2010

Bookmarklet Builder for Offline Data Retrieval

Sheetal Naidu
San Jose State University

Bookmarklet Builder for Offline Data Retrieval

A Writing Project

Presented to

The Faculty of the Department of Computer Science

San José State University

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

By

Sheetal Naidu

May 2010

SAN JOSÉ STATE UNIVERSITY

The Undersigned Writing Project Committee Approves the Writing Project Titled

BOOKMARKLET BUILDER FOR OFFLINE DATA RETRIEVAL

by Sheetal Naidu

APPROVED FOR THE DEPARTMENT OF COMPUTER SCIENCE

_____

Dr. Chris Pollett, Department of Computer Science Date

_____

Dr. Mark Stamp, Department of Computer Science Date

_____

Dr. Robert Chun, Department of Computer Science Date

ABSTRACT

BOOKMARKLET BUILDER FOR OFFLINE DATA RETRIEVAL
By Sheetal Naidu

Bookmarklet Builder for Offline Data Retrieval is a computer application which will allow users to view websites even when they are offline. It can be stored as a URL of a bookmark in the browser. Bookmarklets exist for storing single web pages in hand-held devices and these web pages are stored as PDF files. In this project we have developed a tool that can save entire web page applications as bookmarklets. This will enable users to use these applications even when they are not connected to the Internet. The main technology beyond Javascript used to achieve this is the data: URI scheme. With the data: URI scheme we can embed images, Flash, applets, PDFs, etc. as base64 encoded text within a web page. This URI scheme is supported by all major browsers and in Internet Explorer from version 8 onwards. The application could be made available online, to users who are typically website owners and would like to allow their users to be able to view their websites offline.

ACKNOWLEDGEMENTS

I take this opportunity to express my sincere gratitude towards my project guide Dr. Chris Pollett. His help and guidance have led me towards completing this project. I thank him for his invaluable patience and for helping me overcome technical difficulties throughout the project.

I am very thankful to my committee members, Dr. Mark Stamp and Dr. Robert Chun for their time and feedback.

I would also like to thank my family and friends for their continued support. Raghu, Reena, Ram, Madhavi M., Balki, Madhavi D., Nilesh, Kanaka, Smita and Bhuvi have had strong direct and indirect influence in helping me complete my project. I would also like to thank my Mom, Dad and Deepti for believing in me. Sarika and Rohan have been a great inspiration throughout.

Table of Contents

1. Introduction 1
2. Technologies Used 2
    2.1 data:URI 2
    2.2 Document Object Model 4
    2.3 Nutch 7
3. Preliminary Work 9
    3.1 Javascript 9
    3.2 Using data:URI 10
    3.3 Working with Nutch 12
    3.4 Code Obfuscation 15
4. System Implementation 18
5. User Interface of Bookmarklet Builder 18
6. Integrating Nutch 19
7. Converting Individual Pages to data:URI 20
    7.1 Converting Images 21
    7.2 Converting Links 22
    7.3 Converting CSS Files 23
    7.4 Converting Javascript Files 24
8. Advantages and Disadvantages 25
    8.1 Advantages 25
    8.2 Disadvantages 26
    8.3 Note about Recursive data:URI Conversion 26
9. Performance Testing and Analysis 26
    9.1 Time Based Test Results 27
    9.2 Length of URI 29
    9.3 Differences Between Firefox and Opera 29
10. Conclusions 31
11. Future Enhancements 32
12. References 33
13. Appendix A 35

List of Figures

Figure 1: Sample Code to demonstrate DOM structure 5
Figure 2: DOM Tree Structure 6
Figure 3: Page Structure of Sample Website 8
Figure 4: Javascript Demonstration 10
Figure 5a: Original Web Page 11
Figure 5b: Web Page with Images converted to data:URI 11
Figure 6: <img> tag 12
Figure 7: UI of Sample Nutch Usage 13
Figure 8: Output After running Nutch 14
Figure 9 : Sample input to Stunnix 16
Figure 10: Obfuscated Code Output by Stunnix 17
Figure 11: UI of Bookmarklet Builder 19
Figure 12: Algorithm to Convert Individual Web Page 20
Figure 13a: Original <img> tag 21
Figure 13b: data:URI of <img> tag 21
Figure 14: HTML code generated by PHP program 22
Figure 15a: Original <link> tag for CSS file 23
Figure 15b: <link> tag with data:URI for CSS file 23
Figure 16a: Original <script> tag for Javascript file 24
Figure 16b: <script> tag after data:URI is inserted 24

List of Tables

Table 1: Time Based Performance Results …..... 27
Table 2: System Performance based on Depth of Crawl …..... 28
Table 3: System Performance based on Type of Web Pages …..... 28
Table 4: URI length Proportional to Number of Pages …..... 29

1. Introduction

Bookmarklet Builder for Offline Data Retrieval is a system that lets you create a bookmarklet and add it to a browser's bookmarks. The created bookmarklet can be used to view web pages when not connected to the Internet. In this system, the bookmarklet that is created is a data:URI (data: Uniform Resource Identifier) of a web page or a set of web pages. One main web page is connected to other pages or has links to other web pages which are also converted to data:URI. A data:URI is a URL (Uniform Resource Locator) scheme which allows for including small data items such as text, images, etc. as immediate data items in a web page. The “immediate” inclusion is as if the data had been included externally. With this scheme, a web page would have a data:URI instead of an external link pointing to the source. In the general definition of a bookmarklet, it could be a Javascript program wrapped around a string of HTML code performing some action once it is loaded in a browser. It could be saved as a bookmark in any browser. In this project the system is designed such that it saves an entire website as a single, long string of a data:URI.

Currently, many tools exist that convert web pages to PDF format. There are also many online tools that convert web pages to data:URI. These systems work only on a single page at a time. It is similar to saving a file in a browser's cache but it also allows to use the URI within another web page. If a user wishes to save multiple such files, like multiple pages of a particular website, the connection or links between the pages is not maintained. For example clicking on a link in the data:URI of Page A which is connected to Page B, will not open the data:URI of Page B.

Many users use the feature of saving their PowerPoint presentations as HTML files and publish them. These files have basic HTML code wrapped around a GIF or JPEG image of the presentation slides. If these files can be converted into data:URI, then they could be easily transported onto any computer and you would not need PowerPoint or even Internet connection and would be able to present the slide show from your computer anytime.

On some hand-held devices, entire web sites are converted to PDF. The idea of this project is based on a similar idea where web pages are converted to data:URI. The conversion is such that links on all the pages are still maintained; while at the same time, images and text content are converted appropriately. Once a set of web pages are converted this way, users can bookmark the initial page and view the bookmarklet any time later. Even when they go offline, they will be able to browse through the site as if they were online.

The project is a web-based system and has a front-end and a back-end. In the front-end there a basic UI through which the user gives input. At the back-end, there are mainly two parts. The first part is a crawler used to crawl the given website and the second part involves functions written in PHP that convert the required HTML elements to data:URI. This report explains how the system was developed. In Section 2 we describe the main concepts and technologies that are deployed in the project. In Section 3, we discuss about the preliminary work that was conducted in order to lay the foundation for the implementation. Sections 4, 5, 6 and 7 describe how the system was actually implemented and go into details of the UI, integration of the crawler module and algorithm for converting web pages to URI. Section 8 discusses the main advantages and disadvantages of such a system. Following that is Section 9 which describes test scenarios and analyzes the results. Section 10 concludes the report.

2. Technologies Used

Beyond Javascript and PHP, the important concepts that are used in this project are data:URI and Document Object Model. In the following sections more light is thrown on the two concepts. There is also an introduction to Nutch, which is the crawler that is used in this project.

2.1 data: URI

In order to better understand what a data:URI scheme is, it is required to understand a URI in general and then look at how a URL is a subset of the URI scheme.

A Uniform Resource Identifier (URI) is a compact string of characters for identifying an abstract or physical resource. URIs provide a simple and extensible means for identifying a resource. URIs provide a uniform way of identifying different resources which could be used in the same contexts while the means to access those resources may be different. A resource in a URI could be anything that has identity. It could be a text file, an image file or even a resource like a shopping cart on an e-commerce website. The definition of a resource does not limit it to be retrievable through a network. A resource is only a conceptual mapping to an entity at any given instance of time. Thus, the entity itself can change over time as long as the mapping remains the same. An identifier in URI is an object that acts as a reference to an accessible source that has identity. Such an object is a sequence of characters with defined syntax.

The URI syntax is dependent upon the scheme. In general, absolute URI are written as follows:

<scheme>:<scheme-specific-part>

An absolute URI contains the name of the scheme being used (<scheme>) followed by a colon (":") and then a string (the <scheme-specific- part>) whose interpretation depends on the scheme. Below is an example of a URI

ftp://ftp.is.co.za/rfc/rfc1808.txt

-- ftp scheme for File Transfer Protocol services

A URL is a URI scheme which identifies a resource mainly by the way it is accessed. That is, its network “location”. Although URLs are named after a protocol, it does not imply that that URL's resource is accessible only via that named protocol. For example, a URL with 'http' scheme uses both DNS and HTTP protocols.

