

University of Dublin



## TRINITY COLLEGE

The design and development of a device-responsive educational app  
(for mobile, tablet and web) based on a historical, educational  
program

**Charles Hebert**

B.A.(Mod.) Computer Science and Business

Final Year Project April 2016

Supervisor: Richard Millwood

School of Computer Science and Statistics

O'Reilly Institute, Trinity College, Dublin 2, Ireland

# Declaration

I hereby declare that this project is entirely my own work and that it has not been submitted as an exercise for a degree at this or any other university.

---

Charles Hebert - April 11th 2016

# Abstract

This report describes the process in designing and implementing a modern adaptation of the educational application, 'WordRoot'. WordRoot aims to teach the meaning of words and their roots whilst entertaining the user.

In particular, this project explores Single Page Web Application Development, Test Driven Development and The Reproducible Build.

# Table of Contents

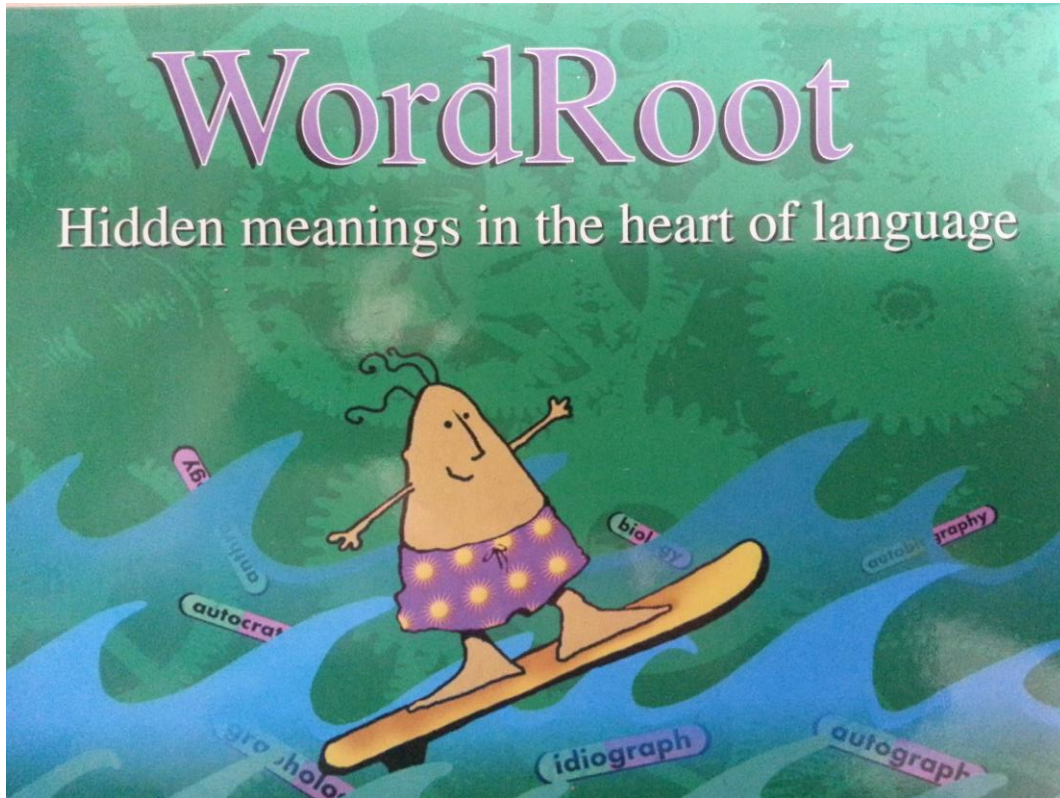
Declaration.....	1
Abstract.....	2
1 Introduction .....	5
1.1 Historic App Background.....	5
1.2 Web Applications.....	7
1.2.1 Client-Server Model .....	7
1.2.2 Comparison with Desktop/Native Applications .....	8
1.2.3 Traditional Web Applications .....	8
1.2.4 Single Page Web Applications .....	8
1.3 Test-Driven Development .....	13
1.3.1 Benefits.....	14
1.3.2 Limitations.....	15
1.3.3 Test Types .....	16
1.4 The Reproducible Build .....	16
2 Design.....	18
2.1 Single Page Client-Server Model.....	18
2.2 Database .....	19
2.2.1 What's in a Word? .....	19
2.3 Restful API .....	20
3 Implementation .....	21
3.1 Build and Tooling.....	21
3.1.1 Shell/Batch Scripts.....	21
3.1.2 Jake .....	21
3.1.3 Browserify .....	22
3.1.4 ShellJS.....	22
3.1.5 JSHint .....	23
3.1.6 Karma .....	23
3.2 Application.....	24
3.2.1 Client.....	24
3.2.2 Server .....	25
3.2.3 Client and Server .....	25
3.2.4 Restful API Data.....	25

4 Conclusions .....	27
4.1 The Application.....	27
4.2 Difficulties .....	27
4.3 Further Work .....	29
4.4 Additional Tools .....	30
4.4.1 React .....	30
4.4.2 Sassy CSS (SASS) .....	30
4.4.3 Clojure/ClojureScript .....	30
4.5 Closing Remarks .....	31
5 References .....	33
6 Appendix.....	33
6.1 Appendix 1 .....	33
6.1.1 WordRoot Mobile View.....	33
6.1.2 WordRoot Desktop View .....	34

# 1 Introduction

This final year project is carried out in support of the UK National Archive of Educational Computing apps<sup>1</sup>. The aim of this project is to design and develop a device-responsive educational app based on a historical educational program. The author has chosen to base their project on the app, 'WordRoot'.

## 1.1 Historic App Background



*Figure 1: Front cover of WordRoot CD*

WordRoot (WR) is an educational application that was first released in the 90's by John Davitt. It was originally designed to teach children the meaning of words in an interactive, entertaining manner. The application is made up of three main components: 'wordlink', 'the place to study individual words'; 'accent map', listen to regional accents recorded live around the UK; and 'wall of talk', a place to 'browse that sound'.

WR is a desktop application. It was originally created using Macromedia Flash and Quicktime.

This project is focused solely on developing the 'wordlink' component (see figure 1). From now on, the reader should assume that the author's use of 'WordRoot' applies to this 'wordlink'

---

<sup>1</sup> <http://www.naec.org.uk/>

component. Figure 2 shows the user interface and potential actions the user could take in this component:

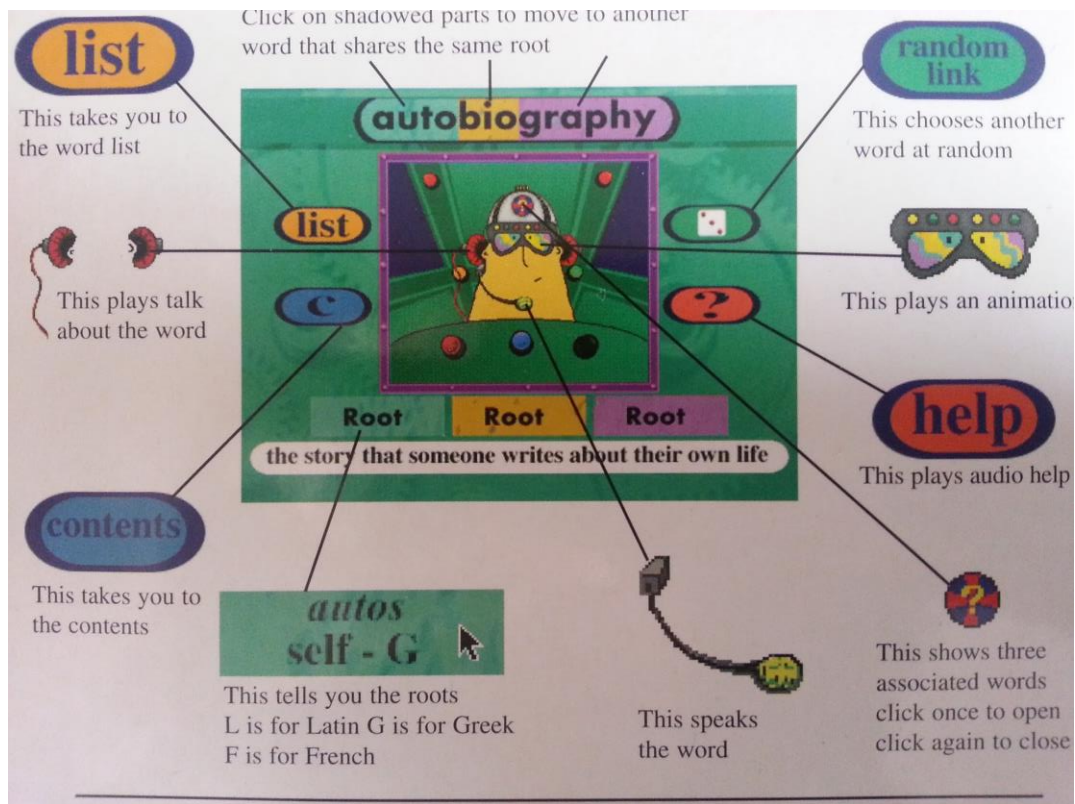


Figure 2: Back cover of WordRoot CD

There are a number of key features of this component:

- Displays the meaning of the word
- Displays information about each root of a word
- Plays an animation based on the word
- Plays a recording of the word
- Plays a recording of a story about the word
- Navigates to related words
- Displays a list of words

To see a video explanation from the creator, the reader may visit [this page](https://www.youtube.com/watch?v=gkoliNBrWao&ab_channel=newtools)<sup>2</sup>.

