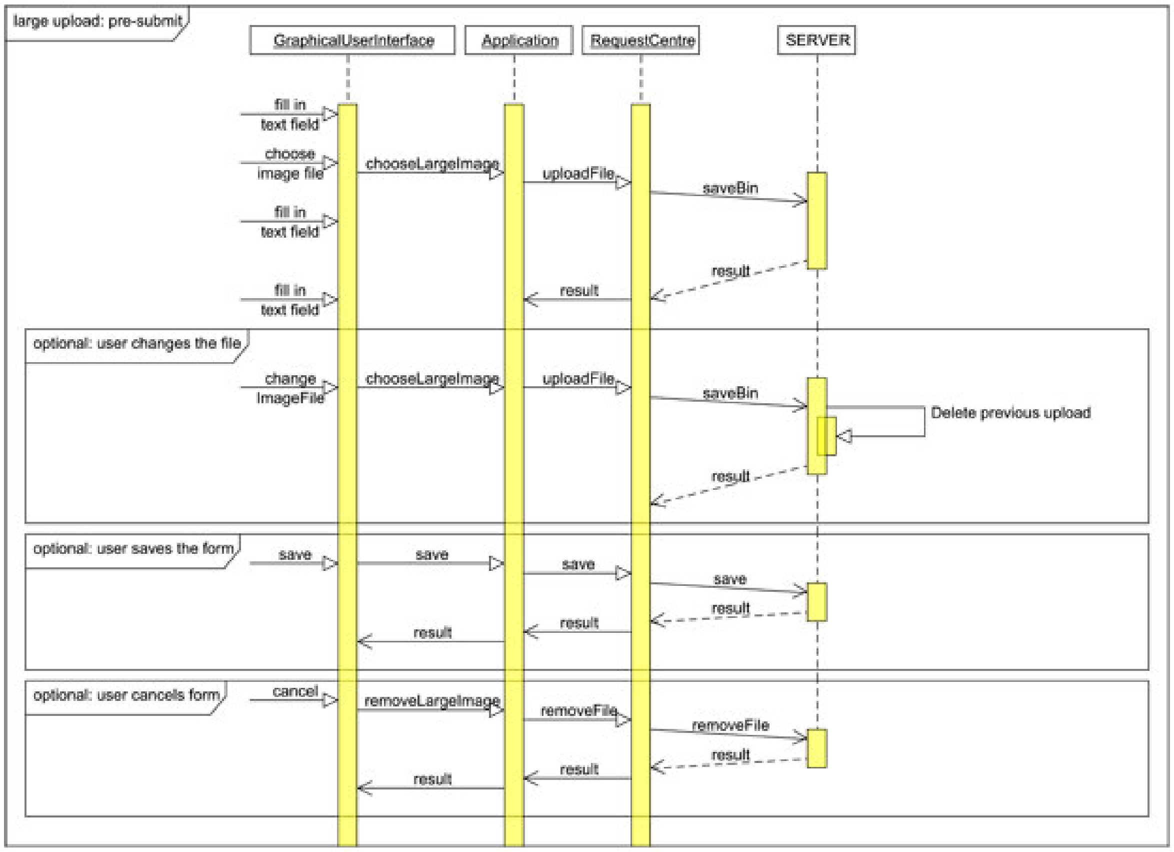


*Figure 13. Presubmitting a large Image.*

# 6 Evaluation

This evaluation is divided in a section devoted to Ajax for content management and a section about general application. The final section is a future vision.

## 6.1 Ajax for content management

The tutorial and the application design showed the strengths and weaknesses of Ajax when used for content management systems. The assumption that Ajax only improves a website is wrong.

**Ajax in the visitor client**
Ajax for the CMS visitor client encounters some key disadvantages. First browser compatibility is an issue. The visitor client has an unknown audience with a wide variety of browsers that all should be supported (as stated in the accessibility guidelines). GWT supports many browsers, but a full Ajax visitor client is a risk. If a visitor's browser does not support a tiny piece of the Ajax Engine, the complete website is inaccessible for that visitor.

The client side Ajax Engine requires more power of the visitor's computer than a traditional website. When a website becomes large - in number of pages and content – it might take too much for the client to handle. With traditional CMSs the power is partly a server side process which the developers are in control of, but also where they pay for.

Another reason not to choose for a full Ajax visitor client with front-end templating is Search Engine Optimization (SEO). Because the website is dynamically built by the visitor client, the search engine's robots might not be able to access it [1]. Therefore the website is not indexed, or wrongly indexed, in search engine. Also, programming the tricks that are needed to include header tags at the visitor client are tedious.

The disadvantages mentioned above make Ajax unsuitable for a complete visitor client to build. The advantages, like polling, and prefetching, could also be used only for parts of a page. For instance poll news headlines only. Or prefetch only large images. In that situation, most of the website is still cross-browser and SEO friendly. This type of including Ajax is currently done with gadgets.

**Ajax in the administration client**
The issues that hold for the visitor client do hold for the administration client, but their impact is lower. Browser compatibility is not needed, because as a developer for a specific public, you know what browser is used or you can impose one. Also, SEO is not needed.

In addition the advantage of building the administration client entirely in Ajax is great. Large uploads can be done with pre-submission and the interface can be enriched. The desktop look and feel that can be created with GWT and Ajax is a great advantage.

## 6.2 General Ajax applications

The differences between administration client and visitor client in the CMS that is designed in this thesis shows in what situations Ajax is useful. Ajax is useful in applications that are used by a controlled group of users. The developer can then at least ensure that the application works at the user's workstation. The rich desktop-like interaction can improve traditional applications.

One should take care with Ajax applications that are meant for a wider public. Browser issues could pop up and search engines do not index the Ajax content. A balance between Ajax content and 'traditional' content could be the solution. Coverage of systems and browsers, SEO and the enrichment would have to be weighed against each other.

The rich graphical interface and interaction model is not only positive. Ajax applications go against the click-and-wait protocol. Especially for the visitor client this means unexpected behaviour of a website. With polling, in example, the website changes without visible reload and with prefetching the request page appears in a split second. Breaking the conventions like this probably takes some time for users (visitors) to get used to.

Building an Ajax application requires expertise in many different fields. This is a result of the fact that Ajax is not a language itself, but is a combination of technologies. Bundling of these technologies does not only affect the implementation of an application, but it is fundamental to the design and the design process. Some characteristics of Ajax have high impact on design decisions.