A data URI is a URL scheme which provides a way of including small data objects as immediate data in a web page rather than specifying the object as an external resource. The basic syntax of such URLs is

data:[<mediatype>][;base64],<data>

<mediatype> is an Internet MIME with optional parameters. In the absence of mediatype, the default is text/plain;charset=US-ASCII. It is possible to also omit "text/plain" but the charset parameter should be supplied. If ";base64" is present it implies that the data is base64 encoded. If ";base64" is not present, it means that the data is represented as ASCII encoding.

2.2 Document Object Model (DOM)

In order to convert links and images in each of the web pages to a suitable data:URI form, we need to convert the web pages to the Document Object Model representation. DOM provides a language independent platform to access the properties and elements of a web page. It is an Application Programming Interface to represent and manipulate the content of HTML and XML documents. Thus, when we convert a web page to a DOM representation, we can access individual elements of the page and perform actions to modify the elements. It is also possible to add, delete or replace individual elements of the document. The structure of DOM can be viewed as a tree. Thus, the elements in an HTML document can be viewed as nodes of the tree. Consider the following HTML code.

```
<html>
        <body>
            <h2>List of files</h2>
            <ol>
                <li>Readable file</li>
                <li>Writable file</li>
                <li>Executable file</li>
            </ol>
        </body>
</html>
```

Figure 1: Sample Code to demonstrate DOM structure

The graphical representation of the above code can be represented by the tree diagram shown below.

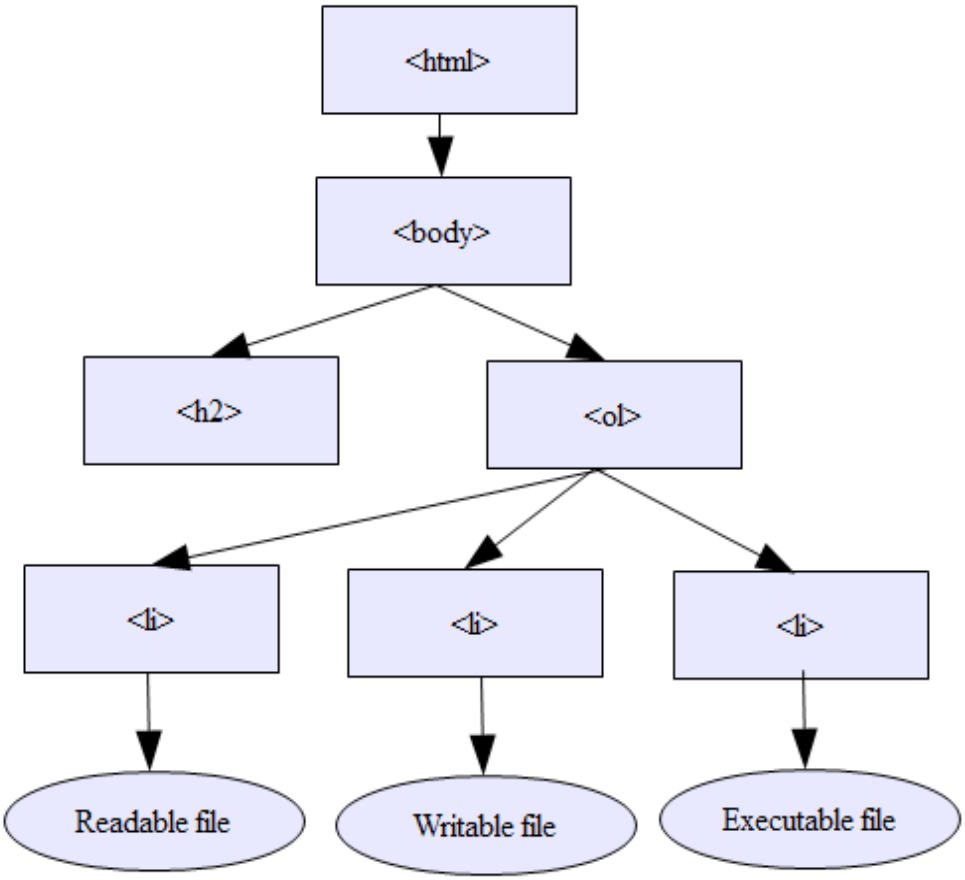


Figure 2: DOM Tree Structure

Each of the nodes can be accessed from this tree structure. Using constructs from any language, it is possible to change, add or delete nodes from this tree. The above figure is only a graphical representation which describes the logical structure of the HTML code. The tree structure is only a choice and is not necessarily the only way a DOM document can be represented. The DOM is a logical model that may be implemented in any structure and does not specify a particular manner to do so.

The Document Object Model was originated in order to provide a standard to make Javascript scripts and Java programs portable across different browsers. It provides a way for any Javascript code to look at its

containing HTML document. Thus, serving a way for the Javascript to access and manipulate its containing web page. The intention of the DOM was to provide a way to convert documents to objects so that they can be used in object oriented programs. It is important to note that the tree structure is not how the internal data structure of the document is; it is only a logical view of the parent-child relationships defined by the programming interface of that document. For XML files, the information set is defined by the XML information set that that file uses. The DOM is only an API to that information set.

In this project, PHP constructs are used to convert HTML pages into their DOM representations, which are referred to as DOM documents. Each such DOM document is defined by a DOMDocument class which represents an entire HTML file and which serves as a root of the document tree. Any node belonging to such a tree can be accessed using XPath queries. For example, to find all the link tags in a document, we can use the query "/home/body//a"

This will return all <a> tags that are nested inside the <body> tag, inside the <html> tags. For each of these <a> tags we can access their attributes and change them such that they do not link to pages on the WWW but instead link to data structures stored in the project. After the changes have been made, the DOM documents can then be converted back into their HTML format.

2.3 Nutch

The crawler used in our system is part of a search engine called Nutch. Nutch is an open source Java search engine. For our system, only the crawling capabilities of Nutch are used. Using Nutch involves using its various commands in a sequence. In our system, the main commands that are used to crawl and then read the necessary related data are the '*crawl*' command and '*readdb*' command.

As a case study, lets use the following example of a simple website. We will use this site to first crawl and then convert the web pages to data:URI form.

Page A is linked to Page B and Page D.

Page B has links to Page A, Page C and Page D.

Page C has link to Page B.

Page D has link to Page B.

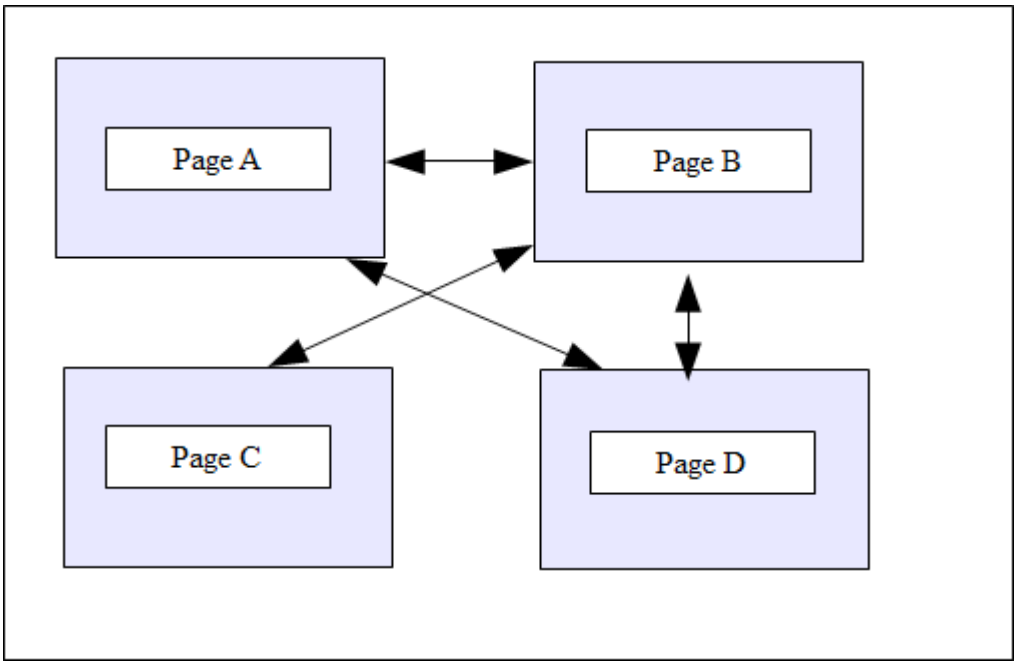


Figure 3: Page Structure of Sample Website

Once the user enters the URL of the home page, that is, PageA.html, the system creates a flat file which contains this URL. This file is used by Nutch for its crawl purpose. One significant setting is the domain which the crawl needs to be restricted to. This information is added to the Nutch configuration file. This piece of information is important because this will restrict Nutch from crawling pages that are outside of the specified domain. Without this detail the crawler would add links that go out of the domain of the website which might be unnecessary web pages. A typical example might be links to web sites through ads on the home page.

Nutch's *crawl* command involves the following options. A flat file with the URL of the home page, a destination directory where all the crawled data is stored, the depth of the crawl and the number of links to be

fetched at each depth. In our example website, when depth is 1, only pages A, B and D are fetched. Page C is omitted because it is reachable from Page A at depth 2. When depth is 2, all four pages are fetched.

From the crawl data, we gather a list of web pages that were crawled and using this list, we convert each page to the data:URI form. The command used to generate this list is the "*readdb*" command.

