F20FC – Coursework 1

Rohan Veit

Contents

Contents

Introduction

You can find the video demonstration of the CW here:
https://www.youtube.com/watch?v=e2VXFPmGdTE

The purpose of this report is to detail the development process of my simple web browser 'Gongle'. It will cover the full process from the requirements, to design decision, to the specifics of implementation, how testing of each feature was conducted and the results from said tests, and finally a conclusion detailing what could be done differently given more time.

The aim for this project is to deliver a lightweight, user-friendly web browser that allows them to 'surf' the web. The browser aims to make the user experience better by offering a few functionalities, such as saving a homepage, named favourites, bulk download options as well as providing an easy-to-interpret user interface displaying the page, the history of navigation as well as a GUI for the aforementioned functionality.

Requirements Checklist

Functional Requirements

Requirement Code	Description	Achieved
FR-1	The user can send an http request to a URL defined by the user	X
FR-2	Once a website has be navigated to, the user will be displayed relevant information about the website.	X
FR-2.1	The status code of the HTTP request will be displayed.	X
FR-2.2	The raw HTML body of the page will be displayed.	X
FR-2.3	The title of the web page will be displayed.	X
FR-3	The user can reload the webpage, i.e.: send another HTTP request to the website they are currently on and display the contents of the newly returned page.	X
FR-4	The user can define a 'homepage' URL.	X
FR-4.1	If defined, the user's 'homepage' URL will be loaded on browser startup.	X
FR-4.2	A user's homepage URL will persist between restarts of the browser.	X
FR-5	Users can save and remove 'favourite' pages.	X
FR-5.1	Users can navigate between all their favourite pages.	X
FR-5.2	Users can define a custom name for each of their favourite pages.	X
FR-5.3	Favourite pages persist between restarts of the browser.	X

FR-6	The browser should maintain a history of searched web pages.	X
FR-6.1	The browser should allow for the navigation of the history using 'previous' and 'next' buttons.	X
FR-7	The user can initiate a 'bulk download' of specified websites	X
FR-7.1	The browser displays the following information regarding each of the websites specified in the bulk download: • Status code of request made to website • The number of bytes of information retrieved from the website • The URL of the website	X

Non-Functional Requirements

Requirement Code	Description	Achieved
NFR-1	The user should be able to seamlessly utilise all the functions listed above using a simple-to-use and visually appealing GUI.	X
NFR-2	The browser should utilise some key-bindings to make the execution of certain functionalities easier and provide the user with a choice.	X

Design Considerations

This section will cover the justification for various design patterns within the code.

Class Design

Field Accessibility

Consider Figure 1 which shows one of the simpler classes within the project.

As we can see on line 12, for simplicity, fields that need to be accessed via other classes have been defined via C#'s getter/setter syntax. This helps reduce boilerplate code that would otherwise be present in alternative languages such as Java (requiring a method for each operation of each field) and thusly helps streamline development while making code more readable and therefore easier to maintain.

Guard Clauses

As we can also see on line 27 of Figure 1, the 'guard clause' design pattern has been utilised to make code more readable and therefore easier to maintain. The benefits of this are that it reduces indentation as well as clutter.

Structs

As seen in Figure 2, for classes that essentially just serve as encapsulation for a set of fields and don't need any methods attached to them, I have used structs. This is because the data they encapsulate for all instances they are employed is immutable and needn't be updated after retrieval (in the case of a WebResponse) and therefore they can be passed by value instead of requiring reference.

Handlers

Finally, I have decided to split functionality into smaller class 'handlers' (as seen in Figure 3). This means that there are no thousand-line classes that cover a variety of functionalities and exist only to serve a singular function. This is beneficial as it means that debugging a function is confined to only an individual class and upholds the modularity of the overarching program.

Choice of Data Structures

Lists

I chose to employ lists for storing all the favourited websites. This was because it allowed for easy traversal of all the 'FavouriteItem' objects I created. Furthermore, because of the iterable nature of the List collection, it made checking each favourite for a particular feature (such as checking whether the currently entered URL was already favourited) much simpler, as it meant I could just use a foreach loop to compute this.

Furthermore, I coupled this concept with a pointer for the history logic. This meant that I could store a user's whole search history as a list, and if they traversed backward, I could use the pointer as a reference to the location (or how 'far back' they had traversed). This was beneficial as the 'ordered' aspect of lists perfectly represented a user's timeline of visited websites; alternative unordered collections would obviously not have supported this.

GUI Design

For this project I took functional inspiration from Google as well as inspiration from design of early web browsers (such as earlier iterations of Internet Explorer, see Figure 4) due to the ease-of-implementation of their aesthetics and the navigable interface they offer. Finally, I thought that I could reinforce the user experience by offering some key-usage functionality, rather than requiring the user to explicitly click buttons for each step of their navigation (see Project Requirements).

Design Brief

Figure 5 displays a typical screenshot of the browser with the features annotated. Find a table describing each feature below.

Component ID	Feature Name	Additional Notes
C-1	URL input	-

C-2	Search button	-
C-3	Website HTML output	-
C-4	Response Status Code	-
C-5	Website Title / Favourite Name	By default on a page load, this will fill with the page title, however the user may change it to save the name as a favourite as something else.
C-6	Refresh button	Will reload most recently navigated to page
C-7	Favourite button	-
C-8	Navigate to home button	Will navigate user to saved home button
C-9	Edit home button	Will save overwrite homepage to currently visited page
C-10	Selected favourite box	User may select a favourite from C-11, once a favourite has been selected into C-10, the user may press ENTER to transfer it to the URL input.
C-11	Favourites list	-
C-12	History view	-
C-13	Bulk download button	May be used in many ways, see User Guide for more.
C-14	History back button	-
C-15	History forward button	-

Additional Design Choices

Unicode characters were chosen for all the buttons as they are universal and do not rely on the user having knowledge of one particular language. Furthermore, they are beneficial to agile development as they do not require each button's image to be graphically designed, thus saving time which could otherwise be spent on functional requirements.

Use of Advanced Language Constructs

Regex

As seen in figure 6, I used regex for pattern matching URLs to derive their domain from them for display in the history view. This meant that one generic pattern could be used to extract the domain from every possible URL format, this was necessary as displaying full URLs in the history view would cause it to be crowded and not easily interpretable, this is supported by the fact that a lot of repeated (arbitrary) information such as the http:// prefix that denotes the Internet Protocol would be included, and to the user this irrelevant to the history of websites they've visited.

Null-Coalescing Operators

I also used the null-coalescing operator to allow me to shorten type checking and assignment of lists in scenarios where I wasn't sure if an expression would be true. One

example of this is in the HistoryHandler class, where I try read from a file and return a new list if the line read is null. See figure 7 for example.

Nullable Types

Likewise, in some scenarios I was unsure if certain files would be present or if exceptions would be thrown, therefore it made sense to use nullable types as the return for some functions (such as for the getBulkDownloads function shown below). By using nullable types I was able to pass this upward and check (and de-nullify) my types in a context that was less imperative to the nature of the underlying function. See Figure 6 for usage.

Ternary Operators

In the getBulkDownloads function I also make use of ternary operators to help condense the code and make it more legible. This is used as I wanted to offer a few different ways of specifying a bulk download and after checking whether the criteria had been outright satisfied, I needed to set a variable depending on this. You may see Figure 8 for reference.

Records

Initially, I wanted to use records for classes that were essentially wrappers for a few fields. However, due to the limitations of Visual Studio 2019 edition, I was only able to select up to .NET core 5.0 as a version for my project, which does not contain any of the features of C# 10, one of which being records.

Annotations

Some of the libraries I made use of required fields to be annotated using in-built annotations. While the only standard annotation I used in the testing phase of development was [Serializable] (as shown in Figure 9), I did make use of different annotations which were part of one of these libraries. See more on this in Developer Guide.

'Using' keyword for automatic resource release

Finally, I made use of the 'using' keyword – particularly when interfacing with files – to ensure that interfaces that require explicit closing or shutting down after their use were properly disposed of, to prevent memory leaks and make the most efficient use of resources.

Performant Relevant Choices

History Caching

I decided that saving and loading the whole history and each page's HTML content each time the browser was stopped/started was extremely costly, to avoid this, I only saved the history of URLs and their names to an external file each time, and instead opted to only re-retrieve the body of a website if a user navigated to it, at which point it would be cached in memory so that if they decided to navigate to it again during the same session it would be readily available. This is a particularly noticeable performance increase for websites that have a very long latency and take a while to retrieve normally.