This study did not only teach us about programming Ajax or about final products, but it did teach about the process of designing and creating an Ajax application. The architecture needs no detailed information about SQL queries, PHP statements or JavaScript issues. The architect should combine requirements specified by dataset experts, PHP programmers, JSON experts, Java/JavaScript programmers and the end users. Every expert programmes his part, but the Ajax expert (the architect) should combine it.

The tutorial and the design of the CMS were created without expert knowledge on either of the languages. Instead basic knowledge about every aspect was present and more advanced knowledge and thinking about typical Ajax issues was included.

## 6.3 Future

This thesis shows that Ajax changes the web interaction model. Websites' potential grew. This section sums up some issues might gain interest in future.

**Ajax vs. Flash**
Ajax competes with Flash/Flex. Their strengths and weaknesses differ. Flash needs a (commercial) plug-in, Ajax does not. Flash has just incorporated some asynchronous communication options[23]. In contrary, Flash offers richer gui possibilities, whereas Ajax relies on HTML and CSS and is thereby limited. Flash offers better and more animation and video options. Both techniques will probably converge, but never be combined. Both can live next to each other.

**Payment model**
Ajax has impact on the payment model that is used for the web. This can be explained by the example of a traditional news website. The owner might earn his money from sponsors that place a banner next to the headlines. Every time a user visits the headline page, the hit counter increases, and the sponsor pays one cent more to the news website owner. More visitors means more money. But also returning visitors means more money. Even people that refresh the headlines, which is not unlikely, earn money for the website's owner. What happens when the website is 'improved' with Ajax?

[23] As stated in Adobe's Flash Player Feature List on http://www.adobe.com/uk/products/flashplayer/productinfo/features/#model

The traditional website turns into a 'web 2.0' website, with build-in Ajax requests to refresh the headlines every two minutes. The headlines change separately from other content. Now, the news junkie opens the website early in the morning and keeps it opened all day without ever hitting the refresh button. There is no need for it! This visitor now earns only 1 cent a day, whereas with the old website he earned a lot more cash. When websites (partly) behave like applications, what does it mean for the payment model?

**Search engine optimization**
Search engines index pages, but an Ajax application like the CMS visitor client build in the tutorial is a single page. Therefore it will not be or badly be indexed. The same issue happens with Flash applications.

Preferable is a situation in which search engines are improved in order to handle Ajax better and Ajax frameworks (like GWT) are improved in order to facilitate search engines. In addition, web designers should always care for SEO when they develop a new website.

In the paragraphs above the assumption is made that SEO is always preferred, but there are situations in which SEO is unwanted. For instance the data inside an internet banking application should not be indexed.

**Mobile applications**
Ajax is a combination of technologies that are (W3C) standard and it can be programmed to be lightweight. It does not require a plug-in. These characteristics make Ajax suitable for mobile application. Most of them already support JavaScript, XHTML and CSS; the main technologies of Ajax are covered.

# 7 Conclusions

The term Ajax was invented in 2005 for Asynchronous Javascript And Xml. Its use developed rapidly, from small scripts to libraries, from small web enhancements to powerful desktop-like web applications. Content Management Systems existed before Ajax did. CMSs – specifically Web CMSs - are web applications to maintain websites. Some of them incorporated Ajax in the administration client, either to improve parts of the system or to create a full single page – desktop like – application. In the visitor client, Ajax is supported by some systems. However, some nice Ajax patterns remain unused. This thesis discussed what Ajax could add to CMSs.

Ajax is a suitable technique to use in a CMS visitor client. Though, it should be implemented with care. Ajax developers are limited by the fact that running the Ajax engine requires a certain performance level of the visitor's computer. Also, it requires the user's browser to support Ajax functionality and –the other way round - the application should support multiple browsers. Search engine access is an important visitor client topic. Dynamic content might not be readable for search robots, which makes SEO harder. These limitations should be weighed against the advantages: prefetching and polling functionality. Ajax on widget base, with only parts of the page using Ajax, is beneficial solution. The tutorial's single-page visitor client with templating mechanism is no good when it comes to browser compliance and SEO.

The administration client has different constraints and therefore Ajax is more suitable. The user knows what systems he or she uses, so the Ajax developer does not need to support every possible browser. The same holds for performance; the user (or customer) decides what system to use and should make sure that his work environment meets the system requirements. The rich desktop like interaction that is possible with Ajax is a great advantage for administration client use.

These conclusions show that the current CMSs that use a full Ajax administration client and an Ajax plug-in based visitor client are best of both worlds. That does not mean that the current CMSs satisfy: prefetching or polling (parts of a page) pattern are not included.

Attention should be paid not only to the technical impact of Ajax (for developers), but also to the use of the applications. Users or website visitors are not used to the desktop-like way of interaction. They might for instance ignore drag-and-drop functionality just because they don't know that it is now possible to drag and drop in a web application. It may take some time for them to get used to it. Also, the payment model – in which a website owner gets paid for every refresh – changes, because Ajax applications need no refreshes.

For developers the downside of Ajax is that the combination of technologies requires experts in every field. For this study, which was performed by the author who could not be expert in every topic, it is a downside, but for enterprise applications it might not be. When multiple developers are involved the separation of expertise could be useful. Another conclusion from a developer's perspective is that this study showed that frameworks like GWT take away lots of the cross browser issues. This allows developers to focus on higher level issues: Ajax patterns, like polling.

# 8 Epilogue

Building the original project's deliverables, a (prototype) visitor client and administration client, turned out to be not feasible in the time given. Programming the visitor client took me so much time that I, together with my supervisor, decided to finish that visitor client prototype and only design a complete system on paper. In my opinion programming the administration client would only add to my programming experience, but not so much to my understanding of Ajax and asynchronous communication. The latter, understanding Ajax, was most interesting to me.

The things I wrote about Ajax do not only hold for CMSs. Many statements are generic and can be applied to other complex systems as well. It is true that some (or: most) improvements could be beneficial for other systems too, and my focus might have been too much on Ajax and CMSs. My co-supervisor made me realize afterwards that I did not write much about CMSs. I believe I should have made more explicit what is generic and what I not.

I'd like to thank dr. Željko Obrenović, who acted as a co-supervisor and was very responsive with useful comments. He made me include the exact definition of a CMS, which existed only in my mind and was missing in previous versions of the thesis. Also, I did not use the words front-end and back-end for the right purpose and changed it to visitor client and administration client. In addition, he made me realize that my focus was too much on Ajax, but I should have connected this more to CMS systems. Finally, dr. Obrenović corrected me in my claim that current CMSs did not use Ajax. I did some research before, but did not feel the need to include that in my thesis. Afterwards, I did some extra research, with his help, and mentioned and compared current systems in the thesis. This resulted in an extra chapter about current CMSs.

