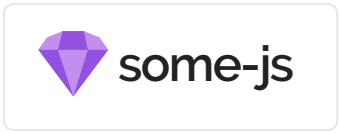


Third Year Project (2015-16)

Simplifying Web Development

Tom Golden

Supervisor: Alexandra Cristea

28th of April, 2016

i: Abstract

This report documents the development of a system that generates webpages without requiring knowledge of existing web languages. Part of the system involves the creation and (successful) implementation of a more intuitive language than the current blend of HTML, CSS and JavaScript, along with a compiler and an interface for users to develop their own webpage.

The main purpose is to make web development simpler for users by reducing the time taken to learn the syntax and subsequently write it. Other solutions implement a drag and drop interface which constrains the author's individual creativity. My research's main aim of simplicity will be tested through a tutorial and a questionnaire.

Keywords:
Web Development, Language Design, Markup Language, Compiler Design, Regular Language, Markdown, JavaScript

ii: Acknowledgements

I received considerable support throughout the course of my project, and it would be unreasonable to not show my appreciation, so here it is...

Thanks to my friends: especially Thomas O'Brien, for his unfiltered input and reminding me when I should have a break, and to Hamish Lacmane, for his help testing my tutorial, but also to the rest of my friends that gave to me help and welcomed encouragement.

I'd like to thank my supervisor, Alexandra Cristea, for everything she has done this year, providing very interesting feedback and more generally supervising my project.

Also Adam Chester, who has helped with advice throughout my project. His module in first year ultimately persuaded me to take on a project involving web design and coding, and I thank him for it.

Lastly, I would like to thank Yvie for her help and support with the tutorial and the report.

iii: Contents

1: Introduction

1.1: Background

The World Wide Web was conceptualised by Tim Berners-Lee in 1980 during his time at the European Organization for Nuclear Research (CERN). It would later become the most popular medium by which the web would operate. It standardised how the web is structured, and the languages by which web page resources can be produced in.

These languages were created specifically for the purpose of creating web pages. HTML, the backbone of web pages, was created as an ISO Standard Generalized Markup Language (SGML); not as a version of XML as many imagine (see Appendix II). Though it is no longer an absolute implementation of a SGML since the fifth version, HTML5 (somewhat as a result of legacy constraints[1]), it still retains much of the same philosophy and syntax.

One of the two postulates for SGML's development is that:

> **Markup should be rigorous** so that the techniques available for processing rigorously-defined objects like programs and databases can be used for processing documents as well.

For web programmers as technically minded as Tim Berners-Lee and his colleagues at CERN, or those in the universities that commissioned ARPANET (the first internet implementation) this attitude would have been appropriate for the majority of the Internet's contributors. **However**, in the 25+ years that have passed since this decision was made, a *much* larger fraction of those who want to contribute to the web would be better served with a different philosophy.

1.2: Motivation

During my time at Warwick, I have made various websites for societies, companies and even the student newspaper. My web knowledge comes from studying at Warwick, mostly from the first year module, Web Development Technologies.

Looking back, the code I wrote was awful and the styling was understandably amateurish. Much of what was written didn't even work.

Most of all though, the amount of time spent doing the most basic things was excessive. Everything was much easier to explain in unambiguous English than to actually code. Indeed, when translated into what I thought was code equivalence, it rarely, if ever, worked.

Rarer again was that same translation to be best practice. Web development is often considered to have more exceptions than rules. Here are examples of this; seemingly correct code which doesn't work in the first case and doesn't conform to the HTML5 specification in the second.

```
<body>
    <b style="text-align:center">This bold text isn't centred!</b>
</body>
```

```
<center>This is, but this is not included in HTML5.</center>
```

This seems to be a very common feeling amongst those new to web development. These issues prove to be the obstruction for a lot of people who want to create their own website. Drag and drop user interfaces provide an alternative but the user loses control of the design of the page.

It is my aim in this project to reduce how much time and knowledge of web development someone needs to develop a website, whilst still providing a platform for absolute control.

1.3: Objectives

The original specification predicated that the project's objective was to develop a method for developing a website that reduces the amount of time required to create a website.

It also defined targets for the method and the resultant website.

> - **The method should be:**

- fast to write (important)
- require little knowledge of web languages (important)
- further accommodate those with knowledge of web development
- documented

- **The resulting website should have:**
  - a level of consistency
  - content-aware display
  - basic modern styling
  - responsive design
  - fast to load
  - small (file size) without compromising functionality

These objectives were formalised in the Requirement Specification which is explored in depth in the report, including some numerical targets.

1.4: Initial Ideas

The idea for the project came to mind when I discovered Markdown, which is a small markup language that was developed to compile to HTML elements. In short, a language whose design is to output rich text. My first impression was how elegant it looked and how simple; as all elements are made using a very simple, intuitive and abbreviated syntax. It was also so quick to learn that I sat down and learnt it in 5 to 10 minutes using the Minimalist Markdown Editor developed by Philippe Masset. This tool had a tab with a quick reference which contained most of Markdown's structures.

It also contained a link to the original specification of Markdown laid down by John Gruber.

> The overriding design goal for Markdown's formatting syntax is to make it as readable as possible. The idea is that a Markdown-formatted document should be publishable as-is, as plain text, without looking like it's been marked up with tags or formatting instructions. While Markdown's syntax has been influenced by several existing text-to-HTML filters, the single

> biggest source of inspiration for Markdown's syntax is the format of plain text email.

The two statements that have been highlighted have been integral to Markdown's success.

Github and StackOverflow both use it; the former for documenting projects on its repository pages and the latter as the input language for users questions and answers on it.

Especially in the case of StackOverflow, which is a website that offers peer to peer computing questions and answers, which is excelling due its aforementioned aims. It also has two notable extra benefits over other markup languages. *One*, is the ease of learning. As previously mentioned, it really is a simple learning curve, and it's most well known and used extension, Github Flavoured Markdown (GFM) builds on those principles with deliberate and concise syntax, adding key constructs like line-breaks, strikethroughs and simple tables. *Two*, it is a regular language and can be parsed with one single regular expression, leading to immense compilation speed. Indeed, using the marked parser I have not yet seen a delay resulting from this that would be noticeable by a human.

Here is an example of some standard Markdown code and the HTML that would be output:

Markdown

```
# Large Header

## Sub-header

Paragraph that contains **bold text** as well as
_italic text_. In standard Markdown there isn't
any styling. John Gruber aimed for a styleless
output that could include inline HTML, such as
<span style="color:blue">this!</span>
```

HTML

```
<h1>Large Header</h1>
<h2>Sub-header</h2>
<p>Paragraph that contains <strong>bold text</strong> as well as
<em>italic text</em>. In standard Markdown there isn&#39;t
any styling. John Gruber aimed for a styleless
output that could include inline HTML, such as
<span style="color:blue">this!</span></p>
```

Clearly, the Markdown code is simpler to read, write and maintain.

2: Related Work

There has only been one project with the same objective (to use Markdown to generate webpages client-side), **Strapdown.js**. It is no longer in further development. It provides a single-page static solution that fully follows the Markdown specification.

Strapdown also tries to give a solution for simpler web development by using Markdown as a means to generate HTML and comes with stylesheets designed for Twitter Bootstrap websites.

The source code does this:

> 1. load the *marked* Markdown compiler with default settings
> 2. loads the *Google's Prettify* code highlighting module
> 3. hides the <body> and its content while the compiler is executing
> 4. adds a shim to provide the getElementsByClassName function to Internet Explorer
> 5. copies the scripts and styles and pastes them in the new page
> 6. adds a meta tag that enables a mobile browser to view in a more responsive style
> 7. takes the Markdown from the first <xmp>...</xmp> or <textarea>... </textarea>
>
> ```
> <xmp theme="slate">
> # Markdown Content
> </xmp>
> ```
>
> 8. compiles the Markdown into HTML
> 9. places it into the body
> 10. dynamically loads a Bootstrap theme stylesheet based on a theme attribute
> 11. unhides the <body> and its content

2.1: Before discovery

Before I started development on some-js, I looked for similar projects that had been developed; but at that point I could not find any.

During the project I found a link to *Strapdown.js* (on November 3rd, 2015). I had already made a partially functional piece of software at that point. As my project has been documented on GitHub throughout the project, I have a record of what my project could at that point do.

My project did and still does use *marked* and seeing *Strapdown.js* using the same compiler was very encouraging as I had done component research preceding this. Beyond this there were a surprising amount of shared steps and features, like adding a <meta name="viewport"> tag.

There were also some major differences between *Strapdown* and some-js at this point and they were:

- Strapdown used **html tags to house the Markdown** whereas I had just started to load the source code from an external file through `XMLHttpRequest` objects.
- Strapdown uses **themes**; specifically those designed for Bootstrap. some-js then had a single “vanilla”-type css layout.
- Strapdown uses the global scope; some-js splits functionality into distinct functions
- Strapdown includes Google’s Prettify syntax highlighting code

...and most importantly, Strapdown had not extended the Markdown compiler.

2.2: After discovery

Difference	Conclusion
HTML Tag Container	Very buggy. <xmp> is deprecated. Full explanation in Project Development.
(Bootstrap) Themes	Unconvinced, Bootstrap files too bulky. See Component Analysis.
Global scope	Bad practice! Nightmare to test. Abstraction helped facilitate a **WYSIWYG editor**.
Google Prettify	Slow to execute compared to other highlighters. Large source file too

2.2.1: Changes made

The <textarea> tag is a potential alternative interface for the users.

The <xmp> tag is deprecated in HTML5, and also xmp is a non-descript name to even the more experienced web developers (e**X**tensible **M**etadata **P**latform, of course). The <textarea> tag has neither issue.

It doesn't mitigate the core issue however; if your Markdown contained </textarea> (which my code for this project at one point did) then you will end up with an error somewhere or at the very least, an undesirable result.

From this, I decided to permit <textarea> tags. If the HTML document contains exactly one <textarea> it takes the Markdown from it instead of an external file.

3: Methodology

The project fits in an area where parts are available online and through open source. There is from my research between 30 and 50 open source Markdown compilers that each compile a different strand of Markdown (though most just compile the original specification from 2004). Furthermore, front end web software is almost always open source, as it is near impossible to protect the source code as it is, by nature, sent to the client that is viewing the website on execution.

As a result it seemed apt to choose a software development model to best represent the great work done by the community. From the Software Engineering module's slides I learnt about the **Reuse-oriented model**. The module is based around using '*Common off the shelf*' (**COTS**) systems developed elsewhere and stitching them together.

The rationale revolves around:

- Reduced development time
- Using the open source community's software to develop even better things for the community
    - Encourages programs with settings to encourage further development using your work as a foundation
- More capable and tested code
- Often a choice of components with separate benefits to suit the need

However there are negatives too.

The main issue is that sometimes compromises have to be made. An initial scan of the potential resources gave me the impression that I would not find this **too** much of an issue, but the project would require careful and in-depth research to ensure that the right components are found and selected.

Certainly, it would help to get feedback throughout the process. Indeed, the CS261 module seems to give the impression that the Reuse-oriented model is disjoint from both Waterfall and Agile models, but in practice it seems to share

plenty with the latter. The model includes a 'Requirements Modification' stage which seems a stage you might find in an Agile project.

Also, the project in any case would involve a large amount of code that is not particularly related to the Reuse model. In these cases I would like to make it apparent that I am following an Agile model, which aimed to involve two feedback cycles before the end of the project. The plan-driven nature of Waterfall is only really considered down to the level of which Reuse development stages should be being done at any particular time. This was formalised in a Gantt chart style timetable in the initial specification and updated in a progress report.

