Honours Project Report

CALJAX

Matthew Hirst

Supervised by Hussein Suleman

Category		Min	Max	Chosen
1	Software Engineering/System Analysis	0	15	0
2	Theoretical Analysis	0	25	0
3	Experiment Design and Execution	0	20	10
4	System Development and Implementation	0	15	15
5	Results, Findings and Conclusion	10	20	15
6	Aim Formulation and Background Work	10		10
7	Quality of Report Writing and Presentation	10		10
8	Adherence to Project Proposal and Quality of Deliverables	10		10
9	Overall General Project Evaluation	0	10	10
Total marks		**80**	**80**	

Department of Computer Science

University of Cape Town

2009

Abstract

Digital repositories are systems used to store, access and manage digital collections.

Most often these systems are Web-based but in places where there is limited or no Internet connectivity, a distributed system on a digital medium such as a CD or DVD would be preferred.

In order to create a compatible and lightweight digital repository that can be distributed on removable media, this paper investigates and finds that using AJAX to create a system to browse a digital repository is a feasible and practical solution.

This is done by investigating digital repositories, the AJAX technology and investigating existing applications like Greenstone and DSpace. This report identifies what can be learnt from each implementation and attempts to recreate this functionality using AJAX.

The report investigates the design and implementation of such a system. The system is then evaluated through compatibility, performance and user testing.

The system is found to be successful based on the testing. However, the nature of AJAX limits the more advanced functionality.

Acknowledgments

I would like to acknowledge my project supervisor Hussein Suleman, as well as my project partners Marc Bowes and Suraj Subrun.

Contents

List of Figures

List of Tables

1. Introduction

There are many digital collections that contain an abundance of information, but this information is of no use unless there is a way to access and distribute it. This is where Digital Repository Systems come in. They allow collections of information to be searched, browsed and managed.

There have been many digital repository systems created in the past. However, these systems tend to either be made specifically for a collection, such as the system created to manage the Bleek and Lloyd collection [9], or tend to be heavyweight. They are heavyweight in that they require a slew of software to be installed, such as a database manager or a Web server.

This leads to digital collections traditionally being accessed as Web-based applications [11]. This makes them difficult to access in places where Web connectivity is not available, or limited, and leads to a need for a more lightweight and distributable system.

A distributable repository is a collection of information along with a system to access it that can be distributed on a removable medium such as a CD or a DVD. The information is therefore static if one uses a read only medium.

There have been systems created that could be distributed on CD and DVD, such as Greenstone. However, systems like Greenstone need to install software on the computer to function [14].

One technology that could be used to create a repository that is both lightweight and easily distributable is Asynchronous JavaScript and XML (AJAX). AJAX is included in most modern browsers, such as, FireFox, Chrome, Internet Explorer 7 and up, as well as safari. This means that as long as the user has access to one of these browsers, no extra software would need to be installed.

The background to AJAX and digital repositories is discussed further in section 2.

Due to the limitations previously mentioned, as well as advances in Web browsers, users have access to, this project proposes, implements and evaluates an AJAX based digital repository system.

The system this report proposes should be lightweight, extensible and distributable with a minimal software footprint, but without sacrificing usability or functionality. The system therefore needs to be able to browse, search and manage a digital repository without any additional software.

This project was undertaken by a group of three, with each member of the group taking responsibility of one of the pieces of functionality.

The browsing and searching parts of the project interface with the repository, which is managed by a third system, as indicated in figure 1.

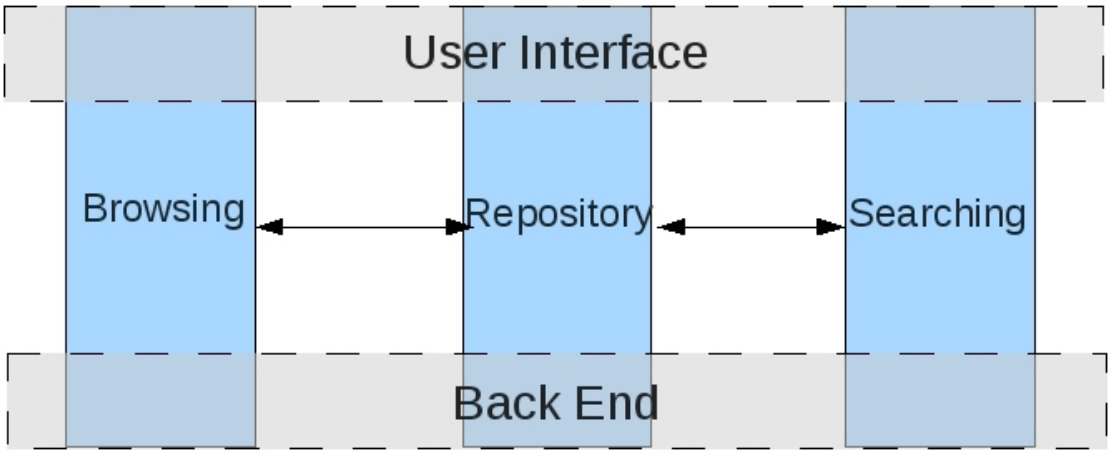

Figure 1: Project Division

This report focuses on the browsing section of the project.

1.1 Problem Statement

This section outlines what this project aims to achieve and gives a more formal definition of the problem that is being investigated. Section 1.1.1 identifies the research question, while section 1.1.2 defines the problem of browsing a digital repository.

1.1.1 Research question

Digital Repository systems are typically heavyweight, requiring a Web server in both online and offline forms. This project will attempt to overcome this through the use of AJAX in a Web browser. The research question asks whether this approach is feasible and able to provide at least the basic functionality expected of a Digital Repository System, while maintaining a minimal software footprint.

1.1.2 Browsing a digital repository

Browsing through a digital collection is a primary function of Digital Repository Systems and entails users being able to navigate through items in the digital collection based on the various categories.

This is simple to do using database management software (DBMS), but as AJAX is not able to interface with a DBMS and the definition of the project is to minimise the software footprint, the browsing system needs to be implemented from scratch.

The design and implementation of the browsing system is discussed further in section 3.

1.2 Expected Outcomes

The system will be considered fully successful if it manages to satisfy the research question, that the system is able to provide all the functionality of a regular browsing system, while still being lightweight and distributable.

Success will be measured through compatibility, performance and user testing and is discussed further in section 4.

1.3 Overview

This report starts by researching the technologies involved, including digital repositories as well as AJAX, in Section 2. The AJAX system's design and implementation is then described in section 3. The evaluation of the system follows in section 4, finally the report in concluded in section 5.

2. Background

As no research has been previously performed in the area of AJAX repositories, this report presents the technologies involved as well as looking at previously created systems with similar goals in order to better understand the problem.

In order to gain a better understanding of what the problem of creating a digital repository entails, section 2.1 is about at research done in the area of digital repositories. Section 2.2 then investigates the AJAX technology, what it is, its capabilities and its limitations. Section 2.3 is about systems already created in the area of digital repositories, the problems they were made to solve and their implementations.

2.1 Digital Repositories

Digital repositories are obviously stores of information. However, like digital libraries, they tend to be difficult to define.

Digital repositories and digital libraries are similar in definition and it could be argued that the terms could be applied interchangeably [4]. However, in a library the librarians have control over what resources are to be placed in the library whereas repositories, on the other hand, place an emphasis on people being able to contribute to the collection [4].

Because the terms are difficult to define there are many descriptions of digital repositories. One example would be to describe it as a shared database of information with a repository manager that supports various actions geared towards accessing and managing the information. [3]

Another general description of a digital repository is a system used to store and retrieve any digital material. These materials can consist of text, hypertext, images, videos and many other digital formats. [4]

Digital repositories are not just about safely storing digital materials, but more about sharing and reuse [4].

In general, a digital repository is a digital collection with some or other system to access and manage the collection, a similar definition to a digital library [12].

Now that there is a definition of a digital repository to work with, this paper will present the systems that make up a repository, namely the storage system and the system to access the objects/data in storage.

2.1.1 Storage

Storage in a repository is an integral part of the system and can be done in a variety of ways. Examples include file systems, database systems or even hard-copy filing cabinets. A variety of information about sources in the repository can be stored as well, such as a source's

location, its revision history, the tools and processes that were used to build it and even who is able to access the file [3].

More than one system may be used to store information in the repository. For example, a document or source code program may be stored in a file system, while the descriptive attributes of the object, such as its location and other information mentioned before, are stored in a DBMS [3]. (See Figure 2)

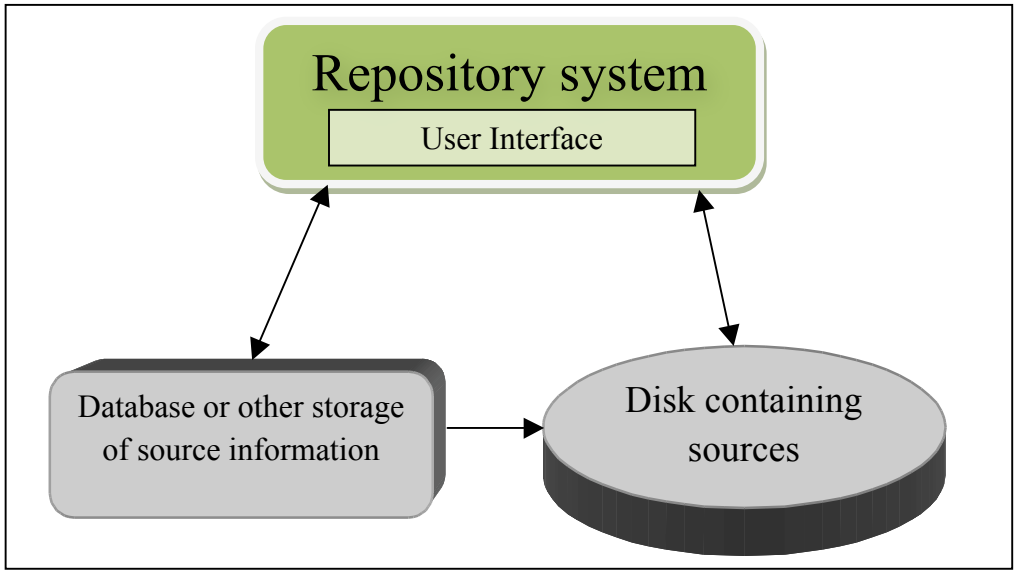

Figure 2: Diagram showing the layout of a repository system using two storage systems.

2.1.2 Data Access System

Once information is stored in a repository, one needs a system to access the objects and manage the repository. A repository manager must provide services for modelling, retrieving and managing the objects in a repository. The repository manager uses the repository's storage manager to store and retrieve the objects. So, using the previous example with the file system and the DBMS, when a user asks to retrieve an object, the repository uses the DBMS to look up the object's current location attribute and then uses that to load the object from the file system [3].

Beyond simple storage and retrieval, tools are needed for managing metadata. These services are the main added features of a repository and provide workflow control, version control, configuration control, content management and checkout/check in [3].

In the example of an AJAX repository, this would be the AJAX interface and the pre-processing to create AJAX-readable information.

The tasks needed to be performed by a repository system were be summarised by looking at a table from the IMS Digital Repositories White Paper [4].