Finally, many thanks go to my supervisor prof.dr. Anton Eliëns. He supported me during the whole project, lent me books and gave useful feedback. He made me pay more attention to the CMS visitor client. I really appreciate his help.

This study combined knowledge and experience that I acquired throughout the Computer Science courses: XHTML/CSS, Java, Javascript, XML/JSON, server side languages, architectural issues and communication patterns, usability… It gave me insight into the process of building a larger application. By doing this on my own I missed the teamwork that could have improved the result, but on the other hand it forced me to dive into every aspect of Ajax development. This broad expansion of my knowledge means more to me than knowing every detail of a programming language.

# 9 References

[1] **Johnson, Dave, White, Alexei and Charland, Andre.** *Enterprise AJAX, Strategies for Building High Performance Web Applications.* Boston : Pearson Education, 2008.

[2] **W3C [World Wide Web Consortium].** XHTML™ 1.0 The Extensible HyperText Markup Language (Second Edition). [Online] 2002. [Cited: September 24, 2008.] http://www.w3.org/TR/xhtml1.

[3] —. Cascading Style Sheets Level 2 Revision 1 (CSS 2.1) Specification. [Online] 2007. [Cited: October 29, 2008.] http://www.w3.org/TR/CSS21/.

[4] **Zakas, Nicholas C, Jeremy, McPeak and Joe, Fawcett.** *Professional Ajax.* 2nd edition. Indianapolis : Wiley Publishing, Inc, 2007.

[5] **Flanagan, David.** *JavaScript: The Definitive Guide.* Sebastopol, CA : O'Reilly Media, Inc, 2006. 5th edition.

[6] **W3C [World Wide Web Consortium].** HTML 4.0 changes. [Online] 1998. [Cited: September 24, 2008.] http://www.w3.org/TR/1998/REC-html40-19980424/appendix/changes.html.

[7] **Crane, David, Bibeault, Bear and Sonneveld, Jord.** *Ajax in Practice.* s.l. : Manning Publications Co., 2007.

[8] **W3C [World Wide Web Consortium].** Document Object Model (DOM). [Online] 2005. [Cited: September 26, 2008.] http://www.w3.org/DOM/.

[9] **Mozilla.** XMLHttpRequest. *Mozilla developer center.* [Online] 2008. [Cited: September 26, 2008.] http://developer.mozilla.org/en/XMLHttpRequest.

[10] **W3C [World Wide Web Consortium].** W3C Technical Reports and Publications. [Online] [Cited: September 2008, 26.] http://www.w3.org/TR/#last-call.

[11] —. The XMLHttpRequest Object. [Online] 2007. [Cited: September 26, 2008.] http://www.w3.org/TR/2007/WD-XMLHttpRequest-20071026/#xmlhttprequest.

[12] —. Extensible Markup Language (XML) 1.0 (Second Edition). [Online] 2000. [Cited: October 08, 2008.] http://www.w3.org/TR/2000/REC-xml-20001006#sec-origin-goals.

[13] **Garrett, Jesse James.** Ajax: A New Approach to Web Applications. [Online] 2005. [Cited: October 01, 2008.] http://www.adaptivepath.com/ideas/essays/archives/000385.php.

[14] **Mahemoff, Michael.** *Ajax Design Patterns.* s.l. : O'Reilly Media, Inc., 2006.

[15] **O'Reilly, Tim.** What Is Web 2.0. [Online] 2005. [Cited: October 01, 2008.] http://www.oreillynet.com/pub/a/oreilly/tim/news/2005/09/30/what-is-web-20.html.

[16] **Best, David.** *Web 2.0: Next Big Thing or Next Big Internet Bubble?* Eindhoven : Technische Universiteit Eindhoven, 01 11, 2006.

[17] **Noda, Tom and Helwig, Shawn.** *Rich Internet Applications: Technical Comparison and Case Studies of AJAX, Flash and Java based RIA.* Wisconsin : UW Business Consortium, UW E-Business Consortium, 2005.

[18] **W3C [World Wide Web Consortium].** Introduction to SGML. *On SGML and HTML.* [Online] 1999. [Cited: October 15, 2008.] http://www.w3.org/TR/REC-html40/intro/sgmltut.html#h-3.1.

[19] **JSON.org.** JSON The x in Ajax. *Introducing JSON.* [Online] 2006. [Cited: October 08, 2008.] http://www.json.org/json.pdf.

[20] **Sam Stephenson and the Prototype Team.** Prototype 1.6: The Complete API Reference. *Prototype JavaScript framework.* [Online] 2008. [Cited: October 24, 2008.] http://globalmoxie.com/bm~doc/prototype-160-api.pdf.

[21] *C Component- and Push-based Architectural Style for Ajax Applications.* **Mesbah, Ali and Van Deursen, Arie.** Delft : Elsevier, 2008.

[22] **Google.** Google Web Toolkit. *Google Code.* [Online] Google. [Cited: October 29, 2008.] http://code.google.com/webtoolkit/.

[23] **Johnson, Bruce and Webber, Joel.** Google Developer Day US - Fast, Easy, Beautiful: GWT. *YouTube.* [Online] 2007. [Cited: November 5, 2008.] http://www.youtube.com/watch?v=NvRa-CxkpZI&feature=related.

[24] **Vliet, van, Hans.** *Software Engineering, Principles and Practice.* 2. Chichester : John Wiley & sons, LTD, 2000.

[25] **Bell, Donald.** UML's Sequence Diagram. *IBM.* [Online] IBM, 16 02 2004. [Cited: 18 12 2008.] http://www.ibm.com/developerworks/rational/library/3101.html.

[26] —. UML Basics: The class diagram. *IBM.* [Online] IBM, 15 09 2004. [Cited: 04 12 2008.] http://www.ibm.com/developerworks/rational/library/content/RationalEdge/sep04/bell/.

# 10 Appendix: Tutorial

## 10.1 Introduction

In this tutorial a CMS visitor client is build that heavily uses Ajax technologies. The first setup is discussed using separate examples. The visitor client is not build in this tutorial, but walked through step-by-step.

For this tutorial the Google Web Toolkit (GWT) is used. This toolkit allows programming Ajax in Java and compiling it to JavaScript afterwards. It takes away the nasty cross-browser issues that come with JavaScript programming. GWT includes packages that handle Ajax requests, JSON and gui elements.

GWT hides the low level Ajax issues. This allows developers to focus on a more architectural level. Google has put some videos of GWT presentations online that promote GWT[24].