<sup>2</sup> [https://www.youtube.com/watch?v=gkoliNBrWao&ab\\_channel=newtools](https://www.youtube.com/watch?v=gkoliNBrWao&ab_channel=newtools) [Accessed 1 April 2016]

## 1.2 Web Applications

Today, everyone who has access to the Internet is familiar with web applications. Google, Facebook, Wikipedia, Youtube - these are all household names. Merely a couple decades ago, network infrastructure was in a state such that it was only feasible to distribute basic, static web pages. If developers wished to distribute a complex application to users, it had to be burned to disk. The user then had to physically place this disk in a media drive in order to install/execute it.

As technology has advanced, internet speeds have reached a point where users can simply load applications through a web browser by visiting a web page. Nowadays, therefore, Web Applications have become the most prevalent applications created today due to how simple it is to distribute them to users. When developers make changes to the application, they simply update the application on the server. Whenever users revisit the application, they will download it in its latest form and therefore it is simple for developers to ensure that users are updated with the latest changes of the application.

### 1.2.1 Client-Server Model

Fundamentally, all web applications can be described by the Client-Server model.

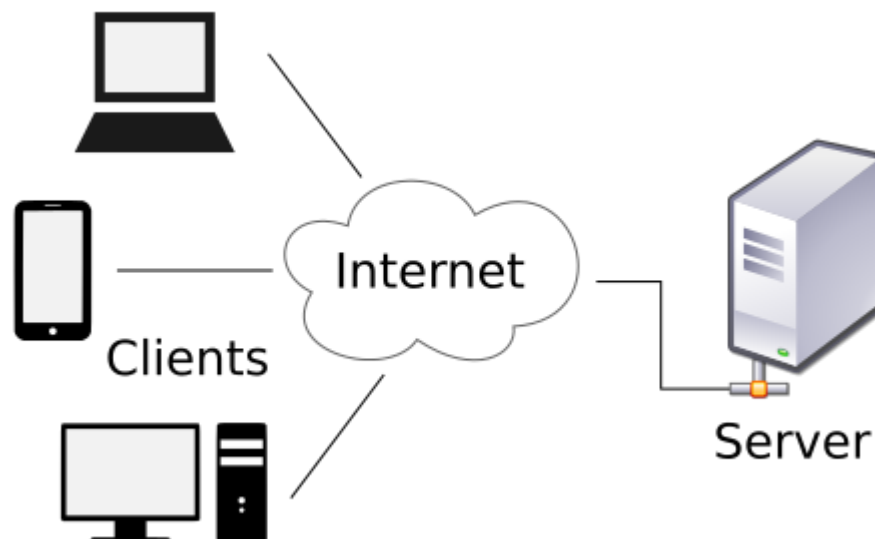


Figure 3<sup>3</sup>: The Client-Server Model

Simply put, the *server* component provides a service to the *clients* that initiate requests for the service.

In the domain of web applications, a server is described as a *web server*. The web server serves web pages and data. The clients are known as *web clients* and are usually in the form of

---

<sup>3</sup> Source: [https://en.wikipedia.org/wiki/Client%E2%80%93server\\_model](https://en.wikipedia.org/wiki/Client%E2%80%93server_model) [Accessed 1 April 2016]



web browsers or mobile applications. The web clients simply send requests to the web server. The server then determines how to react to these requests.

### 1.2.2 Comparison with Desktop/Native Applications

Nowadays, native applications are usually downloaded from the Web. Once downloaded, they are installed onto the user's machine.

Native applications must be developed specifically for a host platform such as OS X, PC or Linux. There are trends emerging that no longer make this the case such as the advent of hybrid applications, however this topic is out of the scope of this report. Web applications are developed such that any device with a full browser can run them.

### 1.2.3 Traditional Web Applications

Traditional web apps are such that every client (a web browser) makes an *HTTP*<sup>4</sup> request to a web server (see Figure 4). A client accesses a resource on a particular server through *URLs*<sup>5</sup>. When a client visits a resource and then visits another similar one on the same server (i.e. navigating from one Wikipedia article to the next), it downloads all the assets associated with that resource. It must redownload all the assets despite many of them being shared across pages. For example, the navigation bar of the wikipedia site is the same across various articles.

A web browser requests an HTML document from the web server. The browser then parses the document, renders the elements and downloads any linked documents such as images, CSS<sup>6</sup> files and JavaScript<sup>7</sup> (JS) files.

### 1.2.4 Single Page Web Applications

Single page web applications (SPAs) are an evolution of traditional applications that were first made feasible by a number of techniques and technologies collectively known as 'AJAX'<sup>8</sup>. These techniques combine JavaScript, CSS, the DOM<sup>9</sup> and XMLHttpRequest<sup>10</sup> to update the view of the application without having to load an entire HTML page.

---

<sup>4</sup> HyperText Transfer Protocol - the protocol by which information is communicated on the World Wide Web ([https://en.wikipedia.org/wiki/Hypertext\\_Transfer\\_Protocol](https://en.wikipedia.org/wiki/Hypertext_Transfer_Protocol)) [Accessed 1 April 2016]

<sup>5</sup> Uniform Resource Locator - a reference to a resource ([https://en.wikipedia.org/wiki/Uniform\\_Resource\\_Locator](https://en.wikipedia.org/wiki/Uniform_Resource_Locator)) [Accessed 1 April 2016]

<sup>6</sup> Cascading Style Sheets - describe the representation of a web page

<sup>7</sup> A scripting language supported by web browsers

<sup>8</sup> Asynchronous JavaScript and XML

<sup>9</sup> Document Object Model - A protocol by which to interact with objects within an HTML document

<sup>10</sup> An API that provides client functionality for transferring data between a client and a server (<https://developer.mozilla.org/en-US/docs/Web/API/XMLHttpRequest>) [Accessed 7 April 2016]

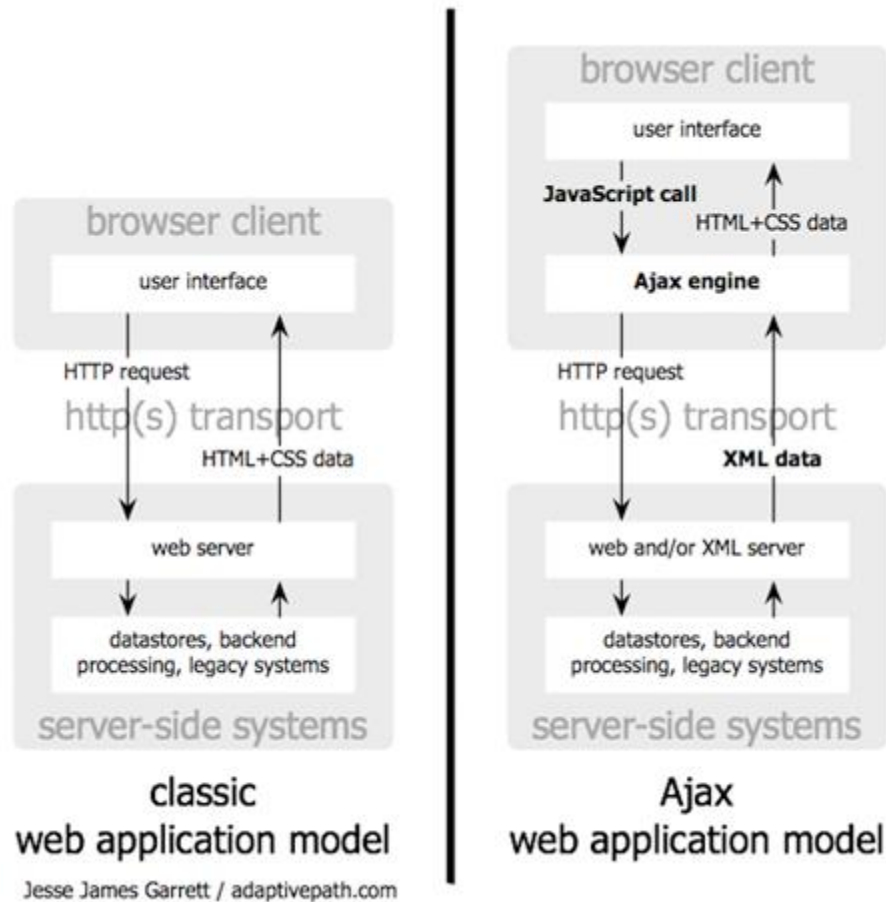


Figure 4<sup>11</sup>: The classic and AJAX web application paradigms

The AJAX model (see figure 4) is vastly different from a user interaction point of view because it means that the user is never blocked in their actions like they would be in the classical model. For example, when a user navigates between pages, traditionally they'd see a white flash as the page transitioned or a loading animation.

The AJAX model, on the other hand means that a user can remain on the same page and view new content without the jarring flash of the screen. An example case of this paradigm is the Facebook news feed, which uses AJAX to implement a concept known as *infinite scrolling* (figure 5). As a user scrolls down through the content, new content is added on the fly without it becoming noticeable or requiring a full page load.

