Examining the Performance and Suitability of Node.js for Web Application Development

Gavin Flood

Thesis presented for the degree of
B. Sc in Software Design (Web Development)
to the
Department of Software Engineering,
Athlone Institute of Technology.

Supervisor: Seán Kennedy

Table of Contents

Chapter 1

Lead Section

(1.1) List of Figures

(1.2) List of Tables

(1.3) Abstract

The planning, design and implementation of a web application built using server side JavaScript and Node.js is presented in this documentation. The overall goal of the project is to determine how suitable Node.js is as a web development framework and how well it performs under a variety of load levels. This was done by testing two simple applications running on Apache and Node servers, as well as a full-scale application running on a Node server.

The result of this project showed that an application running on a Node server performs better than an identical application running on an Apache server when dealing with large amounts of concurrent requests.

(1.4) Acknowledgements

I would like to thank Seán Kennedy, whose assistance and guidance throughout this project has been critical to seeing it through. In addition, a thank you to Paul Jacob for his part in helping me understand some otherwise unfamiliar technologies and to Enda Fallon for providing insight and encouraging me to focus on one particular area.

(1.5) Declaration

This thesis is a presentation of my original research and work. Every effort is made to indicate contributions of others. All work was done under the guidance of Seán Kennedy at Athlone Institute of Technology.

(1.6) Glossary of Terms

Node
Node is the commonly used method of referring to Node.js, a framework developed by Ryan Dahl that implements an event-based model for handling input and output. It utilizes Google's V8 engine to achieve top performance with server side JavaScript.

Evented-I/O
Event-driven input and output involves using a single thread to process events from a queue and being able to process other events before the original event has finished.

MVC
Model-view-controller is a software architecture pattern that aims to separate business logic from the user interface.

XP
Extreme Programming is a software development methodology that advocates frequent releases (or iterations) over documentation.

FDD
Feature Driven Development is a software development methodology that aims to produce working software repeatedly over short iterations and in a timely manner.

REST
Representational State Transfer is a style of software architecture for distributed systems such as the World Wide Web.

RUP

Ramp-Up Period is referred to in terms of load testing. It is the number of seconds after which all users will be up and running.

Load Testing

This is a type of testing method that evaluates how an application reacts to different loads of traffic. The end-goal is to find the load at which the application is most productive, while also finding the approximate load at which the application's performance begins to deteriorate.

JSON

JavaScript Object Notation is a text-based open standard designed for human-readable data interchange, which was originally specified by Douglas Crockford.

XML

Extensible Markup Language is a markup language developed by the W3C that defines a set of rules for encoding documents in a format that is both human-readable and machine-readable.

UNIX

UNIX is a computer operating system originally developed in 1969. It is mainly programmed in C and Assembly language, with many modern operating systems (Linux, for example) being based on the UNIX system.

GUI

A Graphical User Interface provides users with a way to interact with devices using images and visuals rather than text-based interfaces.

Chapter 2

Introduction

(2.1) Problem Statement

The modern nature of web application development requires developers to have a firm knowledge of multiple languages. Front-end developers are usually competent with HTML, CSS and JavaScript while back-end developers can work with a wide array of languages, frameworks and database systems.

Node.js has attracted a lot of attention since development began on it in 2009, mainly due to it providing a way to use JavaScript on the front and back-end. While this was potentially great news for developers, the project is still under active development and rapid releases can sometimes hinder development, since functionality can quickly become deprecated.

Ideally, all web application development could be done using JavaScript, with JSON being perfectly suited for communicating data structures (JSON, 2002). However, there have been multiple server side JavaScript platforms in the past, and few have gathered a notable following. Node.js is different in it's implementation of this, but there are still questions asked about it's suitability for web application development and whether it offers any noticeable performance benefits.

In addition to this issue, most popular web servers currently in use provide a lot of modules and functionality out of the box. This can, of

course, be advantageous but it could be argued that it may impact negatively on the performance of these web servers. Node provides the developer with the power to tailor the server to suit the needs of the project, which greatly reduces the amount of unnecessary functionality (Teixeira, 2013). The benefits this provides may not be as important as the event-driven process it uses to handle requests, but the power it gives to the developer may be of significant importance.

(2.2) Project Overview

The goals of this project are to examine how the Node software system performs at multiple levels of stress and also how suitable it would be for developing web applications.

To examine the performance of web applications running on a Node web server, load tests were carried out on a small Node application. The results of these tests were compared with the results of load tests on an identical PHP application running on an Apache server.