User Guide

Step	Figure	Description
Initial starting of browser	Figure 10	Upon initial starting of the browser, the interface should look as demonstrated in the figure.
Viewing a page	Figure 11	As seen in the figure, a URL may be entered into the search bar. Subsequently, either the ENTER key or the search button (highlighted in figure) may be clicked to navigate to the page. Once navigated to, status code and title will be displayed as shown.
Saving a favourite	Figure 12	A user may save a navigated to page as a favourite simply by clicking the ☆ icon. The text shown in the title box will be the associated name with the favourite (defaults to website title). A user may load a favourite by selecting it from their favourites list and pressing ENTER. A favourite may be removed simply by clicking the ★ icon.
Adding a homepage	Figure 13	A user may save a favourite by clicking the 🖋🏠 button while on a page. This will save it as their homepage which will be loaded whenever the browser is started, the title bar will indicate this. A user may also click the 🏠 button to navigate to this page at any time.
Navigating the history	Figure 14	A user may 'backtrack' through their history by using the highlighted arrows. When done, the history view will highlight the current page to make clear to the user the options of going back/forth.
Initiating bulk download	Figure 15, Figure 16	A user may initiate a bulk download via two ways: - Selecting the items they wish to download via the history - Inputting a file name into the search bar After doing so, the user must click the download icon (highlighted in figure) to initiate the bulk download. Once complete the results of the bulk download will be shown in the main area.

Developer Guide

Organization of Classes

I chose to divide code into three main 'types' of class:

- Main class (Form.cs)
- Handlers (I.e.: FavouritesHandler, HistoryHandler)
- Wrappers

Their definitions are as follows:

Main Class

This is the body of the program, it handles the logic for instantiation (creating all visual fields in the form), closing (activating the save functions), and finally all component interfacing (button clicking, text entering, etc.). It has a few helper functions within it, but the logic of the more complex functions (such as saving favourites) isn't contained within it.

Handlers

There are three handlers within the project, being: FavouritesHandler; HistoryHandler; and HomepageHandler. These all respectively handle a larger task, so that the code doesn't pollute the main class. Due to the similar nature of the tasks these components handle, they are somewhat similar internally (they all handle loading/saving values and interfacing with the renderer in some way). They may use 'dependency injection' to access the visual components they interact with (such as the constructor for the HistoryHandler).

Wrappers

These are smaller classes that have no methods within them and only serve purpose to bundle a few relevant fields together into one contained object. Examples of this type of class are FavouriteItem or WebsiteResponse.

General Technical Information

This project was made using Visual Studio 2019 Edition. The framework on which this project is based on is .NET Core 5. This means that it has been built using C# 9. Finally, WinForms has been selected as the framework on which the GUI has been built, this is because of its simplicity and extensive documentation, which makes debugging and troubleshooting much easier.

Utilized Libraries

Aside from standard libraries (such as Generics.Collections for Lists), I only used three notable libraries for functionalities:

System.Text.Json

I used this library for JSON serialization. This was particularly useful as it meant that I could save my favourites list in its current form, and have it parsed back into a List<FavouriteItem> by one cast, rather than having to write my own serialisation method for it; originally, I had opted for this method which stored FavouriteItems in plaintext following the format: '<url>|<title>', however I realised this wasn't suitable as some objects may have a | in their name, malforming this. Therefore, I opted for JSON serialization as it was more robust and – as it turned out – much simpler to implement.

System.Text.RegularExpressions

As previously mentioned, I used Regular Expressions for inferring the domain of a website from its URL, as well as for extracting the title of a webpage from its HTML body. This made these operations much simpler as before I would have had to implement my own form of pattern matching which would have consisted of iterating over my whole input text and checking if the following tokens matched my input; let alone considering the many forms a title may come in (such as the <title> tag having parameters within it) that I would have accounted for, which a singular regex expression now handles.

System.Net.Http

This is the library that I used for the retrieval of HTML given a URL. It provides a simple class 'HttpClient' which can be used to retrieve an HTML response from a website in a single method. Furthermore, it contains objects within it that allow for the extraction of raw HTML body, status code (both number and plaintext meaning) and helps make error mitigation much easier.

Testing

Test Number	Relevant Figure(s)	Test Definition	Expected Result	Actual Result
1	Figure 17	Application starts without saved data	Application starts with no saved history, memory or homepage	Same as expected
2	Figure 18	Browser can navigate to non-real page	404 Error	Same as expected
3	Figure 19	Other status codes work	Correct status code is shown	Same as expected
4	Figure 20	Non-valid URL navigated to	User is given error telling them URL is invalid	Same as expected
5	Figure 21	Users can't favourite invalid web pages	Favourite button does nothing	Same as expected
6	Figure 22	Users can't save invalid web page as home page	Home page is not saved	Browser does allow this
7	Figure 23	Tampered data	Browser still starts if save data has been tampered with / malformed / corrupted	Same as expected
8	Figure 24	Invalid bulk download item	Error code and 0 bytes is shown for that item,	Same as expected

			other bulk items displayed fine	

Conclusion

I think that this project was very successful, as all the requirements (defined in the Requirements Checklist) were met. The testing was also very successful, with the only 'failed' test, being one allowing you to save an invalid webpage as your home page. However, I think if I was to attempt this project again, I would try utilise either generics or delegates (or possibly even more OOP methods) to try and support creating a template for a saved data handler (such as HistoryHandler), and then reusing that for all my use cases. This would be better as it means there would be less repetition of code across classes that have similar functionality, as all my handlers serve the similar function of saving/loading data and then performing some function once it has been loaded on startup.

Appendix

```
10  class HomepageHandler
11  {
        7 references
12      public string HomepageUrl { get; set; }
13      static string HOMEPAGE_PATH = Path.Combine(Form1.APP_DIR, "homepage.txt");
14
        1 reference
15      public HomepageHandler()
16      {
17          HomepageUrl = loadHomepage();
18      }
19
        1 reference
20      public void saveHomepage()
21      {
22          File.WriteAllText(HOMEPAGE_PATH, HomepageUrl);
23      }
24
        1 reference
25      public string loadHomepage()
26      {
27          if (!File.Exists(HOMEPAGE_PATH)) return "";
28          using (StreamReader reader = new StreamReader(HOMEPAGE_PATH))
29          {
30              return reader.ReadLine();
31          }
32      }
33  }
```

(Figure 1, HomepageHandler.cs)

```
6 references
public struct WebsiteResponse
{
    3 references
    public int responseCode { get; set; }
    3 references
    public long bytes { get; set; }
    3 references
    public string url { get; set; }
    1 reference
    public string htmlBody { get; set; }

    2 references
    public WebsiteResponse(int responseCode, long bytes, string url, string htmlBody)
    {
        this.responseCode = responseCode;
        this.bytes = bytes;
        this.url = url;
        this.htmlBody = htmlBody;
    }
}
```

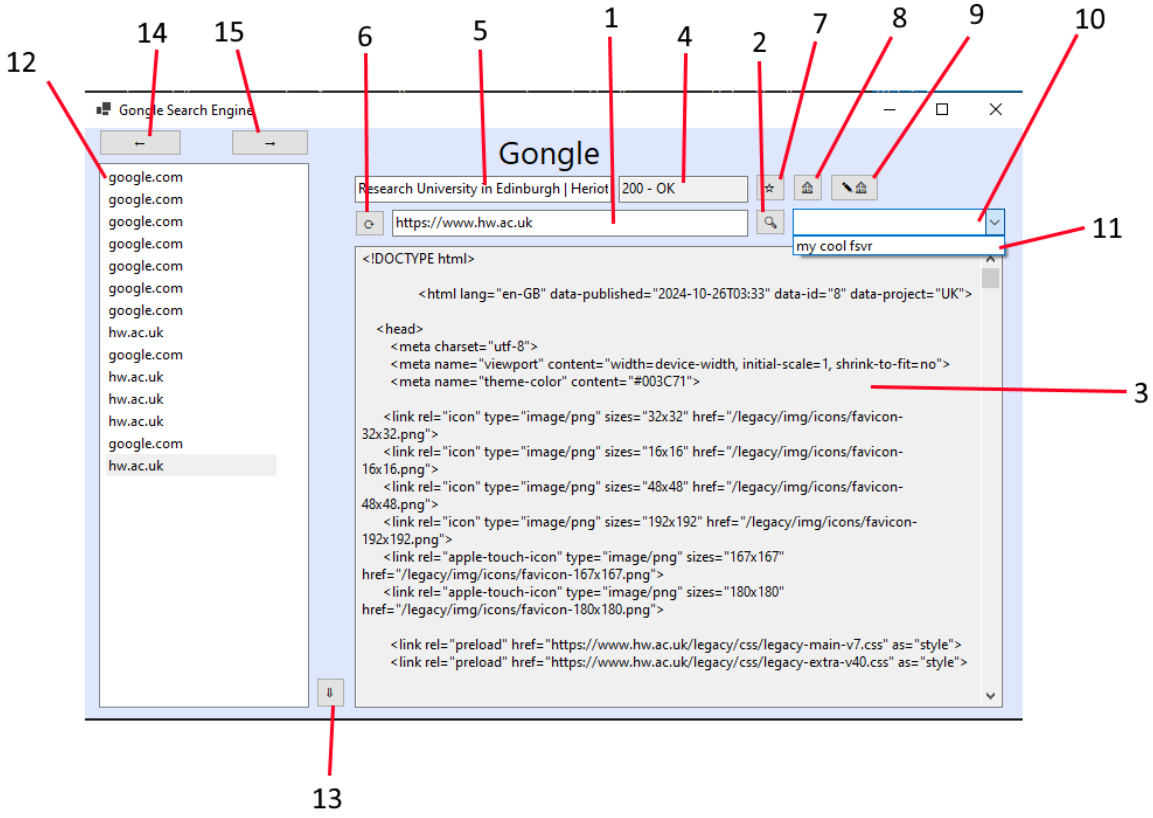
(Figure 2, WebsiteResponse.cs)

```
private HttpClient client;
private FavouritesHandler handler;
private HistoryHandler historyHandler;
private FavouriteItem currentPage;
private HomepageHandler homepageHandler;

1 reference
public Form1()
{
    InitializeComponent();
    this.client = new HttpClient();
    this.handler = new FavouritesHandler();
    this.historyHandler = new HistoryHandler(this.historyView);
    this.homepageHandler = new HomepageHandler();
```