<sup>11</sup> Source: (Garrett, J. 2005)

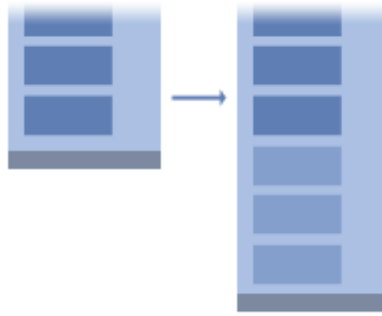


Figure 5<sup>12</sup>: Infinite Scrolling as seen in the Facebook News Feed

Single page web applications employ this concept of updating parts of the DOM on the fly, but take it to an extreme whereby an entire page of content is replaced with a new one.

This has profound effects on the ‘flow’ of the program execution such that one progresses from a synchronous path of execution in the classic model to an asynchronous path in the AJAX model (Figure 6).

From a user point of view, their activities are never blocked. Garrett, J. (2005) explains this succinctly:

*“Every user action that normally would generate an HTTP request takes the form of a JavaScript call to the Ajax engine instead. Any response to a user action that doesn’t require a trip back to the server — such as simple data validation, editing data in memory, and even some navigation — the engine handles on its own. If the engine needs something from the server in order to respond — if it’s submitting data for processing, loading additional interface code, or retrieving new data — the engine makes those requests asynchronously, usually using XML, without stalling a user’s interaction with the application.”*

---

<sup>12</sup> Source: <http://www.paulirish.com/2009/infinite-scroll-14-is-out-twitter-style-now-supported/> [Accessed 1 April 2016]

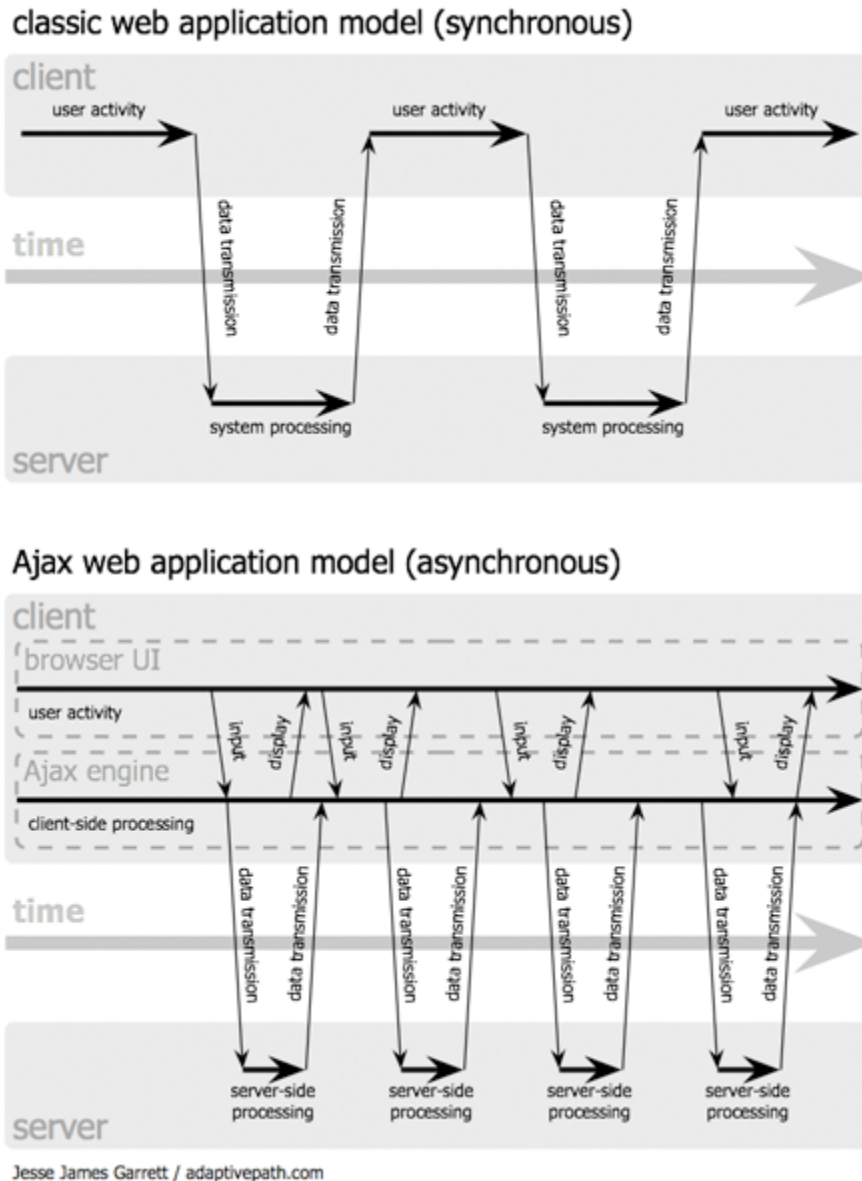


Figure 6<sup>13</sup>: Comparison between synchronous and asynchronous models of execution

SPAs make regular use of AJAX. A client downloads an HTML page. The browser engine then parses the page for JS resources, downloads them and then executes them. This JS code contains the asynchronous calls for remote resources. When a user navigates to a different page, the asynchronous calls are executed and, once complete, the original page 'element' is destroyed or hidden and replaced by a DOM object composed of the new data.

The author has decided to adopt this model for the development of WordRoot primarily because SPAs tend to provide a better user experience. This improved user experience is due to the fact

<sup>13</sup> Source: Garrett, J. (2005)

that they don't suffer from the 'stop-start' nature of traditional web applications. SPAs are more similar to a native application and are perceived as such by users.

One issue that will arise from making this choice is that an SPA is unable to function if JavaScript is disabled by the user. This will prevent the web browser from executing the JS code and so the SPA will not be usable. A traditional web application would not suffer from this problem but would offer a much more simplified, archaic experience to a modern SPA. In almost all cases, the user will be using a browser with a JS engine and have JS enabled, so the severity of this drawback is limited.

Some of the other weaknesses of SPAs, such as Search Engine Optimisation (SEO) are not applicable to this application's use case. However, this may change as new features are added or requirements change.

## 1.3 Test-Driven Development

Wikipedia summarises this topic succinctly:

*"Test-driven development (TDD) is a software development process that relies on the repetition of a very short development cycle: first the developer writes an (initially failing) automated test case that defines a desired improvement or new function, then produces the minimum amount of code to pass that test, and finally refactors the new code to acceptable standards."*

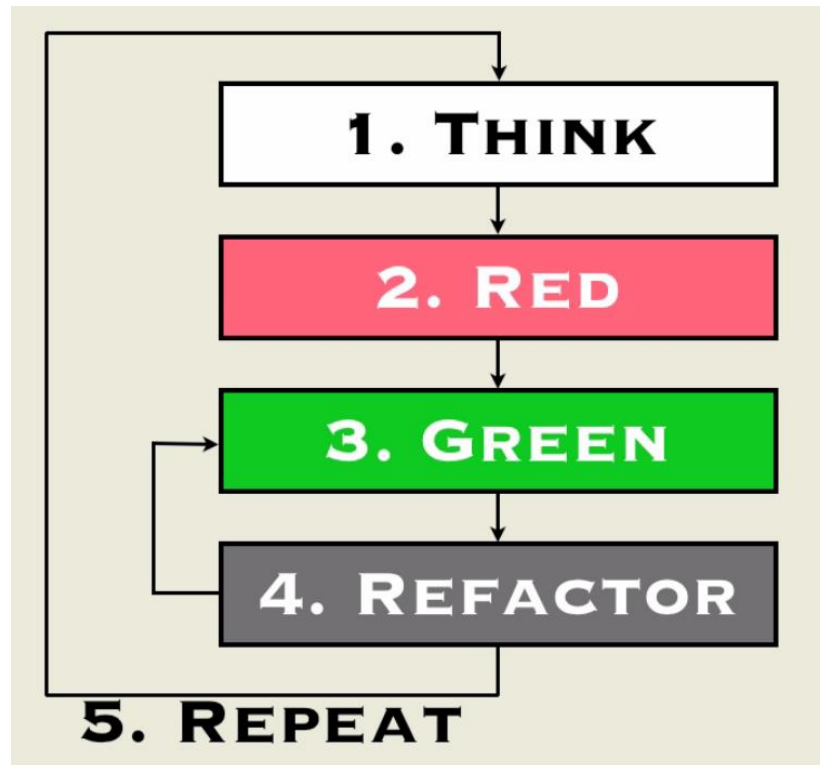


Figure 7<sup>14</sup>: The TDD Cycle

Shore, J. (2005) clarifies this process by distinguishing 5 key steps (figure 7). He explains each step:

1. **Think:** Figure out what test will best move your code towards completion. (Take as much time as you need. This is the hardest step for beginners.)
2. **Red:** Write a very small amount of test code. Only a few lines... usually no more than five. Run the tests and watch the new test fail: the test bar should turn red. (This should only take about 30 seconds.)
3. **Green:** Write a very small amount of production code. Again, usually no more than five lines of code. Don't worry about design purity or conceptual elegance. Sometimes you

<sup>14</sup> Source: [http://www.letscodejavascript.com/v3/comments/how\\_to/20](http://www.letscodejavascript.com/v3/comments/how_to/20) [Accessed 1 April 2016]

can just hardcode the answer. This is okay because you'll be refactoring in a moment. Run the tests and watch them pass: the test bar will turn green. (This should only take about 30 seconds, too.)