The first 8 weeks of the academic year had a timetable as follows:

Week (from 5th Oct)	Requirements Specification	Component Analysis	Requirement Modification	System Design with Reuse	Development and Integration	System Testing	Evaluation & Report
Week 1							
Week 2							
Week 3							
Week 4							
Week 5							
Week 6							
Week 7							
Week 8							

The updated timetable produced for the progress report, that was followed until project's end, is left below *(with additions in green)*:

		Component Analysis	System Design with Reuse	Development and Integration	System Testing	Evaluation & Report
Week 9	30/11/2015					
Week 10	07/12/2015					
Week 11	14/12/2015					
Week 12	21/12/2015					
Week 13	28/12/2015					
Week 14	04/01/2016					
Week 15	11/01/2016					
Week 16	18/01/2016					
Week 17	25/01/2016					
Week 18	01/02/2016					
Week 19	08/02/2016					
Week 20	15/02/2016					
Week 21	22/02/2016					
Week 22	29/02/2016					
Week 23	07/03/2016					
Week 24	14/03/2016					
Week 25	21/03/2016					
Week 26	28/03/2016					
Week 27	04/04/2016					
Week 28	11/04/2016					
Week 29	18/04/2016					
Week 30	25/04/2016					

The stages of development named in the headers of the tables are defined as follows:

3.1: Stages of Development

- **Requirements Specification**
  - Requirements analysis and planning as per any software project.
  - Identify components from pre-existing libraries and frameworks.
- **Component Analysis**
  - If there is more than one choice for a component, research and test each to assess which is more appropriate for the specification.
  - If none are suitable, look for any partial solutions; the remainder will need to be purpose-built.
- **Requirement Modification**
  - If there were any requirements in the specification that cannot be realistically met in the context of the available components, amend the specification to allow the project to progress.
- **System Design with Reuse**
  - Create a high-level design for the program and plan the order of development.
- **Development & Integration**
  - Create the program itself, ascertaining the connections between the components that are linked and create the components that need to be custom made.
- **System Testing**
  - Test through the development process at customary intervals and do full testing at the end, fixing any bugs that are found.
- **Evaluation & Report**
  - Assess how the project matches with the quantitative targets and address this in the program.
  - Also observe how users work with the software and use the feedback received to fine-tune any oversights.
  - Finally evaluate the end product to discern how it matches/contrasts with the specification laid out.

3.2: Ethical Considerations

Ethical consent for the assessment methods including the tutorial and the

questionnaire was seeked, and approval attained.

The software licences are all open source. In addition they are all free to be modified and redistributed, though this redistribution is not allowed to be profitted from.

My project will be open source and distibuted under Apache License Version 2.0 from January 2004.

4: Design and Implementation

4.1: Requirement Specification

This was a separate document compiled after the project specification. As opposed to the original specification which detailed the aims of the project as a whole, this document was a much more quanifiable set of benchmarks. The core of this original specification has been input as the Objectives section of this report.

These points were formalised and quantified in the 'Requirement Specification'.

A big part of getting people to use a new technology is to show them the capabilities as quickly as possible, whilst also getting them to use it, and get results, quickly.

A great example of how a piece of software achieved this is *Less.js*, which is a JavaScript runtime compiler of *Less*, a CSS preprocessing language which allows for better structured and more concise styling. In this case, the runtime compiler which can be linked with a single **<script src="less.js">** tag. Unlike its main usershare competitor as a CSS preprocessor, *Sass*, this option allowed people to get using it without having to compile the preprocessed code every time before opening the browser, or even installing the compilation software.

As a result of its quick usage growth, I would emulate this attitude. In particular, a condensed safe-bet stylesheet which looks modern as a standalone style but doesn't impose any great restrictions on the user.

This 'one size fits all' strategy would underpin many of the project's decisions. Where possible decisions would be taken to provide universal support but a line was drawn at users who lie in the 5%. This evaluates as approximately 95%.

This was applied throughout the Requirement Specification, from users' download speeds and JavaScript execution speed to the number of users who use a browser that supports a certain feature.

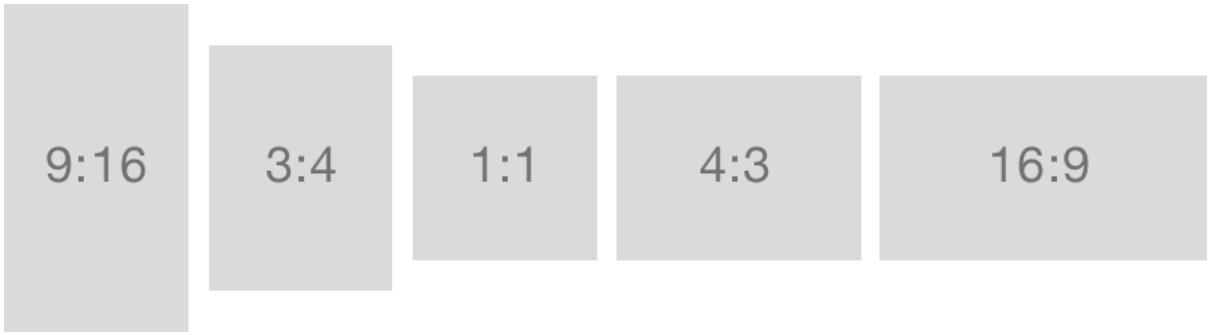
For instance, this table used to determine which screen resolutions would fit

within the 95% (although in this particular case the user experience of the 5% did not have to be fully sacrificed).

Screen resolution	Display ratio	Usage	Screen size / type
1366x768	16:9	19.1%	14” Notebook / 15.6” Laptop / 18.5” monitor
1920x1080	16:9	9.4%	21.5” monitor / 23” monitor / 1080p TV
1280x800	8:5	8.5%	14” Notebook
320x568	9:16	6.4%	4” iPhone 5
1440x900	8:5	5.7%	19” monitor
1280x1024	5:4	5.5%	19” monitor
320x480	2:3	5.2%	3.5” iPhone
1600x900	16:9	4.6%	20” monitor
768x1024	3:4	4.5%	9.7” iPad
1024x768	4:3	3.9%	15” monitor
1680x1050	8:5	2.8%	22” monitor
360x640	9:16	2.3%	
1920x1200	8:5	1.7%	24” monitor
720x1280	9:16	1.6%	4.8” Galaxy S
480x800	3:5	1.1%	
1360x768	16:9	0.9%	

1280x720	16:9	0.9%	720p TV	
		84.1%		***Source:*** [2]

all of these (provided the short side is at least 320px, and the long side is no more than 1920px):



The results of the requirement specification came out as:

Objective	Target
Fast to write	***to be minimised***
Little knowledge	***to be minimised***
Further accommodate	Each web language could still be easily included
Documented	All encompassing documentation
Consistency	The website should have continuity of style
Content-aware	At minimum, looks for different 'types' of page content to match
Modern style	Use modern principles and generally not look dated
Responsive design	Must work for $[\ \mathbf{x}\ \text{px} \times \mathbf{y}\ \text{px} \mid 320 \leq \mathbf{x}, \mathbf{y} \leq 1920 \wedge 0.5625 \leq \frac{\mathbf{x}}{\mathbf{y}} \leq 1.\dot{7}\ ]$
Fast to load	1 second maximum based on 7.4Mbit/s
Small file size	Upper bound 6.1Mbits to comply with above

The full Requirement Specification has been included as Appendix C.

4.2: Research & Component Analysis

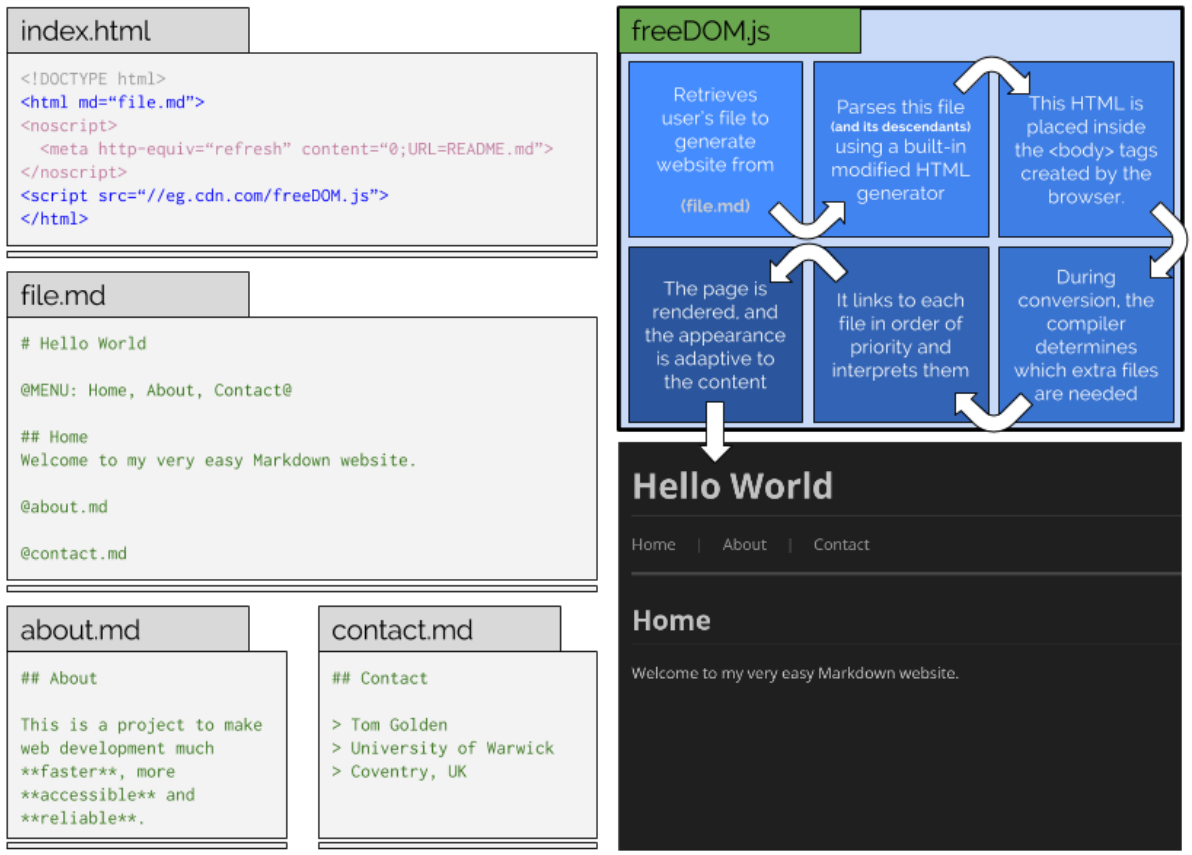
I completed this, alongside the Requirements Specification by Friday of Week 3, which was one week ahead of schedule.

An excerpt from the Component Analysis report has been added to the Report in Appendix IV.

In it, it is described how *Markdown* (specifically *Github Flavoured Markdown*), the *marked* Markdown parser and the *Skeleton* bootstrap framework was chosen

to serve as the foundation from which the project would develop from.

Now that I had a model for how the project would look like, I drew a quick diagram as a more structured overview. This diagram was drawn on 31st October, 2015. As a result, the syntax has changed in a few areas but the overview has not.



Note that the box in the top right which "contains" the script. The name was originally FreeDOM (**Free D**ocument **O**bject **M**odel) but was changed as a result of a separate project of the same name.

4.2.1: Requirement Modifications

In the Reuse software development model a component is included in the development cycle which is called the Requirement Modifications section. Usually this section is in place to lower/remove the target objectives in face of the Component Analysis. Yet in this project, the analysis confirmed that each of the objectives were not only realistic, but under ambitious! This applied most to the file size of the library. The sum of the minified sizes of the chosen parser and

design framework came to less than 0.37% of the upper bound specified in the Requirement Specification.

This led to a choice from two:

- One, lower the upper bound significantly and focus on creating a minified framework.
- Two, add more language features to the project that would not otherwise be included.

I chose the latter after a long period of consideration; the reasoning being that there are some features that are integral to modern web development that have to be added to over 5% of websites.

There is also a possibility to design the project such that it only loads resources if they are actually used. While this is a feature that would add value to the project, it was not executed as part of the project, but might be considered as part of work that could be done as part of future work.

4.3: System Design

To reach a point of minimum completeness, the system would require three components.