(Figure 3, Form.cs)

(Figure 4, Internet Explorer 2)



(Figure 5, Typical application layout screenshot)

```
278     public static string? getDomain(string url)
279     {
280         string domain = url;
281         string pattern = @"https?:\/\/(?:www\.)?([^\/]+)";
282         Match match = Regex.Match(url, pattern);
283         if (match.Success)
284         {
285             // match.Groups[1] contains the domain (e.g., google.com)
286             domain = match.Groups[1].Value;
287             return domain;
288         }
289         return null;
290     }
```

(Figure 6, getDomain function)

```
if (cacheToUpdate != null)
{
    cacheToUpdate.cacheHTML(responseBody);
    return;
}

string domain = getDomain(url) ?? url;
HistoryItem newItem = new HistoryItem(url, domain);
```

(Figure 7, Null coalescing operators)

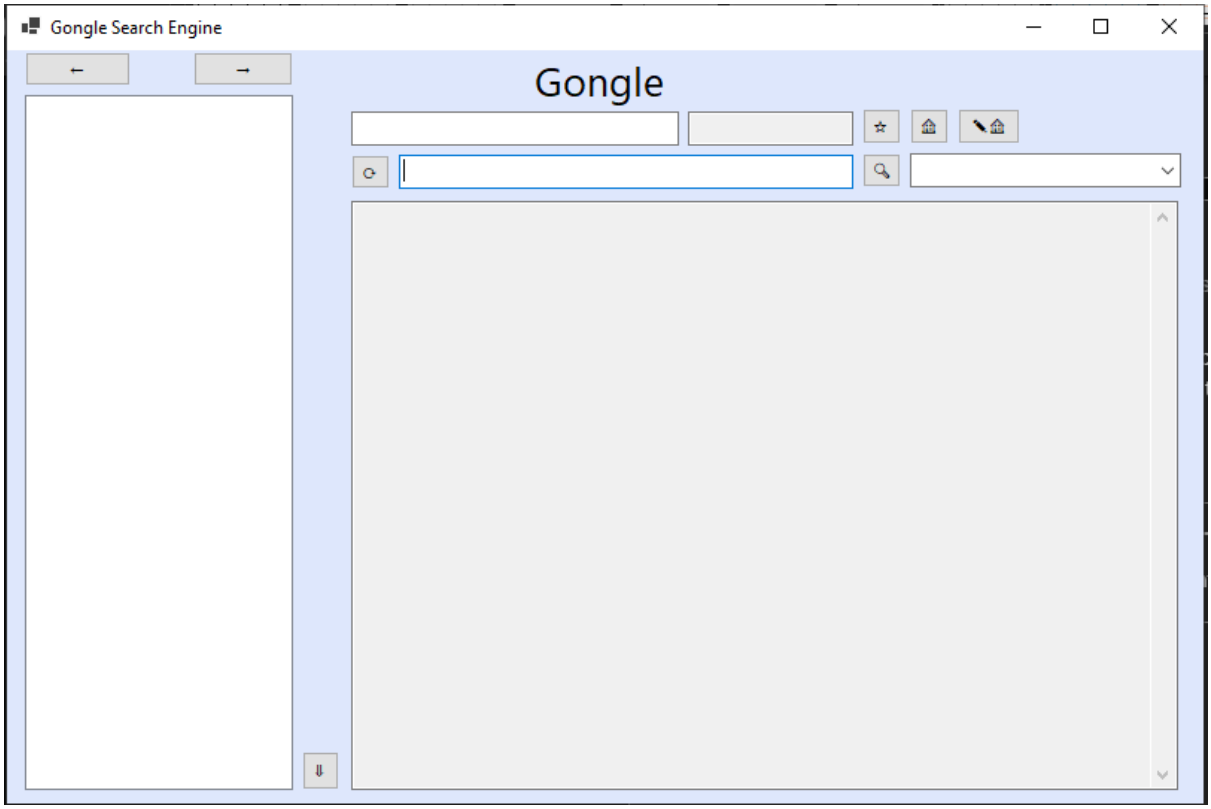
```
322     private List<string>? getBulkDownloads(string fileName)
323     {
324         string fileInternal = Path.Combine(APP_DIR, fileName); //get internal path
325         bool internalExists = false, absoluteExists = false;
326         if (File.Exists(fileInternal)) { internalExists = true;  } //check if file exists interally or absolutely
327         else if(File.Exists(fileName)) { absoluteExists = true; }
328         if(!absoluteExists && !internalExists) //if neither, report to user and return null
329         {
330             titleTextbox.Text = "<ERROR> File does not exist <ERROR>";
331             return null;
332         }
333         string filePath = (absoluteExists) ? fileName : fileInternal; //check WHICH of the two exists (with bias towards absolute)
```

(Figure 8, getBulkDownloads function)

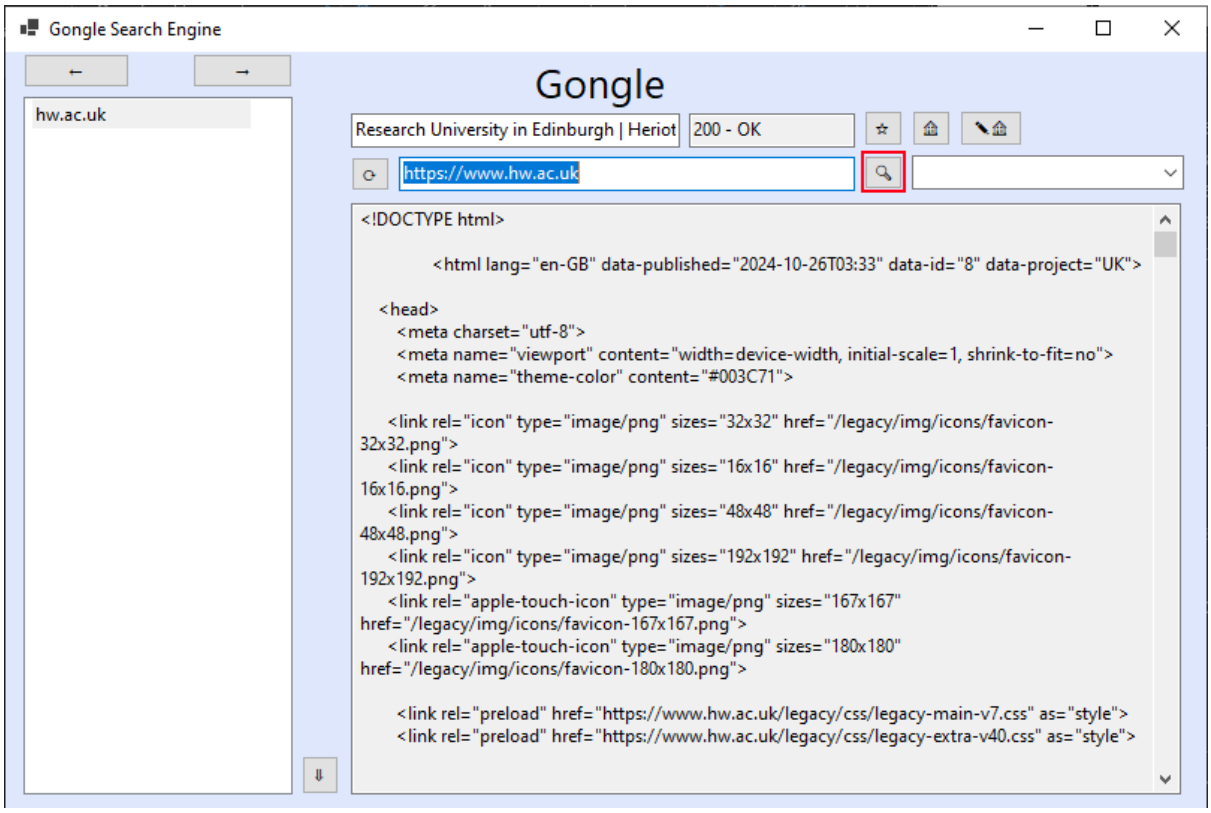
```
[Serializable]
21 references
class FavouriteItem
{
    7 references
    public string Url { get; }
    2 references
    public string Name { get; }

    3 references
    public FavouriteItem(string url, string name)
    {
        Url = url;
        Name = name;
    }
}
```