Task defined by use	Response of Digital Repository
Search, gather, alert, browse	Expose
Configure	Interface change
Request	Deliver
Publish	Store
Deliver (from one repository)	Store (in another)

Table 1: Tasks and the way a digital repository might respond (based on Digital Repository Interoperability Problem Space from IMS Digital Repositories White Paper [4])

Now that a basis for the functionality and storage needed for a repository has been defined, this report presents the AJAX technology in order to better understand how it can be applied to the task of creating a repository.

2.2 AJAX

Web applications for a long time have been less rich and less responsive than their desktop counterparts. However with the introduction of Web 2.0 and technologies such as AJAX this gap is closing. AJAX stands for Asynchronous JavaScript and XML and it represents a shift in what is considered possible on the Web [5].

AJAX isn't really a new technology. It is several technologies working together in powerful new ways [5].

AJAX incorporates [5]:

- standards-based presentation using XHTML and CSS;
- dynamic display and interaction using the Document Object Model;
- data interchange and manipulation using XML and XSLT;
- asynchronous data retrieval using XMLHttpRequest; And
- JavaScript binding everything together.

In the classic Web interaction model the communication is one way with the user performing an action which is then post back to the server which does some processing and then returns an HTML page to the client [5]. This limits the interactivity and feedback of a website [6].

AJAX allows two way communication to occur between the browser and the Web server without having to install additional software [6]. It does this by introducing an AJAX engine which acts as an intermediary between the user and the server. Instead of loading a Webpage, when a session is started the browser loads an AJAX engine written in JavaScript. In cases where AJAX is used, this AJAX engine is then responsible for rendering the user interface as well as communicating with the server or other sources [5]. (see Figure 3)

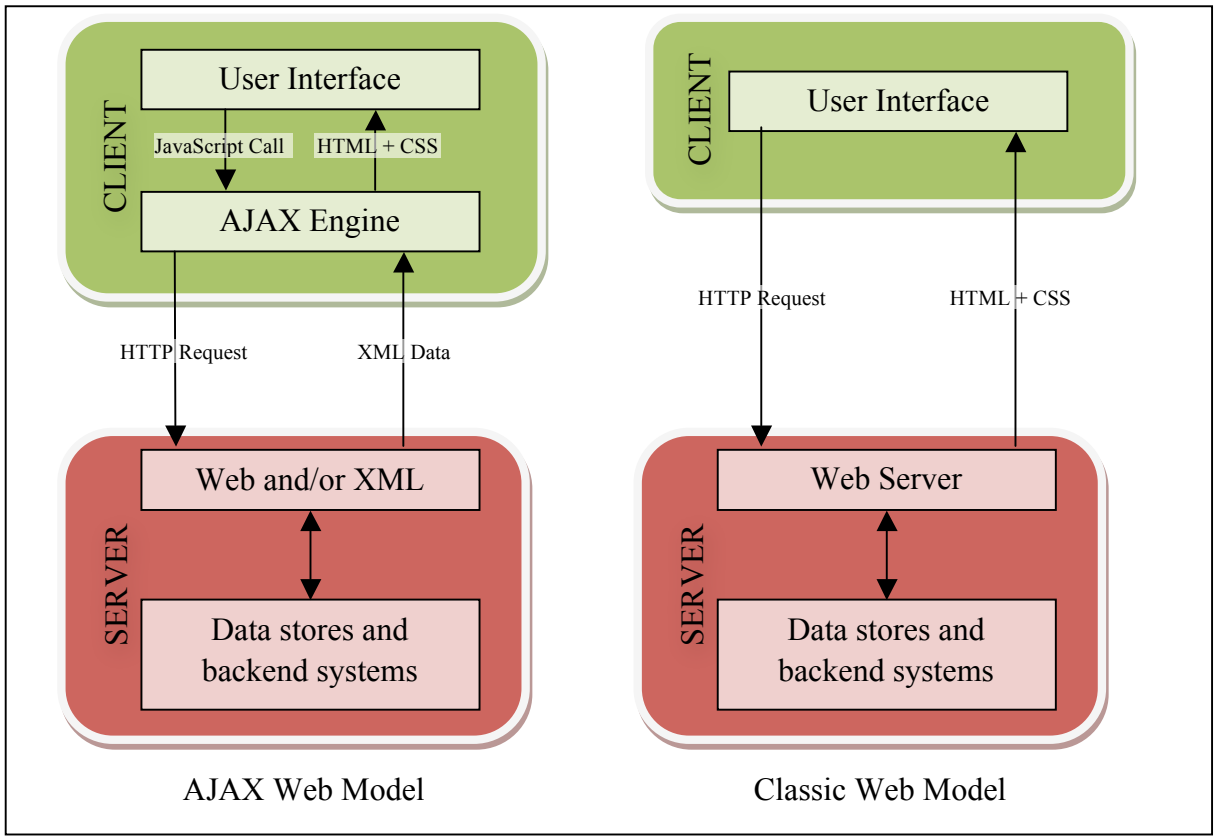

Figure 3: The different models for AJAX (left) and traditional (right) Web applications.

This enables creating a Web application that works without using a Web server, the application is able to just do all the work through the AJAX engine, never having to contact a Web server. This allows for creating a distributable repository that uses a Web browser without any extra software needed.

Furthermore, using the AJAX engine, one can manipulate the Document Object Model or DOM which is basically the data structure that represents the current Webpage [5]. By manipulating the DOM it is possible to create functionality without a Web server. Manipulating the DOM allows an application to create new elements in a Web page, remove them, reorder them and change their attributes. It basically gives the application the ability to directly edit the source of a page [5].

AJAX also is already a very widely used technology with large companies like Google using it in many of their applications such as Google Maps, Google Groups and Google Suggest. The core AJAX technologies are mature, stable and well understood [5]. As such, almost all modern browsers have support for AJAX, making any application written with it widely compatible with a variety of operating systems and architectures.

One limitation is that JavaScript is not able to write to a local filestore due to security issues. This makes it more difficult for AJAX to modify information in repository. Although AJAX is unable to browse a filestore, using the AJAX engine AJAX is able to perform requests through the operating system, and read files from the filestore.

Using AJAX with an intermediate data representation of the digital collection one could use the AJAX engine to browse through the digital collection by modifying the DOM and styling the data using some styling method, such as CSS.

2.3 Previous work performed in the area

Although no repository has been created using AJAX, previously there have been systems created with similar objectives, either to be a distributable medium for information or to create a lightweight system that is easy to use. In order to gain background information and to learn from what the other systems have done, this paper will investigate several previously created systems, including Greenstone [14], A Student friendly repository for teaching [1], DSpace [8], DotWikIE/TiddlyWiki [7] and The Bleek and Lloyd collection [9], focusing specifically on how the systems browsed and stored information.

2.3.1 System to access the repository

On the topic of browsing of digital repository systems, this report investigates 2 well established digital repository systems: Greenstone and DSpace. These systems were created with slightly different objectives. DSpace was created in an attempt to address the problems that the MIT faculty were having in collecting, preserving, indexing and distributing research materials and scholarly publications [8]. Greenstone was created to help people build their own collections [14].

One commonality between the systems was the goal to design the system to make participation by contributors easy [8][14]. Another commonality is that both systems use a Web-based interface for the users to interact with the digital repository. This would suggest that using a Web interface is the easiest or at least most common or most comfortable way for a user to interact with the system.

Using AJAX to build a digital repository may be a natural step in creating a user friendly system. This can be further motivated by the fact that in the student friendly repository for teaching, AJAX was chosen specifically to emphasize the ease-of-use of the system [1].

Another observation is that the systems keep the different interfaces separate, i.e. the interface for users to browse the data is kept separate from the interface to manage the repository [14] [1] [8]. This makes it simpler and easier for the users to browse through the data as well as making the application to browse the data more lightweight and easier to distributable.

In Greenstone data in the repository is browsed based on the metadata. The data in the repository can therefore be browsed by in a multitude of ways, e.g. by author, by tile. The browsing facilities are therefore created during the building process by accessing the metadata for each object in the collection [14].

DSpace, on the other hand, allows data to be added to the repository after it has been created, so statically creating the browse functionality at build time was not an option. DSpace's solution was to have users browse based on a specific index. It does this by providing an API

that allows a user to specify an index and a subsection of that index. The indices that may be browsed are item title, item issue date and authors [13].

The information within a distributable repository is likely to be static or not frequently updated, so building the browsing facilities at creation is an option.

2.3.2 Storage

Often the storage system of choice for a digital library system is a database [9]. This can be seen in systems such as DSpace [8] where mySQL is normally used [9]. This is because databases provide an efficient and easy way to manage the data in the collection [9]. However in order to use AJAX one would have to find another way to store the data as AJAX is unable to connect to a database [7].

One technology that has already been suggested and tested is XML [9]. It has been used in collecting data in the Greenstone system [14] and it also fits well into the AJAX schema as XML is one of the core technologies AJAX works with.

The solution TiddlyWiki used was to store the data embedded in the Webpage in the form of text and hyperlinks [7]. In TiddlyWiki the sources are known as ‘tiddlers’. ‘Tiddlers’ are created, edited and deleted through the use of JavaScript embedded in the same file. By editing the DOM as stated in the AJAX section of this paper one is able to hide, display and load information through hyperlinks onto the page.

3. Design and Implementation

CALJAX, being a proof of concept system is experimental at heart. However, the design and implementation play a very important role. As such a proper design cycle was used in developing the system. The iterative approach used by the project team is described in section 3.1.

The CALJAX system as specified before is a digital repository system created using AJAX, thus allowing users to access the system locally without installing any special software. This paper focuses on the browsing section of the system, so only this will be described.

Browsing includes allowing users to navigate through a collection looking for an item in the collection they are interested in. Creating browse able lists out of a collection is essentially sorting data in that collection based on each of its sortable fields and letting users navigate through the collection sorted in these different ways. This way users can find what they are looking for by browsing through the category they needed to sort by.

The end product must then be a Webpage with access to files in lists sorted by each of the sortable fields.

The files in the collection can be anything from pictures or music to documents or plain text. The data describing a file in the collection, which is used in browsing is stored along with that file as a .metadata file. The CALJAX system assumes that these are XML documents where each child of the root node is a sortable field.

For the CALJAX project sorting could be done in two different places (see Figure 4): In a pre-processor where the files would be pre-sorted and written out in a simplified form for the JavaScript to use, or in the webpage itself where JavaScript could do the sorting.