4. **Refactor:** Now that your tests are passing, you can make changes without worrying about breaking anything. Pause for a moment. Take a deep breath if you need to. Then look at the code you've written, and ask yourself if you can improve it. Look for duplication and other "code smells." If you see something that doesn't look right, but you're not sure how to fix it, that's okay. Take a look at it again after you've gone through the cycle a few more times. (Take as much time as you need on this step.) After each little refactoring, run the tests and make sure they still pass.
5. **Repeat:** Do it again. You'll repeat this cycle dozens of times in an hour. Typically, you'll run through several cycles (three to five) very quickly, then find yourself slowing down and spending more time on refactoring. Then you'll speed up again. 20-40 cycles in an hour is not unreasonable.

### 1.3.1 Benefits

The question arises: why should one follow this process? Shore argues that this process works so well for software development because it breaks work down into small incremental steps that are simpler to understand and reason about. Furthermore, as the developer iterates through each cycle, they are "constantly forming hypotheses and checking them". By doing this work in small chunks, when the developer makes a mistake, the failing test immediately identifies the mistake and it is easy to find and solve because only a few lines of code have been written. This tight feedback loop is vital in writing code that is as close as possible to bug-free.

Shore also argues how TDD can improve the overall design of the implementation:

"The other reason this process works well is that you're always thinking about design. Either you're deciding which test you're going to write next, which is an interface design process, or you're deciding how to refactor, which is a code design process. All of this thought on design is immediately tested by turning it into code, which very quickly shows you if the design is good or bad."

TDD can thus lead to code that is more modular, flexible and extensible because, throughout the TDD process, the developer is thinking about how each unit of code can be tested independently and then be integrated with other units later. One can argue that it leads to cleaner interfaces, more specific responsibilities of units of code and looser coupling between components of the application.

Proponents of TDD have made many claims on its benefits. Erdogmus, H. (2005) concluded that developers who followed TDD wrote more tests. Moreover, programmers who wrote more tests tended to be more productive.

Llopsi N. (2005) argues that, by following TDD practice, there is less need for a debugger: "Comparing it to the non-test-driven development approach, you're replacing all the mental

checking and debugger stepping with code that verifies that your program does exactly what you intended it to do.” Essentially, TDD helps to automate debugging. Furthermore, Lloplis also concludes that TDD “flattens out the change vs. cost curve”. In other words, changes and refactorings to the code base do not accrue a large cost as the lifetime of the application increases. This is due to there being a test suite in place to check that the program is functioning according to one’s assumptions.

He adds that this test suite ensures that a large amount of the code base is covered by tests which gives the programming team, and users, greater confidence in the code. Lloplis also notes that tests are a low-level form of documentation that can be considered superior to traditional, commented documentation. This is because commented documentation can quickly become outdated, needlessly state the obvious, mislead readers and bloat the code base. Tests remain relevant because they are executed - if they’re incorrect then they will fail. This failure highlights changes in the program that require attention.

Muller and Padberg (2003) propose a model that suggests the total code implementation time could be shorter with TDD, despite having to write the unit test code. A large factor that contributes to this conclusion is likely to be how defects are caught early in the TDD development cycle. This prevents them from becoming more expensive problems later on and helps eliminate the need for tedious debugging in the later stages of the project.

Madeyski (2010) has done extensive research into the effects of TDD and come to the conclusion that a test-first strategy (TDD) is superior to the classic test-last approach when considering the coupling between objects. Therefore, TDD code is more modular and easier to reuse and test. In other areas of his research, Madeyski has found that measurement of branch coverage (thoroughness) and fault detection effectiveness factors are improved by the TDD practice.

After analysing the practice of TDD and its positive effects, the author decided to adopt this methodology in the creation of the WordRoot application.

### 1.3.2 Limitations

TDD is not a perfect solution to writing bug-free code. Bugs may still occur that fail to be identified by the test suite. Furthermore, tests only test the programmer’s assumptions. If those assumptions are incorrect, the tests pass these incorrect assumptions. They don’t guarantee that the right choices are made. Moreover, if the programmer has blind spots, the tests will also have blind spots.

If tests are written poorly, known as ‘fragile tests’, then they may become a maintenance burden that can slow down a project. Like source code, tests require refactoring. Additionally, the practice of TDD must be followed religiously in order to reap its benefits. This can be particularly problematic for diverse teams where individuals don’t all share the same philosophy.

Despite these limitations, the practice of TDD is incredibly useful in ensuring a high-quality codebase and a minimum of bugs. The practice may be compared to the double-entry practice



followed Accountants. Although it is not necessary, the ability to constantly check assertions and correctness is invaluable to individuals working on these large, complex tasks.

### 1.3.3 Test Types

A key rule that applies to all tests is that the results of a test must be deterministic. Each test must be isolated from the other tests. They don't depend on any sort of external state of the application and may be run in any order.

#### 1.3.3.1 Unit Tests

Unit tests test small units of an application. They may be at the function level, class level or even include multiple classes. It is up to the developer to determine the granularity of each unit test. Martin Fowler clearly distinguishes unit tests from other types of tests<sup>15</sup>:

“A test is not a unit test if:

1. It talks to the database
2. It communicates across the network
3. It touches the file system
4. It can't run correctly at the same time as any of your other unit tests
5. You have to do special things to your environment (such as editing config files) to run it.”

Furthermore, unit tests should only ever use components that operate on the same process. Unit tests are the most common tests in a test suite and execute the most quickly.

#### 1.3.3.2 Integration Tests

Integration tests are the next most common type of test and execute slower than unit tests. Integration tests combine a variety of components (that are tested by unit tests) and evaluate whether they function together.

#### 1.3.3.3 End-to-End Tests

End-to-end tests are the least common type of test and the slowest to execute. They test that an application is behaving as expected from start to finish. Essentially, they test that separate components of the application, such as the database and server process are working together correctly.

## 1.4 The Reproducible Build

Fowler, M. describes the reproducible build as it applies to a software system such that “at any point you should be able to take some older version of the system that you are working on and build it from source in exactly the same way as you did then.”

---

<sup>15</sup> <https://groups.yahoo.com/neo/groups/extremeprogramming/conversations/topics/111829>

The motivation behind the reproducible build is that it allows a team to easily share the development environment of the system with each other and ensure that each developer is working in an environment that is consistent across the team.

A reproducible build requires that it be versioned under a version control system. This includes all internal source code as well as external dependencies. A build tool is used to automate the tasks required to build the system. This includes steps such as: performing static analysis of the source code, running tests to check everything is passing, copying files from a source folder to a distribution folder and compiling source files to bytecode.

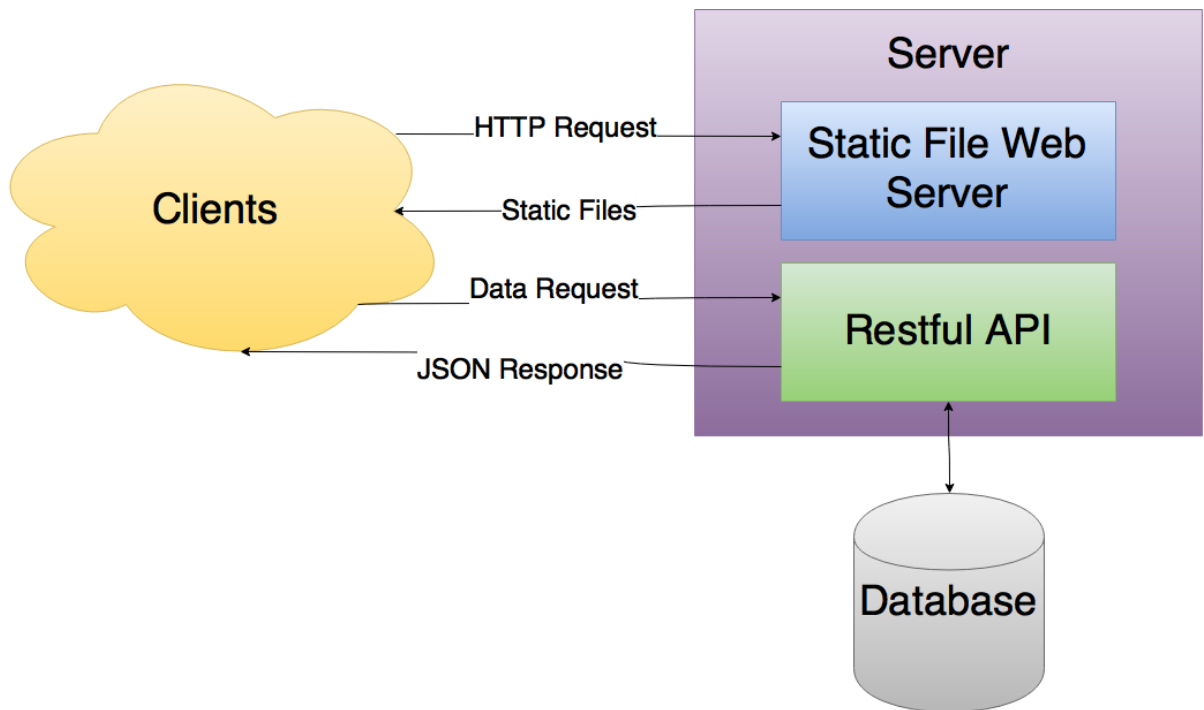
The reproducible build is a requirement for developing applications professionally because it automates the repetitive tasks required to build the system, saving vast amounts of time over the lifetime of the project and reducing costs. Developing the build system will slow down the initial progress of the project due to the fact that focus is not on developing the actual application itself. Therefore, initial progress is slower towards the beginning but increases relatively as the lifetime of the project increases.