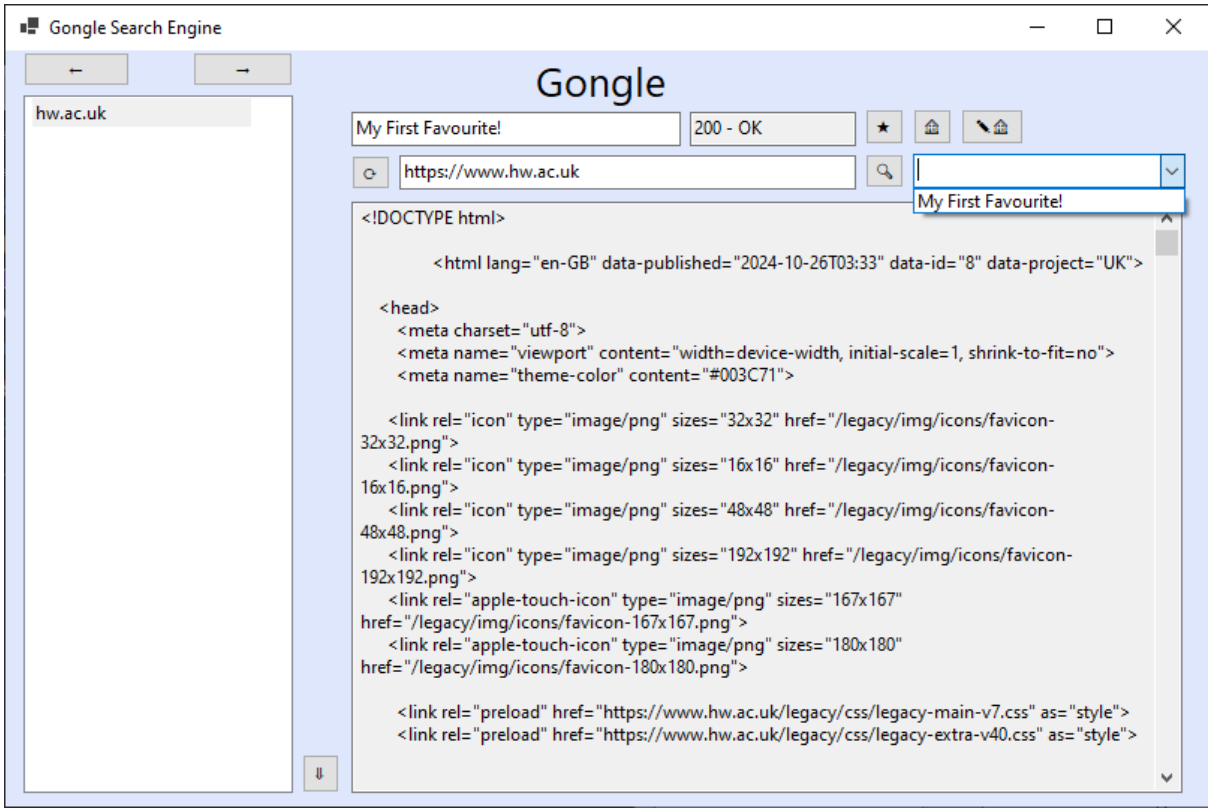
(Figure 9, Annotations)



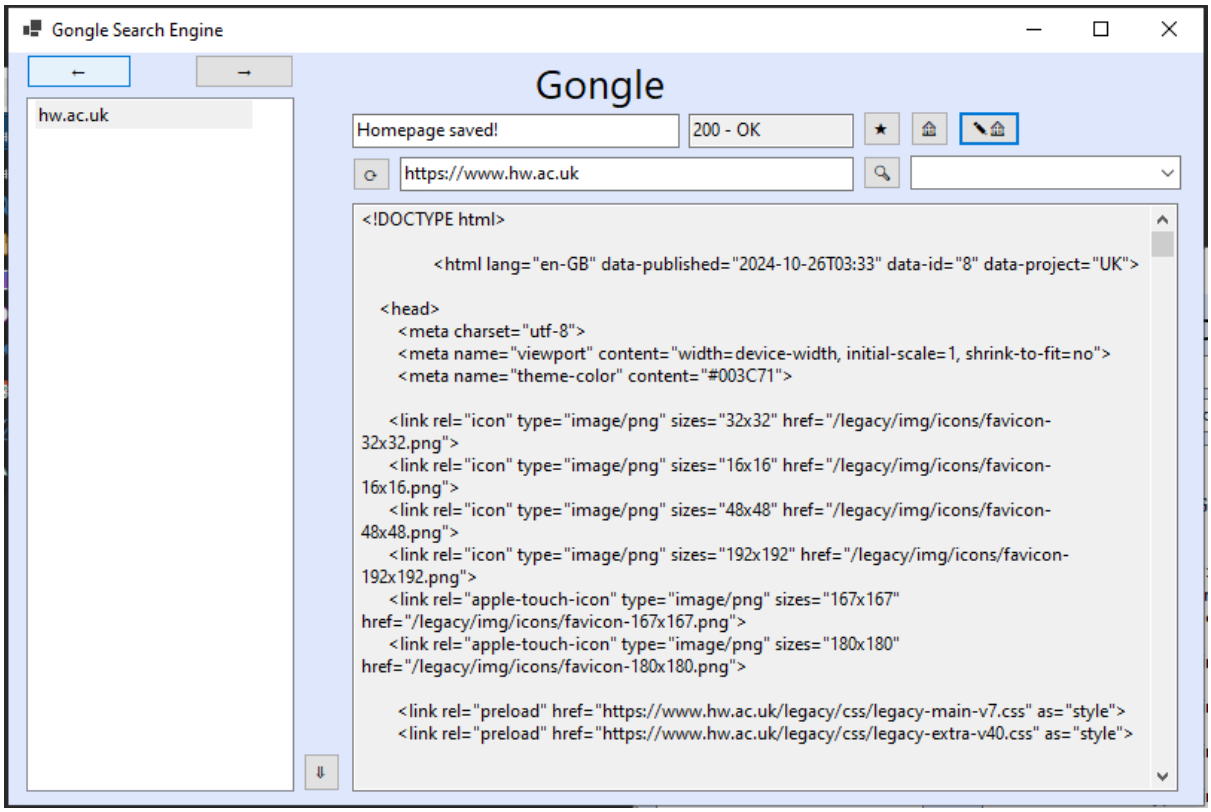
(Figure 10, Initial view)