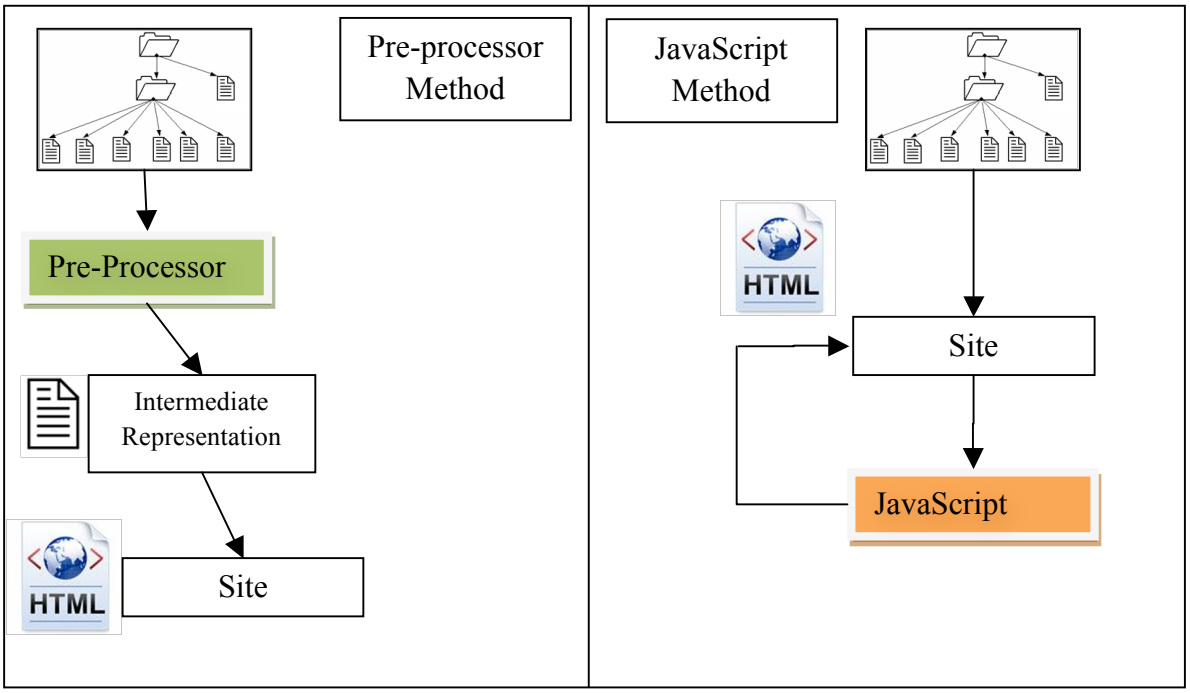

Figure 4: Diagram of two implementation methods

However, because JavaScript is unable to write to disk, one would have to re-sort the collection for each of its sortable fields every time the webpage was opened. This may be viable for very small collections where the sorting would be fast and would allow the system to be very flexible. With no pre-processor needed users could add and remove files from the collection easily on the local machine with no real reconfiguration required.

However with JavaScript being slow and collections potentially being very large (the Bleek and Lloyd collection testing was done with contained 35 950 files) sorting the data multiple times every time the page is loaded becomes infeasible.

Thus the pre-processor route was chosen as it allows the very intensive calculations to be done only once and the results stored. Using a pre-processor also allowed the intensive work to be performed in a language other than JavaScript.

The overview of the system can be seen in Figure 5 below. The data is pre-processed into an intermediate form which the JavaScript from the site uses to browse. The site also has access to the actual data so that it can display the files when a user finds a file they want to access.

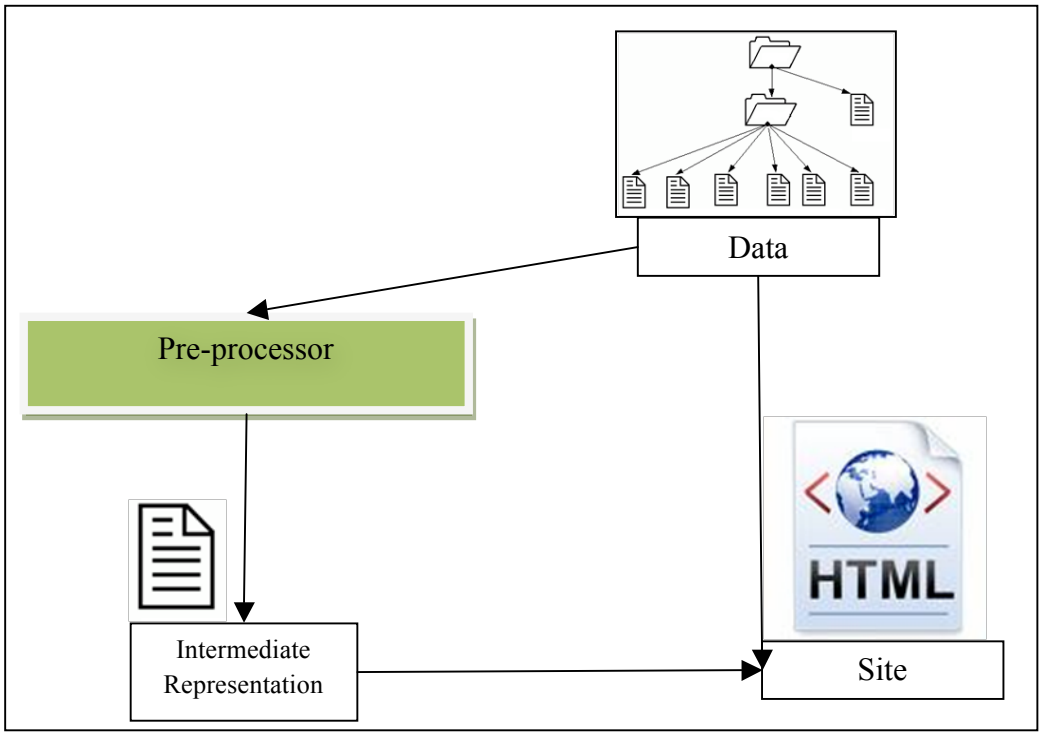

Figure 5: Overview of the browsing system

The design of the system is thus broken up into two sections. Section 3.2 describes the design and implementation of the pre-processor and section 3.3 describes the website.

The system was also made to be generalised and, as such, a default structure for the files and metadata is assumed. The directory structure may be tiered but files in the collection must be accompanied by an XML .metadata file.

3.1Overview - Iterative Design

The iterative design proposed for the project contained 3 phases, namely: Design, Development and Evaluation phases. However, due to time constraints only the 3rd and final phase had a proper evaluation phase.

The iterations began with the group meeting the project supervisor to discuss what was to be done in that iteration, the iteration was then implemented and the iteration and ended with a demo of the system to the project supervisor and some other students where feedback was given for the next iteration. This demo was used as an evaluation phase for iterations 1 and 2.

3.1.1 Iteration 1

The first iteration of the system, or the prototype, was where the ideas for an AJAX repository were first tested. This included the first version of the pre-processor which simply sorted the files and wrote output lists (This will be discussed further in section 3.3). The prototype also included the first version of the site which simply displayed these lists, allowing the user to browse through them.

The prototype although seemingly rudimentary in functionality was where most of the browsing logic was worked out.

The below figure is a screenshot of the completed first iteration.

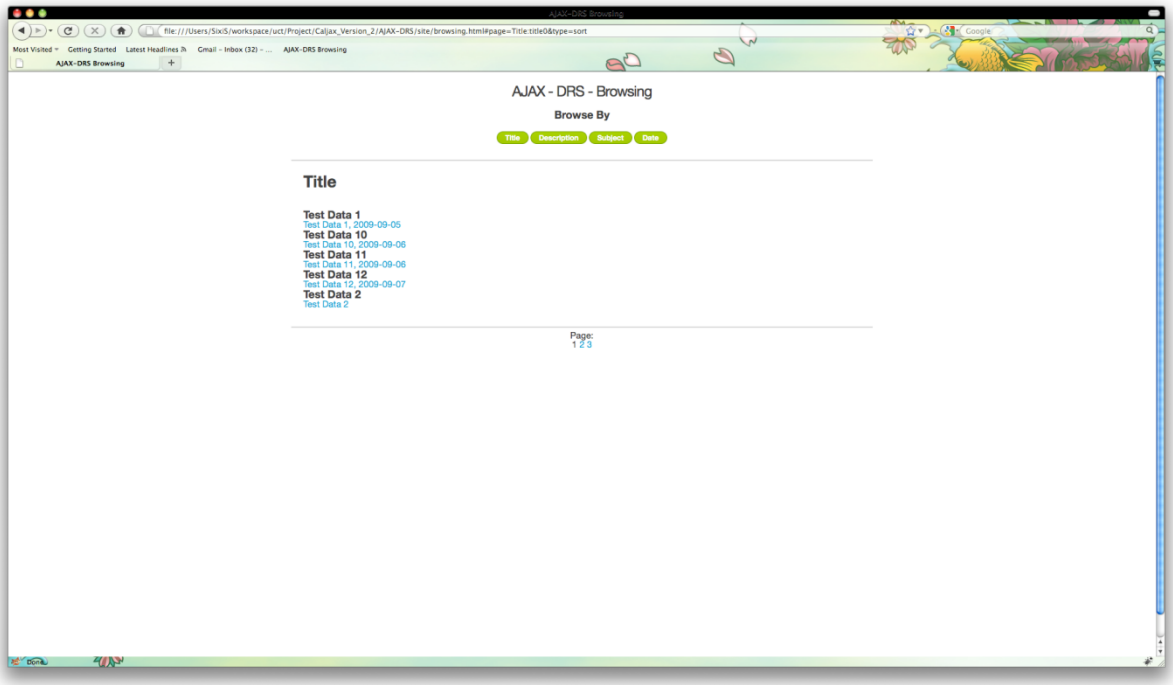

Figure 6: Iteration 1

3.1.2 Iteration 2

The second iteration then started to include basic functionality. As users need to be able to access files using the system, basic viewing of the metadata as well as links to the files in the collection was introduced.

When listing the files on the website, because the system is generalised, it does not know which field to display as the name of the file. This is because for one collection the field to be displayed may be called the “title”, whereas in another collection it might be the “name” field.

As the system cannot determine which fields need to be displayed as a representation of that file on the lists of sorted files, a configuration file needed to be created, allowing the user to specify which fields were to be used as the file’s representation.

The system was made to support collections in directory structures other than a simple flat structure and online integration was introduced. Online integration is where the system fetches updates from a server and includes them in the browsing results. This will be discussed further in section 3.3.

JavaScript to create normal page access using the history was also introduced in this version. This allowed users to make use of the browsers “back”, “forward” and “favourite” functionality even though the system is just a single web page. This is done using a jQuery plug-in and is discussed further in section 3.3.1. [2]

The look and feel of the page was also improved in this iteration. These improvements can be seen in the below figure which is a screenshot of the completed second iteration. The screen is displaying the metadata for a single file.