Lastly, the build steps provide low level documentation for the steps taken in building the system. This is invaluable in sharing knowledge across the team and introducing new developers to the project.

## 2 Design

WordRoot adheres to the basic Client-Server model. However, it being a single page application, the server has two primary roles: serving static files and providing a restful API through which clients may request data.

### 2.1 Single Page Client-Server Model



*Figure 8: A high-level overview of WordRoot's Core Design*

## 2.2 Database

### 2.2.1 What's in a Word?

Every word has some symbolic representation and a meaning associated with it. Furthermore, a word is composed of multiple parts. A part also has a symbolic representation and the position at which it comes in the word. Every part may be derived from some root word. Likewise, a root part has a symbolic representation, a meaning and a language from which it originates. This relationship can be outlined by an entity relationship model (figure 9).

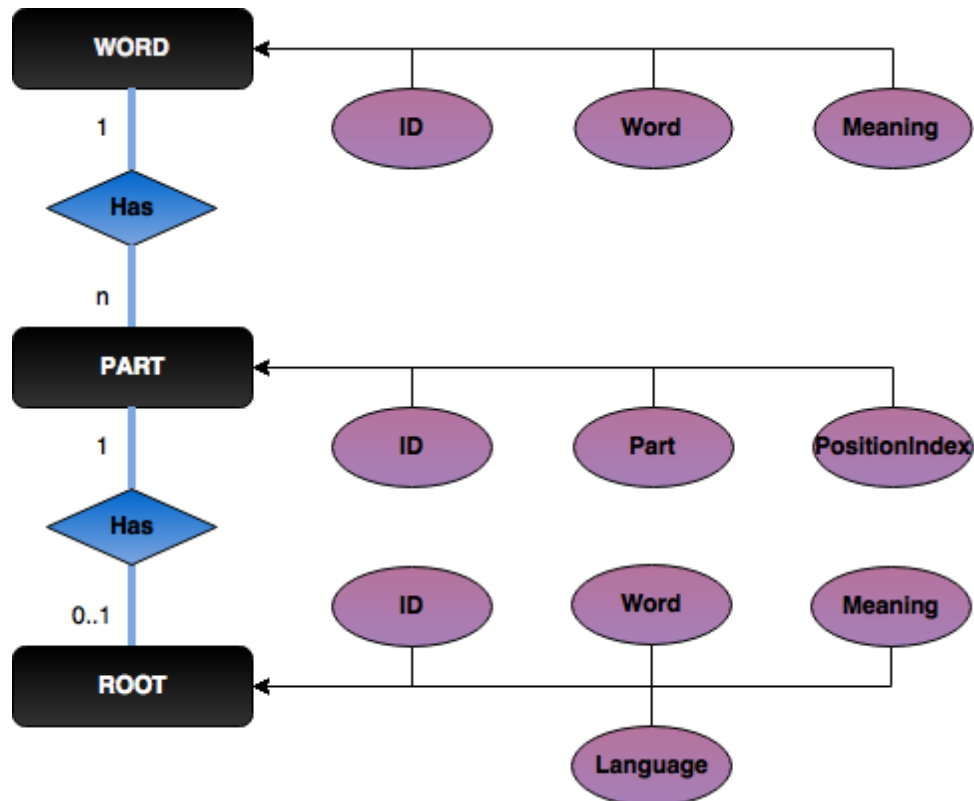


Figure 9: Entity Relationship Diagram

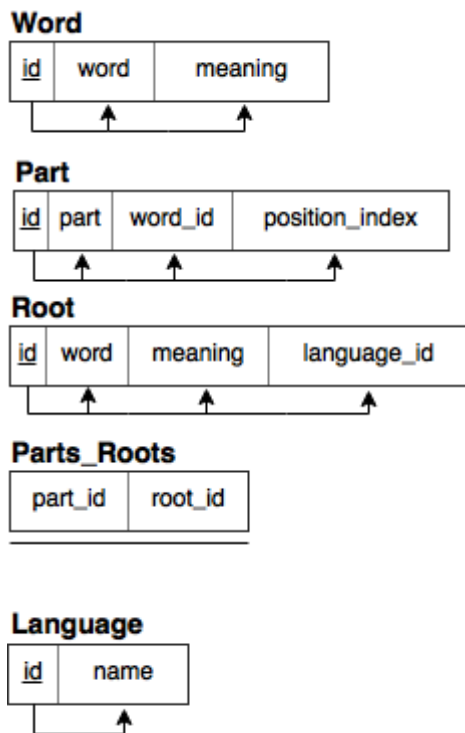


Figure 10: Functional Dependencies

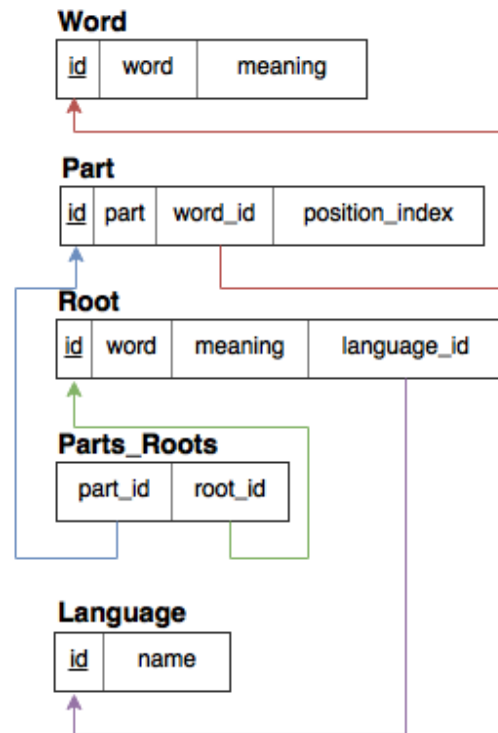


Figure 11: Relational Schema

Normalising this model (figure 10 and figure 11), we can represent this model in a database such that all values are atomic and there is no duplication.

## 2.3 Restful API<sup>16</sup>

URI	Description
GET /words	Get a list of words
GET /words/:word	Get the word that matches the value of :word

Currently, the API supports getting a list of words and getting information about a particular word.

<sup>16</sup> Representational State Transfer Application Programming Interface - An architectural style for which to access data ([https://en.wikipedia.org/wiki/Representational\\_state\\_transfer](https://en.wikipedia.org/wiki/Representational_state_transfer)) [Accessed at 7 April 2016]

## 3 Implementation

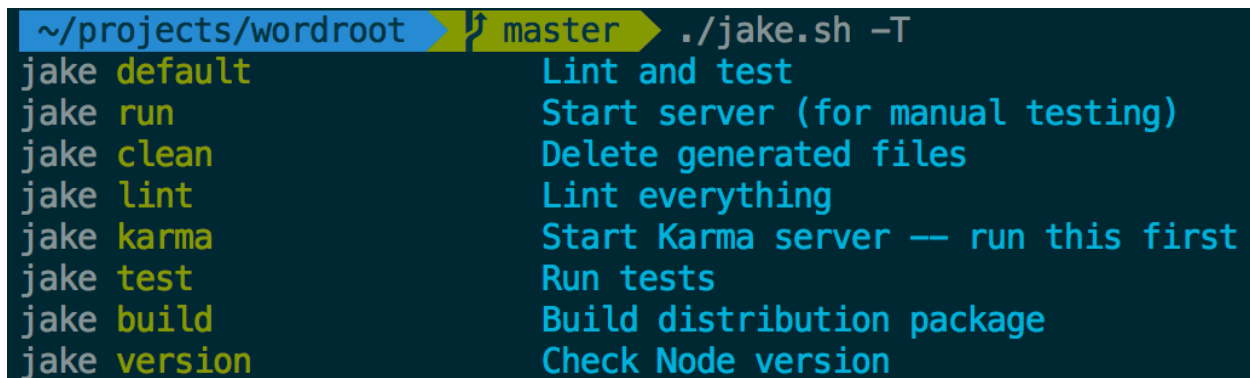
### 3.1 Build and Tooling

#### 3.1.1 Shell/Batch Scripts

These scripts are used to shorten the commands required to launch certain programs like the task runner. I.e. instead of inputting the command `./path/to/my/task/runner`, one could instead type something like `./run` into the console. This increases the ease at which the developer can launch automated tasks.

#### 3.1.2 Jake<sup>17</sup>

Jake is a JavaScript build tool that runs on Node. It provides a way for the developer to write and compose custom tasks that may be run by entering a short command into the console.