1. A compiler which turns an extended Markdown into HTML code.

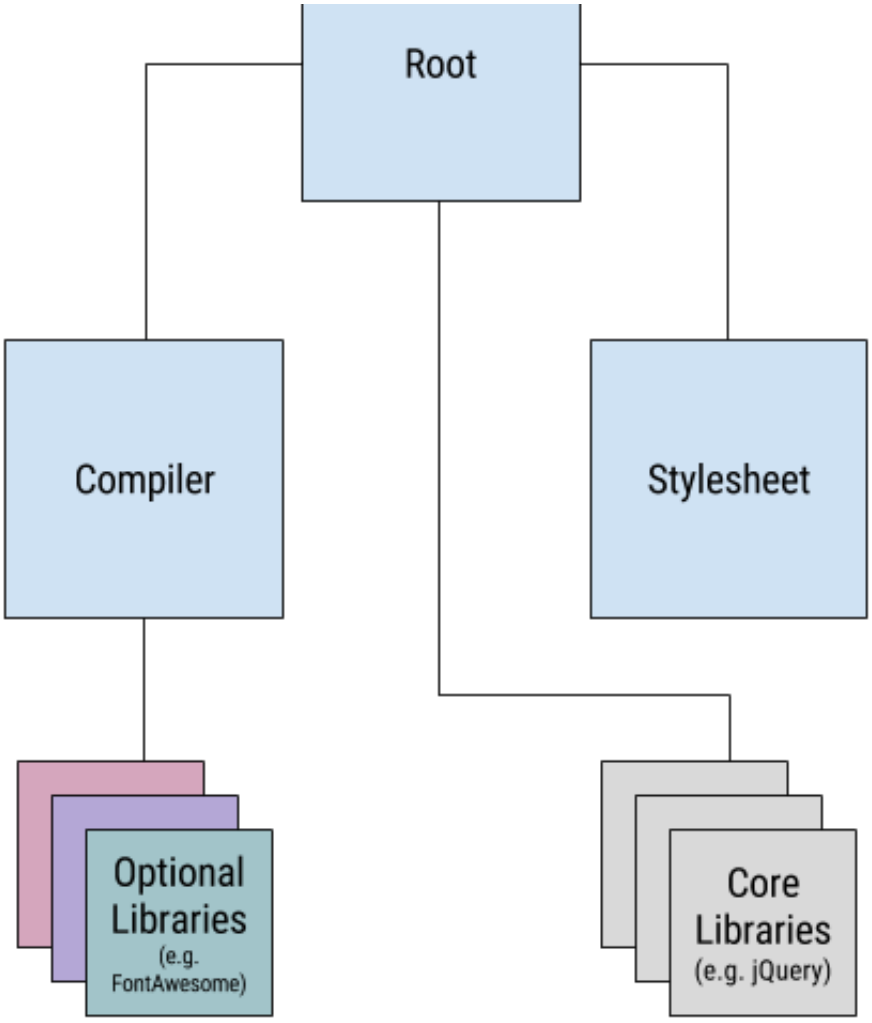
2. A stylesheet which styles this HTML code.

3. A 'root' file which retrieves all necessary libraries including the compiler and stylesheet, and retrieves the source Markdown from which the webpage will be generated. It then passes this source to the compiler and inserts the HTML and the stylesheet into the page.

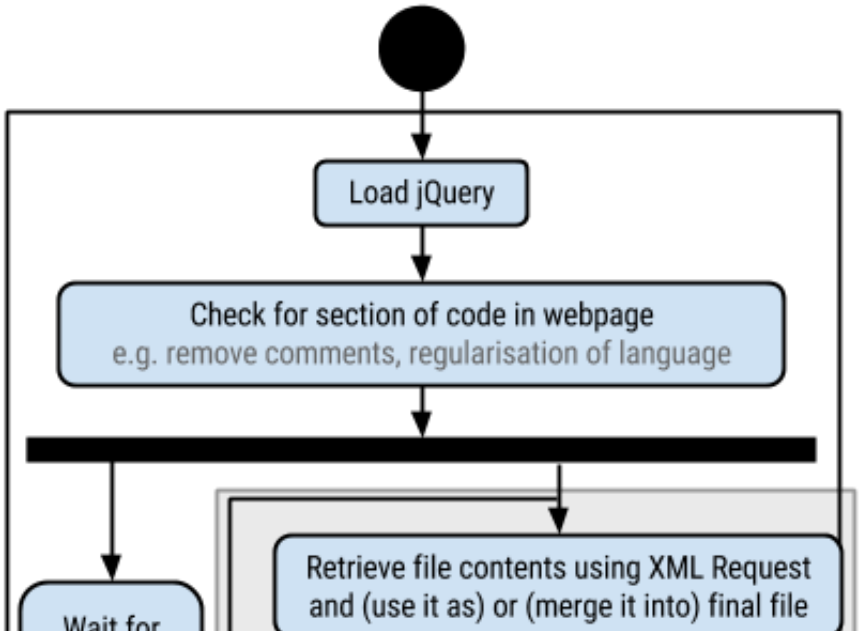
Here are some diagrams for these components.

Diagrams

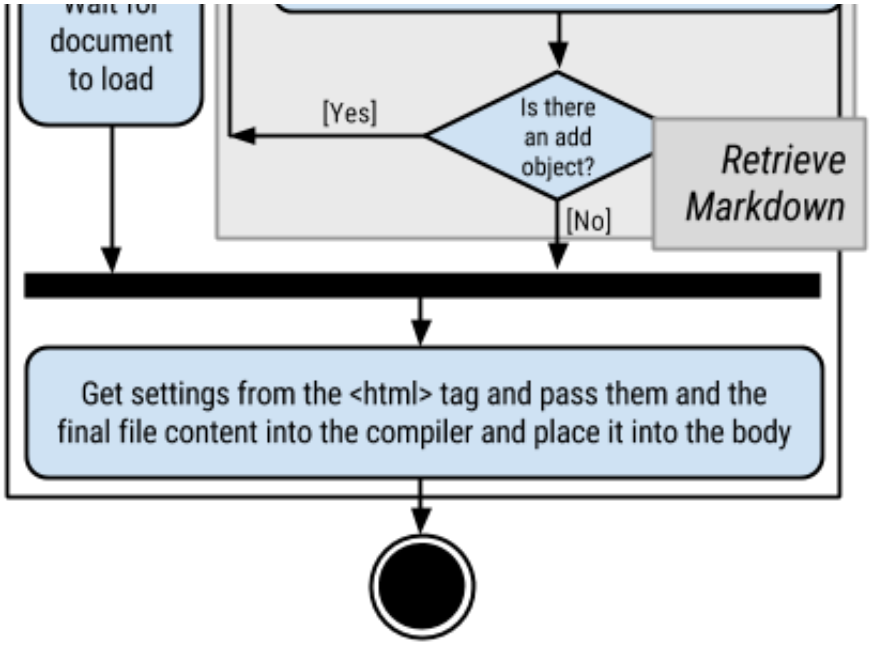
Overview



That two nodes are linked acts to demonstrate which component is loads which. The root at the top is loaded, and then it loads the others through this arrangement.

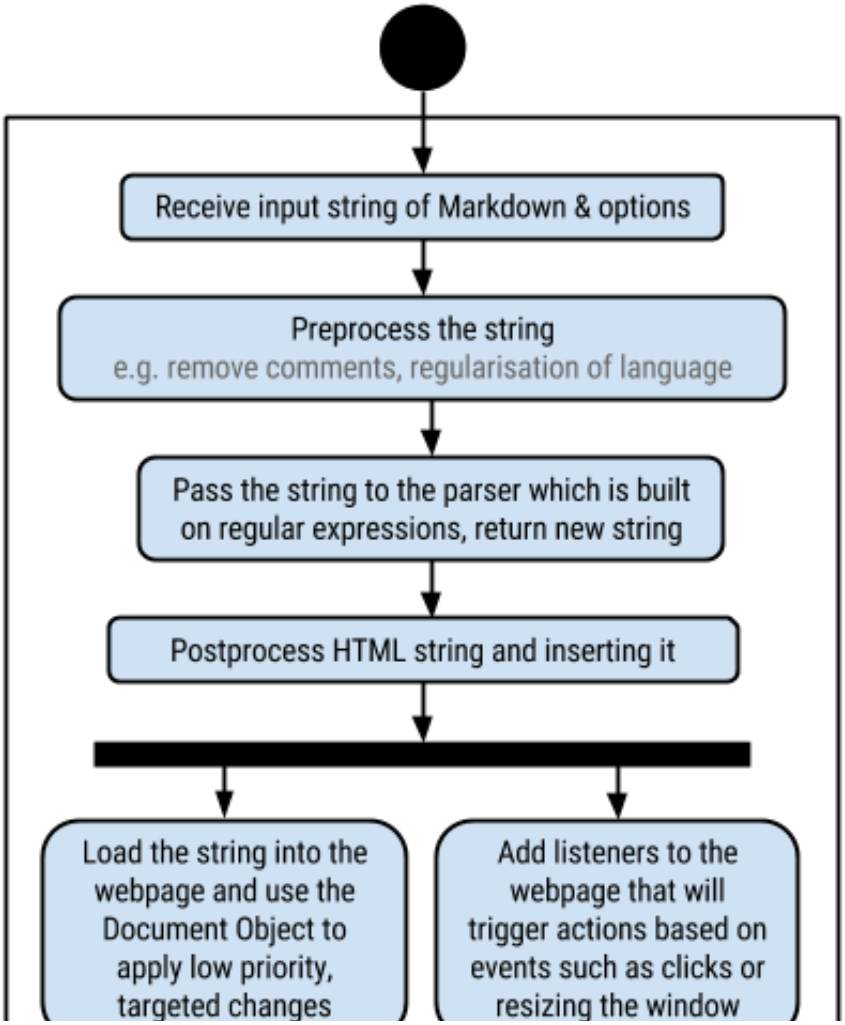


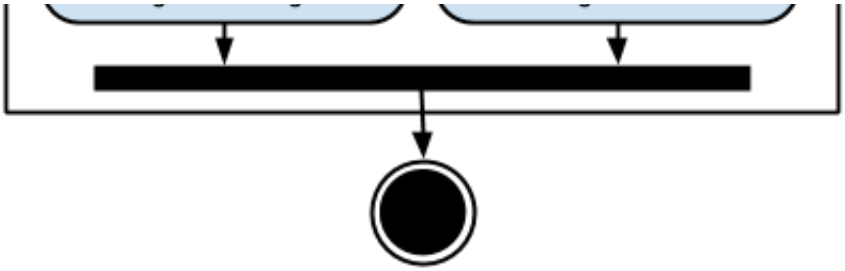
Root
`root.js`



Activity diagram to show the functionality of the root script which serves as a “main method”.

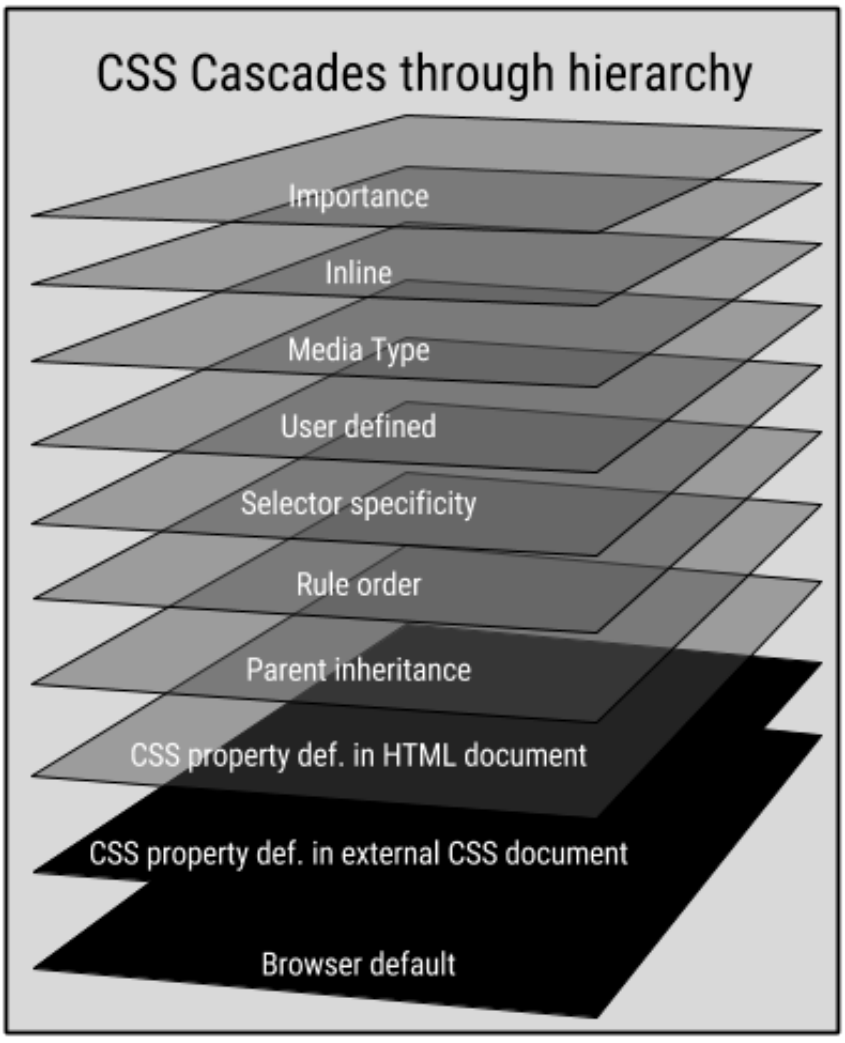
Compiler
`compiler.js`





Activity diagram to show the steps and dataflow of the compiler.