**Tutorial structure**
The first part of the tutorial (section10.2 and 10.3) is a step by step guide to **build** your first Ajax application. The second part (section10.4) serves as a guide to **understand** the much more complex CMS visitor client application.

**Preknowledge**
GWT should not be your first Ajax programming experience. A glance under the hood, some basic steps in Ajax, helps getting a feeling with the nature of Ajax. Please consult one of the many books and articles available for this. This tutorial requires basic knowledge about Object Oriented and Scripting languages. Basic Java experience is required; PHP and MySQL experience is favoured.

## 10.2 Project setup

GWT compiles Java to JavaScript, but the can even be run without compiling in hosted mode. GWT integrates easily with Eclipse. We will use that environment for this tutorial.

Ensure that you have installed the Java SDK[25]. For this tutorial we download and install Eclipse Classic from www.eclipse.org[26]. Download the most recent GWT release from Google Code[27]. Extract the GWT package to a directory of your choice.

**Create a project**
Run the commands below to create a project. Working on the Windows platform you can create two '.bat' files with these lines:

```
projectCreator -eclipse AppName -out "D:\path\eclipse workspace\AppName"

applicationCreator -eclipse MasterFE -out "D:\path\eclipse workspace\AppName"
  com.package.client.AppName
```

[24] References are in the final section of this tutorial.
[25] http://java.sun.com/j2se/1.4.2/download.html
[26] http://www.eclipse.org/downloads/
[27] http://code.google.com/intl/nl/webtoolkit/

If the GWT creator applications are not in your system variables, you should run the bat-file from the directory in which these applications are placed (where GWT is extracted to). Adding these lines to one .bat file doesn't work.

**Add the project to Eclipse**
Eclipse should be able to access the GWT library. Click `File > Properties`, select `Java Build Path`, on the tab `Libraries` ensure that `gwt-user.jar` is listed. If not, select it by clicking `Add External JARs….`

In Eclipse choose `File > Import…` and select the import source `Existing Projects into Workspace`. Select the newly created project directory and click `finish`.

**Running the example**
Run in hosted mode by clicking the `Run` button (play icon) in the toolbar or by invoking `Testproject-shell.cmd`. You should now see the 'Hello World!' example in a hosted browser.

Compile the application by invoking `Testproject-compile.cmd` (from within Eclipse). You can now run the application which is in the `www` directory in the Eclipse workspace.

Try to change the 'Hello World!' application step by step and discover the (non-Ajax) functionalities. Consult IBM's webpage about setting up GWT[28] to learn basic GWT programming. The installation steps are also covered on this webpage.

## 10.3 Doing Ajax

Basic Java knowledge together with the GWT API (see section 10.5) can help you learn programming Ajax. This section of the tutorial focuses on communication.

**Ajax Setup**
Before we can use HTTP Requests and JSON we should add two lines to the setup XML file. Browse with Eclipse in the project 'src' directory to the package and open the file `AppName.xml`. Add these lines between the module tags:

```
<inherits name="com.google.gwt.http.HTTP"/>
<inherits name="com.google.gwt.json.JSON"/>
```

**Setup a request**
We create a separate class to show some simple Ajax communication functionality. `Click File > New > Class`. Type in `AjaxExamle` in the name field and add `RequestCallback` (`com.google.http.client.RequestCallback`) to the (empty) list of Interfaces.

The new class implements `RequestCallback` and thus has at least these two functions already: `onError` and `onResponseReceived`. These are the callback functions. We will fill those with code later in this tutorial. Remove the `@Override` line above those functions. GWT does not support it.

In the new `AjaxExample` class create a function `ajaxRequest()`. Import classes from the GWT package when needed.

[28] Build an Ajax application using Google Web Toolkit, Apache Derby, and Eclipse, Part 1: The fancy front end http://www.ibm.com/developerworks/library/os-ad-gwt1/

```
public void ajaxRequest() {

       StringBuffer postData = new StringBuffer();

       postData.append(URL.encode("id")).append("=").append(URL.encode("5"));


       RequestBuilder builder = new RequestBuilder(RequestBuilder.POST
                                                                   "file.txt");

       builder.setHeader("Content-type", "application/x-www-form-urlencoded");

       try {

              Request response = builder.sendRequest(postData.toString(), this);

       } catch (RequestException e) {

              throw new Error("geen request");

       }

}
```

The code shows a `POST` request with data (`id = 5`). We send this request to a local file, so there is no need for `POST` variables. You can leave them out and even use a `GET` request. We include them as an example, because in more advanced request you probably want to post some variables.

The `RequestBuilder` does a local request, so we should create a file that is requested. Choose `File > New > Untitled Text File`. Type "Hello World!" into the file and save it in the project's `src/com/package/public` folder.

**Handling the result**
An Ajax requests needs a callback function that handles the result. In our `AjaxExample.java` we keep it simple and just output the text that is received:

```
public void onResponseReceived(Request request, Response response) {

       RootPanel.get("slot1").add(new Label(response.getText()));

}
```

In your applications you would probably pass the result back to a callback function that is set when `ajaxRequest()` was called. Or maybe your object parses the result first and then passes it back. Outputting directly to the RootPanel like we did now is bad practice, but useful for learning purposes.

**Test drive**
Remove the original "Hello World!" functionality in the main java file `AppName.java by` deleting the code that is in the `onModuleLoad()` function. Now add these lines:

```
AjaxExample example = new AjaxExample();

example.ajaxRequest();
```

You create an object of the type class you have just written and you invoke the function that does the Ajax request. Run this example in hosted mode. The output will be the text that you wrote in `file.txt`.

**Parsing JSON**
In the previous example we sent and outputted a simple text file. Though, for more advanced applications we want to send over complete objects. Then we need a communication format that is supported by server and client, like JSON. The advantage above a self-made format is that JSON parsers are available out-of-the-box. GWT also includes JSON functionality.