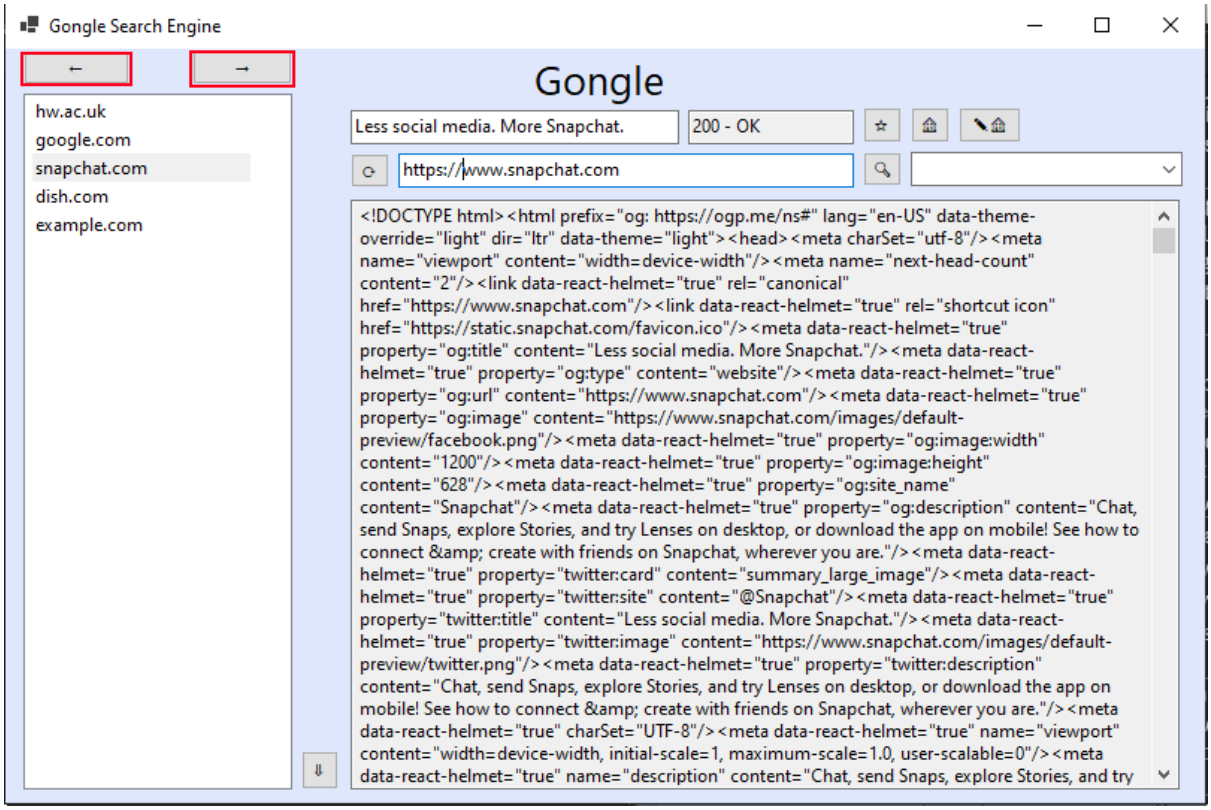
(Figure 11, Viewing a page)



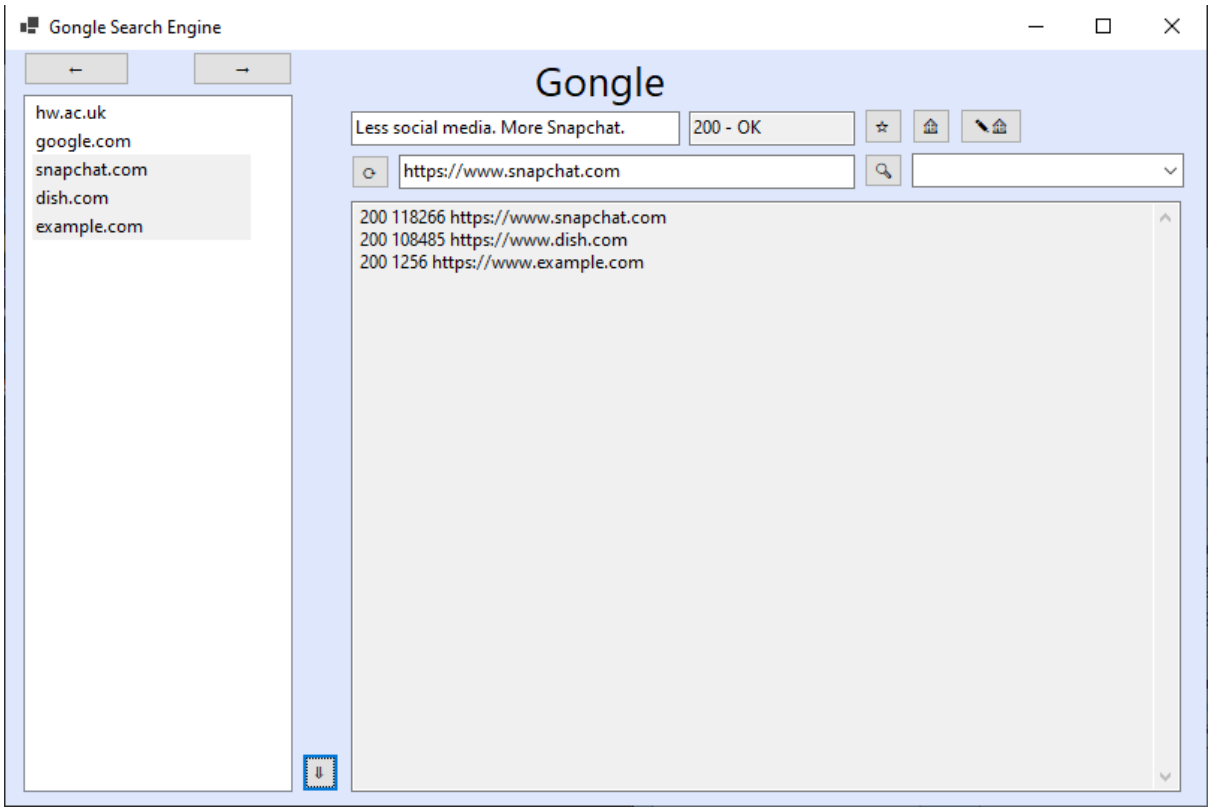
(Figure 12, A favourited website)



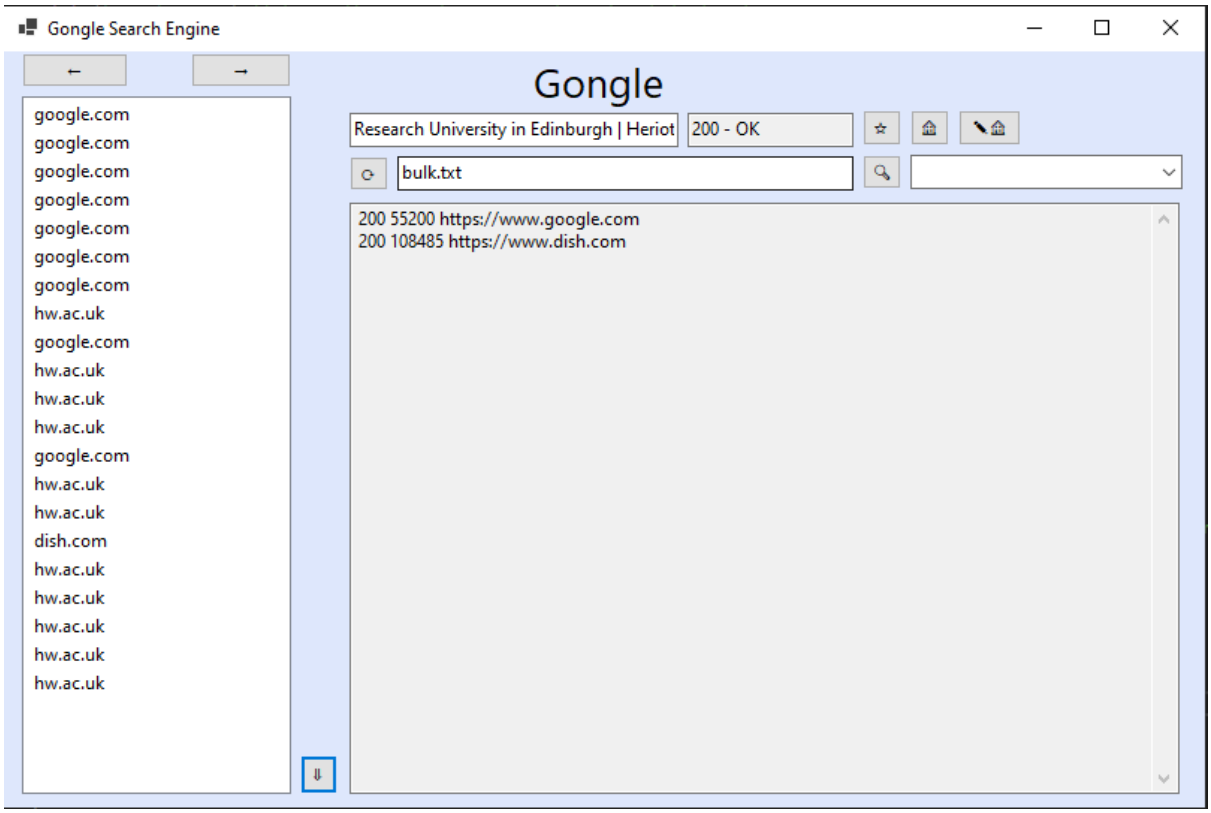
(Figure 13, Adding a homepage)