A terminal window with a dark blue background. The top bar shows the path `~/projects/wordroot` in blue, a branch indicator `master` in yellow, and the command `./jake.sh -T` in white. Below this, a list of Jake tasks is displayed in yellow text, each followed by its description in light blue text.

```
~/projects/wordroot master ./jake.sh -T
jake default      Lint and test
jake run          Start server (for manual testing)
jake clean        Delete generated files
jake lint         Lint everything
jake karma        Start Karma server -- run this first
jake test         Run tests
jake build        Build distribution package
jake version      Check Node version
```

*Figure 12: A list of all runnable Jake tasks and their descriptions*

For example, by entering `./jake.sh run`, the `run` task is executed which automatically boots up the WordRoot server on a particular port.

---

<sup>17</sup> <http://jakejs.com/>

```

~/projects/wordroot master ./jake.sh
Checking Node.js version: .
Linting Node.js code: .....
Linting browser code: .....
Testing Node.js code:
.
1 passing (56ms)

Testing browser code:
[2016-04-10 19:03:27.179] [DEBUG] config - Loading config /Users/chooie/projects/wordroot/build/config/karma.conf.js
Mobile Safari 9.0.0 (iOS 9.3.0): Executed 22 of 24 (skipped 2) SUCCESS (0.181 secs / 0.067 secs)
IE 9.0.0 (Windows 7 0.0.0): Executed 22 of 24 (skipped 2) SUCCESS (0.337 secs / 0.151 secs)
Firefox 41.0.0 (Mac OS X 10.11.0): Executed 22 of 24 (skipped 2) SUCCESS (0.229 secs / 0.026 secs)
Safari 9.0.3 (Mac OS X 10.11.3): Executed 22 of 24 (skipped 2) SUCCESS (0.196 secs / 0.029 secs)
Chrome 49.0.2623 (Mac OS X 10.11.3): Executed 22 of 24 (skipped 2) SUCCESS (0.37 secs / 0.025 secs)
TOTAL: 110 SUCCESS
Bundling browser code with Browserify: .
Copying client code: .
Copying server code: .
Running local smoke tests:
.
1 passing (229ms)

BUILD OK (4.41s)

```

Figure 13: Output from running the default task

Whenever changes are made to the source code of the application (iterating through the TDD cycle), the default task is run so that the JS code is linted, the application is built and the developer may see what effect their changes have on the application via the results of the tests.

### 3.1.3 Browserify<sup>18</sup>

Browserify ‘lets you require(‘modules’) in the browser by bundling up all of your dependencies.’ JavaScript that runs in web browsers does not currently support a module system, a common feature of other languages/platforms like Node.js<sup>19</sup>. This tool parses the developer’s client-side JS code (which contains require statements) and replaces them with a notation that is supported natively by browsers. Therefore, the developer can use the module system that is found in Node.js without this module system being officially supported by web browsers.

### 3.1.4 ShellJS<sup>20</sup>

ShellJS provides ‘portable UNIX shell commands for Node.js.’ It allows the developer to use familiar shell commands that make programming tasks like copying directories very simple.

<sup>18</sup> <http://browserify.org/>

<sup>19</sup> <https://nodejs.org/en/>

<sup>20</sup> <https://github.com/shelljs/shelljs>

### 3.1.5 JSHint<sup>21</sup>

JSHint is a 'static code analysis tool for JavaScript.' When run, it can detect common errors in the code such as using undeclared variables. It provides a large host of configuration options so that it may be as strict in its analysis as desired and help enforce certain coding conventions across the team.

### 3.1.6 Karma<sup>22</sup>

Karma is a test runner for client-side JS code. By starting a Karma server, one can attach browsers to it like Chrome, Firefox and Internet Explorer by accessing the Karma server and downloading the client application. When the developer runs their tests, the test code and application code is sent to the Karma server. The Karma server then sends this all to each connected client. Each client then executes all the client-side tests and sends the results back to the Karma server which then sends these results to the console:

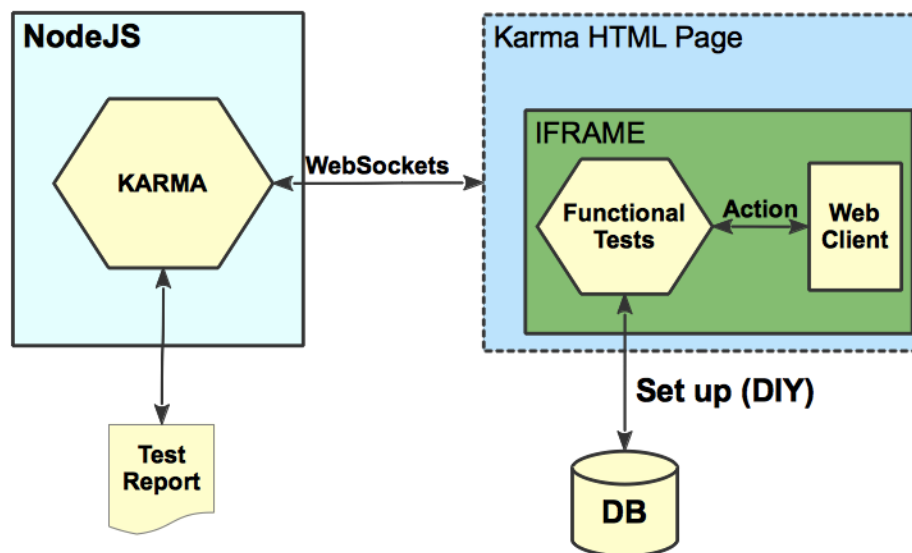


Figure 14<sup>23</sup>: A basic model of Karma

By using this tool, the team may test their application's JS code on a variety of different browsers and automatically check whether there are cross-browser incompatibilities (figure 15).

<sup>21</sup> <http://jshint.com/about/>

<sup>22</sup> <https://karma-runner.github.io/0.13/index.html>

<sup>23</sup> Source: <http://eamodeorubio.github.io/bdd-with-js/img/karma-bdd.png>



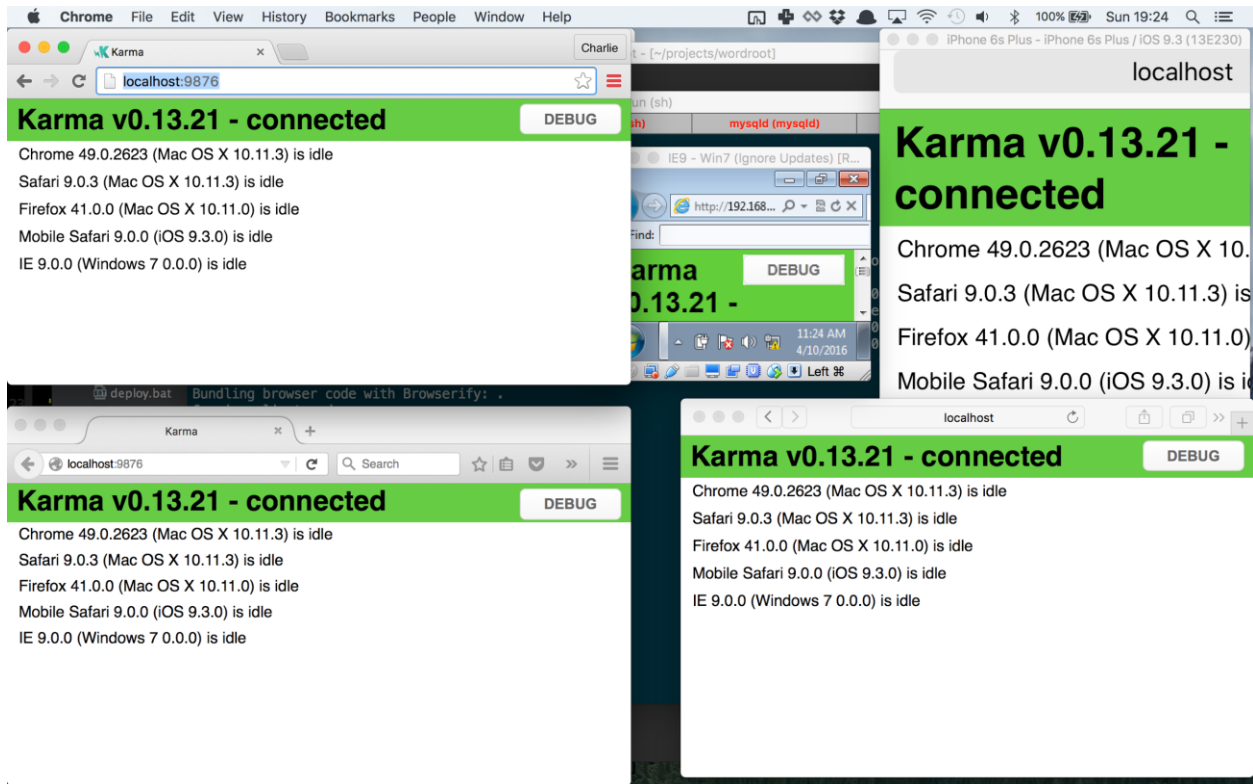


Figure 15: Chrome, Firefox, IE, Mobile Safari and Safari test clients

## 3.2 Application

### 3.2.1 Client

The client application is built using native JavaScript. In addition to the native JS, a 'shim' for the 'classList' API is used so that this API can be used in older browsers that don't support it. In the latest browsers, this API is supported natively. The 'classList' API makes it easier to manipulate the classes of DOM elements.

In addition, the other basic tools for creating web applications are used: HTML and CSS. HTML is the format in which one specifies the base elements of the application. Furthermore, CSS is used to style all the application's elements.