Stylesheet
`style.css`



I am avoiding the bottom two layers of the CSS hierarchy by adding Normalize.css as a core script by adding the CSS file dynamically to the page in a <style> tag. This is to reduce the number of hidden display errors and to provide complete inter-browser uniformity.

These three files would then be merged into one using a Python script. I would

remove the whitespace, remove other unnecessary characters and merge the files together into one single script which could be added in one line of HTML.

```
<script src="scriptname.min.js"></script>
```

The design places emphasis upon code reuse, for the reasons that it is easier to test, produces smaller code for file transfer and useful functions will mean that over the long term development will speed up.

4.4: Development & Integration

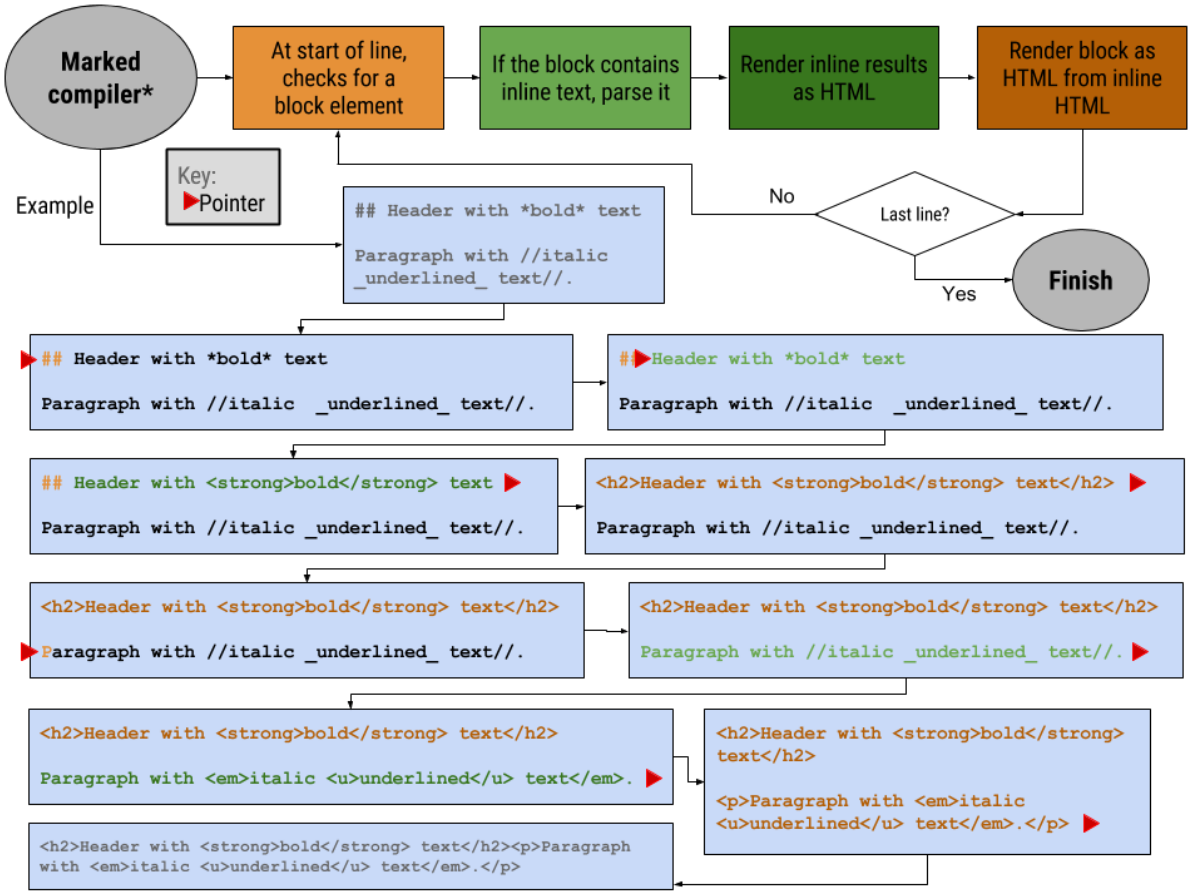
4.4.1: Getting an Understanding

4.4.1.1: Marked Compiler

Development started in Week 5, one week before scheduled. The progress made and the lack of changes needed for the Requirement Specification meant that I could get started earlier than expected.

The additions in the schedule were down to system design updates that I imagined would be made through the process, and indeed they were very necessary.

To best use the marked compiler, an understanding of the compiler's structure and principles needed to be developed.



The compiler uses the default JavaScript RegExp object to match syntax. The HTML syntax, and consequently the Markdown syntax, is split into block and inline objects. This is reflected in the way the marked parser is structured. There are two separate parsers, the inline and block which can call each other. The flowchart diagram above shows the order of parsing that marked uses. This recursive descent style parsing and rendering results in a very efficient compiler, that executes in a fraction of a second (even large documents could compile twenty times a second).

There is an asterisk in the diagram that is a disclaimer; there are a few objects that do not follow this model. Blockquotes, for instance, are Markdown block elements that can contain other block elements inside them. Another issue is that reference links (links or images whose URL is kept elsewhere in the file) require extra information (the URL) and consequently cannot be immediately rendered. Indeed, there are as many more exceptions than rules.

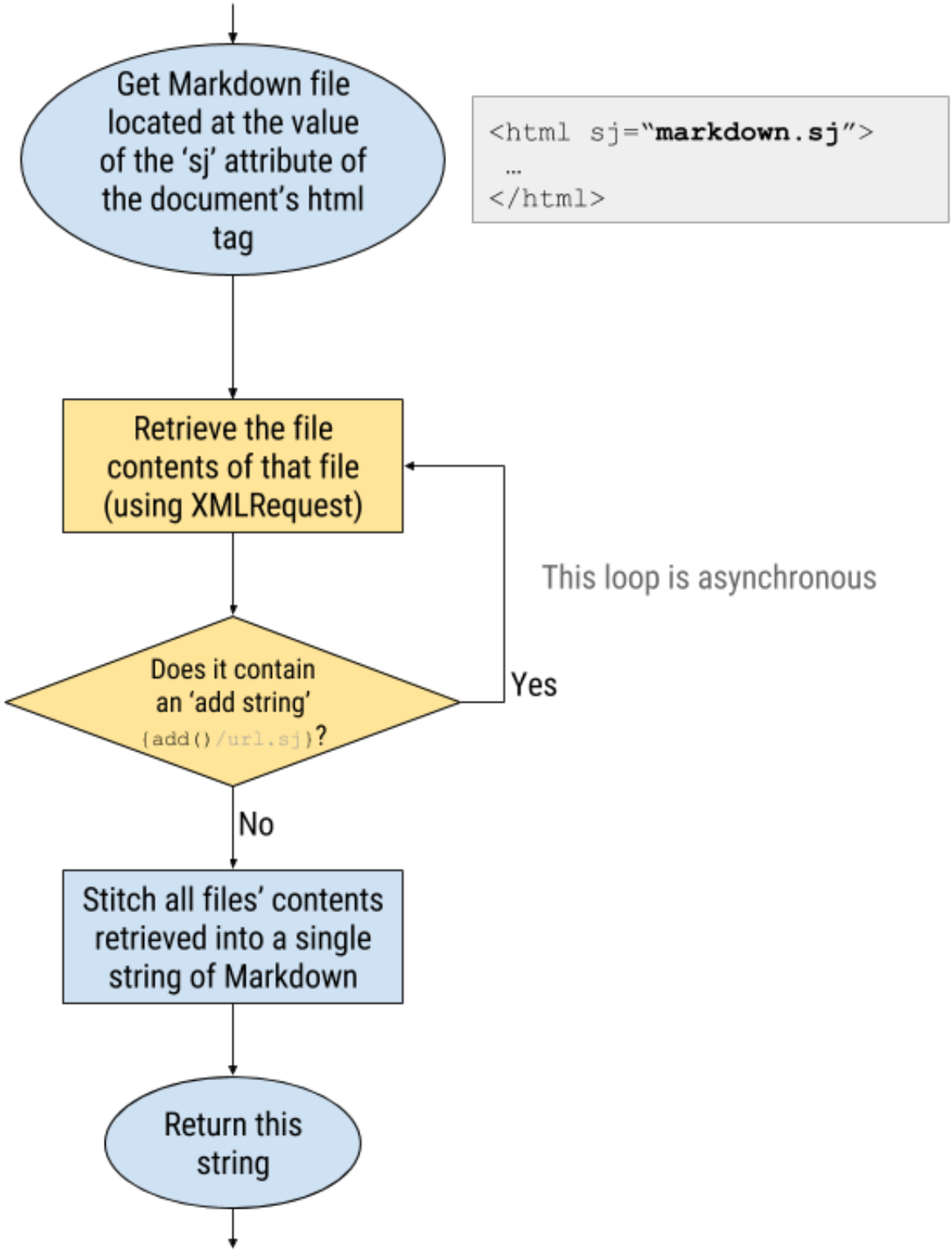
Block Element	Contains
New lines	Nothing, ignored in output
Code block	Raw text
Code fences	Raw text
Horizontal rule	Nothing
Heading	Inline Markdown
Table	Multiple cells of inline Markdown
Blockquote	Block Markdown
List	Multiple lines of inline Markdown
Raw HTML	Text nodes (Optionally read as inline Markdown)
Link definition	Raw text
Paragraph	Inline Markdown

4.4.1.2: The Root Script

Whereas the stylesheet and the compiler would use a proportion of others' code in combination with an extension written by me, the root would be completely self-written.

A fundamental part of the root would be to perform XML requests to retrieve a complete Markdown document.

This would follow this recursive procedure.



This was the most difficult part of the project, because:

1. XML Requests do not have a cross-browser compatible implementation

2. Browsers have enforced AJAX (Asynchronous JavaScript And XML). This means an XML Request cannot be synchronous as the setting for it has been

deprecated universally.

3. The index of a loop will change during an asynchronous call.

4. Recursively performing asynchronous calls in a synchronous way for an indefinite amount of times requires a way of halting JavaScript execution, something which JavaScript is not designed to do.

5. All intuitive solutions to halt execution, like:

```
while (true) {
  if (asynccondition) break;
}
```

will not work in most JavaScript implementations as JavaScript is a single threaded language.

Additionally, this component would be responsible for providing an interface to pass the compiler options. The <html> tag is the interface.

```
<html sj="file.sj" compatibility="true">
    <script src="some.min.js"></script>
</html>
```

This would put the compiler in compatibility mode.

These attributes would be merged into a settings object and this object would be passed as the settings object for the compiler to use.

For example, the compatibility mode would allow those who wish to use standard Markdown notation of bold and italic (`**bold**` , `_italic_` , `<u>underline</u>`) instead of my preferred variation (`*bold*` , `//italic//` , `_underline_`).

4.4.1.3: The Stylesheet

The stylesheet used Skeleton as a frame. A few modifications were made to the original source to better align it with the Markdown produced. Additionally, styles had to be designed to align with the additional objects added. Some were very basic, such as the underline object, whereas some were far more complex,

such as the contents object, to create the period separated number format and to make sub lists of a progressively smaller font size.

Most of it was simple however. Variable width declarations to ensure a good user experience at variable screen sizes.

```
p { font-size: 12px; }

@media (min-width: 600px) {
    /* This rule overrides the former
    when the screen is wide enough */
    p { font-size: 14px; }
}
```

An extension to the stylesheet (and compiler) was made for the purpose of printing. This extension meant three extra objects:

- `{print()contents to show if in print mode}`
- `{donotprint()contents to show if not in print mode}`
- `{break()}` This breaks the page and folds down to the next one.