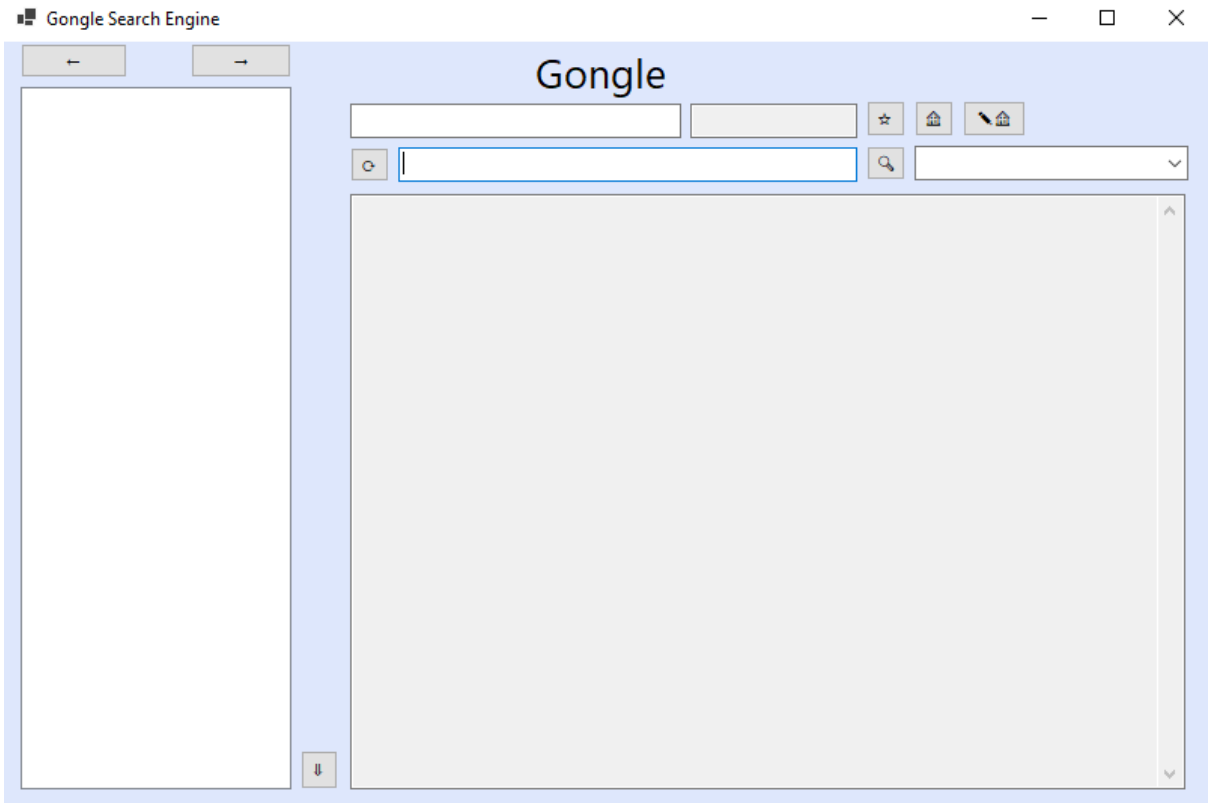
(Figure 14, Navigating through the history)



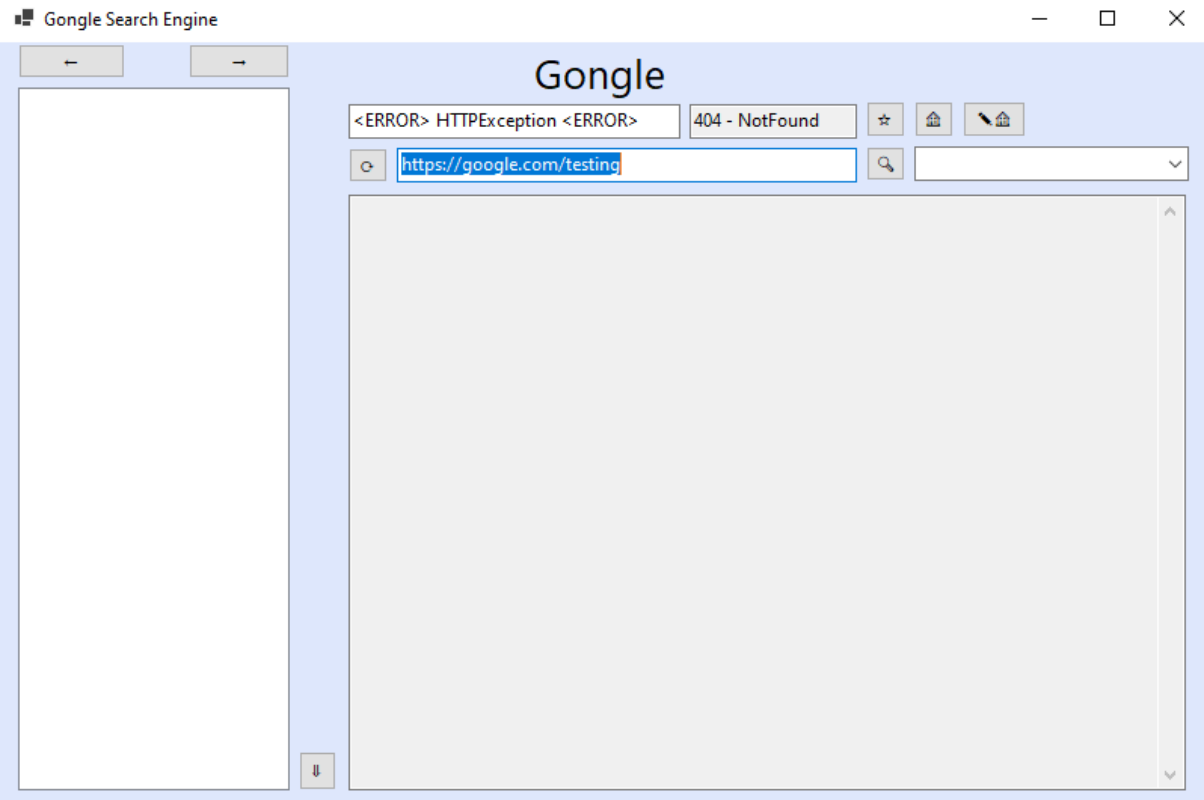
(Figure 15, Initiating bulk download via history)



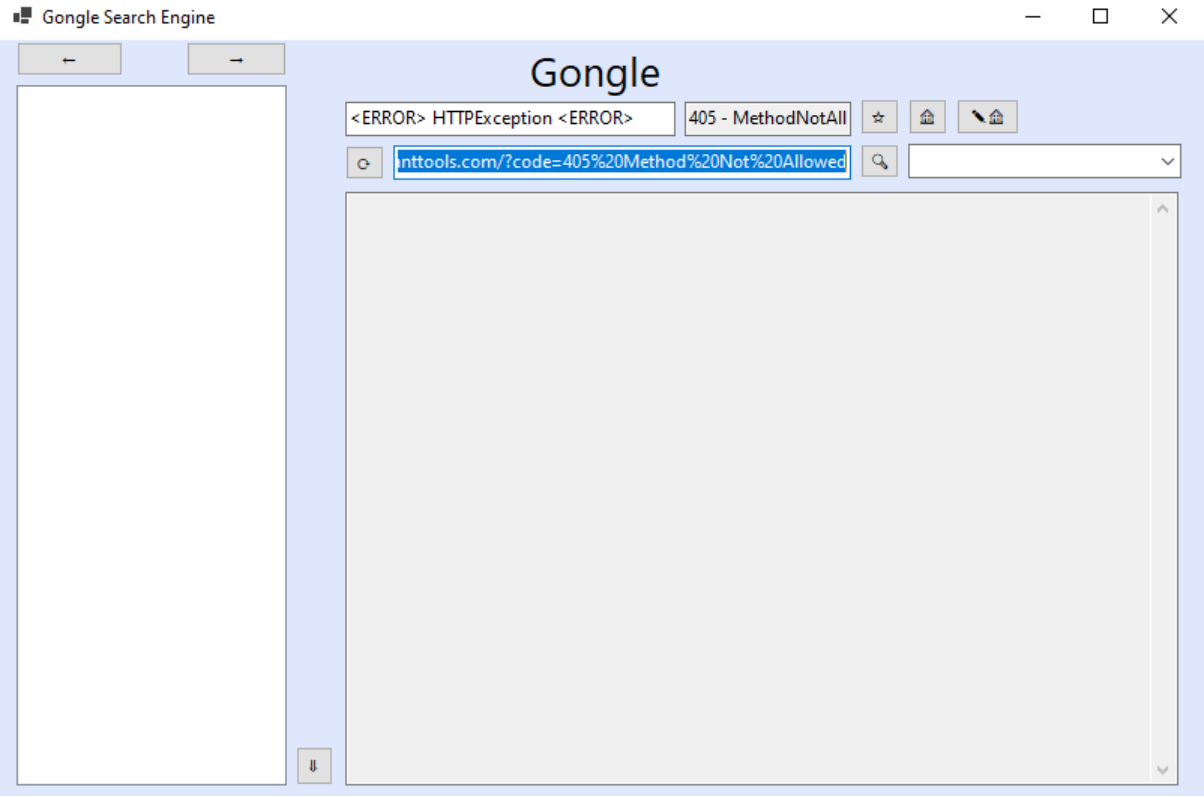
(Figure 16, Initiating bulk download via file)