We use the JSONParser together with JSONValue, JSONArray and JSONObject. Furthermore JSONBoolean, JSONNumber and JSONString represent the values that for most programming languages are native. First the response, which is now JSON (instead of "Hello World!") is parsed. It results in a JSONValue:

```
JSONValue jsonValue = JSONParser.parse(response.getText());
```

JSONValue is the superclass of all JSON value types[29]. It has some functions to check what JSON value it is: isArray(), isBoolean(), isNumber(), isObject() and isString(). When you design you application, you probably create a data model. Based on that model, you might expect a type of value. You might expect for instance an array of objects, in which every object contains a string and a number, like the JSON shopping list below.

```
[
  { "article" : "Apple",
    "amount" : 2 },
  { "article" : "Egg",
    "amount" : 6 },
  { "article" : "Banana",
    "amount" : 5 }
]
```

We can rewrite the callback function from the "Hello World!" example such that it displays a shopping list. It should place every item as many times on the list as the amount says. So, the word "apple" is displayed twice. We then need the `amount` value to be read as a number, and not just invoke the toString() function that every Java object has. The callback function below decomposes the JSON string.

```
public void onResponseReceived(Request request, Response response) {

       HTML result = new HTML("Shopping list<br />");

       JSONValue jsonValue = JSONParser.parse(response.getText());
       JSONArray jsonArray;
       JSONObject currentObject;
       JSONString article;
       JSONNumber amount;

       if ((jsonArray = jsonValue.isArray()) != null) {
```

[29] See the GWT JSON API, http://google-web-toolkit.googlecode.com/svn/javadoc/1.5/com/google/gwt/json/client/package-summary.html

```
            int i = 0;

            while (jsonArray.get(i) != null) {

                  currentObject = (jsonArray.get(i)).isObject();

                  if (currentObject != null) {

                        article = (currentObject.get("article")).isString();

                        amount = (currentObject.get("amount")).isNumber();

                        if (article != null && amount != null) {

                              for( int j = 0; j < amount.getValue(); j++)

                              result.setHTML(result.getHTML() +
                                           article.stringValue() + "<br/>");

                        }

                  }

                  i++;

            }

      }

      RootPanel.get("slot1").add(result);

}
```

In the example you see that the JSONValue is 'asked' if it is really the expected type, like (jsonArray.get(i)).isObject(), and then the objects functionality is used. We do this check, because we are not really sure that the object is really a JSONArray or some other JSON value type. We could have made a typo in the JSON file for instance. If we would skip the test, runtime errors appear. These checks are even more useful in applications where you don't know what JSON format to expect.

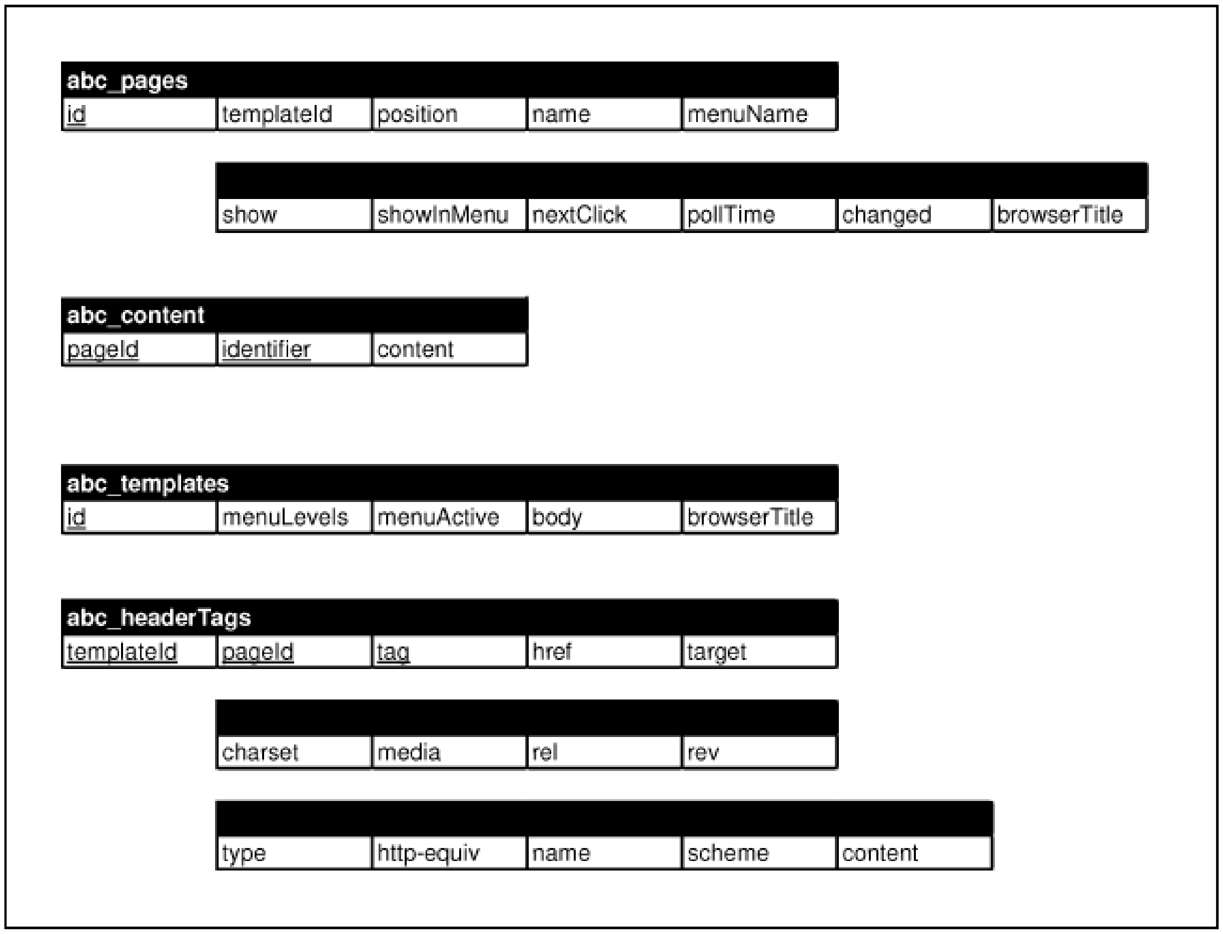
**Server side**
There was no server in our example. Everything was localhost on your computer. The file `file.txt` that we accessed with the Ajax request was static. In real applications the file that you will access is probably a server side script that dynamically generates output. For instance a PHP script that reads from a MySQL database and outputs the result in JSON format. Therefore, POST data was already included in the example.

Why didn't we connect to a 'real server'? Because of the cross domain issue. Ajax does only allow connections to a server from which the Ajax Engine itself was fetched. So, if we want to access server side script, we should put the Ajax application on the server too. But then we can't use the hosted mode (which can be useful for debug purposes) of GWT. Another solution would be installing PHP and a MySQL database on localhost.

## 10.4 CMS visitor client

This chapter studies the example Fronted CMS created for experimental purpose. The working example can be reached from http://www.few.vu.nl/~teunis/master-project. There is also a link to the code that is discussed in this chapter.