Here is the actual CSS which handles the logic:

```
/** Word Processor Printing **/
.__print__ { display: none }
.__donotprint__ { display: block }
@media print {
    /* These rules activate when in print mode */
    .page-break {
        height: 1px;
        width: 1px;
        margin: 0;
        padding: 0;
        page-break-after: always;
    }
    .__print__ {
        display: block;
    }
    .__donotprint__ {
```

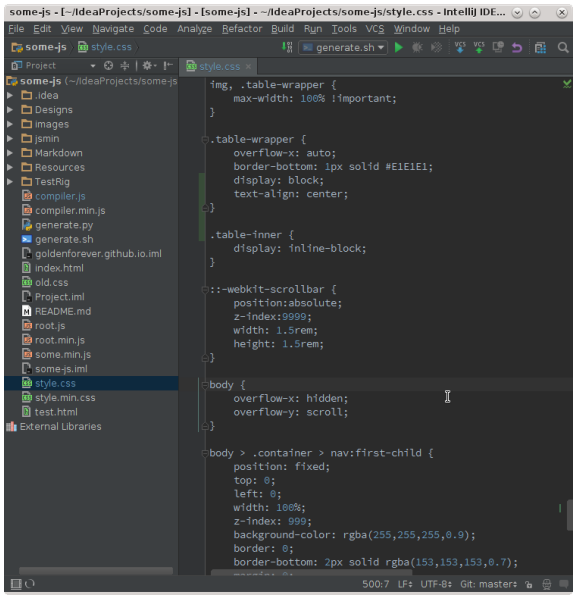
```
        display: none;
    }
    body > .container {
        margin: 0;
        padding: 0;
        width: 100%;
    }
    html {
        font-size: 48%;
    }
    body {
        font-size: 1.7rem;
    }
    h1, h2, h3, h4, h5, h6 {
        margin-bottom: 3rem;
    }
}
```

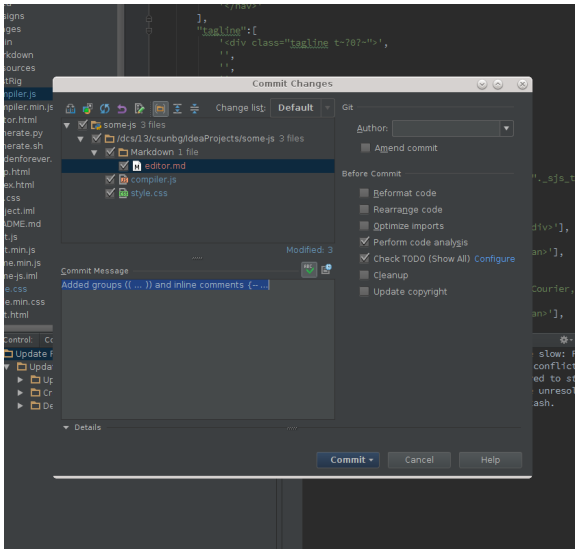
4.4.2: Branding

Halfway through the process, the name of the project was changed from *FreeDOM* to *some-js*. This was as a result of another library called *freedom.js*. This name was chosen as a working name; quite simply I couldn't think of any name and as I own the domain name http://some-website.com it made short term sense.

When I reach version 1.0.0 of the project I will find a better name for permanent use.

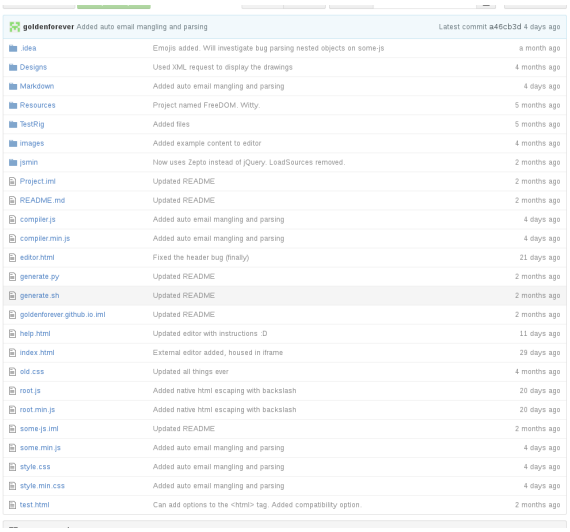
4.4.3: Development Environment

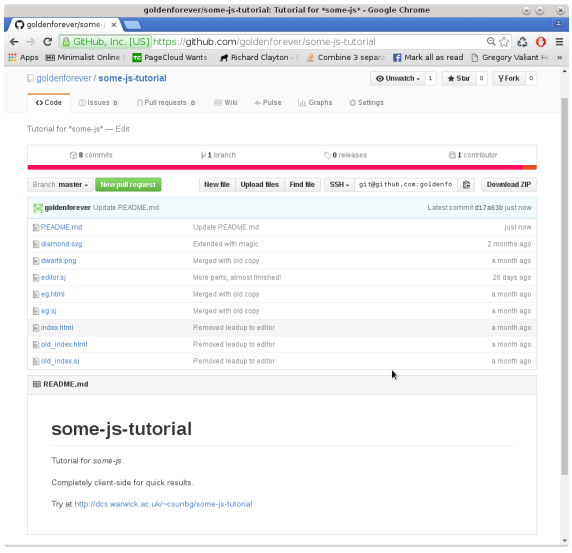




IntelliJ Idea
The IntelliJ IDE was used for development; the green ▶ button was linked to compile the compiler to a single file

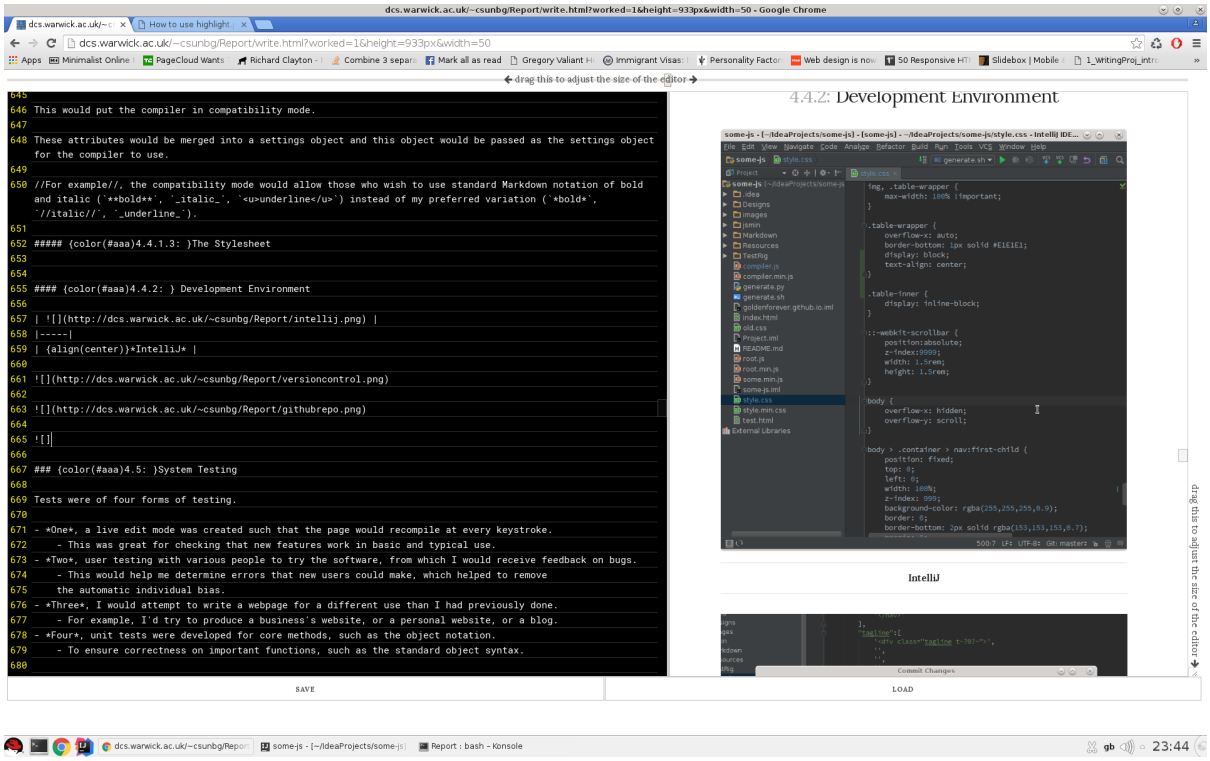
IntelliJ's Github Integration
IntelliJ Idea has built in Github integration which allowed for quick commits and merges.





The some-js Github repository
Changes have been made right up until the 23rd April and this repository will be further developed after the project's end.

The some-js-tutorial Github repository
The some-js tutorial was maintained in a separate repository from the project.



Report Writing
This report was written in the project's own some-js code, in a CodeMirror text editor which would compile after each content change. It was then converted to a *.pdf* using Google Chrome's webpage print service

---

4.5: System Testing

Tests were of four forms of testing.

- **One**, a live edit mode was created such that the page would recompile at every keystroke.
    - This was great for checking that new features work in basic and typical use.
- **Two**, user testing with various people to try the software, from which I would receive feedback on bugs.
    - This would help me determine errors that new users could make, which helped to remove the automatic individual bias.
- **Three**, I would attempt to write a webpage for a different use than I had previously done.

- For example, I'd try to produce a business's website, or a personal website, or a blog.

- **Four**, unit tests were developed for core methods, such as the object notation.
  - To ensure correctness on important functions, such as the standard object syntax.

Notable tests

The Editor	Not strictly an individual test but rather a platform for them, `editor.html` (See **'One'**, above)
Hamish Lacmane	In giving him the tutorial, he found a number of flaws; his technical knowledge also allowed him to be specific with his feedback and showed me a problem with nesting the {modify()} object
Yvette Garfen	As she did not have a background in web development she showed me where my project was less intuitive to the uninitiated, not least the importance of getting inline syntax to extend over the end of lines
Alexandra Cristea	When she used the editor using the HTML source of her website as input she showed me that Markdown would not render when between HTML tags and this was partially resolved (*inline objects do now render inside HTML a separate compiler would be needed for block elements*)
some-js webpage	The project webpage was written using the project. This webpage featured a menu based design. This was the first website created with the project.
some-js-tutorial	The tutorial was also written using the project. As a large part of the tutorial was written in JavaScript and CSS, this allowed me to see how the project would respond when there are large sections of JavaScript. This allowed me to find a very complex problem which stemmed from preprocessing the Markdown string.
My own webpage (1)	The original web page was an attempt at trying to use responsive design alongside the project; it was not a very good experience!

My own webpage (2)	The webpage currently located at http://dcs.warwick.ac.uk/~csunbg/ was an attempt to use as much of some-js's capabilities in one small page
This report	This report is written in the project. The first few weeks of writing the report was plagued with halts from wanting to add a feature to use it myself! For instance, the contents page showed me the issues involved in using CSS to automatically generate section numbers.
Academic webpage	Created a website for my supervisor using the content of her webpage. This allowed me to perform direct comparisons between my language and standard HTML.
Header tree unit test	The tree of headers generated was tested by checking that every header smaller than the previous was a child of it, that every header of equal size was a sibling and that every header larger was a sibling of an ancestor. This was done using a random testing platform; each test string generated using randexp.js[3] and a regular expression.
General web object test	A parameterised tests which tested 11 different input-output pairs; check was if toObject(input) == output. **Test is shown below.**

Here is the test to see if a string of general web object values parses correctly to a JavaScript array.

```
var testPairs = [
    ["", []],
    ["item", ["item"]],
    ["list,of,items", ["list", "of", "items"]],
    ["[list,of,items]", ["list", "of", "items"]],
    ["list,of,items]", "FAIL"],
    ["[list,of,items", "FAIL"],
    ["list,[of,lots,of],items", ["list", ["of", "lots", "of"], "items"]
    ["li\[st,o\,f,ite\]ms", ["li[st", "o,f", "ite]ms"]],
    ["!\"#$%&'()*+,-./0123456789:;<"+
```

```
     "=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ"+
     "[\]^_`abcdefghijklmnopqrstuvwxyz"+
     "{|}~", "FAIL"],
    ["!\"#$%&'()*+,-./0123456789:;<="+
     ">?@ABCDEFGHIJKLMNOPQRSTUVWXYZ\\"+
     "[\\\\]^_`abcdefghijklmnopqrstuvwxyz"+
     "{|}~", ["!\"#$%&'()*+", "-./012345678" +
     "9:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[" +
     "\\]^_`abcdefghijklmnopqrstuvwxyz{|}~"]],
    ["[a,b,c],[d,e,[f,g,h]],i,j", [["a","b","c"],["d","e",["f","g","h"]
];
function testToObject() {
    for (var i=0; i<testPairs.length; i++)
        try {
            if (testPairs[i][1] != toObject(testPairs[i][0]))
                console.log("MISMATCH! [toObject test]" +
                "\nInput:         " + testPairs[i][0] +
                "\nOutput:        " + toObject(testPairs[i][0]) +
                "\nWas expecting: " + testPairs[i][1]);
        } catch (e) {
            if (testPairs[i][1] !== "FAIL")
                console.log("MISMATCH! [toObject test]" +
                "\nInput:         " + testPairs[i][0] +
                "\nOutput:        " + e.message +
                "\nWas expecting: " + testPairs[i][1]);
        }
}
```

4.6: What was achieved

The project is at Version 0.8.12. In it's current state it is very stable and has no (known) major bugs when given well formed code.