#### 3.2.1.1 Device-Responsive

In order for the application user interface to adapt to a range of different device sizes, three concepts are applied in CSS:

- The fluid grid
- Flexible images
- Media queries

When applying the concept of the fluid grid, elements are sized in percentage units so that they are always styled relative to the bounds of their container. Elements must not be styled in absolute units if the developer wishes to make the elements responsive. Likewise, flexible images and other components such as videos are styled using percentage units so that they don't exceed the bounds of their containers.

Media queries are used for changing the layout of the application more drastically when the width of the device reaches certain points. For example, at mobile device sizes, the side-bar navigation menu is hidden and only appears when the menu icon is clicked. Once the device width reaches a desktop-size, the menu is styled such that it is constantly in view and sits to the left of the main content.

### 3.2.2 Server

The server application is built using Node.js<sup>24</sup>. The author decided to use the Express<sup>25</sup> web framework that provides a number of utilities that aid in setting up a static file server and restful API. When visiting the application's homepage, the server sends the client a static HTML file that contains all the code necessary to launch the application. Furthermore, the application will then request data from the restful API. The server extracts all data from a MySQL database.

### 3.2.3 Client and Server

On both the client and server, the Q<sup>26</sup> Promise library is used. This provides a Promise abstraction that makes writing and reasoning about asynchronous code much easier. In particular, it helps developers avoid 'callback hell', whereby they are forced to nest functions within other functions in order to access the data generated from an asynchronous call. Promises instead allow the developer to separate these functions and call them in a clear, sequential manner.

In addition, the Mocha<sup>27</sup> test framework is used to provide utilities for structuring and running tests. The Chai<sup>28</sup> assertion library is used in combination with the Mocha test framework in order to use a variety of assertion techniques that will check the developer's assumptions against the application.

### 3.2.4 Restful API Data

The Restful API is currently very simple in that it only allows for the requests for a list of all words (figure 15) in the database or all the information associated with a particular word (figure 16). The data is in the JavaScript Object Notation<sup>29</sup> (JSON) format.

---

<sup>24</sup> <https://nodejs.org/en/>

<sup>25</sup> <http://expressjs.com/>

<sup>26</sup> <https://github.com/kriskowal/q>

<sup>27</sup> <https://mochajs.org/>

<sup>28</sup> <http://chaijs.com/>

<sup>29</sup> <http://www.json.org/>

```
[  
    "autobiography",  
    "idiosyncrasy"  
]
```

*Figure 16: List of words*

```
{  
  word: "idiosyncrasy",  
  meaning: "a mode of behaviour or way of thought peculiar to an individual",  
  - roots: [  
    - {  
      part: "idio",  
      - root: {  
        word: "idios",  
        meaning: "private",  
        language: "Greek"  
      }  
    },  
    - {  
      part: "syn",  
      - root: {  
        word: "sun",  
        meaning: "together",  
        language: "Greek"  
      }  
    },  
    - {  
      part: "cracy"  
    }  
  ]  
}
```

*Figure 17: Information for a particular word*

## 4 Conclusions

### 4.1 The Application

Overall, the application provides a set of useful core features and is a solid base from which to extend upon. The application is also responsive so that it presents well on a range of device sizes (see appendix 1).

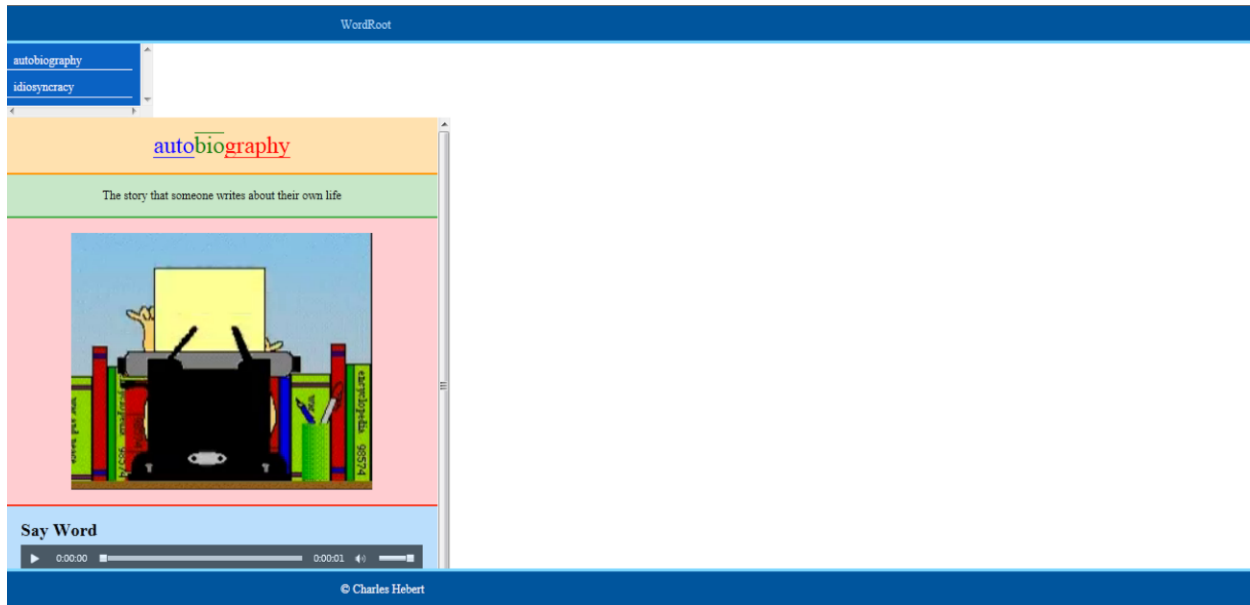
Users may view a list of words; they may view a word page containing the parts and roots of a word, its meaning, a video and sounds; and they may navigate to words through the list of words.

These features were provided with minimal use of frameworks. This has meant that the data requirements of the application are very low (<400kb) on initial load and even lower when navigating to other pages.

Although it was more difficult to develop the application without some frameworks, this does mean that the application is less coupled to large frameworks. Therefore, there is greater flexibility in developing it in the long run and the author believes that it is more understandable due to there being a reduced amount of framework abstractions littered throughout the codebase. With that said, however, there is ample opportunity going forwards to include certain minimal libraries like a template system that will improve understandability without increasing complexity.

### 4.2 Difficulties

A developer faces many difficulties when developing a web application. A notable problem is the browser incompatibilities - IE 9 was particularly problematic. Although the JS code is working well within IE 9, the styling is broken because IE 9 does not properly support the Flexbox model in CSS (a way of arranging elements within a page). This can be seen in figure 18:



*Figure 18: WordRoot in IE 9*

Furthermore, the Document Object Model is a particularly difficult interface to use with native JS. There are many pitfalls in terms of performance that are not immediately obvious without a great deal of experience and knowledge. Great effort must be expended in understanding its 'quirks'.

Managing asynchronicity was also particularly difficult until the Promise abstraction was used. The Q library was incredibly important in simplifying the structure of the codebase and making it easier to reason about.

The lack of a module system on the front end was a particular nuisance. As the application grew in size and complexity, it became increasingly difficult to work with because code had to be required in a particular order within the HTML page. When Browserify was introduced, code could be properly separated on the front end, as if the developer was using the module system in Node.js. Furthermore, only a single JS file had to be included in the HTML file, removing the clutter of all the script tags that required each JS file separately.

Lastly, testing was particularly difficult due to the author's lack of experience in writing unit tests, integration tests and end-to-end tests. The author decided to focus on testing the most critical sections of the application with unit tests. Less emphasis was placed on writing integration and end-to-end tests. Despite the challenges involved with following TDD and writing good tests, the author believes that following this practice is unequivocally worth the investment. The suite of tests written in a TDD manner undoubtedly saved a huge amount of debugging time.

## 4.3 Further Work

The bare features of the application are in place (a subset of the features offered by the original application), but there is still much work to be done in finishing the application.

Tasks to complete (in a basic ordering) include:

- Make the video and audio controls into more minimalistic buttons
  - Currently they use the default HTML5 'look' that should be replaced with controls that better fit the aesthetic of the application
- Introduce a feature to upload words
  - This will allow users to upload their own words to the application
  - The developer will also need to transfer all the remaining words to the database. A program should be written to automatically extract the words from a comma-separated-value (CSV) file and input them to the database as it is not pragmatic to do this manually.
- Fix remaining visual bugs due to browser inconsistencies
- Improve front end routing
  - HTML5 push state should be used so that one can properly navigate throughout the web browser history
  - Support for links should be implemented
- Implement remaining core features:
  - Search related words
  - Navigate to next random related word
  - Search for words in a list
  - User accounts
  - Social integration
- Deploy the application to a cloud host provider
  - The Amazon Web Services (AWS) platform is under consideration
- Interface pop-ups for errors
  - The user should be informed when an error occurs - for example, when data can't be loaded because the device has lost connection. This feature will be particularly important for mobile users.
- Caching
  - Cache visited pages

## 4.4 Additional Tools

There are a number of tools that the author wishes to introduce to this project:

### 4.4.1 React<sup>30</sup>

React is ‘a JavaScript library for building user interfaces’. It provides an interesting abstraction around the view component of the application. This library could contribute to making interface elements more modular and composable. Furthermore, it demonstrates features that make UI components easier to reason about in isolation. More research needs to be done by the author in this area but it looks to be the next great tool for simplifying web development.