**Server side: database layout and PHP scripts**
The server side uses PHP and a MySQL database to store and retrieve data. The example data is inserted in the database with a Database Management System.



| abc_pages | | | | | | |
|---|---|---|---|---|---|---|
| id | templateId | position | name | menuName | | |
| | show | showInMenu | nextClick | pollTime | changed | browserTitle |

| abc_content | | |
|---|---|---|
| pageId | identifier | content |

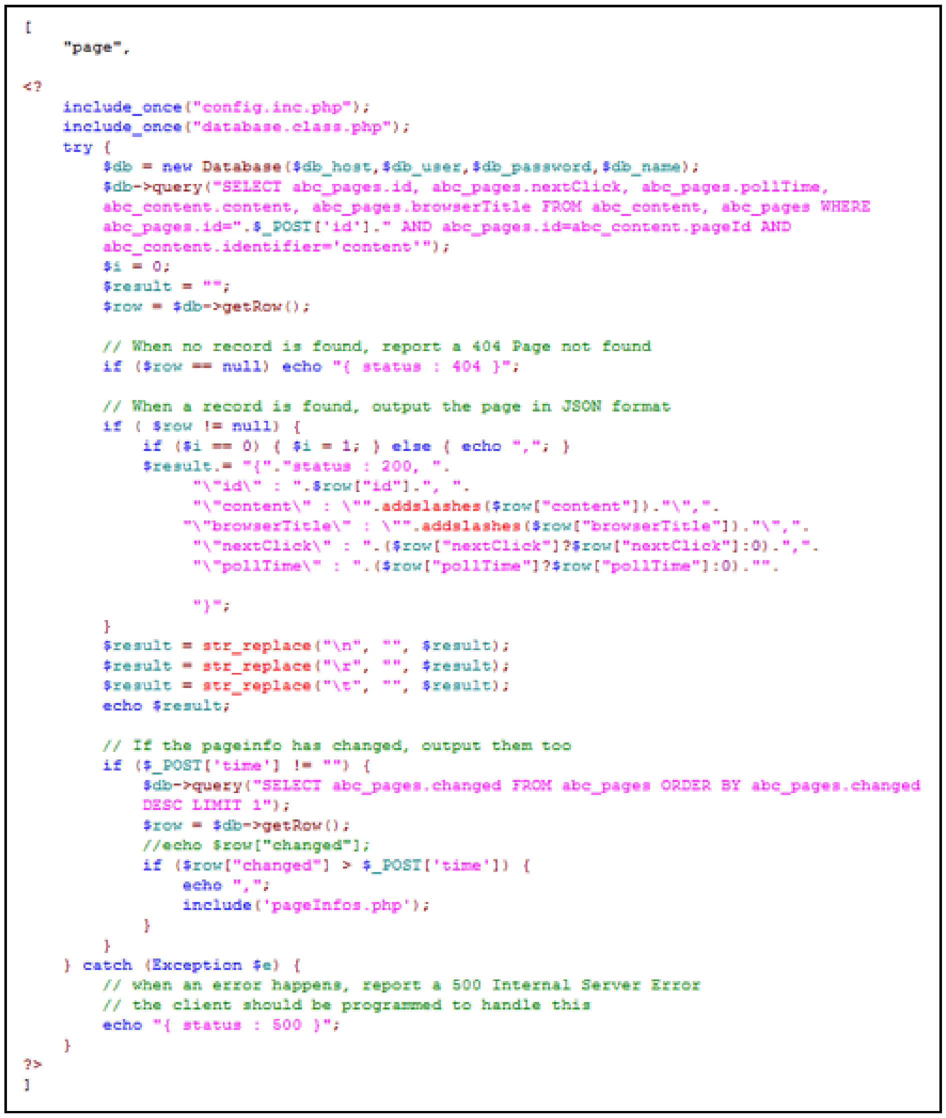
| abc_templates | | | | |
|---|---|---|---|---|
| id | menuLevels | menuActive | body | browserTitle |

| abc_headerTags | | | | | |
|---|---|---|---|---|---|
| templateId | pageId | tag | href | target | |
| | charset | media | rel | rev | |
| | type | http-equiv | name | scheme | content |

*Figure 14. The database layout.*

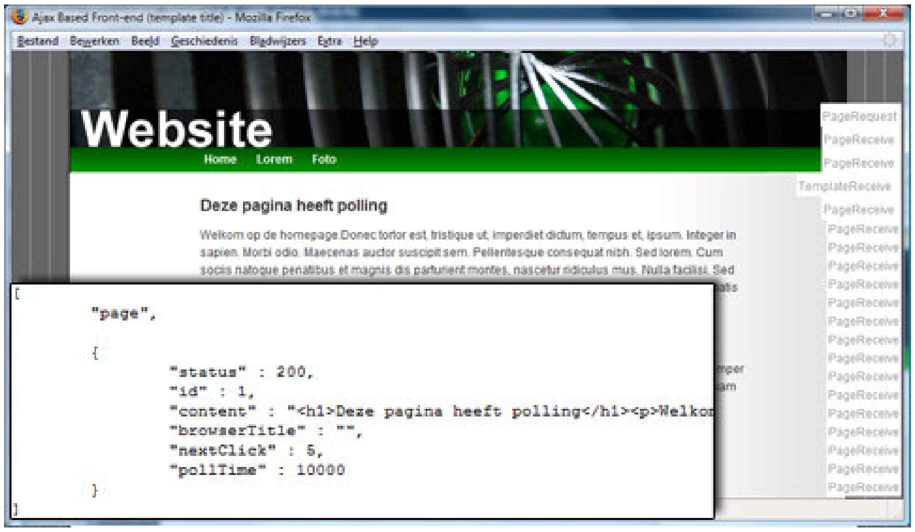
Figure 14 shows the database layout. In the eyes of a database expert this might not be a satisfying solution, but it is at least a working solution. Because database design is a specialist's topic, we do not discuss this any further.