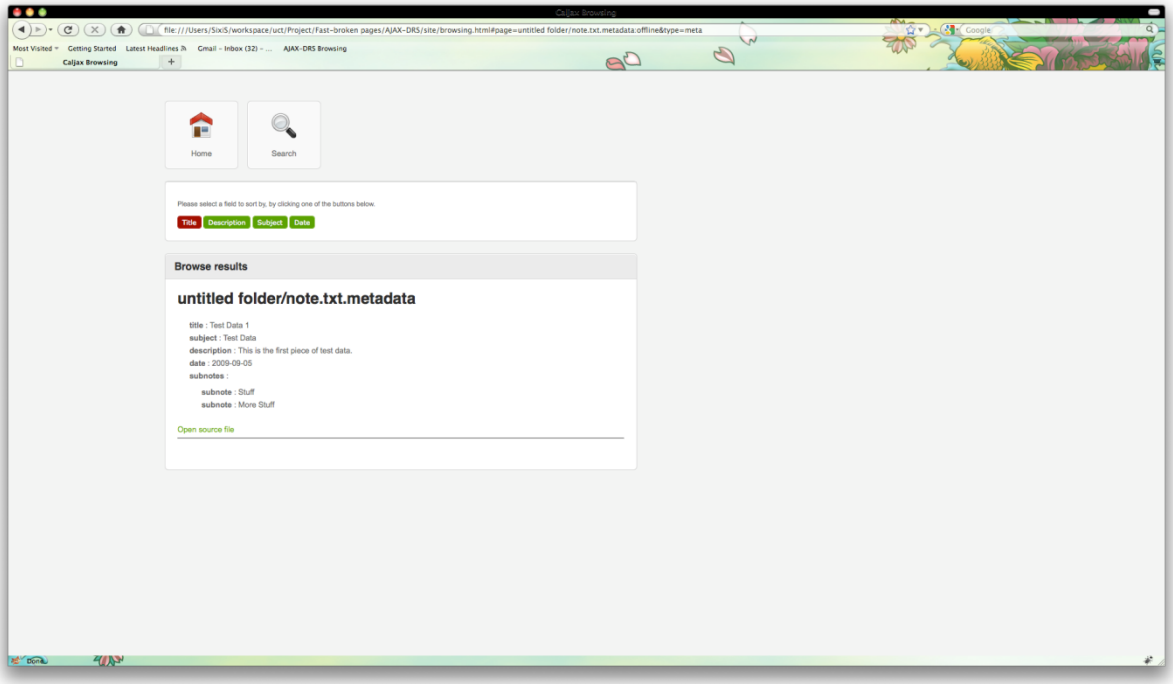

Figure 7: Iteration 2

3.1.3 Iteration 3

The third and final iteration was to tune up and finalise all the functionality as well as improving the usability of the system. In iteration 2 the basic functionality had been introduced, users could browse through pages of files sorted in different ways and access the files they wanted to, however, the site was not easy to use. This was because users had to

navigate through a very large number of pages with no way of quickly jumping to the content they were looking for.

In order to ameliorate this difficulty, page jumping functionality was introduced, allowing users to specify a page number and navigate directly to that page; next and previous page buttons were also added.

A “quick nav” panel was created which allowed users to quickly identify the categories in that sorted field. Once they found the category they were looking for they could use the “quick nav” to jump to the page that category started on.

The online update functionality was improved in this iteration, making it much more stable and allowing users to enable and disable it by ticking a box which stored the setting in the local cookies.

Local cookies are settings for Web pages saved by the browser that the Web page can access later.

The look and feel of the site was improved again in this iteration, making full use of the real estate present on the screen. This can be seen in the figure below. Where there used to be unused space there are navigation tools.

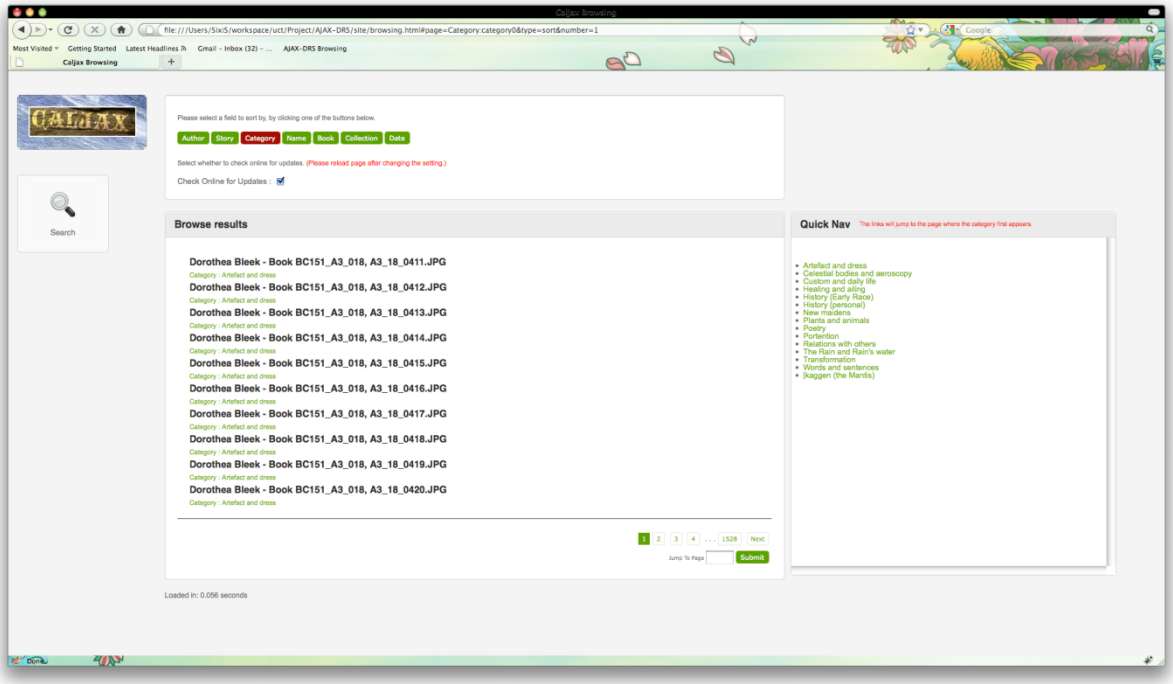

Figure 8: Iteration 3

3.2 Pre-processor

As stated before the sorting cannot be done by JavaScript as it is unable to write to disk, therefore a pre-processor is needed to pre-sort the data by its various metadata fields, as well

as performing other tasks described below, in order to allow JavaScript to offer functionality without having to access every metadata file.

The pre-processor was written in Java, to keep cross platform compatibility.

The tasks performed by the pre-processor are:

- Sorting the data
    - Reading metadata for each valid file from an unknown directory structure.
    - Sorting the data based on each of the fields in the metadata.
    - Creating a representation for each file.
- Writing out the lists for the site
    - Omitting fields the user does not want to browse.
    - Creating pagination.
    - Creating quick navigation links.
    - Linking the lists to the website.

3.2.1 Reading and sorting the data

The basic task performed by the pre-processor is sorting the data based on its various metadata fields. In order to do this the pre-processor needs to read the metadata for each file from an unknown directory structure.

The base location of the data is set using a configuration file. From the base location the pre-processor then recursively searches through the directory structure looking for file and metadata pairs.

An example of a file and metadata pair is the file “file.txt” and its associated metadata file “file.txt.metadata”.

Once a file and metadata pair have been found the pre-processor loads the metadata into memory. Once the entire directory structure has been searched and all the metadata has been loaded, the pre-processor can then sort the files into lists based on the various fields in the metadata (seen below in Figure 9).

```
<?xml version="1.0" encoding="UTF-8"?>
<output>
  <author>!|kabbo (Jantje) (II)</author>
  <book>BC_151_A2_1_013</book>
  <category>|kaggen (the Mantis)</category>
  <collection>Lucy Lloyd |xam notebooks</collection>
  <date>23-27 October 1871</date>
  <keywords>|kaggen Ichneumon dreams arrow Ostrich ostrich eggs hunting food</keywords>
  <name>A2_1_13_00720.JPG</name>
  <story>The story of the Mantis and the Ostrich who talks (|kaggen and !kaken-!kaka-!k'aui)</story>
  <subkeywords>
    the Mantis his 'thinking strings' dreams and Ostrich who talks or !kaken-!kaka-!k'aui real
    ostrich arrow misses advised by Ichneumon hunts eggs stick to him discourse of |ni advises
    |kaggen tells about hunting ostriches |kaggen's comes back at is different its magic methods
    drinking for implements finding what
  </subkeywords>
</output>
```

Figure 9: Example of an XML metadata file

The sorting only deals with flat fields, in that if a field in the metadata has children, the pre-processor will not sort based on the child values.

For the above metadata example in Figure 9, a list of files will be created for each field, where each list will be sorted alphabetically based on the values for that field. So the list for the name field will have a list of files sorted by the name field.

Finally, a representation for each file needs to be created out of the metadata. This is set in the configuration file. The representation setting is simply a list of the fields that the user wants to be displayed for that file on the system.

```
<root>
  <version>2009-10-12 20:00</version>
  <representation>story,name</representation>
  <do_not_browse>keywords,subkeywords,summary</do_not_browse>
  <category_browse>book,story,collection,category</category_browse>
  <alphabet_browse>name</alphabet_browse>
  <data>../site/data/</data>
  <site>../site/</site>
  <items_per_page>10</items_per_page>
  <force_page_count>false</force_page_count>
</root>
```

Figure 10: Example of a configuration file

In the example configuration file (Figure 10) the story and name field would be used as the representation for the file when it is displayed in the lists on the site.

Other fields in the configuration file include the path of the site and the path for the data. These settings are used to tell the pre-processor where to write the files to and when to read the data from.

The version setting is the timestamp to be used for online updates.

The do_not_browse field specified which fields the site must not browse. The fields specified in the do_not_browse fields are ignored when reading in the metadata and as such are not sorted.

Then the category_browse and alphabet_browse settings are used to define the quick nav which is discussed in section 3.3.

3.2.2 Writing the lists and linking to the site

Once the sorted lists have been created they need to be written out in an intermediate representation so that the JavaScript on the Webpage can use it to browse the collection.

The first consideration was that JavaScript would not be able to handle loading incredibly large lists and as such some sort of chunking would be needed. Chunking in this case is to write the results out in batches so that JavaScript could dynamically load them, piece by piece.

As pagination would be needed on the Webpage a natural way to split the lists up would be by page. If the lists were split by page, when the site needed to load a set of results it could simply load the list for that page and display the results without having to do any more work.

The number of items per page is set in the configuration file and affects the size of the lists.

The example list file in the figure below has a page size of 10, so 10 items are linked to by the file.

```
Wilhelm Bleek notebooks/BC_151_A1_4_010/A1_4_10_01025.JPG.metadata;Blank pages in Bleek's Book X, A1_4_10_01025.JPG;A1_4_10_01025.JPG
Wilhelm Bleek notebooks/BC_151_A1_4_010/A1_4_10_01026.JPG.metadata;Blank pages in Bleek's Book X, A1_4_10_01026.JPG;A1_4_10_01026.JPG
Wilhelm Bleek notebooks/BC_151_A1_4_010/A1_4_10_BKCOV.JPG.metadata;Covers and first pages of Bleek's Book X or BC151_A1_4_010, A1_4_10_BKCOV.JPG;A1_4_10_BKCOV.JPG
Wilhelm Bleek notebooks/BC_151_A1_4_010/A1_4_10_FRCOV.JPG.metadata;Covers and first pages of Bleek's Book X or BC151_A1_4_010, A1_4_10_FRCOV.JPG;A1_4_10_FRCOV.JPG
Wilhelm Bleek notebooks/BC_151_A1_4_010/A1_4_10_IBCOV.JPG.metadata;Covers and first pages of Bleek's Book X or BC151_A1_4_010, A1_4_10_IBCOV.JPG;A1_4_10_IBCOV.JPG
Wilhelm Bleek notebooks/BC_151_A1_4_010/A1_4_10_SPINE.JPG.metadata;Covers and first pages of Bleek's Book X or BC151_A1_4_010, A1_4_10_SPINE.JPG;A1_4_10_SPINE.JPG
Wilhelm Bleek notebooks/BC_151_A1_4_011/A1_4_11_01027.JPG.metadata;The Mantis and !goe !kweitentu, A1_4_11_01027.JPG;A1_4_11_01027.JPG
Wilhelm Bleek notebooks/BC_151_A1_4_011/A1_4_11_01028.JPG.metadata;The Mantis and !goe !kweitentu, A1_4_11_01028.JPG;A1_4_11_01028.JPG
Wilhelm Bleek notebooks/BC_151_A1_4_011/A1_4_11_01029.JPG.metadata;The Mantis and !goe !kweitentu, A1_4_11_01029.JPG;A1_4_11_01029.JPG
Wilhelm Bleek notebooks/BC_151_A1_4_011/A1_4_11_01030.JPG.metadata;The Mantis and !goe !kweitentu, A1_4_11_01030.JPG;A1_4_11_01030.JPG
```

Figure 11: Example of a list file

The problem then became telling JavaScript how many pages there were, so that it could link to them. The solution used in this case was to have Java manually write out a JavaScript file containing arrays of the list files. The Webpage could then include this JavaScript file and use these arrays to access the file lists.