3. Preliminary Work

Writing Javascript programs, researching about what data:URI is and finding a suitable crawler for the project were part of preliminary work for the project. In the following sections we will discuss about what was done while researching about the necessary tools needed to implement the project, and how the selected tools would be used.

3.1 Javascript

Javascript is an essential part of implementing this project. As part of foundation, learning to write code in Javascript was one of the initial and concrete deliverables. In order to demonstrate the same, a program that demonstrated user interaction with an HTML object was written using Javascript. In this program, there was a black box on an HTML page and this black box was defined using tags and not <img> tags. There were four buttons below the black box that allowed users to move the box around the screen. Each click moved the box by 15pts in the direction specified by the button. e.g Top moved the box 15pts in the top direction, Left moved the box 15pts in the left direction and so on.

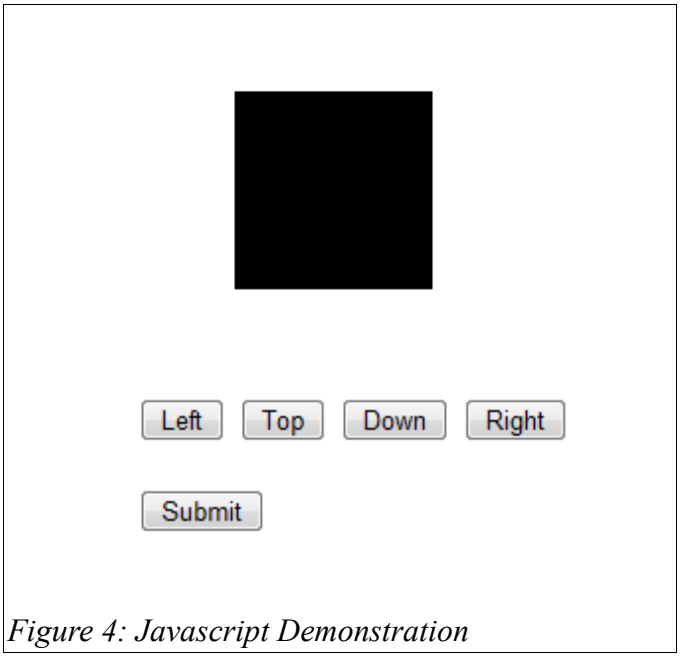


Figure 4: Javascript Demonstration

By clicking on Submit, the user is taken to the next page which displays the x and y coordinates of the last position of the black box.

Ex: The x coordinate is : 525px and the y coordinate is: 400px

This deliverable helped learn about the use of tags, accessing the query string from a URL and using regular expressions for pattern matching. On the first HTML page there were two hidden form elements which contained the present x and y coordinates of the black box. When we hit on Submit, these two values were sent as part of the query string to the next page. On the second page, these values were retrieved from the query string and displayed in the browser.

3.2 Using data: URI

As part of preliminary work, a complete PHP script was written that took the URL of a page as input from the user and fetched the page using cURL() function in PHP. It then parsed it to see if there were any image files embedded in the file. It then fetched each of the image files from their address, again using cURL() and then

base64 encoded these image files individually and displayed the images back in the browser in the data:URI scheme.

The format for using data in the data:URI form is

data:[<MIME-type>][;base64],<data>

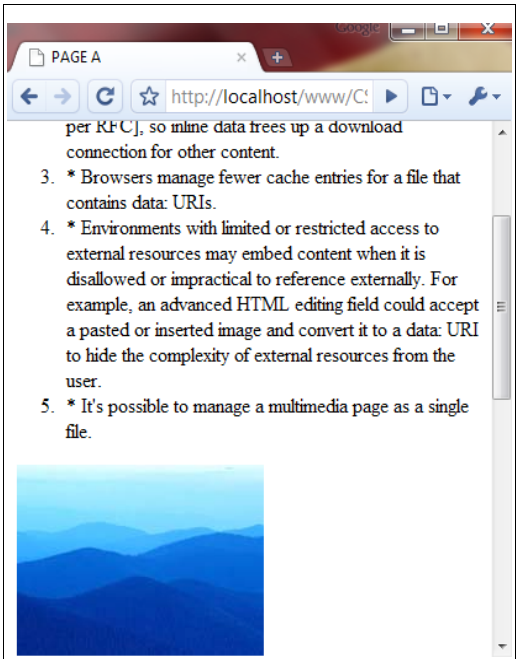


Figure 5a: Original Web Page

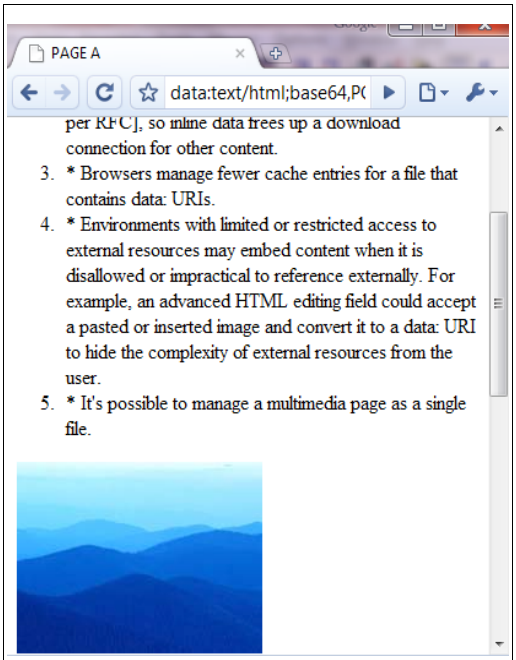


Figure 5b: Web Page with Images converted to data:URI

The two figures shown above, show the HTML pages before and after the base64 encoding has been applied to the image files. Although they appear the same in the browsers, their sources look significantly different.

For Figure 5a, on the left side, the source code looks like:

```
<img src="http://localhost/CS298/Images/Image1.jpg" />
```

Figure 6: <img> tag

Where as for the new page in Figure 5b, on the right side, the source code of the <img> tag looks like:

```
<img
src="
wESAAMAAAABAAEAAAEaAAUAAAABAAAAYgEbAAUAAAABAAAAagEoAAMAAAABAAIAAAExAAIAAAA
UAAAAcgEyAAIA7AAAUAAAAhodpAAQAAAABAAAAnAAAAMgAAAEsAAAAAQAAASwAAAABQWRvYmU
gUGhvdG9zaG9wIDcuMAAyMDA3OjExOjI3IDEyOjAzOjM4AAAAAAOgAQADAAAAAf//AACgAgAEAAAA
AQAAAMigAwAEAAAAAQAAAJYAAAAAAAAABgEDAAMAAAABAAYAAAEaAAUAAAABAAABFgEbAAUAA
AABAAABHgEoAAMAAAABAAIAAAIBAAQAAAABAAABJgICAAQAAAABAAAGJwAAAAAAAABIAAAAAQA
AAEgAAAAB/9j/4AAQSkZJRgABAgEASABIAAD/7QAMQWRvYmVfQ00AAv/uAA5BZG9iZQBkgAAAAH
/2wCEAAwICAgJCAwJCQwRCwoLERUPDAwPFRgTExUTE...” />
```

The above example is shortened for this report. The actual base64 encoded representation of a typical image file is bigger than the string that is shown above. Appendix A provides more examples of data:URI.

3.3 Working with Nutch

Nutch was chosen for crawling purpose because it is an open source project and hence available freely and is fairly straight forward to install and use. Nutch is a web crawler written in Java which is part of an open source search engine which uses Lucene for its search and index component. Nutch was previously a part of Apache but now it is a sub-project of Lucene. This preliminary work was conducted in two parts. In the first part we learned to install Nutch and use it from the command line. Nutch maintains a database of pages and links that it fetched during the crawl. The pages have scores that are assigned by analysis. Every time the crawl is made, it looks for high-scoring, out-of-date pages and fetches them to update its database.

The second part of this deliverable was to write PHP code so that Nutch commands could be run from within this script. For this, the user was presented with a front-end where he/she would enter the URL of the site to

be crawled. This URL was given as input to Nutch's *crawl* command and the crawled data was stored in a directory. Next, some useful data was fetched from the database and displayed to the user in human-readable form. This data is extracted using *readdb* command in Nutch. This command could be used to read data such as the URLs of all the pages visited during the crawl.