Version	Summary
0.1	Added marked and provided interface for use, added Skeleton stylesheet. All files separate for now.
0.2	Created Header Object Model and Menu object to use it. Simple syntax changes like adding underline and changing bold/italic.
0.3	Created General Web Object syntax and added to compiler. Added file loading.
0.4	Recursive file loading. General Web Object rendering better structured. Comment syntax added.
0.5	Menus responsive to width of screen subject to width of content.
0.6	Objects can now nest within each other, indicating the first break from regular language design. This required a major preprocessing step.
0.7	Most structures added. Also general web objects include objects such as bold and italic, which already have their own syntaxes. This is to provide intuitive use without need for a manual.
0.8	Shortcuts and short form css stuctures added to the general object renderer; (for instance {emoticon()} is a shortcut to {emoji()}).

Additional Syntax

The syntax I added to the compiler:

Object	Syntax
General Web Object	`{objectname(setting1,setting2)Comma,Separated,Values`
Underline	`_underlined text_` but not example_user_name; detects word boundaries
Checkboxes	`- [x] checked box`, `- [ ] unchecked box`; used in static lists
Email	Automatic, detects person@place.com and mangles to prevent crawlers from reading

Comments	`@@@`
	`Block comment`
	`which works for multiple lines`
	`@@@`
	
	`## Report @@ Only this is commented out`

The syntax (by default) changed:

Object	Previous Syntax	New Syntax
Bold	`__bold__` or `**bold**`	`*bold*`
Italic	`_italic_`, `*italic*`	`//italic//`
Underline	[none]	`_underline_` (replacing italic)

Web Objects

The general web object gave me a framework where I could add objects quickly. There's no elegant way to list them out; so here they are:

Object	Description
`menu`	Creates a menu that 'opens' a header and it's children for viewing, values are the header ids
`modify`	Applies a CSS style to the containing element; e.g. {modify(opacity,0.5} would make the element translucent
`tagline`	*Shortcut to subtitle*
`subtitle`	Auto sizes font based on contents
`break`	Breaks the page
`print`	Contents only appear when the webpage is in print mode
`donotprint`	Contents only appear when the webpage is not in print mode
`icon`	Produces a FontAwesome icon using the first setting as the icon id
`monospace`	Changes font to a fixed-width font
`typewriter`	*Shortcut to monospace*
`mono`	*Shortcut to monospace*
`fixedwidth`	*Shortcut to monospace*
`font`	First setting is the name of the font to use
`header`	Produces a header equal in size to one produced by first setting's number of hashes
`heading`	*Shortcut to header*

`color`	First setting is the colour of the text
`vspace`	vertical space, first setting is the space you want to add (or subtract)
`hspace`	horizontal space, first setting is the space you want to add (or subtract)
`vertical`	*shortcut to vspace*
`horizontal`	*shortcut to hspace*
`date`	Generates a human readable date of the current day (with respect to the client)
`bold`	Generates bold text
`italic`	Generates italic text
`underline`	Generated underlined text
`super`	Allows content to be treated as superscript text
`sub`	Allows content to be treated as subscript text
`group`	Groups elements together so they can be modified with {modify()}
`align`	Aligns the adjacent text to the first setting, one of {left, right, center, justified}
`highlight`	Highlights the content, first setting is the highlight colour; uses filters to imitate slight darkening of text
`background`	*Shortcut to backgroundimage or backgroundcolor*, checks if setting looks like a file address to determine
`backgroundimage`	Sets the background image of the container to the first setting

[illegible]	first setting
`backgroundcolor`	Sets the background colour of the container to the first setting
`comment`	Removes contents from output
`escape`	Does not read contents as Markdown or HTML, escapes each character
`rule`	Generates a horizontal rule
`contents`	Generates the contents based on the heider heirarchy. First setting is where to start from, second is where to end the introduction list and start the main contents, third is where to finish the main contents and start the appendices, the fourth argument is where to remove to finish. Each setting is a string that will select the first heading title which contains it fully
`emoji`	Generates SVG scalable EmojiOne emojis, the content is a comma separated list of emoji codes
`emoticon`	*Shortcut to emoji*
`up`	Move the contents up the first argument distance. Does not affect surroundings
`down`	Move the contents down the first argument distance. Does not affect surroundings
`left`	Move the contents left the first argument distance. Does not affect surroundings
`right`	Move the contents right the first argument distance. Does not affect surroundings

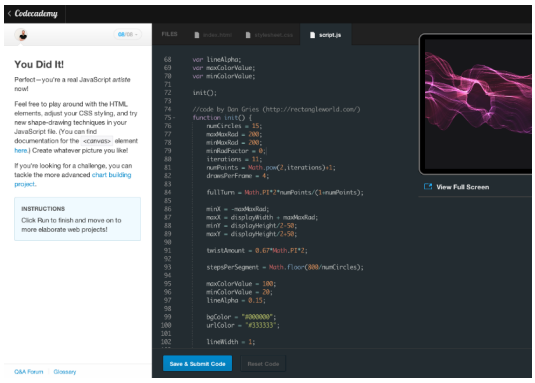
Tutorial

The interactive tutorial was an important part of getting feedback and priming someone with the project quickly so I could get feedback from as many people as possible.

The tutorial used Codecademy, and similar projects, as inspiration for the design.

Features:

- Written using some-js itself
- Automatically detects if the code is correct
- You can go back and forward through the exercises
- You can edit freely
- Has over 10 steps
- Led straight into Google Forms feedback form at end

Codecademy screenshot	some-js-tutorial screenshot
	

The editor has handy cheatsheets:

And it looks like this towards the end:

The tutorial is live at http://some-website.com/some-js-tutorial/.

5: Evaluations

Evaluation came in many forms through the project.

The user testing not only helped me spot bugs but it also was a fantastic source of feedback throughout.

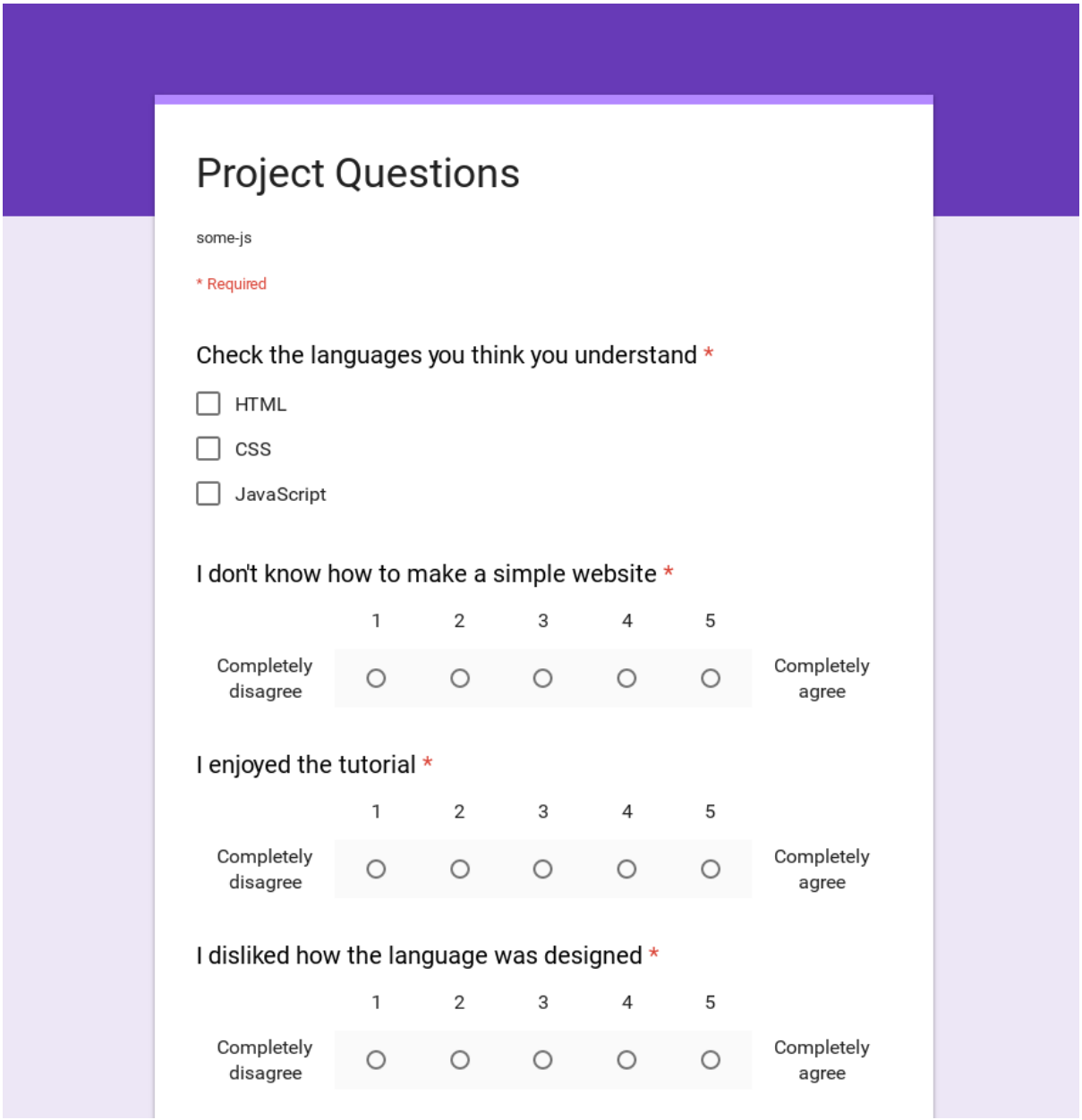
I got in touch with a world leading expert in the field of markup languages (Professor John McFarlane, developer of Pandoc, a universal document converter; he also wrote BabelMark which is a Markdown compiler benchmark) to ask him what he thought of the project's motivation and any advice he would give.

> There are many ways you might add these sorts of features. Usually people use the Markdown for the site's content, and use templates to add things like menus and the overall page structure.
> - John McFarlane

He also suggested some links where I might find some help in terms of forms and inputs.

The best source of evaluation was undoubtedly from the forms filled out by those who completed the tutorial. I received 41 valid and complete forms. The questions asked on the online form have been placed in Appendix V.

Here is a picture of what the form actually looked like to those questioned:



The form is accessible at this address: http://goo.gl/forms/XJakqxGmu1

The form has an optional open feedback section but unfortunately, none of the 41 used it.

The main three results to come out were that 27 out of 41 (65.8%) strongly agreed that they learnt something, 25 out of 41 (61.0%) 'strongly' liked writing the language but just 10 out of 41 (24.4%) said that they will definitely use it on their next website.

The sample size was two small to provide any hypothesis testing or form any conclusions to a significance level < 10%, but the results are encouraging.

6: Discussion and Further Work

6.1: Author's Assessment of the Project

6.1.1: General Conclusions

The project has been fantastic from start to finish. By spreading the work over the 2 terms and holidays evenly I've managed to do a lot of work on an ambitious project. As a member of the set of target users I get to benefit from using the project to make websites but also documents such as this one. Given the volume of work I've done using it even during it's development I have no doubt of its value to myself.

However, this project is not about creating something for yourself, which is why care had to be taken at every stage to ensure the standard was high and the project was generalised. Putting the US spelling of colour as the web object name is a pertinent case where I've tried to stick to it as best as possible.

When my friends who study in the University of Warwick Department of Computer Science used it, they were impressed, and as they are part of this user group too, it is a good sign.

Facts always triumph over impressions and in order to assess how much more simple my project was in practice, I created two websites with it.

One was my own. http://dcs.warwick.ac.uk/~csunbg/

It had a strong emphasis on merging some-js code with HTML to produce code with brevity, a blend between the fast writing of some-js and the full-featured HTML, CSS and JavaScript. It has approximately half some-js to half HTML content ratio.