Other settings from the configuration file were also included in this JavaScript file as variables, as can be seen in Figure 12 below.

```
date[1558] = "date1558";
date[1559] = "date1559";
date[1560] = "date1560";
date[1561] = "date1561";
date[1562] = "date1562";
files["date"] = date;
var version = "2009-10-12 20:00";
var representation = "story,name";
var pageSize = 10;
var force_page_count = false;
```

Figure 12: Part of the generated JavaScript file

Finally, the pre-processor has to create links for the “Quick Navigation” panel on the site. The quick navigation panel is basically a list of the values in that field and a link to the page that value first appeared in. e.g. a category sort for collection would contain all the collection names with links to the pages each collection began on.

The pre-processor also supports an alphabet sort where instead of checking the values of the fields it checks the letter the values start with. This can be used to generate quick links for items such as name, where every item has a different name, but can be quickly browsed based on the letter it starts with.

In order to facilitate this feature without having JavaScript search through all the lists, the pre-processor generates a JavaScript file that contains arrays of these categories with the page number each began on. The webpage can then include this JavaScript file and use the arrays to generate the Quick Navigation links.

In the example of a generated category browse JavaScript file (Figure 13), the "Dorothea Bleek notebooks" collection begins on page 0, while the "Jemima Bleek notebooks" collection begins on page 353.

```
book_category["MP4"] = 1788;
category_sort["book"] = book_category;
var collection_category = new Array();
collection_category["Dorothea Bleek notebooks"] = 0;
collection_category["Jemima Bleek notebooks"] = 353;
collection_category["Lucy Lloyd !kun notebooks"] = 360;
collection_category["Lucy Lloyd Kora notebooks"] = 507;
collection_category["Lucy Lloyd |xam notebooks"] = 539;
collection_category["Wilhelm Bleek notebooks"] = 1519;
category_sort["collection"] = collection_category;
```

Figure 13: Part of the generated Category Browse JavaScript file

3.3 Site

Once the pre-processor has completed generating the lists and JavaScript the site needs, the site can now use these to generate browsable pages for users of the site.

The jQuery JavaScript library [15] was used in order to simplify the JavaScript for the functionality needed.

The site's functionality is split into two sections. The AJAX browsing section, which is the main browsing functionality of the site, is discussed in section 3.3.1.

The online integration section, which deals with adding online updates to the collection, is discussed in section 3.3.2.

3.3.1 AJAX Browsing

The browsing for this system is all created dynamically from the data provided by the pre-processor. The HTML provides a framework for the JavaScript to modify. This is done by creating static content along with empty divs that the JavaScript can then insert objects such as hyperlinks and text into.

The page starts by creating buttons for each of the browsable categories that users can use to start browsing. These buttons have onclick events that alter a fragment URL linking back to the site.

A fragment URL is a URL that ends with "#" followed by anchor identifiers. These anchor identifiers can then be used to direct functionality. For example, the fragment URL

“http://www.site.com#page=4” is a URL pointing to http://www.site.com with a “page=4” anchor.[16]JavaScript can then use this “page=4” anchor to display the 4th pages content.

This is done instead of directly calling a method to display the sorted lists, in order to make the browser believe it has changed pages. This way users can use functions such as forward and backward in the browser because the fragments will change.

The JavaScript then loads information onto the page depending on the fragment every time the fragment changes.

In the case of this site, the JavaScript will load a sorted list or a metadata file depending on the “type” fragment.

When loading a field to browse through, the system checks the included JavaScript arrays provided by the pre-processor for the first page of that sorted field and loads the results in from that file. The files are loaded through a jQuery AJAX call. This performs a GET request to the file system which returns the list file in question.

The loaded list file is then parsed and the links generated using the representation provided and the location of the source file. It also displays the value of that source file for that field.

The links generated for the browsable lists modify the fragments of the page, setting the JavaScript to display that link’s metadata file.

An example of the links generated can be seen in the figure below. The links generated in the figure are for the collection field.

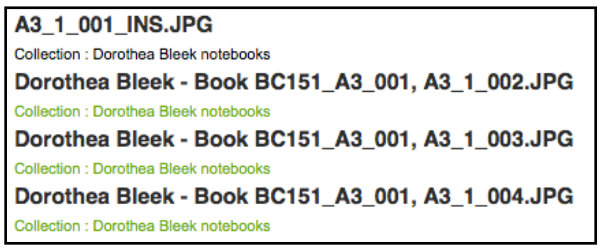

Figure 14: Example of links generated

Loading other pages of a field works much in the same way. The page fragment is altered and the JavaScript loads from that page’s list file.

Various methods of navigating through the pages were created. The page links as well as the next / previous buttons as well as the “Jump To Page” function, depicted in the figure below, work in the same manner of altering the page fragment, causing the JavaScript to load results for that page and field.

Figure 15: Example of pagination links

The other navigation functionality that JavaScript generates is the Quick Nav panel. This panel is generated using the category sort arrays provided by the pre-processor. It does this by looking in the category sort array for that field. If the array does not exist then there is no quick nav for that field and no quick nav box will be generated. If there is a set of items for the quick nav, the JavaScript populates the quick nav box with links, named by the category in question and linking to the page that category starts on.

These links work in exactly the same way as the pagination links. This was done in order to prevent JavaScript having to do any extra work to display the results.

Below (Figure 16) is an example of a "Quick Nav" generated for the collection field.

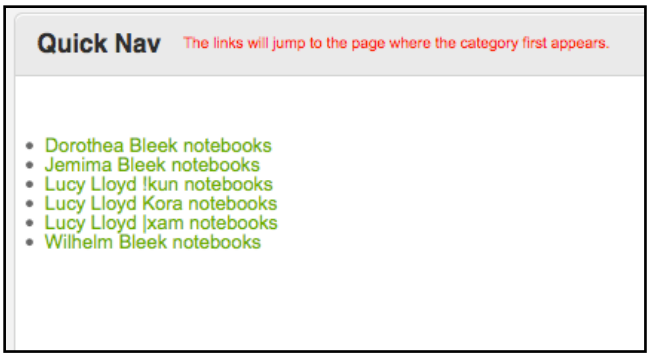

Figure 16: Example of the Quick Nav

The other function performed by the AJAX browsing system is displaying the metadata file. This is done by recursively reading through the file and generating an HTML list out of it.

A link to the original source file is also generated. This can be seen in the figure below.

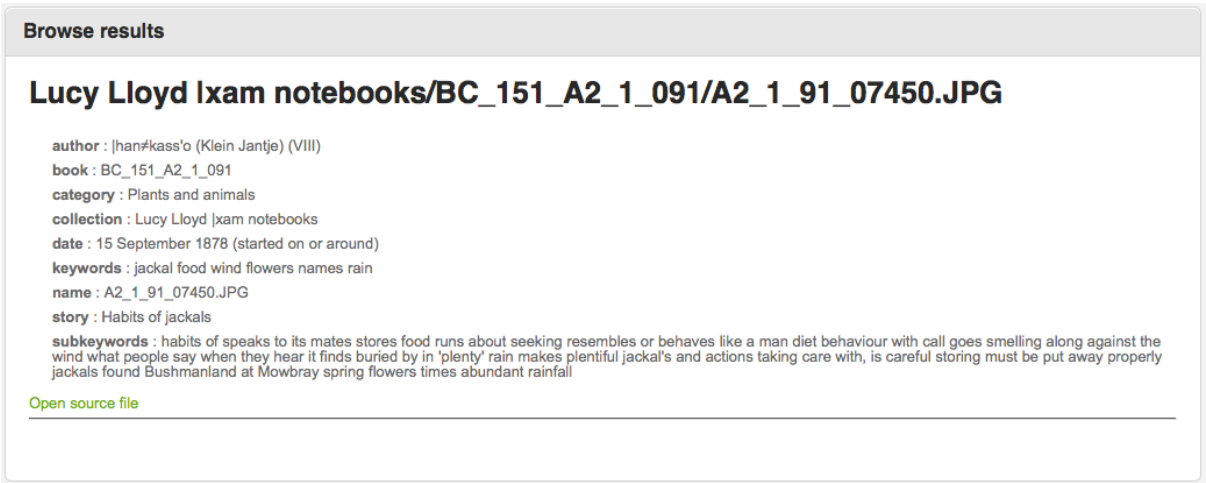

Figure 17: Displaying a metadata file

3.3.2 Online Integration

The other functionality provided within the site is the online integration. The online repository may have new or updated versions of the data collection and in order to include these updates in the system an online integration feature was added.

This worked by sending a "GET" request along with the timestamp for the current version of the collection. The online system would then send an XML response containing a set of updated metadata files. This response is then parsed, separating the retrieved metadata files and saving them to memory. The assumption made at this point is that the updates will be small in size, because if they became very large it would slow the system down greatly.

Once the metadata files are in memory they can be combined with the offline files.

An overview of the process can be seen below in figure 18.