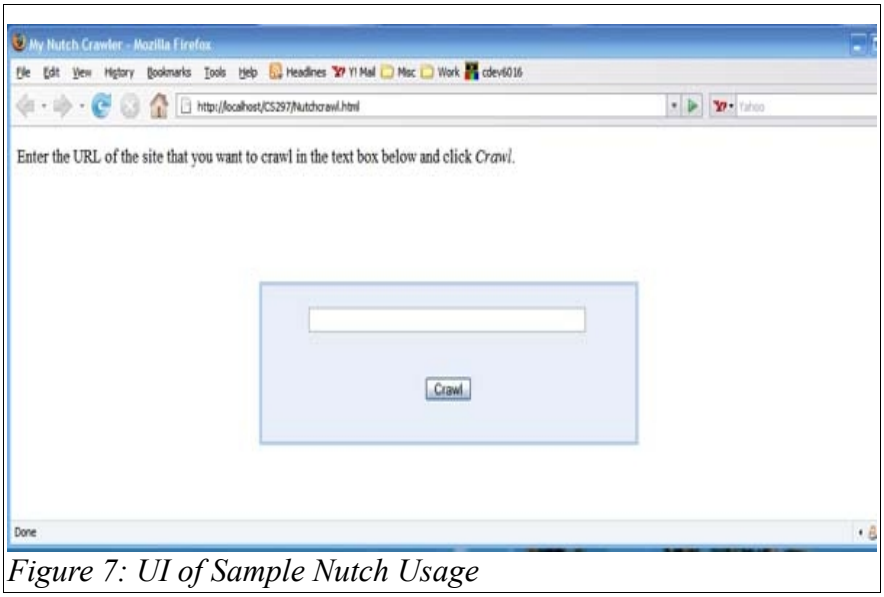


Figure 7: UI of Sample Nutch Usage

The front end provided to the user to enter the crawl site is shown in the screen shot above. When the user enters the URL of the site in the text box and clicks on Crawl, a PHP script is invoked which crawls the site with the URL provided and stores the data in a directory. After the crawl is completed, the user is shown the following message.

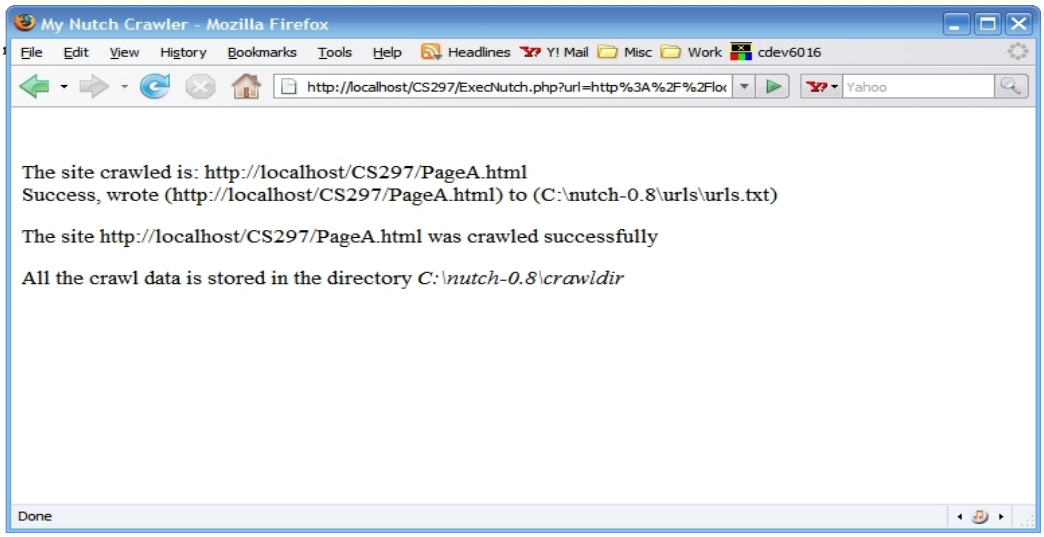


Figure 8: Output After running Nutch

Then, when we used the *readdb* command, it displayed the following information.

http://localhost/CS297/PageA.html Version: 4

Status: 2 (DB_fetched)

Fetch time: Fri Dec 07 16:28:34 PST 2007

Modified time: Wed Dec 31 16:00:00 PST 1969

Retries since fetch: 0

Retry interval: 30.0 days

Score: 1.6666667

Signature: e48ea88ce7aaa83d3115c598205ea05e

Metadata: null

http://localhost/CS297/PageB.html Version: 4

Status: 2 (DB_fetched)

Fetch time: Fri Dec 07 16:28:44 PST 2007

Modified time: Wed Dec 31 16:00:00 PST 1969

Retries since fetch: 0

Retry interval: 30.0 days

Score: 2.0

Signature: a1c8ea6e230e235c5ff1acb2be4bd202

Metadata: null

http://localhost/CS297/PageC.html Version: 4

Status: 1 (DB_unfetched)

Fetch time: Wed Nov 07 16:28:46 PST 2007

Modified time: Wed Dec 31 16:00:00 PST 1969

Retries since fetch: 0

Retry interval: 30.0 days

Score: 1.6666667

Signature: null

Metadata: null

Different options supplied to the *readdb* command provide different kinds of output.

3.4 Code Obfuscation

At the time of conception, the data:URI to be presented to the user was visioned to be wrapped around Javascript and presented as a Javascript function. Thus, we wanted to obfuscate the code using Javascript obfuscation techniques. As the project developed and took shape, we discarded the idea of presenting the data:URI string as a Javascript function. However, in this section we discuss about some research done with regards to Javascript obfuscation techniques.

Obfuscation could be used so that the bookmarklet can potentially be sold without easily being reverse engineered. Part of the research was to get information on the available techniques for code protection. These techniques include

- watermarking – which is a defense against software piracy
- tamper-proofing – which enables the detection of tampered code so that the code becomes unusable, and
- obfuscation - which is a tool for obfuscating code so that it becomes difficult to reverse engineer.

Code is obfuscated mainly by applying certain transformations to the original code. On further study we found out how code can be obfuscated in different ways. There are several ways that code can be obfuscated in. These ways depend on what area of the code we aim at obfuscating. These include:

- Layout transformation – modify variable names
- Data transformation – modify data structures
- Control transformation – change program flow while preserving semantics and
- Preventive transformation – use anti-debugging and anti-disassembly techniques

Most commercially available Javascript obfuscators obfuscate the code only by aiming at transforming the lexical structure of the code or by changing the layout of the code. For our deliverable, we picked a freeware called Stunnix. As an example, let's examine how a piece of code is obfuscated using Stunnix.

Sample code:

```
function foo( arg1)

{

var myVar1 = "some string"; //first comment

var intVar = 24 * 3600; //second comment

/* here is

a long

multi-line comment blah */

document.write( "vars are:" +myVar1+ " " +intVar +" " +arg1);

};
```

Figure 9 : Sample input to Stunnix

The obfuscated code looks as follows:

```
function z001c775808( z3833986e2c) { var z0d8bd8ba25=

"\x73\x6f\x6d\x65\x20\x73\x74\x72\x69\x6e\x67";

 var z0ed9bcbcc2= (0x90b+785-0xc04)*

(0x1136+6437-0x1c4b);

document. write( "\x76\x61\x72\x73\x20\x61\x72\x65\x3a"+ z0d8bd8ba25+

"\x20"+ z0ed9bcbcc2+ "\x20"+ z3833986e2c);};
```

Figure 10: Obfuscated Code Output by Stunnix

We can make the following observations about how Stunnix works:

- The Stunnix obfuscator targets at obfuscating only the layout or the lexical structure of the Javascript code
- As the obfuscator parses the code, it removes spaces, comments and new line feeds
- While doing so, as it encounters user defined names, it replaces them with randomly generated strings
- It replaces print strings with their hexadecimal values
- It replaces integer values with complex equations

In our sample code that was obfuscated, we can observe that the user defined variable foo was replaced with the random value z001c775808; the variable arg1 was replaced with z3833986e2c; the variable myvar1 was replaced with z0d8bd8ba25 and the variable intvar was replaced with z0ed9bcbcc2. The integer value 20 was replaced with (0x90b+785-0xc04) and 3600 replaced with (0x1136+6437-0x1c4b). The print strings “vars are” was replaced with its hexadecimal value \x76\x61\x72\x73\x20\x61\x72\x65\x3a and all spaces were replaced with the

hexadecimal value of \x20. We also saw that all comments and new line feeds were removed from the resulting code. While this kind of code obfuscation makes the resulting code more compact it also has an overhead of having to decode the equations wherever we need to use simple numerical values. But this only adds a constant extra time.

Although any obfuscation techniques were not used in implementing the project, more study might help us understand whether it would be a useful enhancement to the project. One possible use would be to wrap the final resulting URI within some Javascript and then obfuscate that code. Since the final resulting data:URI is a very long string of characters, we could use Javascript to compress this long string and/or break it into smaller sections. Along with this we would also need suitable code that would do the opposite of the above so that the URI gets displayed properly in a browser. It is all this Javascript code that could then be obfuscated.

4. System Implementation

Since the project has a web-based design, there is a web page designed to take input from the user. The back-end of the implementation of the project contains two major parts. In the first section, a given URL is crawled up to a certain specified depth and information on the crawled pages is retrieved. Based on this information, in the second part, the appropriate pages are fetched and then converted one by one to the data:URI form.

5. User Interface of Bookmarklet Builder

Input for the system is a URL of a web site that the user wants to be made into a bookmarklet. The user needs to provide the URL and specify the depth up to which he/she wants the site to be crawled. For example, let Page A be the main URL. Let there be a link to Page B from Page A and let Page C be linked from Page B. Thus, Page B is at depth 1 and Page C is at depth 2 from Page A. When depth is 1, Page C is not among the list of crawled pages. Page C is crawled when depth is specified as 2.

The figure shown below is a screen shot of the UI for taking input from the user.