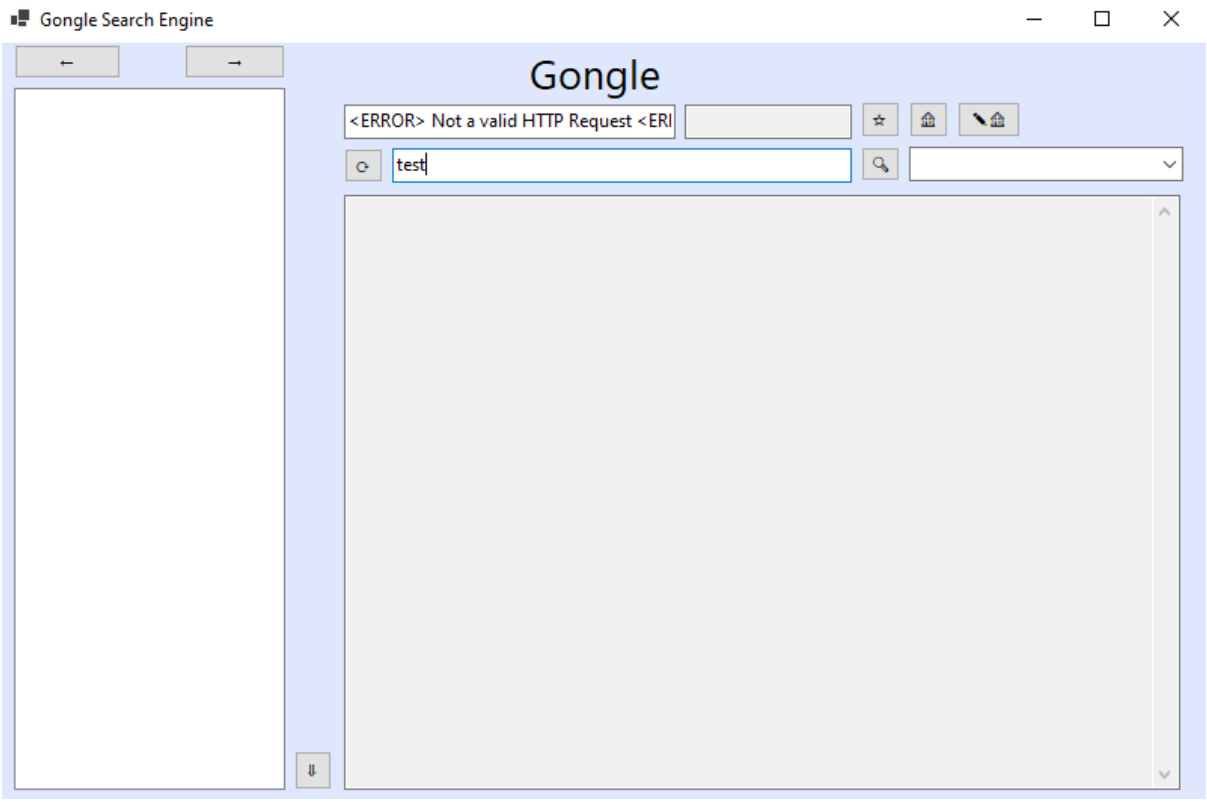
(Figure 17, Default start up test)



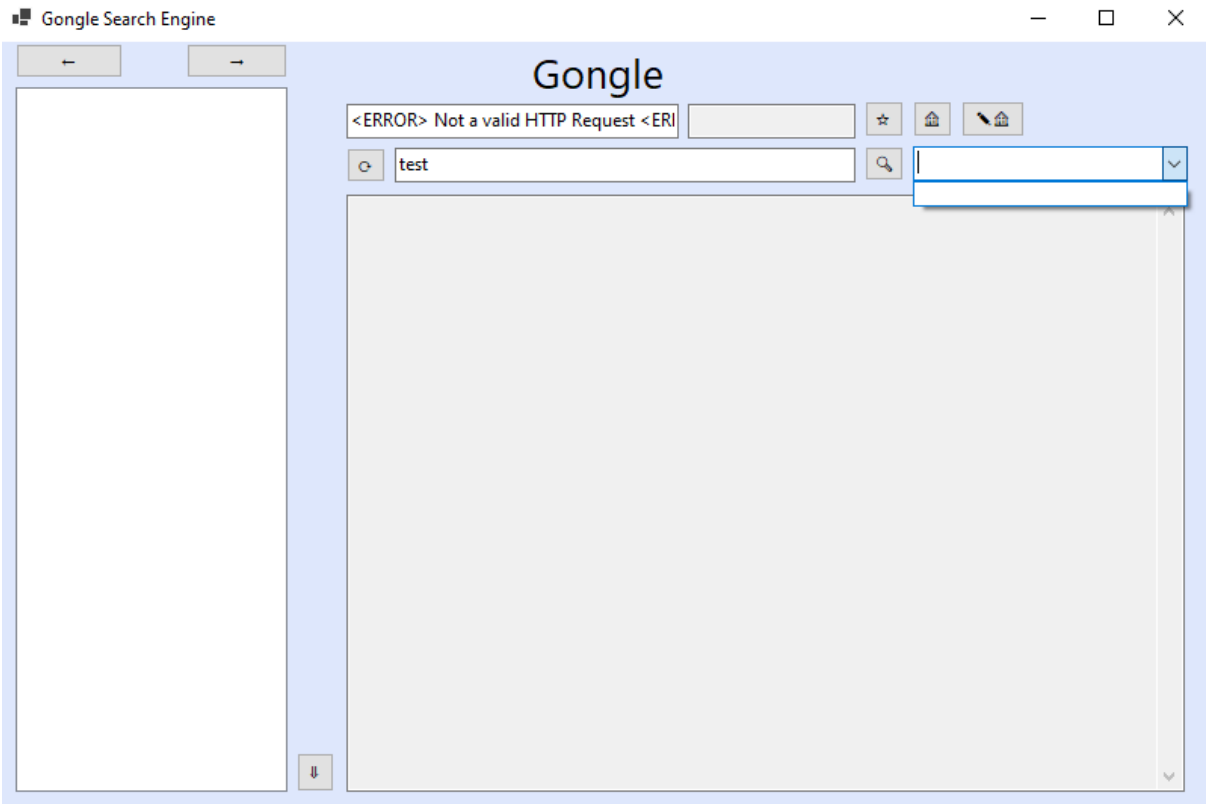
(Figure 18, Non real webpage)