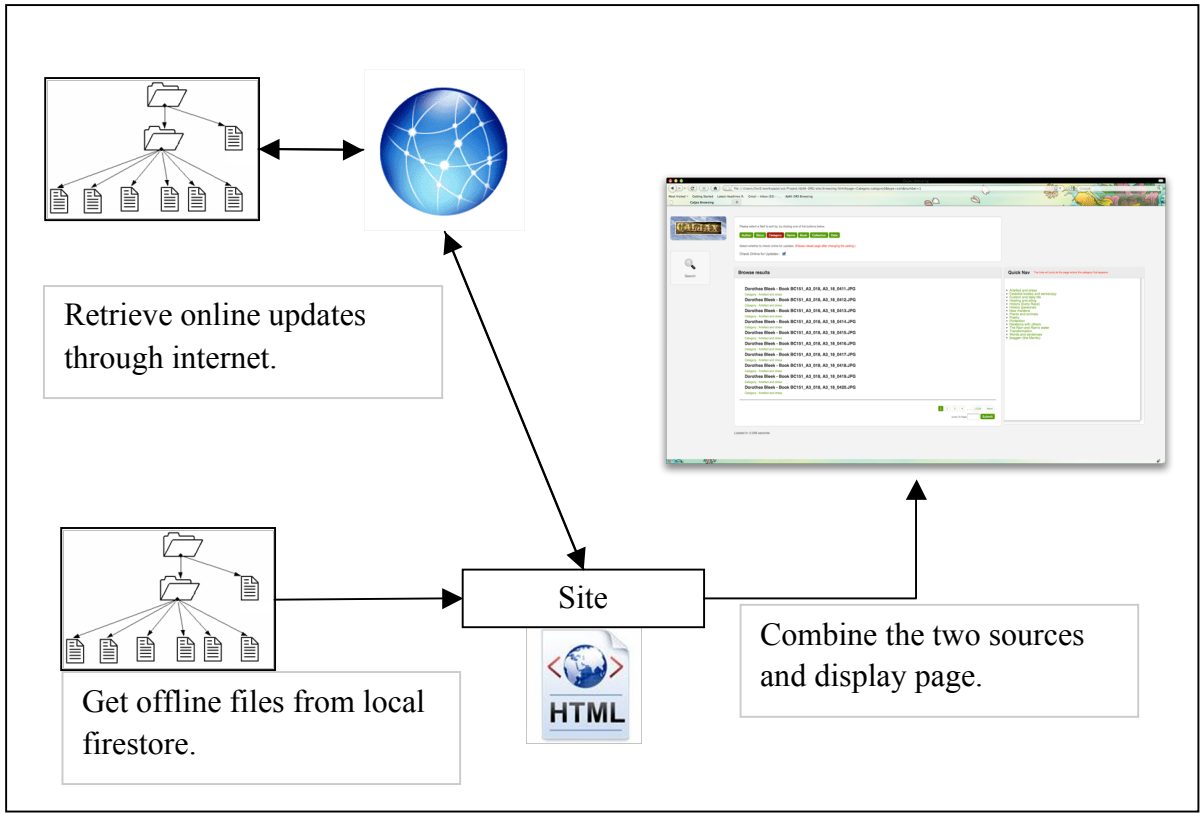

Figure 18: Overview of online integration

Once again the limitation of JavaScript not being able to write to disk creates a problem. As the offline lists cannot be updated only once, the updated files need to be integrated every time they are retrieved.

Two methods of integrating the online files with the offline files were identified at this point. The first method included pre-sorting the online files into the offline lists and remembering the positions they would be saved at in memory. By using this method, when displaying the

lists, the system could simply check if any of the online files belonged in that list and then integrate them.

This also gave the benefit of knowing how many files had been included before the current page and allowing the system to adjust the pagination accordingly. The pagination was adjusted by counting the number of extra online files included up to that point and then starting the listing that many offline files back (as those files would not have been displaced by the extra online files).

However, in very large collections, sorting the online files into the offline lists during runtime becomes infeasible. Due to this another method had to be used.

The second method was to simply check each of the online files when loading a page of results. This was done by loading the current page's list and the list from one page back. The system then checks whether the online file belongs on this page by checking that it is bigger than the last entry on the previous page but smaller than the last entry on the current page.

This is a more basic method and assuming the number of online updates is kept small the method has little impact on the system's performance. The one downside of this method is that there is no way of maintaining the pagination size as the online files have not been sorted into the offline ones.

Once the online files have been integrated with the offline files they are accessed and displayed in the same manner. The only difference is that the link to the source file is an external link as the file is not sent along with the metadata file by the online system.

This method is functional but updating every time the site is opened is an inconvenience. One possibility for future work would be to create an external program to update the collection, or to introduce it into the pre-processor.

3.3.3 Limitations

JavaScript speed limitations make it infeasible to perform actions that require the entire collection to become involved, so certain drill-down functions are complex and slow to implement.

One example of a function that a lot of browsing systems have today but is difficult to implement is that of a filter system.

A filter system is a system where while browsing results one is able to specify extra requirements, such as only displaying items from a specific category.

In a traditional system with a Web server and database management system filtering is simply modifying the SELECT statement with an additional WHERE clause. However, to perform this task in JavaScript one would have to loop through each of the metadata files and check a field. This would take an incredibly long time to do for large collections and as such would not be of any use if implemented at present.

4. Evaluation

4.1. Overview

For this project evaluation entails justifying the research question, “Is it feasible to create a digital repository system completely in-browser, and is it able to provide at least the basic functionality expected of such a system?”.

In order to answer this question a few evaluation techniques will be needed. In order to answer if is feasible to create a digital repository system completely in browser, compatibility tests were conducted to make sure that the program functions in various Web browsers and on various architectures.

Feasibility also entails the system running in an acceptable time. Testing whether the program runs in an acceptable time was done through performance and user testing.

The efficiency of the online updates was also tested at this stage.

The second part of the research question was whether it could provide at least the basic functionality expected of such a system.

In order to answer this question, user testing was performed with users testing the system to see whether the functionality required is there and that it has been implemented adequately.

4.2. Test 1: Feasibility – Compatibility

4.2.1 Aim

The aim of the compatibility test is firstly to test whether the system itself works, that it was a feasible program and secondly to test the system’s compatibility. The compatibility is important because for a system to be easily distributable it needs to work on a variety of platforms.

4.2.2 Methodology

This test was conducted manually. The system was run on a collection of platforms including various operating systems and Web browsers.

Each aspect of the program was checked to make sure it functioned as expected.

The testing was done on three computers running different operating systems.

System 1: Windows

Model Name: N/A
Processor Name: Intel Core 2 Quad CPU @ 2.66Ghz
Memory: 2 GB DDR 2 1066
Operating System: Windows 7 Ultimate RC

System 2: OS X
Model Name: MacBook Pro
Processor Name: Intel Core 2 Duo @ 2.66 GHz
Memory: 4 GB 1066
Operating System: Mac OS X 10.5.8

System 3: Linux

Model Name: Acer TravelMate C312XMi
Processor Name: Intel Pentium M 740
Memory: 512 MB DDR2 533
Operating System: Ubuntu 9.04

4.2.3 Results

	Windows	OS X	Linux
Firefox 3.5.4	Working	Working	Working
IE 8.0.7100.0	Working	N/A	N/A
Chrome 3.0.195.27	Partially Working*	N/A	N/A
Safari 4.0.3	Working	Partially Working**	N/A

Table 2: Table of system compatibility

*All the JavaScript functions properly in Chrome, however Chrome does not support saving cookies for local files. Therefore, the cookie used to set whether to include online update does not function. The fix for this is to run chrome with the command line flag --enable-file-cookies.

**Safari on OS X works almost perfectly. However, when using large collections some of the browsing links become undefined as Safari is unable to read the array from the generated JavaScript file.

4.2.4 Analysis
The system functions perfectly on Firefox in all operating systems. It works on other browsers but there are some glitches. The problems occur not due to JavaScript incompatibilities but due to quirks in some of the browser's engines.

4.2.5 Conclusion
As the system is able to function on all operating systems and has only small problems in specific cases the system can be considered compatible and functional.

4.3. Test 2: Feasibility – Performance

4.3.1 Aim
The aim of the performance test was make sure that system not only works but also works in an acceptable time. If a system is too slow, user will not want to use it, making the solution infeasible.

4.3.2 Methodology

Performance tests were conducted by indexing and browsing data collections of various sizes and recoding the time taken to do so.

The data collections used in the testing were simple files filled with test data. This allowed for strict control over the number of files as the exact number of files could be generated for each test.

The pre-processor performance testing was conducted by simply running the pre-processor on collections containing various numbers of files and recording the time it takes to pre-process them.

The browsing test was conducted by loading 3 pages from the first 3 browsable fields and averaging the time taken to load the pages. The reason more than one browsable field was used is because the time taken to open a page can be influenced by the number of items in the quick nav and this changes between the browsable fields.

Testing was then done in the same manner on the entire Bleek and Lloyd collection.

DVD testing was then performed using the Bleek and Lloyd collection. The collection was pre-processed, written to and run off a DVD, where it was browsed in the same manner as the local testing, and the load times recorded and averaged. This test was only conducted with one collection size as it is shown that collection size has little effect on browsing time.

A follow-up to this test was conducted during the user testing, where users were asked to comment on the performance.

All performance testing was done on a Mac Book Pro with the following specifications.

Model Name: MacBook Pro
Model Identifier: MacBookPro5,2
Processor Name: Intel Core 2 Duo
Processor Speed: 2.66 GHz
Number Of Processors: 1
Total Number Of Cores: 2
L2 Cache: 6 MB
Memory: 4 GB
Bus Speed: 1.07 GHz

Memory: 2 Modules of
Size: 2 GB
Type: DDR3
Speed: 1067 MHz

Total Memory: 4GB DDR3 @ 1067MHz

The test was done with 10 items per page. The online updated were disabled at this time as they were tested in section 4.4.

4.3.3 Results

Pre-processor

Number of files in Data collection	Time taken to index (s)
10	0.044
100	0.251
500	0.753
1000	1.466
5000	9.784
10000	34.32

Table 3: Time taken to pre-process

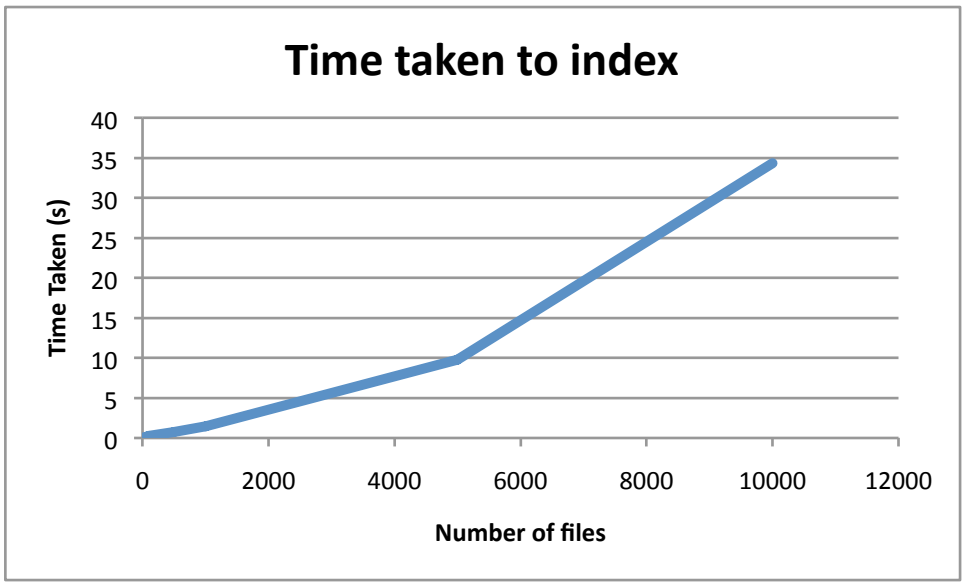

Figure 20: Graph of time taken to pre-process

Browsing

Number of files in Data collection	Time taken to browse (time to open a page) (s)
10	0.01
100	0.018
500	0.018
1000	0.017
5000	0.019
10000	0.022