### 4.4.2 Sassy CSS (SASS)<sup>31</sup>

Currently, the styles are written using vanilla CSS. Going forwards, the styles will be ported to SASS. SASS is a language that compiles to CSS. It provides many useful language features that are missing from CSS such as modules, variable and functions. This tool will prove vital as the size and complexity of the styling code increases.

This tool will be introduced to the build process so that SASS code is automatically generated into CSS code and packaged into the application.

### 4.4.3 Clojure/ClojureScript

Clojure is a functional programming language that runs on the Java Virtual Machine (JVM). ClojureScript is a stripped down version of Clojure that compiles to JavaScript. These languages have come to prominence over the last few years and aim to advance the field of software development by focusing on a number of key concepts such as:

- Limiting mutable state
  - Functions are mostly pure. Areas where mutable state is required are isolated by certain constructs.
- Native abstractions for working with multi-threaded and asynchronous code
- Pure functions are the best unit of abstraction

These languages are paradigm-changing and introduce the ability to build and use groundbreaking tools such as Figwheel<sup>32</sup>. Figwheel provides live code reloading of ClojureScript, JS and CSS code. In addition, it provides a built-in ClojureScript REPL (Read Eval Print Loop).

These features will be of great importance to the future of web development, in particular, because they provide tooling that drastically ease the use of working with the browser and reduce the time taken to modify code and investigate its effects.

---

<sup>30</sup> <https://facebook.github.io/react/>

<sup>31</sup> <http://sass-lang.com/>

<sup>32</sup> <https://github.com/bhauman/lein-figwheel>

For example, suppose a developer changes code that is executed when the program reaches a particular state. Traditionally, the developer has to reload the application and manually navigate to the state it was in before to investigate the changes. Now, with tools like Figwheel, this code is loaded and executed without changing the state of the program. Furthermore, the developer may directly interact with the running program through the REPL, giving them the ability to quickly make minor adjustments. These tools could feasibly save countless hours of developer time.

With these concepts in mind, the author believes that this area merits much research and these technologies could serve as replacements for the Node.js and JavaScript components of the application, simplifying the code base even further.

## 4.5 Closing Remarks

After following multiple practices and adopting the latest habits and tools purported by professional software developers, the author has come to a number of conclusions concerning web development and the best-practices of software development in general.

Firstly, all applications should be developed using version control. The ability to share the current state of an application's environment and revisit old versions is invaluable, especially when working in a team environment.

An automated build should be mandatory when developing a professional application. Although there is an up-front cost in writing the build code, these custom tools save a monumental amount of time over the lifetime of a product. They also aid developers in understanding the context of an application through the low-level documentation provided by the clearly defined build steps.

A test suite is mandatory when developing a professional application. Ideally, the tests should be written in a 'test-first' manner as described by TDD. The advantages have been discussed earlier in this paper. However, to reiterate the main arguments, a test suite is mandatory because it provides invaluable feedback to a developer when making changes to the source code of an application. Furthermore, by following the TDD cycle, a developer may write code with better interfaces, abstractions and an overall superior design. Refactoring is a key step of the process and, with a suite of tests to check any errors arising from refactoring, a developer is likely to be more confident in refactoring large components of a codebase. This will lead to a much cleaner codebase over the long run.

TDD is not a panacea. However, if followed correctly and the team is trained in its practice, it can provide a great deal of benefits to a project at little cost.

After becoming accustomed to writing asynchronous code, the author has come to the conclusion that using nested callbacks should not be a practice when doing asynchronous programming. This is due to the fact that passing around callback functions can be very error-prone and it makes it very difficult to follow the path of execution of code written in this way.



Instead, a developer should wrap asynchronous code with the Promise abstraction or another construct that aims to simplify asynchronous code like ‘async generators’<sup>33</sup>. By using this abstraction, the developer can return to a synchronous style of programming that is much easier to reason about and separates out the calling of functions without all the nesting and ‘callback hell’. These abstractions aren’t perfect, but they greatly improve the ease of working with asynchronous code. It is therefore the author’s opinion that they should be used in practically all circumstances where one works with asynchronous code.

Moreover, the author has studied functional programming whilst carrying out work on this project. It is the author’s opinion that following a functional paradigm in JavaScript, despite there being no explicit language constraints for it, can vastly improve the simplicity and composability of the components of a program. The author will continue their research into this paradigm and, in particular, investigate Clojure and ClojureScript.

From the author’s experience, building single page web applications is much more difficult than building web applications that align with the classical model. There is a need for a great understanding within the server-side and client-side components of the stack. Furthermore, the environments of these components differ greatly. Additionally, single page applications are still poorly supported natively by browsers. This is because browsers have advanced to cater to the classical paradigm of web applications. Therefore, establishing workarounds for issues in browser history and routing can be particularly time-consuming. There is also a less significant body of research in this area in comparison to the classical paradigm.

Overall, the author is satisfied with the progress made in the development of this application. Although there are still more features to implement and areas that need polish, the build environment and application structure is in a state that lends well to continued development and extensibility.

The author also believes that careful consideration of professional practices and tooling have resulted in a product that is much higher in quality than would otherwise be the case. Although it is easy to dive into a project and work quickly, over time this process breaks down and productivity drastically falls. By suffering a few up-front costs, productivity can remain stable over the development of the project and improve the final quality of the product.

---

<sup>33</sup> <https://davidwalsh.name/async-generators>

## 5 References

Garrett, J. (2005), *Ajax: A New Approach to Web Applications*, Available at: <https://web.archive.org/web/20080702075113/http://www.adaptivepath.com/ideas/essays/archives/000385.php> [Accessed 1 April 2016]

Shore, J. (2005), *Red-Green-Refactor*, Available at: <http://www.jamesshore.com/Blog/Red-Green-Refactor.html> [Accessed 1 April 2016]

Erdogmus, H. (2005), *On the Effectiveness of Test-first Approach to Programming*, NRC Institute for Information Technology; National Research, 31(1)

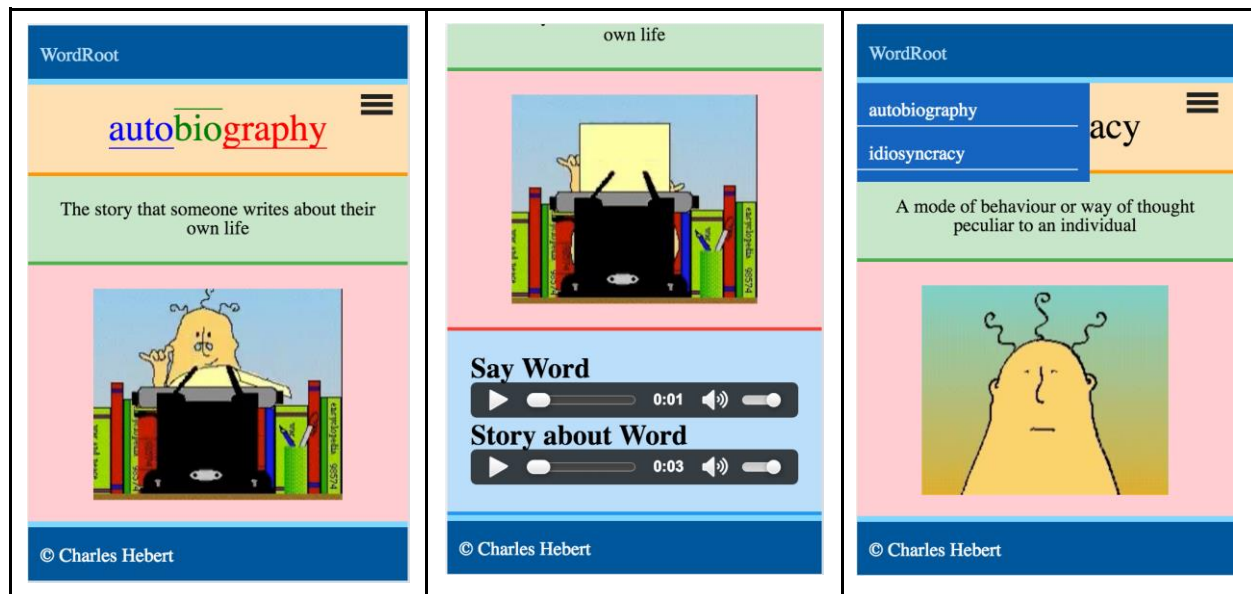
Müller M., Padberg F. (2003), *About the Return on Investment of Test-Driven Development*, In International Workshop on Economics-Driven Software Engineering Research EDSER-5

Madeyski, L. (2010), *Test-Driven Development - An Empirical Evaluation of Agile Practice*, Springer, ISBN 978-3-642-04287-4, pp. 1-245

## 6 Appendix

### 6.1 Appendix 1

#### 6.1.1 WordRoot Mobile View



## 6.1.2 WordRoot Desktop View

WordRoot

autobiography  
idiosyncrasy

**auto****bio****graph****y**

**Word** bios  
**Meaning** life  
**Language** Greek

The story that someone writes about their own life



© Charles Hebert



**Say Word**

▶ 0:01 🔊

**Story about Word**

▶ 0:10 🔊

© Charles Hebert