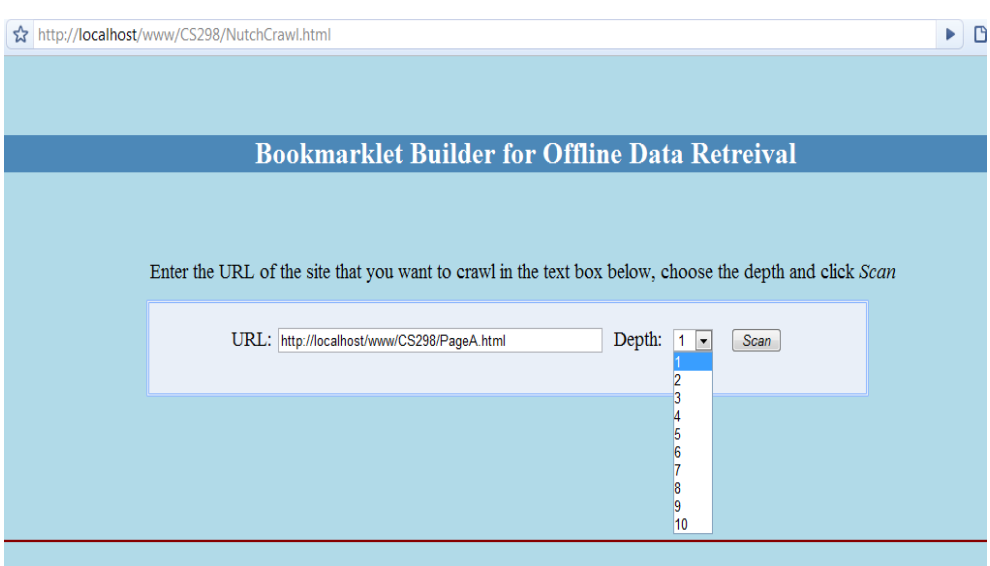


Figure 11: UI of Bookmarklet Builder

6. Integrating Nutch

When the user clicks *Scan*, a PHP program is invoked which runs Nutch's *crawl* command using a combination of predefined parameters and using the input provided by the user.

Consider the site provided as an example in Figure 3: Page Structure of Sample Website. Let the depth be 1. A typical *crawl* command would then look like the following.

bin/nutch crawl url_file -dir crawl_data -depth 1 -topN 10

url_file is the flat file with the URL that is being crawled; crawl_data is the directory in which the crawled data is stored; topN specifies the maximum number of pages that will be crawled at the specified depth – in this case depth is 1.

From the data in the crawl_data directory, we can retrieve the list of pages that were crawled using *readdb* command in Nutch. In the above example, the list of pages are the following.

http://localhost/www/CS298/PageA.html

http://localhost/www/CS298/PageB.html

http://localhost/www/CS298/PageD.html

Notice that Pages B and D are crawled because they are at depth 1 from Page A.

7. Converting Individual Pages to data:URI

Next, each of these pages are fetched and an array is created where the key->value pairs are the url_of_page->content_of_page. The contents of the page is in string form. The following steps are applied to each of the elements of the array.

```
for each (string of content_of_page) {
        Convert to DOM
        Find images and replace image source with data:URI
        Find links and replace with JavaScript
        Find CSS files and replace link to CSS file with data:URI
        Find JavaScript file and replace link to JS file with data:URI
        Convert back to HTML file
}
```

Figure 12: Algorithm to Convert Individual Web Page

When files are converted into DOM documents, the elements of an HTML file can be accessed as nodes of the document tree. We use XPath to locate nodes in the DOM document. XPath is a query/expression language for selecting nodes in an XML file. An expression like /a/b finds all b elements under the root element a. The XPath package (PHP – DOMXPath) used in our project selects nodes from a DOM document. Hence, first the HTML file is parsed into a DOM document and then the XPath query is applied to the DOM document.

7.1 Converting Images

Images in a file are located using the Xpath expression “/html/body//img”. This returns a list of nodes that correspond to each img element in the HTML file. Next we get the “src” attribute of such a node and fetch the image file using cURL() function in PHP. The function returns a string of the image file. This string is base64 encoded and its data:URI is produced. The src attribute which had an external link to the image file is replaced with the data:URI of the image file.

Thus, code that looked like in Figure 13a, is changed to look like in Figure 13b.

```
<img src="http://localhost/CS298/Images/Image1.jpg" />
```

Figure 13a: Original <img> tag

```
<img src="
RgABAgEBLAEsAAD/4QdVRXhpZgAATU0AKgAAAAgABwES
AAMAAAABAAEAAAEaAAUAAAABAAAAYgEbAAUAAAABAAAA
agEoAAMAAAABAAIAAAExAAIAAAAUAAAAcgEyAAIA7AAA
UAAAAhodpAAQAAAABAAAAnAAAAMgAAAEsAAAAAQAAASw
AAAABQWRvYmUgUGhvdG9zaG9wIDcuMAAyMDA3OjE..."/>
```

Figure 13b: data:URI of <img> tag

This is done for each image node that is found in the original HTML file.

7.2 Converting Links

The data:URI of each page is stored in a Javascript array. This array is declared in the head of the HTML page that is served as a home page of the bookmarklet. The HTML code corresponding to this setting is shown below.

```
<html>
        <head>
        <script type ='text/javascript'>
                function change_object_content(url_of_page) {
                        var js_url_array = new Array()
                        js_url_array[Page1]='data:URI of Page A';
                js_url_array['Page2']='data:URI of Page2';
                .
                .
                .
                if url_of_page exists in js_url_array
                then replace object content with new content
                }
        </script>
        </head>
<body class = 'bodycolor'>
<object width ='100%' height = '600' data = 'data:URI of Page>
</object>
</body>
</html>
```

Figure 14: HTML code generated by PHP program

Within the <object> tag's data attribute is the data:URI of the web page. The links within that data are converted such that they point to the function change_object_content() in the head section. This is the main driver function that displays the right page when a link is clicked.

The Xpath query used to find links is “/html/body/a”. This returns a list of nodes that correspond to all the <a> tags in the file. Each of these nodes is parsed to retrieve the “href” attribute and it is changed to something like “javascript:parent.change_object_content('url_of_page)”.

When a user clicks on a link, it invokes Javascript which calls the function defined in the parent window object. This function checks to see if the page referenced by the URL was fetched and displays the new page as an object in the browser.

7.3 Converting CSS Files

CSS files in the DOM document are located using the Xpath expression “/html/head/link”. This returns a list of nodes that correspond to each <link> element in the DOM document. If the “rel” attribute of the tag is specified as “stylesheet” then that means that it is used for styling the current web page and that it could be a CSS file. Now we get the “href” attribute of the <link> tag and fetch the CSS file using cURL() function and replace the “href” attribute with the base64 encoded data:URI of the CSS file.

Thus, code that looks like in Figure 15a, is changed to look like in Figure 15b.

```
<link rel="stylesheet" type="text/css" href="my_styles.css" />
```

Figure 15a: Orininal <link> tag for CSS file

```
<link rel="stylesheet" type="text/css" href="data:URI of CSS file" />
```

Figure 15b: <link> tag with data:URI for CSS file

7.4 Converting Javascript Files

Javascript files in the DOM document are located using the Xpath expression “/html/head/script”. This returns a list of nodes that correspond to each <script> element in the DOM document. Next we get the “src” attribute of each node and fetch the Javascript file using cURL() function in PHP. The function returns a string of the Javascript file. This string is base64 encoded and its data:URI is produced. The “src” attribute which had an external link to the Javascript file is replaced with the data:URI of the file.

Thus, code that looks like in Figure 16a, is changed to look like in Figure 16b.

```
<script type="text/javascript" src="my_javascript.js" />
```

Figure 16a: Original <script> tag for Javascript file

```
<<script type="text/javascript" src="data:URI of JavaScript file" />
```

Figure 16b: <script> tag after data:URI is inserted

After each of the fetched pages in the crawled web site undergoes this cycle of getting images, links, CSS files and Javascript files and converting them to data:URI, the DOM document of each of the web pages is presented back into HTML format. Each HTML file that is stored as a string in an array is again base64 encoded and stored as a text data:URI. The PHP program outputs the code shown in Figure 14: HTML code generated by PHP program. This output is saved in the output buffer without being sent directly to the browser. Once the required Javascript is generated by the PHP program, the contents of the output buffer are taken into a string and converted into data:URI. This final URI is sent to be displayed in the browser. This is a long string of characters

that can be saved as a bookmark in the browser. This string is a bookmarklet and not just a bookmark, because links on this page work such that the content of the page changes when a user clicks on a link of a page that exists as an element in the file's Javascript array.

8. Advantages and Disadvantages

In this section we discuss some of the advantages and uses of this system. We also discuss about some overhead incurred and inconveniences that one may face while using the system.

8.1 Advantages

The main use of this system is that it lets a user save entire website as a single entity in a browser. Once a bookmarklet is created using this system and loaded in a browser, it can then be saved as a file on the user's computer. Thus, by saving a single file, the user gets to view the entire website whose size depends on the depth and breadth of the crawl. Without this system, the user would have to save each page, one-by-one. Using this system, all the web pages of the site would be connected and tied together. In a typical website, the main advantage of using data:URI is that it saves HTTP requests to the server. Other than document size, the number of HTTP requests made by the browser is the main factor that determines the speed at which a page is loaded. Thus, fewer the number of HTTP requests, the faster a page loads. In scenarios where it is required to link to non-XML data within XML files, we can bundle the related media such as simple image files into the XML file by referencing the image file's data:URI. Another example could be: say you have a background image that is used in only a small section of a web page. It would be useful to have it as a data:URI rather than link to the external source. It would also be better to have a data:URI of bigger size within the HTML if it saves more HTTP requests. This project lets you store entire websites in your browser which is not popular in existing online conversion tools. Most existing tools can convert only single web pages into data:URI.