Table 4: Time taken to browse

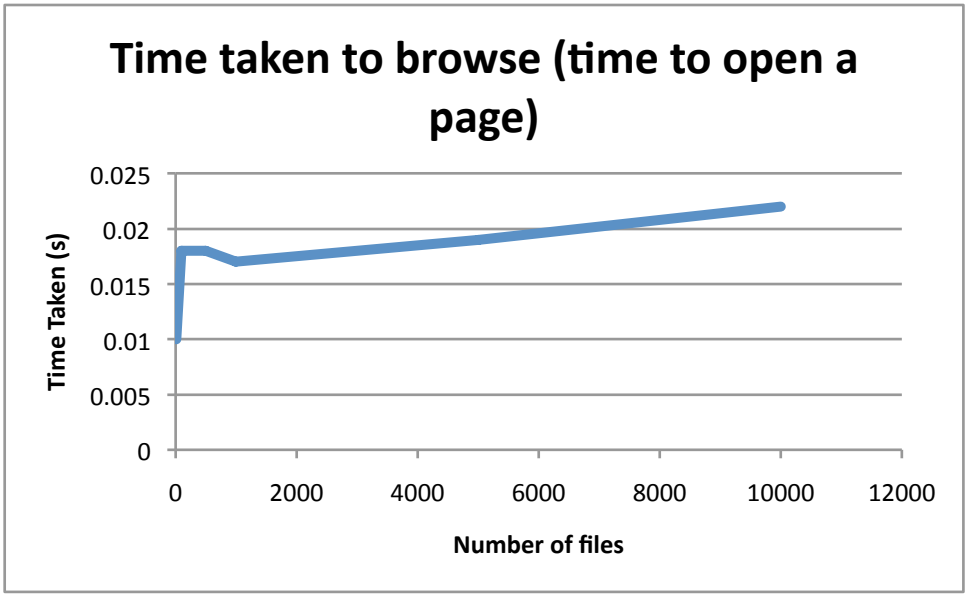

Figure 21: Graph of time taken to browse

Bleek and Lloyd Collection

Time taken to index entire Bleek and Lloyd collection (35 950 files): 109.6 seconds.

Average time taken to browse a page in the Bleek and Lloyd: 0.182 seconds

DVD Testing with Bleek and Lloyd Collection

The following table is of the average time taken to browse a page, using the DVD version of the system, in seconds.

	Windows	OS X
Firefox	0.4158	0.4284
Chrome	0.296	
Internet Explorer	0.999	

Table 5: Time taken to browse on DVD

4.3.4 Analysis

Pre-processor

The pre-processor, as can be seen by the graph, takes progressively longer to index the files. However, even for a 10000 file collection, that time is still below 40 seconds.

Browsing

Due to the way pagination has been done, the system only needs to load from a set number of files to display a page, making the time taken to browse through a collection almost constant. It is less for 10 files as with so few files no pages need to be generated.

DVD Testing

For the DVD testing, browsing took longer as loading files from a DVD is slower than from disk. However, the time taken to load a page of results is still less than a second.

4.3.5 Conclusion

Performance is not an issue for the browsing section as the speed to open up a page of browse results is almost constant. This depends mainly on the size of the quick nav and the medium the files are being loaded from, but even then not varying by more than a second.

The pre-processor, however, does take a longer time for very large collections, but it only needs to be run once, leading to its performance not having an impact on the usability of the system.

4.4. Test 3 – Online Integration

4.4.1 Aim

The aim of this test was assess the efficiency of the online updates. If the updates make the site too slow the gain of having the latest articles may be offset by the loss of usability.

4.4.2 Methodology

In order to determine the efficiency of the online updates, performance testing was done by testing the load times of browsing with various numbers of online updates.

The testing was done using a local stub file. This does not play a role in the results as the only difference between the local stub file and an update from an external machine is the loading time of the website (the update is downloaded when the site is first opened and saved to memory). It thus has no effect on the speed of the browsing.

Calculating the load time was done by browsing through 6 pages in the first sortable category and averaging the time taken to load the pages. As only the difference in browse time due to the number of files in an update is being investigated, the category being browsed is kept constant.

4.4.3 Results

Number of Files in Update	Average Time Taken (s)
0	0.014
5	0.048
10	0.076
20	0.122
50	0.255

Table 6: Time taken to browse with online updates

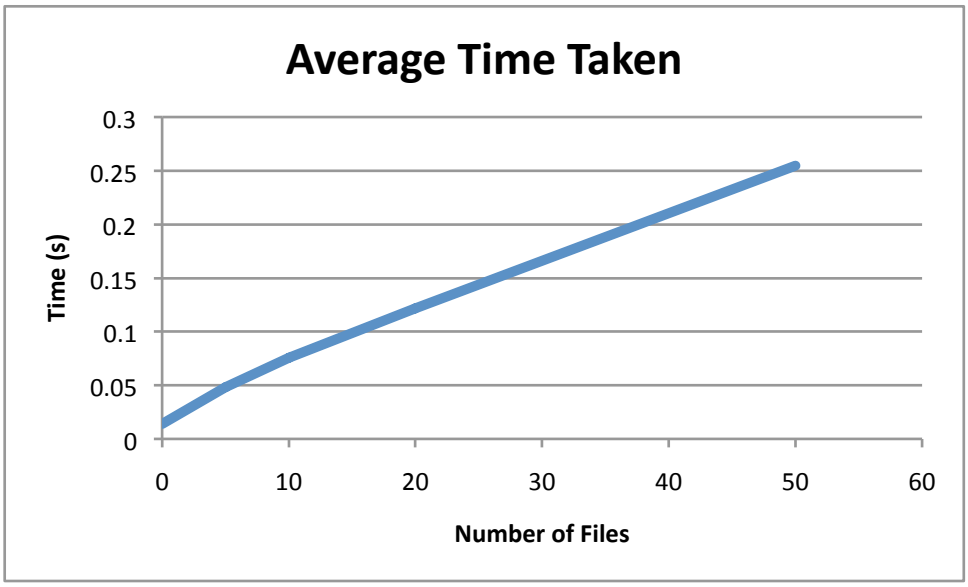

Figure 22: Graph of time taken to browse with online updates

4.4.4 Analysis

The time taken to load a page increases with the number of files in an update.

4.4.5 Conclusion

Due to time taken to browse a page increasing linearly with the number of files in an update, for large updates the responsiveness and performance of the Webpage would be negatively affected and would thus have a negative impact on the usability of the system.

If the updates are small, the online update feature will not have a great impact on the browsing experience. However, if the updates become too large, another method of updating the files permanently would need be to be used.

4.5. Test 4: Functionality – Usability

4.5.1 Aim

A digital repository system is designed to be used by many people, thus the usability of the system is important. The system needs to be easy to understand and use, even for inexperienced users.

The aim is then to test the usability, efficiency, effectiveness and usefulness.

4.5.2 Methodology

Because the system needs to be usable, even by inexperienced users, randomly selected users were asked to use the system with no prior exposure.

Testing users with varying levels of experience and competence with computers was important. However a large majority of the users tested were computer scientists. The other users tested were computer literate but had little or no experience with digital repositories.

First they were asked to simply navigate through the system and were asked for their first impressions. Questions such as how easy they found the system to understand were asked.

Following these simple questions, the users were given a few tasks to make sure they understood the workings of the system and also to give them an idea of what a normal user would want to do with the system.

They were then given a set of questions in the form of statements with answers ranging from Strongly Agree through to Strongly Disagree in order to quantify their understanding and how easy or not the system was to use.

These questions were split up into 3 sections. The first section is questions based on the usability of the system, the second section is questions on the impact of the performance and the third section is on the impact of the online integration.

For the online updates, a small set of updates (10 files) was used.

10 Users evaluated the system.

4.5.3 Results

Question	Strongly Agree	Agree	Neutral	Disagree	Strongly Disagree
1. It is easy to find my way around the site.	4	5	1	0	0
2.I can get to information quickly.	5	4	0	1	0
3.It is easy to remember where to find things.	3	6	1	0	0
4.Screens have the right amount of information.	3	6	0	1	0
5.I always felt I knew what was possible to do next.	4	5	0	1	0
6.Browsing through a digital collection using the system was easy to do.	5	5	0	0	0
7.This system satisfies all the needs of a simple browsing system.	10	0	0	0	0

Table 7: Results of user testing for Section 1.

Question	Strongly Agree	Agree	Neutral	Disagree	Strongly Disagree
1. The site load time was not an issue.	10	0	0	0	0
2. The site was responsive.	10	0	0	0	0
3. Performance was not a hindrance to the browsing experience.	10	0	0	0	0

Table 8: Results of user testing for Section 2

Question	Strongly Agree	Agree	Neutral	Disagree	Strongly Disagree
1. Enabling online updates was easy to do.	10	0	0	0	0
2. Performance did not change after enabling online updates.	10	0	0	0	0
3. There was no noticeable change in the browsing experience after enabling online updates.	10	0	0	0	0

Table 9: Results of user testing for Section 3

4.5.4 Analysis

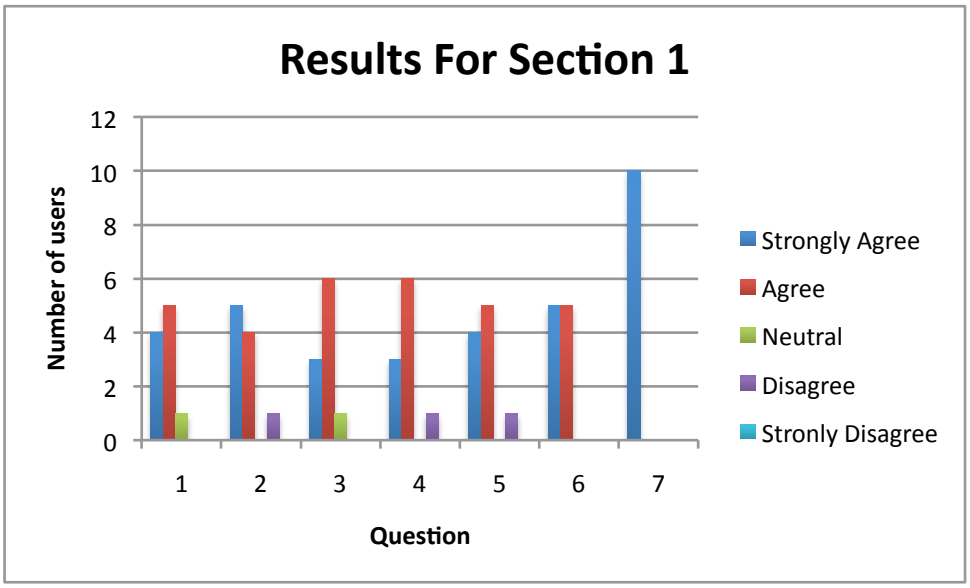

Figure 23: Graph of user testing results for section 1

From the distribution of the results seen in the above figure, it is clear that the users found the system easy to use as well as functional, as most of the results lie in the Agree and Strongly Agree region. The result of question 7 shows that the users unanimously thought the system had all the functionality of a simple browsing system.

This aids in answering the research question, which was if it was possible to provide the basic functionality of a browsing system without a Web server.