Figure 15 shows the PHP script that retrieves page data and outputs it in JSON format. The example JSON and HTML output is in Figure 16.



```
[
    "page",

<?
    include_once("config.inc.php");
    include_once("database.class.php");
    try {
        $db = new Database($db_host,$db_user,$db_password,$db_name);
        $db->query("SELECT abc_pages.id, abc_pages.nextClick, abc_pages.pollTime,
        abc_content.content, abc_pages.browserTitle FROM abc_content, abc_pages WHERE
        abc_pages.id=".$_POST['id']." AND abc_pages.id=abc_content.pageId AND
        abc_content.identifier='content'");
        $i = 0;
        $result = "";
        $row = $db->getRow();

        // When no record is found, report a 404 Page not found
        if ($row == null) echo "{ status : 404 }";

        // When a record is found, output the page in JSON format
        if ( $row != null) {
            if ($i == 0) { $i = 1; } else { echo ","; }
            $result.= "{"."status : 200, ".
                 "\"id\" : ".$row["id"].", ".
                 "\"content\" : \"".addslashes($row["content"])."\",".
                "\"browserTitle\" : \"".addslashes($row["browserTitle"])."\",".
                 "\"nextClick\" : ".($row["nextClick"]?$row["nextClick"]:0).",".
                 "\"pollTime\" : ".($row["pollTime"]?$row["pollTime"]:0)."".

                 "}";
        }
        $result = str_replace("\n", "", $result);
        $result = str_replace("\r", "", $result);
        $result = str_replace("\t", "", $result);
        echo $result;

        // If the pageinfo has changed, output them too
        if ($_POST['time'] != "") {
            $db->query("SELECT abc_pages.changed FROM abc_pages ORDER BY abc_pages.changed
            DESC LIMIT 1");
            $row = $db->getRow();
            //echo $row["changed"];
            if ($row["changed"] > $_POST['time']) {
                echo ",";
                include('pageInfos.php');
            }
        }
    } catch (Exception $e) {
        // when an error happens, report a 500 Internal Server Error
        // the client should be programmed to handle this
        echo "{ status : 500 }";
    }
?>
]
```

*Figure 15. The PHP script that is used to query the database and output a JSON format page.*


Ajax Based Front-end (template title) - Mozilla Firefox
Bestand Bewerken Beeld Geschiedenis Bladwijzers Extra Help
Website
Home Lorem Foto

Deze pagina heeft polling

Welkom op de homepage.Donec tortor est, tristique ut, imperdiet dictum, tempus et, ipsum. Integer in sapien. Morbi odio. Maecenas auctor suscipit sem. Pellentesque consequat nibh. Sed lorem. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Nulla facilisi. Sed

```
[
        "page",

        {
                "status" : 200,
                "id" : 1,
                "content" : "<h1>Deze pagina heeft polling</h1><p>Welko
                "browserTitle" : "",
                "nextClick" : 5,
                "pollTime" : 10000
        }
]
```

```
PageRequest
PageReceive
PageReceive
TemplateReceive
PageReceive
PageReceive
PageReceive
PageReceive
PageReceive
PageReceive
PageReceive
PageReceive
PageReceive
PageReceive
PageReceive
PageReceive
PageReceive
PageReceive
PageReceive
PageReceive
```

*Figure 16. The JSON format page (front) and the final result on the visitors screen (back). On the right are the actions listed that are performed on the back, added for learning purposes.*

**Client side**
Figure 17 is a. UML Class Diagram that shows the internal structure of the Ajax Engine. Details are left out when irrelevant. The objects, which are implemented in Java, are explained below.

- `PageInfo`. Objects of this type are a lightweight representation of a page. That means that most descriptive data is in it, but the real page content (the body) is not. Once the full page is fetched (Page object) these two are linked.

- `Page`. Objects of this type represent a page's content. Together with the `PageInfo`, it is all information that is available about a page. This separation is build in, because the lightweight version is loaded first and the full pages is not if it is not needed. The `pollTime` variable is the interval that is used for polling in milliseconds. The `nextClick` variable is a page id of the next page to fetch.

- `Template`. Objects of this type represent a template: HTML together with information about the header tags and the menu. The `menuLevels` Vector is a list of id's of HTML tags that should contain the menu. Their index in the Vector is in line with the level in the menu.

- `Tag`. This simple class is used for objects that represent a header Tag.

- `Identifier`. Objects of this type are a simple combination of HTML tag id and dynamic content. For example the id could be "menu" and the content could be and HTML object containing the menu.

The real work is done in the Application, which is the application's entry point. This object contains one gui object of type GraphicalUserInterface and on requestCentre object of type RequestCentre. The UML Sequence Diagram in Figure 4 (chapter 4.1) shows the communication between these objects.