8.2 Disadvantages

The size of a base64 encoded image is larger than the binary itself. However, compression techniques can be used to reduce the size of the encoding. Internet Explorer did not support data URLs until version 8 was released. The syntax used in IE is slightly different from the syntax that can be used in all other browsers. Thus, we still need to write browser specific code in the case of IE. Browsers and servers have restriction on the size of a URL. Due to such a restriction, this system is best suited for small websites with small data items.

8.3 Note about Recursive data:URI Conversion

A browser is capable of recursively converting a data URL. However, enough testing has not been conducted to determine up to what level a data URL can be repetitively converted. In our project, elements are converted three times. This happens first, when individual elements are converted. Example: image files in an <img> tag and links within <a> tags. Second, when the web page is base64 encoded in its entirety. And third, when that string is sent to the output buffer and then base64 encoded again. In Firefox, when the final page is displayed in the browser, in the address bar we see the data:URI and when we view the source of that page, we can see the HTML code and the Javascript being used.

In Chrome, the data:URI is displayed in the location bar but when we view the source, nothing is displayed. Opera behaves the same as Firefox where we can view the page's source code.

9. Performance Testing and Analysis

The project was tested on three different browsers using different data. Files with different text and media composition were used in the testing process. Assuming that the size of an average web page is around 300KB, each of the files used in our testing were around 290KB. These files included images within them. The system

was also tested using web pages that contained only text and no other external entities like images. The average size of such web pages was 35KB.

9.1 Time Based Test Results

In this section we will see how the system performed while dealing with different number of web pages and with different sizes. The project was developed mainly on Firefox and Chrome web browsers and for testing purpose, we used Firefox and Opera browsers.

The table shown below lists the system's performance based on the number of pages that are crawled and fetched.

No. of Pages	Time to Crawl	Time to convert to URI	Total time
5	48	7	55
6	52	27	79
8	51	22	73
10	48	40	88
12	50	85	135
14	48	61	109

Table 1: Time Based Performance Results (all times are in sec)

The depth used to get the above data was 2. In the last row you will notice that the time to convert to URI has decreased even though the number of pages has increased. This is because the pages that were added, were 30% smaller in size compared to the other pages.

The following table lists the amount of time the system takes to run when the depth of the crawl increases.

Depth	No. of Pages	Time to crawl	Time to convert to URI	Total time
2	5	48	7	55
3	5	59	15	74
4	6	69	16	85
5	7	82	22	104
6	8	89	33	122
7	9	105	35	140

Table 2: System Performance based on Depth of Crawl (times in sec)

In this test case, all the pages used were of similar size, around 290KB. We can see that as the number of pages increase, the total time taken by the system increases. However, it is important to note that the main increase in the time is coming from the crawl time and the time to convert to URI is not contributing much to the increase in total time. The observation we can make here, is that Nutch uses more time to crawl at higher depths even if the number of pages crawled is the same as that at a lower depth.

Tests conducted using web pages with only text yielded the following results. The average size of the web pages was around 35KB.

No. of pages	Time to crawl	Time to convert to URI	Total time
4	46	0.1	46.1
6	49	0.2	49.2
8	49	0.3	49.3
10	50	0.9	50.9

Table 3: System Performance based on Type of Web Pages (times in sec)

Since the web pages did not have any images, the conversion module takes very little time because there are no images to fetch using cURL() function. In the presence of image files, during the conversion process, it is only the cURL() function that takes significant time to run.

9.2 Length of URI

Browsers and servers have a restriction on the length of URLs that can be displayed. In the case of our project, this might affect the number of pages that can be converted to data:URI depending on the size of the web pages. Thus tests were run to determine the maximum length up to which a browser will display the URI.

The following observations were made in both Firefox and Opera.

No. of pages	URI length (no. of characters)
5	1491318
8	3561366
10	4921554
13	6961830
15	8322798

Table 4: URI length Proportional to Number of Pages

From the data shown in the above table we can deduce that the URI length increases proportionally with the increase in the number of pages in the website. The same can be expected when the average size of the web pages increases. In our test cases, each page added around 680000 more characters to the resulting data:URI.

9.3 Differences between Firefox and Opera

The maximum length of the URI string that can be displayed in the two browsers is different for each browser. In Firefox, URI strings up to about 4921554 characters long are displayed in the location bar. In our tests, this was for 10 pages. When the number of pages increased by one to 11 pages, the data:URI string length was

5601650 characters long and the URI was no longer displayed in the location bar. Nevertheless, the data was displayed in the browser and the system behaved as expected. Further testing was conducted up to URI length of 8322338 characters. The URI loaded in the browser screen but was not visible in the location bar.

While using data:URI, another difference between Firefox and Opera is the way the Back button works. Consider the following scenario: Page A's URI is loaded, then we click on a link to Page B which loads Page B's URI. Now, if we click the Back button in Opera, we are taken back to Page A's URI. But in the case of Firefox, we are taken to the previous non-data:URI page that was last displayed in the browser. Chrome is similar to Firefox in this regard.

10. Conclusions

Using this system, users can view entire websites when they are offline by converting the websites to the data:URI form. The elements of web pages are also converted to data:URI and embedded inline into the HTML code. This helps ensure that media such as images are not missed out in the web page's data:URI. The number of pages that are tied into a single URI depends on the depth and breadth of the crawl conducted in the beginning. If a page was not fetched, then any link to that page will not operate and the current page does not change and the user does not see a “page not found” message. This project is more suited for small websites with small data items. This limitation is mainly because of the length restriction of URLs imposed by browsers.

Since styling elements and Javascript elements are also taken care of, the websites' look and feel is maintained and a user's browsing experience will remain similar to browsing a live site.

11. Future Enhancements

In the current implementation of the project, when a given site's data URL is being generated, the user has to wait for the resulting URI to be displayed. This could be enhanced so that the user can give the URL of the site and get back at a later time to get the generated data URI. Or, the system could be enhanced to let the user know that the URI is available for use once it is ready without the user waiting for the application to complete. One possible way of doing this could be to use email notification.

The project could also be enhanced such that it takes in multiple URLs and converts them into data:URI in a pipeline and stores them for later use.

In the current implementation, text and images are converted to data:URI. It would be an enhancement to try to extend that to other elements like small .swf files and may be other video types.

Since the length of the URI is restricted by browsers, using compression techniques to reduce the same would also be an enhancement.

12. References

[1] The Average Web Page Statistics. May 2, 2008. Available:

http://www.optimizationweek.com/reviews/average-web-page/

[2] C. Collberg, The Obfuscation and Software Watermarking homepage. Available:

http://www.cs.arizona.edu/collberg/Research/Obfuscation/index.html

[3] Changing data content on an Object Tag in HTML. March 24, 2009. Available:

http://stackoverflow.com/questions/676705/changing-data-content-on-an-object-tag-in-html

[4] data: URI converter. Available: http://www.dopiaza.org/tools/datauri/datauri.php

[5] Document Object Model. Available: http://www.w3.org/TR/DOM-Level-3-Core/introduction.html

[6] Document Object Model XPath. Available: http://www.w3.org/TR/DOM-Level-3-XPath/xpath.html

[7] Dr. Chris Pollett. Crawler-Ranker Engine. August 2009. Available:

http://www.cs.sjsu.edu/faculty/pollett/174.1.09f/mysearch.zip

[8] Free JavaScript Obfuscator. Available: http://www.javascriptobfuscator.com/

[9] Introduction to Nutch. January 10, 2006. Available: http://today.java.net/pub/a/today/2006/01/10/introduction-to-nutch-1.html

[10] JavaScript : The Definitive Guide. David Flanagan. O'Reilly. 2006.

[11] JavaScript Bible, Danny Goodman with Michael Morrison. Wiley. 2004.

[12] Javascript Tutorial. Available: http://www.w3schools.com/js/default.asp

[13] Javascript Windows. Available: http://www.infimum.dk/HTML/JSwindows.html

[14] Maximum Length of URL. October 13, 2006. Available: http://www.boutell.com/newfaq/misc/urllength.html

[15] Official page of Nutch project. http://lucene.apache.org/nutch/

[16] Nutch Wiki. Available: http://wiki.apache.org/nutch/

[17] PHP: Hypertext Preprocessor Manual. July 1, 1998. Available: http://php.net/

[18] Programming PHP. Rasmus Lerdorf, Kevin Tatroe, and Peter MacIntyre. O'Reilly. 2006.