The disagree results in question 2 was due to the user not noticing the quick nav, and as such struggling to find the articles.

The disagree result in question 4 was due to the user believing that a page should have had more than 10 results. This however can be configured in the pre-processors settings.

As for the disagree result in section 5, the user, believed that the site should have built-in back buttons, not having to rely on only the browser's functionality to make navigation easier.

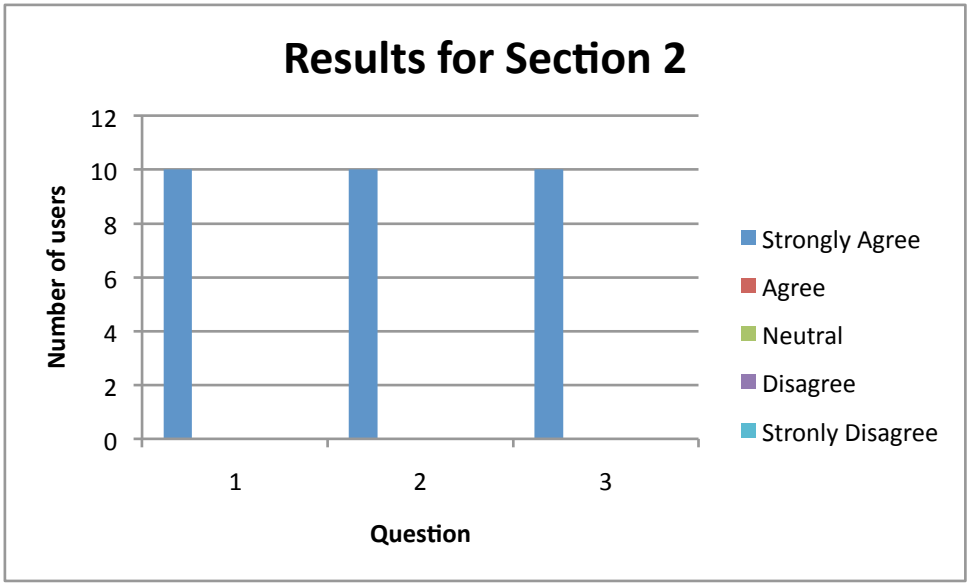

Figure 24: Graph of user testing results for sections 2

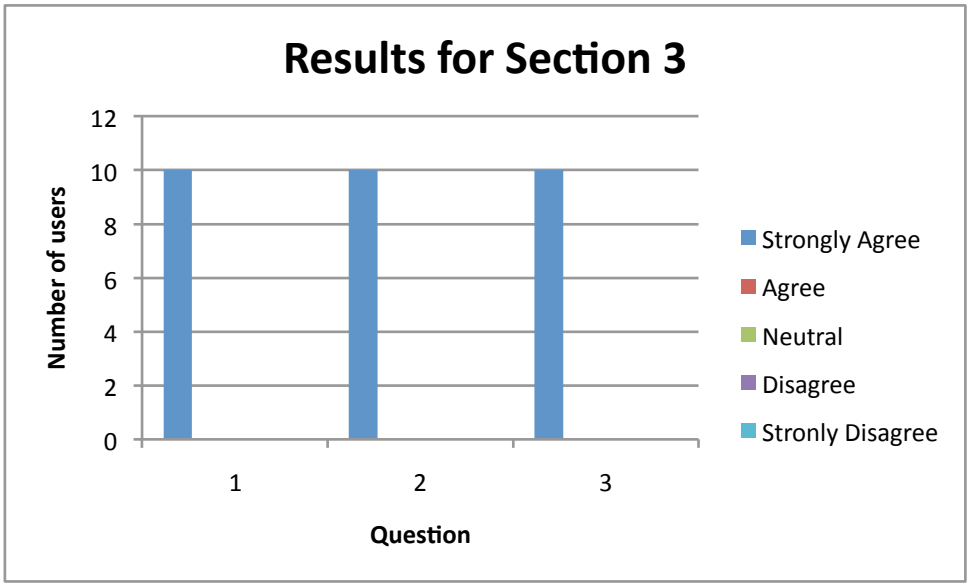

Figure 25: Graph of user testing results for sections 3

For section 2 and 3 as seen in the above figures, it was very clear that users had no problems with performance or the online update as they all strongly agreed that both these factors had no impact on the browsing experience.

4.5.5 Conclusion

From the user testing one can see that the system has the functionality required for a browsing system and that users found the system easy to use.

Also performance was not an issue for the users and the online updates had little impact on the experience.

4.6. Evaluation Conclusion

The system is functional and works on a range of operating systems and Web browsers. This answers the first part of the research question, that the system is feasible.

Secondly, the performance testing along with the user testing on performance shows that performance was not a hindrance to the users. Even on a DVD the load times were still less than half a second on most browsers.

The online updates with small sets of files did not hinder the browsing experience, but if large sets of updates were to be used it could impact negatively on the usability.

The user testing showed that even inexperienced users found the system responsive and easy to use and believed that the system had all the necessary functionality for a simple browsing system.

These test results would suggest that the research question of “Is it feasible to create a digital repository system completely in-browser, and is it able to provide at least the basic functionality expected of such a system?” has been answered, that it is feasible and that the system does provide the functionality needed.

5. Conclusion

The problem of browsing in digital repositories, specifically the problem of replicating the browsing functionality in a lightweight distributable digital repository system was investigated. The background of digital repository systems as well as the AJAX technology used to create the browsing system has been researched in order to better understand the problem of creating an AJAX repository.

Based on the goal of creating a lightweight and distributable digital repository system, an experimental browsing system was designed and implemented that can be used offline and online with no extra software needed other than a JavaScript compliant Web Browser. It showed that although AJAX is a good technology for creating lightweight applications it has limitations in that it cannot browse or write to a local filestore.

Another limitation is that JavaScript is not a particularly fast language, especially when it comes to reading files, as such functions that require a great deal of calculation and reading of data end up taking a very long time.

Due to these limitations extra work had to be done in the form of a pre-processor which converts the digital collections into a form the JavaScript in the system can use to browse the collection, allowing the JavaScript to deal with only a part of the collection at a time.

Once the pre-processing was complete, however, the system was functional and easy to use. This was shown through compatibility, performance and user testing.

The limitations of JavaScript were overcome and the system scored well with users during evaluation.

Online updates were shown to be functional and have little impact on the browsing experience, but only for small updates. A more permanent and long term solution to online updating will need to be found due to JavaScript's read only limitations.

In conclusion, based on the research question, the system was successful in showing that it is feasible to create a general, lightweight browsing system. However, due to the limitations of JavaScript certain functions are not trivial to implement and even some basic functions such as including updated files require a great deal of jumping through hoops to include.

5.1 Future Work

As technology is always moving forward many of the limitations of JavaScript that create complications today may be solved in the future, making this method of creating digital repository system even more feasible.

Already Firefox has created a way to write files using JavaScript, which by itself would solve many of the limitations in creating digital repository systems.

Right now, though, writing files in most browsers using JavaScript is impossible, thus one task for the future would be to create an external program to update the digital collection and re-index it. This could possibly be included in the pre-processor. This would provide a more long term solution to online integration.

Support for hierarchical collections could be added. As the system stands now, only physical files can have metadata, so the metadata has to be stored at the lowest level. This creates a lot of redundant data as files in a collection cannot be organised hierarchically and inherit certain aspects from their parents.

This also makes it so that a user must browse at the lowest level. An example of hierarchical browsing in terms of the Bleek and Lloyd collection would be for the user to be able to browse the stories as items and only view the pages when a story is selected. As it stands now the pages are the actual files and as such are the only items that can be browsed.

More configurations for collections could be added. Configurations such as date formats could be included. This would make browsing through dates more user friendly.

Other functionality that could be implemented would be to improve the UI of the system, possibly including more JavaScript animation to make the site more user-friendly. An example of this would be to create thumbnails of picture files instead of simple links, etc.

6. References

[1] P. Achananuparp and R. B. Allen, “Developing a student-friendly repository for teaching principles of repository management” In *DigCCurr: International Symposium in Digital Curation,* 2007

[2] B. Alman, “jQuery URL Utils: Query String, Fragment and more”, [Online], Available: http://benalman.com/projects/jquery-url-utils-plugin/. [Accessed: November. 5, 2009]

[3] P.A. Bernstein and U. Dayal, “An overview of repository technology” in Proceedings of the 20th International Conference on Very Large Data Bases, Morgan Kaufmann Publishers, San Francisco 1994, pp. 705–713.

[4] C. Duncan, “Digital Repositories: e-Learning for Everyone,” presented at eLearnInternational, Edinburgh, February 9-12, 2003

[5 J.J. Garrett, “Ajax: A New Approach to Web Applications”, [Online]. Available: http://adaptivepath.com/publications/essays/archives/000385.php. [Accessed: May. 5, 2009]

[6] M. Greensade, “Development of web applications using AJAX”, BSc Thesis, University of Portsmouth. [Online]. Available: http://dissertations.port.ac.uk/142/01/GreensladeM.pdf. [Accessed: November. 5, 2009]

[7] M.J. Rees, “Ultra Lightweight Web Applications: A Single-Page Wiki employing a Partial Ajax Solution”, [Online], Available http://ausweb.scu.edu.au/aw06/papers/refereed/rees/paper.html. [Accessed: May. 5, 2009].

[8] M. Smith et al., “DSpace An Open Source Dynamic Digital Repository”, D-Lib Magazine, vol. 9, no.1, [Online]. Available: http://www.dlib.org/dlib/january03/smith/01smith.html. [Accessed: May. 5, 2009].

[9] H. Suleman, “Digital Libraries Without Databases: The Bleek and Lloyd Collection”, 2007. [Online], Available: http://pubs.cs.uct.ac.za/archive/00000433/01/ecdl_2007_dlwd.pdf. [Accessed: May. 5, 2009].

[10] H. Suleman, “in-Browser digital library services”, 2007. [Online], Available: http://pubs.cs.uct.ac.za/archive/00000434/01/ecdl_2007_ajax.pdf. [Accessed: May. 5, 2009].

[11] H. Suleman, “AJAX Repository – Department of Computer Science”, March. 9, 2009. [Online]. Available: http://www.cs.uct.ac.za/teaching/honours/honours-projects-2009/ajax-repository/. [Accessed: May. 5, 2009].

[12] H. Suleman, CSC4000W DL lectures, University of Cape Town, 2009

[13] R. Tansley et al., “The DSpace Institutional Digital Repository System: Current Functionality” In Proceedings of *The 3rd ACM/IEEE-CS joint conference on Digital libraries*, 2003, pp. 87-97

[14] I.H. Witten, D. Bainbridge, S.J. Boddie, “Power to the people: end-user building of digital library collections” in Proceedings of the 1st ACM/IEEE-CS joint conference on Digital libraries, Roanoke, Virginia, United States, January 2001, p.94-103

[15] Unknown. “jQuery: The Write Less, Do More, JavaScript Library”, [Online], Available: http://jquery.com/. [Accessed: November. 1, 2009]

[16] Unknown. “HTML and URLs”, [Online], Available: http://www.w3.org/TR/WD-html40-970708/htmlweb.html. [Accessed: November. 5, 2009]