Time taken	Number of characters in code	Number of characters in generated HTML
25 minutes	4034	16777

The other was a direct equivalent of the academic homepage of my supervisor, Dr. Cristea. http://dcs.warwick.ac.uk/~csunbg/aic/

It had a strong emphasis on being as pure in some-js code as possible. Websites that have a large volume of content benefit most from using some-js as the user can better see the structure of the content in its plain text form.

Time taken	Number of characters in code	Number of characters in generated HTML	Number of characters on original page
15 minutes	10671	24605	43840

If only as a method by which to compress the code sent to the client, this project has shown value. But the projects orignal and main aim is to ensure it's easier too.

6.2: Comparison versus the Specification

Objective	Target	
Fast to write	***to be minimised***	✔
Little	***to be***	Some of those who 'learnt something' included those who had no previous knowledge of

knowledge	***minimised***	had no previous knowledge of HTML, CSS or JavaScript ✔
Further accommodate	Each web language could still be easily included	Through native <style> and <script> tags ✔
Documented	All encompassing documentation	Documentation on the some-js website ✔
Consistency	The website should have continuity of style	Followed the design principles of Skeleton in added features ✔
Content-aware	At minimum, looks for different ‘types’ of page content to match	Minimum not met. Menus collapse and expand based on a menu independent breakpoint, and subtitles sizes are dependant on content but generally this goal was not achieved, nor was it attempted. This objective was removed from the short term scope of the project as it would be difficult to implement and need a slow algorithm to get good results ✖
Modern style	Use modern principles and generally not look dated	Minimalist design which avoids imposing a style on the user ✔
Responsive design	Must work for $[\ \mathbf{x}$ px by $\mathbf{y}$ px $\mid 320 \leq \mathbf{x}, \mathbf{y} \leq 1920 \wedge 0.5625 \leq \frac{\mathbf{x}}{\mathbf{y}} \leq 1.\dot{7}\ ]$	Responsive only by width; broswer support for height and aspect ratio is inconsistent. JavaScript implementation maybe? ❓
	1 second	

Fast to load	1 second maximum based on 7.4Mbit/s	✔
Small file size	Upper bound 6.1Mbits to comply with above	Library < 1 Mbit ✔

6.3: Academic/Research Lessons Learnt

I've shown myself that when researching something I'm interested in, I'm able to produce useful code for a target audience.

The difficulties involved in creating a fair assessment setup through the tutorial took a large amount of time. That said, it was interesting to get the feedback afterwards and it was awesome to hear from John MacFarlane who is to markup as Donald Knuth is to typesetting!

The web community is an immense source of knowledge and I feel as though I've become a part of that community through my work being published on Github.

From an academic point of view, the Compiler Design module has complemented this project well, preparing me for designing a real world compiler. JavaScript is not a very rigid language and whilst that often drives people (inlcuding me) insane, it's uniquities under the surface proved to be of great interest! I've also learnt a lot about synchronous vs asynchronous execution (particularly in JavaScript) and on the most general level, how to design for web efficiently.

6.4: Self Assessment

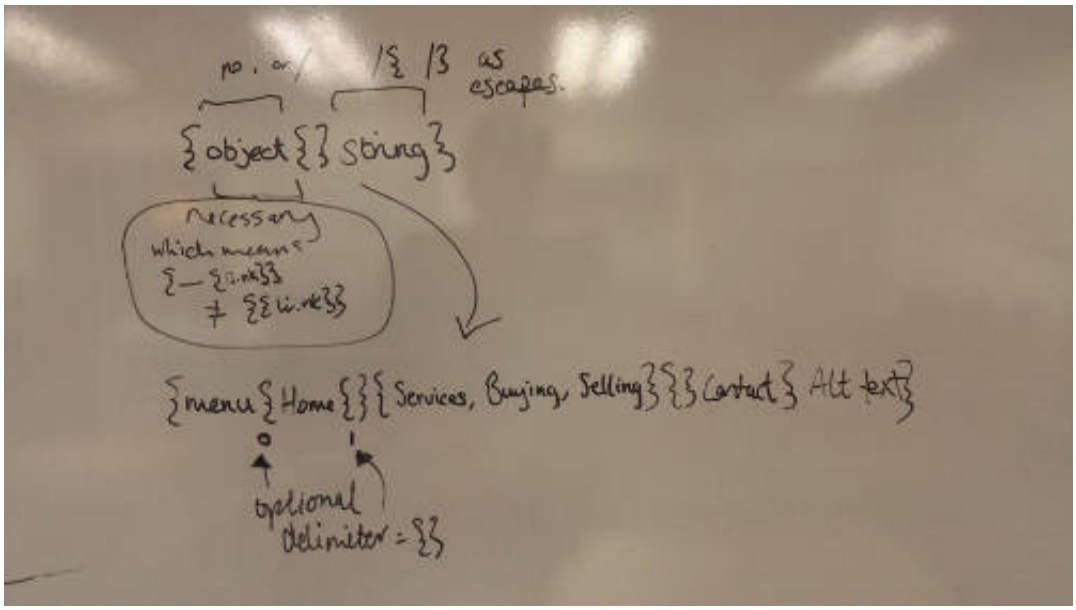
Put simply, I worked hard on this project. It did distract me from other key modules at times and that I regret. Time management across the whole degree is something that I could do with improving.

Nevertheless, my work on the project was very sound. In places it was a bit

unstructured where a plan would have better served, although on the other hand it was probably the reason I managed to get such a volume of work done with the implementation.

I learnt a small wonder about maintaining and developing other people's code. There was very few comments on how the compiler describing how it worked, instead relying on a rigorous appreciation of compiler design which I felt I dealt with well. Sometimes writing down step by step with a pen and paper is a simple way to get a wider understanding of what a program does.

Drawing ideas on a whiteboard, as simple as it sounds, helped me a lot, especially for designing new syntax.



My original sketch for the General Object Notation
The syntax proved to be too difficult for a regular expression to read, even with preprocessing hints, so this design was later changed

6.5: Future work

The project leaves plenty of gaps to fill. More objects, more tests, a through code review.

But in terms of features to add, the most important feature to add would be a well designed general web object interpreter. Currently, when the compiler

detects a well formed web object with an incorrect number of settings it throws an error. This isn't a very encouraging sight for a new user, especially one using the editor, where the errors appear in place of the webpage

6.5.1: Research question

- How many special syntaxes can a small markup language have before additional ones become more of a hindrance than a help in terms of productivity?

- And given the ASCII character set, what is the number of discrete syntaxes that would be needed to minimise the filesize of a typical formatted plaintext document? Is there a limit to the number of syntaxes humans need? And is this number less than the number of syntaxes producable from ASCII?

6.5.2: Completely new algorithms to try

A major extension would be contaxt aware design. I wasn't able to do it as part of my project, but the speed of JavaScript gives me faith that if done in the right way it would be possible to produce something that could generate quickly and have telling improvements. For instance, a serif font for a more serious post or a display font for a large header.

Also I wanted to integrate Less.js into the design from the ground up, as it applies the same principles some-js applies to HTML, to CSS.

Finally, I would have liked to create add a syntax to interact with CSS directly, rather than using <style> tags. Even better would be a merge of this point and the previous, potentially allowing the theme to be chosen through the use of Less.js variables which could in turn be set by this additional syntax.

This would be a great way to efficiently implement modular webpage design that resembles in concept something like Google's Project Ara; the modular smartphone that is comprised of mostly interchangeable and replaceable parts.

7: Conclusion

In conclusion, it would be fair to say the project has been a success, as it has tangibly achieved a purpose, which is to simplify web development. It is a quicker procedure and people using the tutorial managed to learn how to make a small website in just ten minutes.

Perhaps the most important change was to add the general web object syntax which provided word shortcuts to functions even if you forgot the Markdown-esque syntax for them.

And in not restricting the use of inplace HTML, it merges well with any previous knowledge of web development a user might have.

However there are rough edges, and the language is far from a complete alternative yet. Stylesheets still have to be included to change the page's design on a universal level which means there is still a dependency on HTML. That's probably the next issue to tackle.

Yet if that is properly addressed, the project becomes very close to a solution complete enough for some users to avoid any HTML, CSS and JavaScript in their code, and that would be the stimulus to actually release the project as an open-source piece of software!

8: References

[1]: World Wide Web Consortium (2010) HTML5 Reference: The Syntax, Vocabulary and APIs of HTML5, Section 3.2.3.1. Available at http://www.w3.org/TR/2010/ED-html5-author-20100809

[2]: RapidTables.com (2014) Screen Resolution Statistics. Available at http://www.rapidtables.com/web/dev/screen-resolution-statistics.htm

[3]: Roly Fentanes. randexp.js. Available at http://fent.github.io/randexp.js/

[4]: Ofcom. Broadband Speeds UK. http://media.ofcom.org.uk/content/posts/news/2015/one-in-three-uk-broadband-superfast

freedom.js (2014). Available at: http://www.freedomjs.org/

Masset, P. Minimalist Markdown Editor. Available at: http://markdown.pioul.fr/

Gruber, J. (2004). Markdown. Available at: https://daringfireball.net/projects/markdown/

International Organization for Standardization (1986). Standard Generalized Markup Language (SGML). Available at: https://www.iso.org/obp/ui/#iso:std:16387:en

Berner-Lee, T. (1993). Hypertext Markup Language (HTML). Geneva: CERN. Available at: https://www.w3.org/MarkUp/draft-ietf-iiir-html-01.txt

World Wide Web Consortium (2014). HTML5. Available at: http://www.w3.org/TR/html5/

Dubost, K. (2008). 'HTML5, One Vocabulary, Two Serializations'. Available at: https://www.w3.org/blog/2008/01/html5-is-html-and-xml/

Codecademy, Inc. Located at: http://www.codecademy.com

Appendices

I: Program Instruction

Copy and paste this code into a single .html file. There has been an unfixed bug with the compiler's compiler and as a result the script does not read the content from the textarea tag.

If this does not work copy and paste **the contents** of the <textarea> tag and paste it into the live editor located at http://dcs.warwick.ac.uk/~csunbg/some-js/editor.html. This is also included as editor.html in the submission zip.

```
<!DOCTYPE html>
<html>
<script src="http://dcs.warwick.ac.uk/~csunbg/some-js/some.min.js"></sc
<textarea>
# This is the simplest way...
...to get started using some-js.

Copy and paste this file into a HTML document.

You can also load a some-js (.sj) file
externally by setting the sj attribute to
the &lt;html sj="path/to/file.sj"&gt; and
by removing this &lt;textarea&gt; element.

Externally loading a file  will require the
file to be hosted somewhere though, and the
file will have to be on the same domain.

This external loading issue extends to using
null tags in your code.

A free option is to use
Github's free hosting service.

---
```

```
_Disclaimer:_ I've just noticed that the compiler
compiling script; that is the script which merges
the individual files together, has stopped working.

The scripts don't seem to execute yet have the
correct permissions. Maybe you will have better
luck. Simply execute ./generate.sh to compile a new
some.min.js file.

---

Alternatively, you can just use this!

Hope this all helps,
Tom
</textarea>
</html>
```

II: Aside about HTML vs XML

A common misconception about HTML is that it is XML. HTML is neither a derivative nor valid XML. Indeed, XML was created after HTML. In HTML5, XML was included as an option in the World Wide Web Consortium's (W3C) specification for the first time as a disjoint serialisation, XHTML, but the original and current HTML serialization is not XML.

The two main differences are that HTML does not always require:

- forward slashes in empty element syntax
- close tags.

The following HTML is invalid XML for both reasons:

```
<html>
    <head>
    </head>
```

```
    <body>
        <p>This is a paragraph. This tag will
           break the line<br>
           like so.
    </body>
</html>
```

The
 tag indicates a line break and **cannot** have children tags inside it. To indicate this in XML a forward slash is needed.

The <p> tag indicates a paragraph and **can** have children tags inside it. As per the XML specification it must close the tag before its parent tag (<body>) is closed.

This is a valid XML version of the above.

```
<html>
    <head>
    </head>
    <body>
        <p>This is a paragraph. This tag will
           break the line<br/>
           like so.</p>
    </body>
</html>
```

HTML uses the context of tag names to make decisions on how to parse the document. XML doesn't use any context. As a result, XML is faster to parse and has an LL(1) grammar whereas HTML has valid LL(k) grammars.