[19] RFC 3986. Uniform Resource Identifier (URI): Generic Syntax. Network Working Group.

http://gbiv.com/protocols/uri/rfc/rfc3986.html

[20] Shane Ng's GPL-licensed obfuscator “http://daven.se/usefulstuff/javascript-obfuscator.html”

[21] Stunnix JavaScript Obfuscator www.stunnix.com

[22] The "data" URL Scheme. August 1998. Available: http://www.ietf.org/rfc/rfc2397.txt

[23] Uniform Resource Identifiers (URI): Generic Syntax. August 1998. Available: http://www.ietf.org/rfc/rfc2396.txt

[24] Website Optimization: Average Web Page Size Triples Since 2003. Available:

http://www.prleap.com/pr/118836/

[25] Wiki - data URI Scheme. http://en.wikipedia.org/wiki/Data_URI_scheme

[26] Wikipedia - Free Online encyclopedia. Available: www.wikipedia.org

13. Appendix A

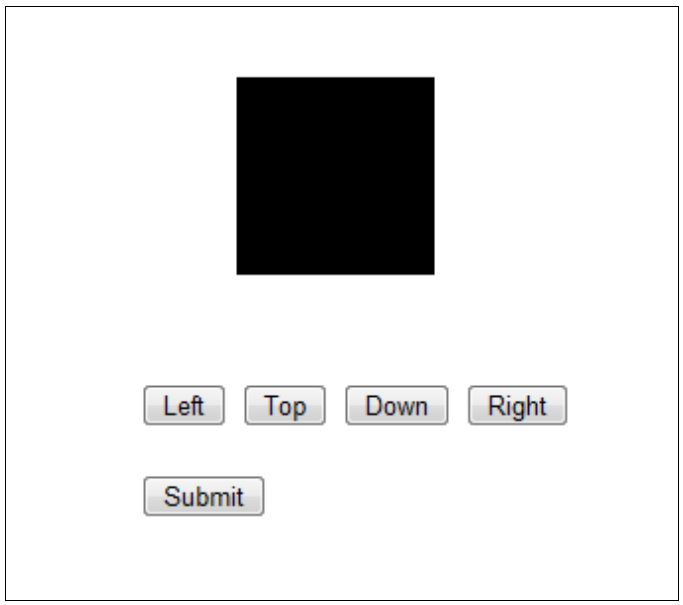
Examples of data:URI

Text = “HTTP request and header traffic is not required for embedded data, so data URIs consume less bandwidth whenever the overhead of encoding the inline content as a data URI is smaller than the HTTP overhead. For example, the required base64 encoding for an image 600 bytes long would be 800 bytes, so if an HTTP request required more than 200 bytes of overhead, the data URI would be more efficient.

Data URI =
“data:text/html;base64,SFRUUCByZXF1ZXN0IGFuZCBoZWFkZXIgdHJhZmZpYyBpcyBub3QgcmVxdWlyZWQgZm9yIGVtYmVkZGVkIGRhdGEsIHNvIGRhdGEgVVJJcyBjb25zdW1lIGxlc3MgYmFuZHdpZHRoIHdoZW5ldmVyIHRoZSBvdmVyaGVhZCBvZiBlbmNvZGluZyB0aGUgaW5saW5lIGNvbnRlbnQgYXMgYSBkYXRhIFVSSSBpcyBzbWFsbGVyIHRoYW4gdGhlIEhUVFAgb3ZlcmhlYWQuIEZvciBleGFtcGxlLCB0aGUgcmVxdWlyZWQgYmFzZTY0IGVuY29kaW5nIGZvciBhbiBpbWFnZSA2MDAgYnl0ZXMgbG9uZyB3b3VsZCBiZSA4MDAgYnl0ZXMsIHNvIGlmIEh
UVFAgcmVxdWVzdCByZXF1aXJlZCBtb3JlIHRoYW4gMjAwIGJ5dGVzIG9mIG92ZXJoZWFkLCB0aGUgZGF0YSB
VUkkgd291bGQgYmUgbW9yZSBlZmZpY2llbnQu”

Total number of characters/bytes is 550.

Image File
Type : PNG file
Dimensions : 337 x 297
Size : 4.58 KB



Data URI of Image File

“
AAARnQU1BAACxjwv8YQUAAAAJcEhZcwAADsMAAA7DAcdvqGQAABHwSURBVHhe7Z3PbtXWFsb9RXFOIH2

R2xBGkEe4ExqRTFNKM0GI2yJQHgDorK0QCF4AVFG1qhCCWcQAUSra8mcAlXqX7WMf+xzbZ1n2Ws32/nnQntj
Le6/9+74vK0VQkn+4IACBmAgkMR2Ws0IAAv+QeUwAgbgIkPm49Oa0ECDzeAACcREg83HpzWkhQObxAATiIkD
m49Kb00KAzOMBCMRFgMzHpTenhQCZxwMQiIsAmY9Lb04LATKPByAQFwEyH5fenBYCZB4PQCAuAmQ+Lr05
LQTIPB6AQFwEyHxcenNaCJB5PACBuAiQ+bj05rQQIPN4AAJxESDzcenNaSFA5vEABOIiQObj0pvTQoDM4wEIx
EWAzMeIN6eFAJnHAxCIiwCZj0tvTgsBMo8HIBAXATIfI96cFgJkHg9AIC4CZD4uvTktBMg8HoBAXATIfFx6c1oIkHkr
DyRcagJWGrBuEwEyb+ULteEpxIRWJmxcF9xWuImynoCVBqzLnPf0gN7xVHrqwl7MeSsPkGQ9ASsNWJc57+kBv
eOp9NSFvZjzVh4gyXoCVhqwLnPe0wN6x1PpqQt7MeetPECS9QSsNGBd5rynB/SOp9JTF/Zizlt5gCTrCVhpwLrMe
U8P6B1Ppacu7MWct/IASdYTsNKAdZnznh7QO55KT13Yizlv5QGSrCdgpQHrMuc9PaB3PJWeurAXc97KAyRZT8B
KA9Zlznt6QO94Kj11YS/mvJUHSLKegJUGrMuc9/SA3vFUeurCXsx5Kw+QZD0BKw1Ylznv6QG946n01IW9mPNW
HiDJegJWGrAuc97TA3rHU+mpC3sx5608QJL1BKw0YF3mvKcH9I6n0IMX9mLOW3mAJOsJWGnAusx5Tw/oHU+I
py7sxZy38gBJ1hOw0oB1mfOeHtA7nkpPXdiLOW/IAZKsJ2ClAesy5z09oHc8lZ66sBdz3soDJFlPwEoD1mXOe3pA
73gqPXVhL+a8lQdIsp6AlQasy5z39IDe8VR66sJezHkrD5BkPQErDViXOe/pAb3jqfTUhb2Y81YeIMl6AlYasC5z3tM
DesdT6akLezHnrTxAkvUErDRgXea8pwf0jqfSUxf2Ys5beYAk6wlYacC6zHlPD+gdT6WnLuzFnLfyAEnWE7DSgHW
Z854e0DueSk9d2Is5b+UBkqwnYKUB6zLnPT2gdzyVnrqwF3PeygMkWU/ASgPWZc57ekDveCo9dWEv5ryVB0iyn
oCVBqzLnPf0gN7xVHrqwl7MeSsPkGQ9ASsNWJc57+kBveOp9NSFvZjzVh4gyXoCVhqwLnPe0wN6x1PpqQt7Me
etPECS9QSsNGBd5rynB/SOp9JTF/Zizlt5gCTrCVhpwLrMeU8P6B1Ppacu7MWct/IASdYTsNKAdZnznh7QO55KT1
3Yizlv5QGSrCdgpQHrMuc9PaB3PJWeurAXc97KAyRZT8BKA9Zlznt6QO94Kj11YS/mvJUHSLKegJUGrMuc9/SA3
vFUeurCXsx5Kw+QZD0BKw1Ylznv6QG946n01IW9mPNWHiDJegJWGrAuc97TA3rHU+mpC3sx5608QJL1BKw0
YF3mvKcH9I6n0IMX9mLOW3mAJOsJWGnAusx5Tw/oHU+lpy7sxZy38gBJ1hOw0oB1mfOeHtA7nkpPXdiLOW/IA
ZKsJ2ClAesy5z09oHc8lZ66sBdz3soDJFlPwEoD1mXOe3pA73gqPXVhL+a8lQdIsp6AlQasy5z39IDe8VR66sJezHk
rD5BkPQErDViXOe/pAb3jqfTUhb2Y81YeIMl6AlYasC5z3tMDesdT6akLezHnrTxAkvUErDRgXea8pwf0jqfSUxf2Ys5
beYAk6wlYacC6zHlPD+gdT6WnLuzFnLfyAEnWE7DSgHWZ854e0DueSk9d2Is5b+UBkqwnYKUB6zLnPT2gdzyVn
rqwF3PeygMkWU/ASgPWZc57ekDveCo9dWEv5ryVB0iynoCVBqzLnPf0gN7xVHrqwl7MeTwAgbgIkPm49Oa0EC
DzeAACcREg83HpzWkhQObxAATiIkDm49Kb00KAzOMBCMRFgMzHpTenhQCZxwMQiIsAmY9Lb04LATKPByAQ
FwEyH5fenBYCZB4PQCAuAmQ+Lr05LQTIPB6AQFwEyHxcenNaCJB5PACBuAiQ+bj05rQQIPN4AAJxESDzcenN
aSFA5vEABOIiQObj0pvTQoDM4wEIxEWAzMeIN6eFAJnHAxCIiwCZj0tvTgsBMo8HIBAXATIfI96cFgJkHg9AIC4CZ
D4uvTktBMg8HoBAXATIfFx6c1oIkHk8AIG4CJD5uPTmtBAg83gAAnERIPNx6c1pIUDm8QAE4iJA5uPSm9NCgMz
jAQjERYDMx6U3p4UAmccDEIiLAJmPS29OCwEyjwcgEBcBMh+X3pwWAmQeD0AgLgJkPi69OS0EyDwegEBcB
Mh8XHpzWgiQeTwAgbgIkPm49Oa0ECDzeAACcREg83HpzWkhQObxAATiIkDm49Kb00KAzOMBCMRFgMzHpTe
nhcDUMn/lypXd3d3/el2yl+yosRGNKTU5sUg7+tf3rLGKdc2kMv/V4eHly5dfvHjxt9cle8mOsm+3TjSmF+TEIu04grJn
6zAr159U5s+dO/fy5csPHz785XXJXrKj7NuNm8b0gpxYpB1HUPaszKR12aQyLz99yTdjvb1GqZQdZd9unWisF+oTi
7TjFJqercOsXH9qmf/48eOfvpfsqMk8jelIObFIO46g6VmZSeuyCWb+j7br0ZefJjvftj6uPfh2J0mvT798tK5eI7Z8U5Cy
5pXSrlYvxcbWjcn6K72N0Na6tuX5UKRNnSel9N026GOS6lE0PVuHWbn+BDP/uu16dGEz2bnV+rj64NZOsnnh0ev
Xilc0YueZX7OzYi9V70XRCI0ttyRYci62l0Hnr7WtN6qgkEbTszKT1mVTy7z8asrvbdePX2wmZ2+2Pq48KCsVr8iOmp
/tuxrLd1bspem9rBmhsdWWbp5NlAx79VovNulcibexTPGupmfrMCvXn2DmX7VdDw82kzM3Gp7eODP/4Tp7mpYtX
ZsHD1tX1Ygt3xSkrHWJ/MFqe4tOygbk1ubBQa3ftmVHaKyBmLAqmllur1KdPZqzzt/lHh4UaLuAvnpl0nnZXfVQ5RG
y1tKG86eFI7IjlPYYagNlJq3LJpj539quB5mcy08Xd9NPmwcPpKC81/xKbQm9QVsbyx8s75Vabd7u4mNaVNxO7+b
9Nl4jNNZw/PJWQ3uL8htnNuUqWKYfssbn3VZedeu8jjDn2oR1GXBWOZINrMOsXH9qmX///v2ztuv+/iw5fbT0tHbz6
HQy27//7Fl5s/GV+gqyo+Zn+67G8gWX9mpsrF7U3d0IjTVsUNxq45bxOzo92z/an6Wwq/XZs9WTrug1Uue1H9aKvS
ucC7Wz/eWLzBxpvyt9jmQDZSatyyaY+adt1z2Rc/to6Wl6s3rN9u89fVpWNr5SX0Fv0NbG8gdLex1ti/ekl/qzeIGtZnn5
ERprOH6xZXN7+V15TTjL5/Kf9dOtozpy55nEC+GL3dN/rxKu9ja2DazDrFx/apl/9+7dk7br7p5k/vrS0+6bjU/rK8iOmjn
f1Vi+4NJetS+vS5b27i4XdXc3QmOrG0gjOcPm9p7l8+29vVnW7PXt2d6efJ0hr9avozp+52nbOcBKJyXU9G5xsMY+
1zUs72t6VmbSumyCmX/cdt05P0tOXVt+Wrlbfqx9mJ2/07qkPNCILd8UpKxrFXm23N61U0nR7uJjWpTMO6oUNC0
9QmNNLRU4Gtt7/PjaqZlcGTJ5u/xcO12zEIszGHQufRXcFrsvYc1oV3sb2wbWYVauP7XMv3379te26/buRv3X4zd2
b2e1V7fm94sbaeXW1fTR/J38i8ZLdtTM+a7G8nUXmxb7LBouGsuLtub9lnetGlshVuPQ0N4c2Lws5bqKtOmk9QOM
gHQVZq6ydFZ9VB4hZZq1uvx0TBsoM2ldNsHM/
+J76Q06Rl8/fL6RfPaNaiXfxlQtKYv+jc6/+SzZ+PwHZYMNZZqercOsXH9qmX/z5s3PvpfsqJnzlzX2fZr5r1Un9G1M1