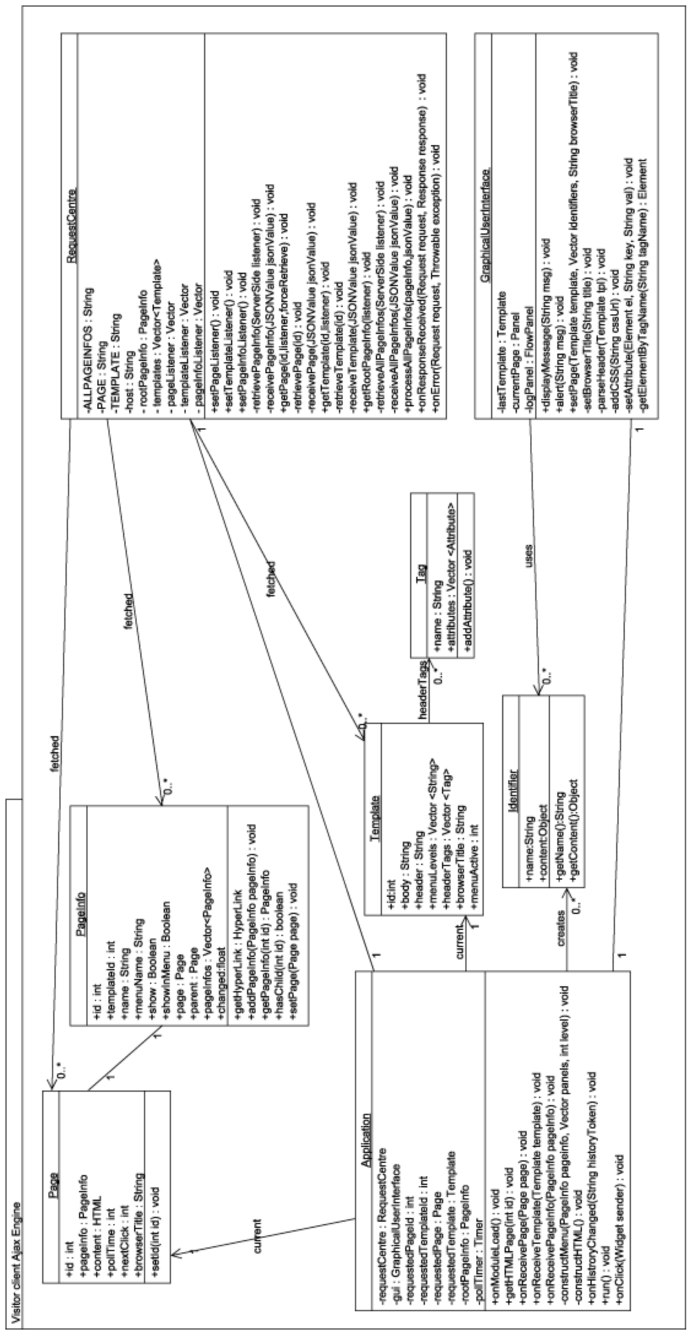

Visitor client Ajax Engine
RequestCentre
-ALLPAGEINFOS : String
-PAGE : String
-TEMPLATE : String
-host : String
- rootPageInfo : PageInfo
- templates : Vector<Template>
- pageListener : Vector
- templateListener : Vector
- pageInfoListener : Vector
+setPageListener() : void
+setTemplateListener() : void
+setPageInfoListener() : void
-retrievePageInfo(ServerSide listener) : void
-receivePageInfo(JSONValue jsonValue) : void
+getPage(id,listener,forceRetrieve) : void
-retrievePage(id) : void
-receivePage(JSONValue jsonValue) : void
+getTemplate(id,listener) : void
-retrieveTemplate(id) : void
-receiveTemplate(JSONValue jsonValue) : void
+getRootPageInfo(listener) : void
-retrieveAllPageInfos(ServerSide listener) : void
-receiveAllPageInfos(JSONValue jsonValue) : void
+processAllPageInfos(pageInfo,jsonValue) : void
+onResponseReceived(Request request, Response response) : void
+onError(Request request, Throwable exception) : void
GraphicalUserInterface
-lastTemplate : Template
-currentPage : Panel
-logPanel : FlowPanel
+displayMessage(String msg) : void
+alert(String msg) : void
+setPage(Template template, Vector identifiers, String browserTitle) : void
-setBrowserTitle(String title) : void
-parseHeader(Template tpl) : void
-addCSS(String cssUrl) : void
-setAttribute(Element el, String key, String val) : void
-getElementByTagName(String tagName) : Element
Tag
+name : String
+attributes : Vector <Attribute>
+addAttribute() : void
Template
+id:int
+body : String
+header : String
+menuLevels : Vector <String>
+headerTags : Vector <Tag>
+browserTitle : String
+menuActive : int
Identifier
+name:String
+content:Object
+getName():String
+getContent():Object
PageInfo
+id : int
+templateId : int
+name : String
+menuName : String
+show : Boolean
+showInMenu : Boolean
+page : Page
+parent : Page
+pageInfos : Vector<PageInfo>
+changed:float
+getHyperLink : HyperLink
+addPageInfo(PageInfo pageInfo) : void
+getPageInfo(int id) : PageInfo
+hasChild(int id) : boolean
+setPage(Page page) : void
Page
+id : int
+pageInfo : PageInfo
+content : HTML
+pollTime : int
+nextClick : int
+browserTitle : String
+setId(int id) : void
Application
-requestCentre : RequestCentre
-gui : GraphicalUserInterface
-requestedPageId : int
-requestedTemplateId : int
-requestedPage : Page
-requestedTemplate : Template
-rootPageInfo : PageInfo
-pollTimer : Timer
+onModuleLoad() : void
+getHTMLPage(int id) : void
+onReceivePage(Page page) : void
+onReceiveTemplate(Template template) : void
+onReceivePageInfo(PageInfo pageInfo) : void
-constructMenu(PageInfo pageinfo, Vector panels, int level) : void
-constructHTML() : void
+onHistoryChanged(String historyToken) : void
+run() : void
+onClick(Widget sender) : void
fetched
fetched
fetched
uses
headerTags
current
current
creates
0..*
0..*
0..*
0..*
0..*
0..*
1
1
1
1
1
1
1

*Figure 17. A detailed UML Class diagram of the visitor client Ajax Engine discussed in this tutorial.*

The gui of type GraphicalUserInterface contains only three public functions. Two of them – `displayMessage()` and `alert()` – are used for debugging, to show short messages (String). The most important public function is `setPage()`. It uses the template together with the vector of identifiers to display a page.

The `setPage()` function takes the template's content and dynamically fills the tags with id's that are in the identifiers Vector. Important is that Widgets like the menu panels are added as a Widget and not as Strings (by calling their `toString()` function). When Widgets are converted like this, they lose their Hyperlink event functionality. Be sure you never include this type of conversion somewhere in the process.

A private function `parseHeader()` that is retrieved from `setPage()` retrieves the headers that are set for the template and adds them dynamically to the page. This is done by using native JavaScript. This is the only way to add CSS that is really interpreted as a stylesheet.

In this tutorial the URLs of the server side scripts are hardcoded in the constants `ALLPAGEINFOS`, `PAGE` and `TEMPLATE`. When testing on a local computer thee variables can easily be changed to point to a local file.

As you can see in the class diagram of Figure 17 the functions in the RequestCentre are prefixed systematically: get, retrieve, and receive. The 'get' function, like in `getPage()`, is the public function that is invoked by the Application. This function checks whether the object is already in memory, it can than directly invoke the callback. If the object needs to be retrieved from the server, the retrieve function is called, which performs an Ajax request. The response is received at the `onResponceReceived()` function of the RequestCentre, which routes the response to the corresponding receive function. The receive function parses JSON, creates the new object, and invokes the callback.

Objects can register themselves to become a listener for an object, with the `set…Listener()` function. Their special functions are then invoked as a callback.

## 10.5 Online programmers references

GWT uses Java as its core language, almost no JavaScript is involved (except after compiling). Therefore, the developers resources listed below are all GWT specific. Though, for basic Ajax knowledge please consult the thesis and its references.

- Google Web Toolkit (Google Code webpage), http://code.google.com/intl/nl/webtoolkit/
- Setting up a GWT project in Eclipse (IBM webpage), http://www.ibm.com/developerworks/library/os-ad-gwt1/
- Building User Interfaces (Google Code webpage), http://code.google.com/intl/nl/docreader/#p=google-web-toolkit-doc-1-5&s=google-web-toolkit-doc-1-5&t=DevGuideUserInterface
- Roughian GWT Examples, http://examples.roughian.com
- GWT API (Google Code webpage) http://google-web-toolkit.googlecode.com/svn/javadoc/1.5/index.html?overview-summary.html
- Introduction (YouTube video), *Google Developer Day US – Fast, Easy, Beautiful: GWT* http://www.youtube.com/watch?v=NvRa-CxkpZI
- Example GWT Application (YouTube video), *Google I/O 2008 – Using GWT to Build a Diagramming Tool* http://www.youtube.com/watch?v=xh5Vo_drhDE
- GWT and Communication (YouTube video), *Google I/O 2008 – GWT and Client-Server Communication* http://www.youtube.com/watch?v=tRJEZgIX8BI