III: Requirement Specification

From the specification:

> The method should be:
>
> - fast to write (important)

- require little knowledge of web languages (important)
- further accommodate those with knowledge of web development
- documented

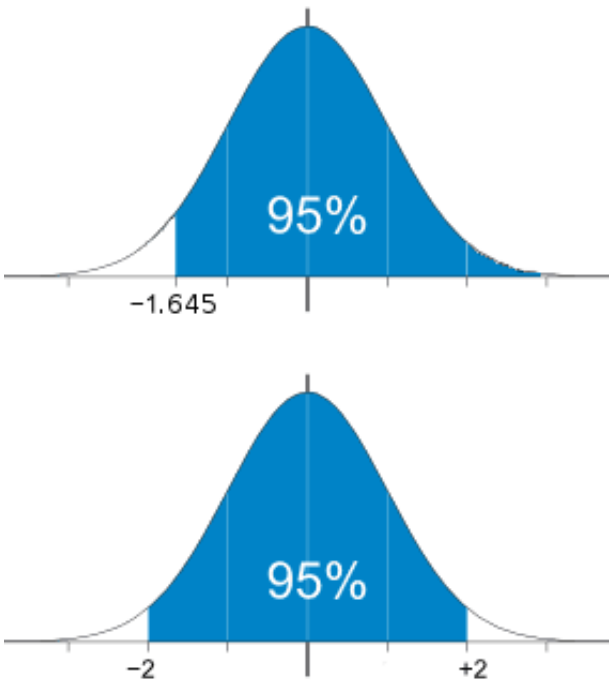
The resulting website should have:

- a level of consistency
- content-aware display
- basic modern styling
- responsive design
- fast to load
- small (file size) without compromising functionality

Targets

The project aims to have a balance between ‘one-size fits all’ and aggressively context-predictive content display.

I will set this balance to be that which satisfies 95% of the population for quantifiable metrics.



Objective	Target
Fast to write	***to be minimised***
Little knowledge	***to be minimised***
Further accommodate	Each web language could still be easily included
Documented	All encompassing documentation
Consistency	The website should have continuity of style
Content-aware	At minimum, looks for different ‘types’ of page content to match
Modern style	Use modern principles and generally not look dated
Responsive design	***see below***
Fast to load	***see below***
Small file size	***see below***

Responsive design

This table shows popular screen resolutions.

Screen resolution	Display ratio	Usage	Screen size / type
1366x768	16:9	19.1%	14” Notebook / 15.6” Laptop / 18.5” monitor
1920x1080	16:9	9.4%	21.5” monitor / 23” monitor / 1080p TV
1280x800	8:5	8.5%	14” Notebook

1280x800	8:5	8.5%	14" Notebook
320x568	9:16	6.4%	4" iPhone 5
1440x900	8:5	5.7%	19" monitor
1280x1024	5:4	5.5%	19" monitor
320x480	2:3	5.2%	3.5" iPhone
1600x900	16:9	4.6%	20" monitor
768x1024	3:4	4.5%	9.7" iPad
1024x768	4:3	3.9%	15" monitor
1680x1050	8:5	2.8%	22" monitor
360x640	9:16	2.3%	
1920x1200	8:5	1.7%	24" monitor
720x1280	9:16	1.6%	4.8" Galaxy S
480x800	3:5	1.1%	
1360x768	16:9	0.9%	
1280x720	16:9	0.9%	720p TV
		84.1%	

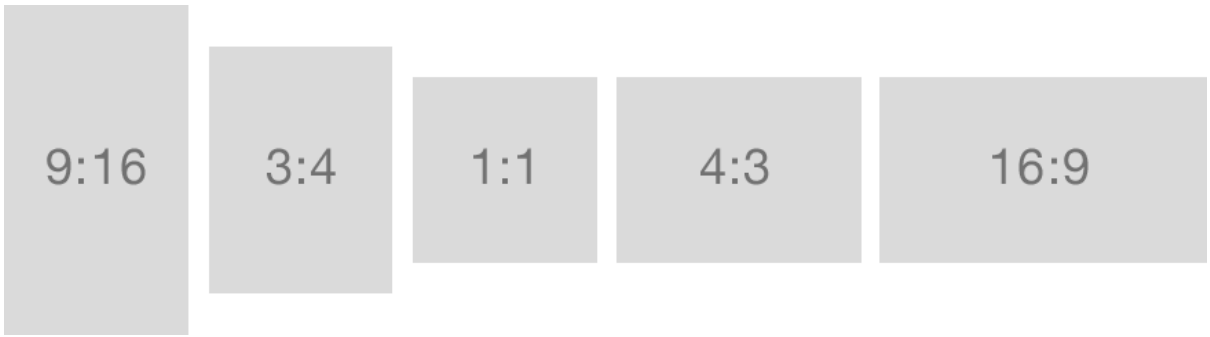
The 320x480 of the first three generations of the iPhone seem to provide a lower bound as they are responsible for over 5% of web-page loads.

The 1920x1080 resolution will serve as an upper bound as larger screens are responsible for less than 5% of web-page loads.

For the same reason, the screen will need to support resolutions between 16:9 in

landscape to 16:9 in portrait.

i.e. all of these (provided the short side is at least 320px, and the long side is no more than 1920px):



Fast to load / Small file size

As part of Ofcom's duties as a regulator, they used this data[4] from SamKnows (11/2014).
The minimum average download speed for a currently offered service (and over 99.5% in use are) in the UK is 7.5Mbit/s. Google says a website should aim to have loaded within a second and so = 7.5(1-x)Mbits where x is the time taken to create the website.

Therefore I will sit an upper bound on 7.5Mbits as the size of the website **including** the assets.

The average size of the images on a web-page is 1.4Mbits (http://httparchive.org/trends.php), so I will subtract the size of these assets from the upper bound to make a new ***upper bound of 6.1Mbits***.

Video would not affect the initial loading time on most modern browsers as it is part of the HTML5 standard.

Adding hosted libraries (js/css/fonts) through runtime adding of links seems to be a very logical way to mitigate this upper bound whilst still retaining a wide range of output formats.

> **Update:** As of 17/12/15 the single JavaScript file size is about 50KB in raw size (unminifed, uncompressed).

In combination of the raw assets, this size increases to 171.1KB ~ 2.8% of maximum allowed.

Interestingly, this website benefited greatly from Markdown being a smaller file than the HTML equivalent.

The Markdown files that generate this website are approximately 44KB in total - the minimised HTML equivalent is 64KB and the unminimised HTML equivalent is approximately 104KB.

This means an approximate 20KB saving in file size, which is almost half the size of the unminimised Markdown compiler itself!

If the file is gzipped (compressed), you can expect the Markdown file size to be reduced by 70-75% as it is not far from plaintext. If the markdown files used to generate this website's content were gzipped they would only be approximately 13.2KB. *This would suggest websites that have a large amount of text content would be require a smaller total file transfer using a Markdown runtime compiler*, which is an interesting but beneficial byproduct of the project.

If the Markdown compiler JavaScript file was cached, the smaller total file transfer would no longer be offset by the size of the compiler and therefore it would provide a smaller total file transfer for almost all websites.

The conclusions of the report are in the Requirement Specification section of this report.

This specification is also located at http://dcs.warwick.ac.uk/~csunbg/some-js/#requirements-specification.

IV: Component Analysis Report

A Markdown-like language, that is one which outputs from a concise and plain text input to a formatted output (typically including HTML), is called a '*lightweight markup language*'. There are other options and they include *Setext* and *reStructuredText*; both of which served as inspiration for Markdown as a language.

There is a very good and in-depth comparison of these languages on Wikipedia (which treats different Markdown extensions as separate ones for comparison's sake.

Language	HTML export tool	HTML import tool	Tables	Link titles	`class` attribute
AsciiDoc	Yes	Yes	Yes	Yes	No
AFT	Yes	No	Yes	Yes	No
BBCode	No	No	Yes	No	No
Creole	No	No	Yes	No	No
deplate	Yes	No	Yes	No	Yes
GitHub Flavored Markdown	Yes	No	Yes	Yes	No
Jemdoc	Yes	No	Yes	Yes	No
KARAS	Yes	No	Yes	Yes	Yes/No
Markdown	Yes	Yes	Yes/No	Yes	Yes/No
Markdown Extra	Yes	Yes	Yes	Yes	Yes
MediaWiki	Yes	Yes	Yes	Yes	Yes
MultiMarkdown	Yes	No	Yes	Yes	No
Org-mode	Yes	Yes	Yes	Yes	Yes
PmWiki	No	Yes	Yes	Yes	Yes
POD	Yes	?	No	Yes	?

reStructuredText	Yes	Yes	Yes	Yes	Yes
Textile	Yes	No	Yes	Yes	Yes
Texy!	Yes	Yes	Yes	Yes	Yes
txt2tags	Yes	Yes	Yes	Yes	?

Markdown will be used as the base of the idea.

This is an obvious choice, there is no similar language, it is very well designed and has plenty of JavaScript compilers.

GFM (**G**ithub **F**lavoured **M**arkdown) is the variant of Markdown which is used by Github.

GFM will be a necessary add-on, and will have to be added to the compiler if it is not already included. This is so the project provides complete compatibility for people who have written material on Github in the past, which is a large proportion (over 5%) of the share of the group of users who are more technical and are using the project for it's speed of development.

Compiler

There are many open licence JavaScript Markdown compilers and these are the most popular.

Compiler	markdown-js	marked	µmarkdown	showdown
Size (bytes)	16,750	16,528	10,097	26,423
Speed (x1000)	17191ms	3727ms	n/a	17191ms
Supports GFM	✖	✔	✖	✔

The most difficult part of GFM to recreate are tables. These would take a large amount of time to add, and as a result I will exclude *markdown-js* and

μmarkdown from the choices.

The difference then comes down to size and speed; ***marked*** is superior to *Showdown* at both.

I will therefore choose *marked* as my Markdown parser.

Framework

There are also a number of HTML/CSS/JS frameworks that are meant to ease web development.

Here the aim is not to make this available to the end user but rather to use it as a base for developing from.

Consequently, I will choose one that has a small file size.

I found this link useful, and decided to compare each one listed.

Framework	Pure.css	Skeleton	Furtive	Min	RocketCSS
Size (KB)	17.2	5.9	10.2	2.3	6.4
Grid	✔	✔	✔	✔	✖
Grid precision*	7	5	4	5	n/a
Font-size in rem	✖	✔	✔	✖	✖
Responsive	✔	✔	✔	✖	✖
Appearance**	2	1	3	5	4

> *Grid precision is here defined as the minimum number such that you cannot use the framework to generate a grid of columns with equal size.
> **Appearance is subjective but is included as it will reduce the amount of

> changes I will need to make.
> Ranked 1(best) to 5.

Whilst Min offers good support, responsive design is a requirement is needed. Equally RocketCSS has the same issue. Therefore neither would be suitable.

Skeleton is equal or better in the remaining categories (with the exception of grid precision, which can be fixed manually) and is much smaller and more attractive, it seems to be the best choice.

I will therefore choose *Skeleton* as my design framework.

V: Feedback Form Questions

Some questions were asked in a positive viewpoint, some in a negative. This mix allowed me to filter entries which selected all 1s and 5s. This removed 3 entries (44), leaving me with 41.

If unspecified, question is from 1 to 5, 1 being completely disagree, 5 being completely agree.

- Check the languages you think you understand
  - ☐ HTML [28]
  - ☐ CSS [15]
  - ☐ JavaScript [14]
- I don't know how to make a simple website

  1 `[13] [12] [0] [4] [12]` **5**
- I enjoyed the tutorial

  1 `[3] [0] [0] [1] [37]` **5**
- I disliked how the language was designed

  1 `[20] [3] [6] [0] [12]` **5**
- I liked writing the language

1 [8] [1] [1] [6] [25] **5**

- I will use this to make my next website

  1 [20] [8] [1] [2] [10] **5**

- If this was language was more widely used I would use it to make a website

  1 [16] [8] [2] [3] [12] **5**

- I find drag and drop interfaces stop me from making the website I want to

  1 [9] [1] [10] [16] [5] **5**

- I feel like I didn't learn anything

  1 [27] [3] [2] [3] [6] **5**

VI: Reading list

Kumar et al (2013). Design Mining The Web: Webzeitgeist. Located at: http://vis.stanford.edu/files/2013-Webzeitgeist-CHI.pdf

The specification for Markdown by John Gruber is a very important read to grasp an overview of the topic. A link is in the references section.