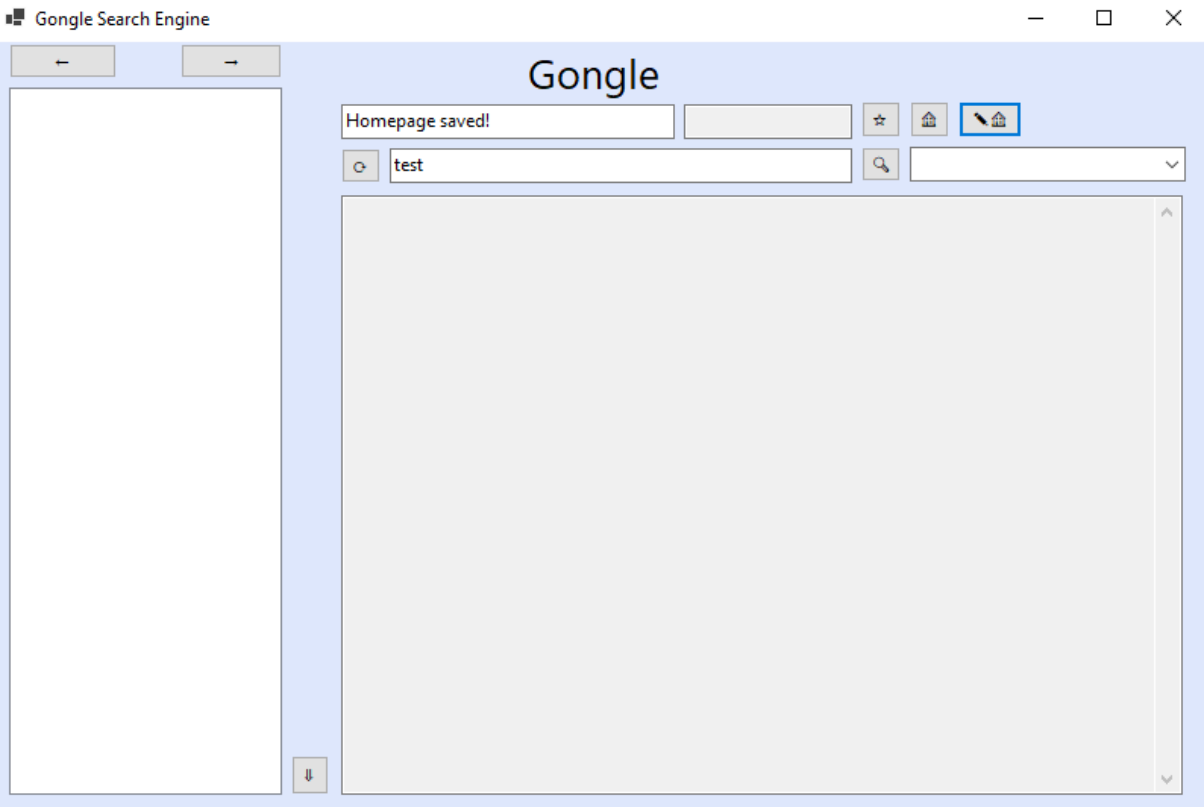
(Figure 19, Other status code test)



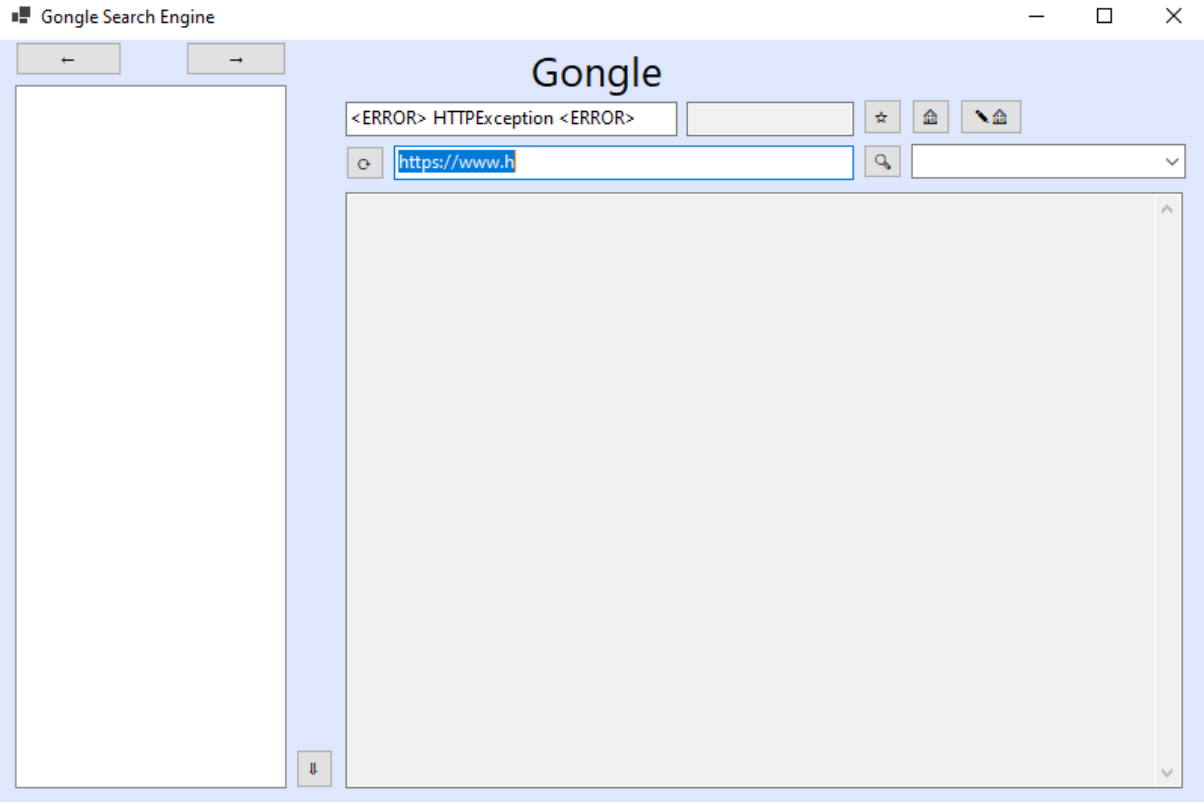
(Figure 20, Invalid URL provided)



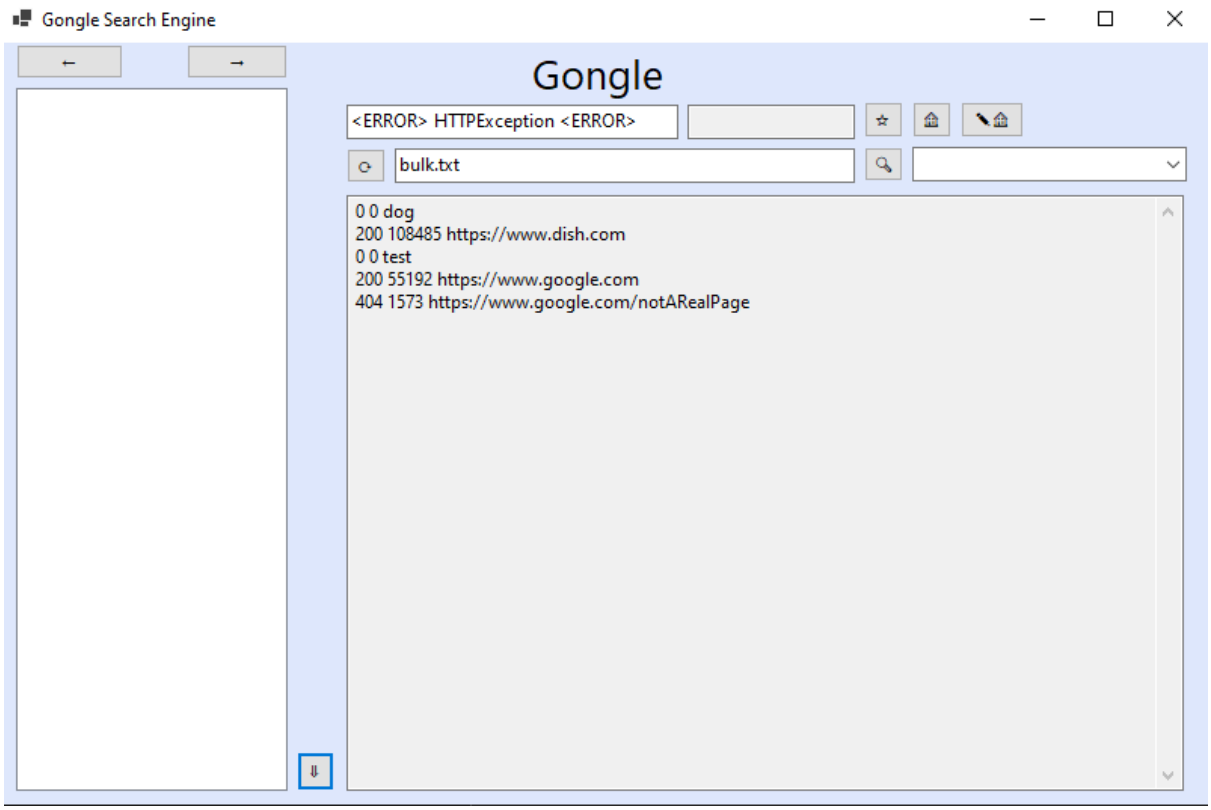
(Figure 21, Users cant favourite invalid URL)



(Figure 22, User adding invalid page as homepage)



(Figure 23, Browser still starting after half of each save file deleted)



(Figure 24, Invalid bulk download items)