ZKyyKnzlOXG599nTX39WaLl2nwITc/KTFqXTSrz8kdWj4+P5XdR/OR1yV6yo+bP0o7U2HfnPkn+87/1x3NvbH1Lyg
rPzlOa8+uTc98pG2woU/ZsHWbl+pPK/OHh4aVLl54/fy7fdH0u2Ut2lH27cdOYXo4Ti7TjCMqelZm0LptU5gXWxYsX
Zeoq/zdMw8tkL9lRIxKNKWmfWKQd/et71IjFumZqmbfmxfoQCJ0AmQ9dQfqHQD8CZL4fL6ohEDoBMh+6gvQPgX
4EyHw/XlRDIHQCZD50BekfAv0IkPl+vKiGQOgEyHzoCtl/BPoRIPP9eFENgdAJkPnQFaR/CPQjQOb78aIaAqETIP
OhK0j/EOhHgMz340U1BEInQOZDV5D+IdCPAJnvx4tqCIROgMyHriD9Q6AfATLfjxfVEAidAJkPXUH6h0A/AmS+Hy
+qIRA6ATIfuoL0D4F+BMh8P15UQyB0AmQ+dAXpHwL9CJD5fryohkDoBMh86ArSPwT6ESDz/XhRDYHQCZD50
BWkfwj0IzC1zF+5cmV3d1f51yT9u2XSp3TbTy6qITCYwKQy/9Xh4eXLI1+8ePF3CJf0Kd1Kz4NFZAEI9CAwqczLXx
X48uXLDx8+/BXCJX1Kt2v/HuseYIlKAQWBSWVeflaXAR9C3uc9SrfSs0ImSiAwGoGpZf7jx49/hnNJt2R+NC+zkI7A
BDP/R9f16MtPk8W18+3a4jUlq+
+nO2hfIvM6l1I1JoEJZv51+3VrJ0l2bhXPH13YrH65+IpasCjvWLfxkeJtMj+ml1ILR2BqmZdfGPu99bp5Ntn84sfK4x+/
2EzO3mytX/O4faPsieJt6Zaf7XVGpWo0AhPM/KvW6+GBDPYzNxqep0+KB4vP2aczZ/L/Ftg8eJi9mN08SFdKL3kp
W3X+uXh+49WN+Wvle01dkfnRjMxCagITzPxvXVcZxTSui8IHWebzrxef00+S2Qfp3fTFrGL55rwgvZ2Vlq9X12xpicy
rjUrhaASmlvn3798/U11Hp/PhfPooLb+/P5t/qn6u3n2WvpAWN5cu7pfPa2839yTd8rP9aF5mIR2BCWb+qf66J1Gf7d
97+jT9sH2Uv7j4XL1b3m4uXbxWPq+93dwTmde5lKoxCUwt8+/evXvSdl3fTpLt67Wnd/ck63Kr+Hf6cPG5eveJvD3b
u9tWurhfvlR7u7kn6ZY5P6adWUtBYIKZf9x63Tk/S5LZ+TtFQfr1/Mtrp5Lk1LX0QVZU+ZgXLO6mn/LH+d3lz+W9yup
tHZF5hUUpGZnA1DL/9u3bXzuvq1uV35OzsXu7KL69u5E/2Njd3Uq2rqb303tbW/MXitrsZvZ4XrD0efF8vmRR3NCW
dMucH9nRLLeOwAQz/0s4F5lf50+ej09gapl/8+bNz+Fc0i1zfnxTs2IngUllXv5c6vHxsfzml59CuKRP6ZY/S0tCnQlMKv
OHh4eXLl16/vy5zM+Tf0mf0q307Cw520VOYFKZFy0vXrwok/Pf/Z9eKXeXPqXbyP3H8f0JTC3z/gTZEQJhESDzYel
FtxAYSoDMDyXI+xAIiwCZD0svuoXAUAJkfihB3odAWATIfFh60S0EhhIg80MJ8j4EwiJA5sPSi24hMJQAmR9KkPch
EBYBMh+WXnQLgaEEyPxQgrwPgbAIkPmw9KJbCAwlQOaHEuR9CIRFgMyHpRfdQmAoATI/lCDvQyAsAmQ+LL
3oFgJDCZD5oQR5HwJhESDzYelFtxAYSoDMDyXI+xAIiwCZD0svuoXAUAJkfihB3odAWATIfFh60S0EhhIg80MJ8j
4EwiJA5sPSi24hMJQAmR9KkPchEBYBMh+WXnQLgaEEyPxQgrwPgbAIkPmw9KJbCAwlQOaHEuR9CIRFgMyH
pRfdQmAoATI/lCDvQyAsAmQ+LL3oFgJDCZD5oQR5HwJhESDzYelFtxAYSoDMDyXI+xAIiwCZD0svuoXAUAJkfih
B3odAWATIfFh60S0EhhIg80MJ8j4EwiJA5sPSi24hMJQAmR9KkPchEBaB/wPgoS4JgiBRNQAAAABJRU5ErkJggg
==”

Total number of characters is 6290.