To examine the suitability of using Node and it's asynchronous programming model for developing web applications, it was necessary to build a fully functional web application handling a reasonable amount of input and output. The application itself is a jobs board for students seeking work placement and internships. This application was also load tested when development was completed, mainly as a comparison to the much less complex Node application.

Chapter 3

Research

(3.1) Concept

The original goal of this project was to examine the benefits and drawbacks of using Node and server side JavaScript when developing web applications. Over the first couple of months this goal changed after load testing a basic Node server which simply responded to requests by sending in-memory JSON data. As a result of this, the end-goal was altered to concentrate on the performance of web applications running on a Node server. To determine whether the Node server performed well, it needed to be compared to another type of server. The final plan was to build two small, identical applications with one running on a Node server and the other running on an Apache server.

Although load testing was intended to play a significant part in the project, the primary goal was still to examine the suitability of Node in the development of web applications. This could be done by developing a full scale, fully tested web application using a typical development process and noting any benefits or drawbacks throughout. As well as that, the application could also potentially contribute to the load tests once development had completed.

(3.2) Node.js

Research was focused on the Node.js software system. It was developed by Ryan Dahl and sponsored by Joyent, his former employer. As detailed on the official website (nodejs.org, 2012), it allows for server side development of applications using JavaScript and follows and event-driven, non-blocking I/O model, making it lightweight and efficient.

It is built on Google's V8 JavaScript engine, which is used primarily in the Chrome web browser. The somewhat unique aspect of Node is that it creates a web server itself which gives the developer full control over how the server works and how it handles requests and responses.

With Node being a relatively recent technology, there is a noticeable lack of published works focusing on it. However, a significant point that was easily conclusive from the research was that Node performs best with applications requiring a lot of I/O. This was notably put down to Node's programming model (Tilkov and Vinoski, 2010 p.81), where “asynchronous interactions aren't the exception; they're the rule.”

JavaScript is quite different in comparison to other languages. It is the language of the web browser and so it manages to divide opinion among developers everywhere. It is loosely-typed, treats functions as first-class objects and provides prototypal inheritance, where objects inherit properties directly from other objects (Crockford, 2008 p.3). Node's strict enforcement of the asynchronous programming model somewhat streamlines the coding style for Node applications, making it much easier to write good JavaScript code. Overall, JavaScript is very different compared to other languages that can be used for server side development (for example, PHP, Python, Ruby) but it also provides some very useful features that make development easier.

(3.3) Measuring Performance

Once the goal of the project changed to focus on performance and suitability of Node in relation to web application development, a significant amount of research was carried out on those topics, mainly using published research papers and previous experiments with the software system. Since Node is still in beta release (at the time of this project), there is a significant lack of peer-reviewed literature on the framework.

McCune (2011) set up experiment to test the performance of Node, EventMachine and Apache servers and provided valuable insight in this project's research process. In his experiment, McCune conducted all tests on an iMac inside a virtual machine running Ubuntu 11.10. The results of this experiment showed that Node out-performed the other two frameworks and this was especially highlighted at increased stress levels.

Paudyal's (2011) work focused on scalable web applications using Node and CouchDB. In his load-testing of his application prototype, he concluded that Node was particularly reliable as a server when dealing with large numbers of simultaneous users. This seemed to be a consistent conclusion across most research documents. Node generally performed adequately when under minimal stress but performed admirably when under large amounts of stress.

A number of performance testing tools are freely available, many in the open-source community. The three that were mainly considered for the load testing section of this project were Apache ab, Apache JMeter and Tsung. “ab” is a small HTTP benchmarking tool designed test the performance of a HTTP server. It is also provided out of the box on a lot of UNIX-based machines. However, it does not provide the detail desired for

this part of the project, unlike both JMeter and Tsung. Tsung uses a detailed XML file to configure the tests to be carried out on the server, whereas JMeter provides a GUI that allows for a similar level of detail. JMeter was ultimately chosen due to it's detailed reports and ability to quickly generate graphs based on test results.

(3.4) Potential Frameworks and Databases

Node.js comes with a lot of modules ready for use and allows for simple extension using it's “require” system. This project utilized some of these modules to avoid reinventing the wheel and gain access to some useful functionality. Outside of Node, there are some very useful JavaScript libraries, many of which were considered for the development of the web application.

(3.4.1) Web Application Framework

ExpressJS is a web application framework built on top of the already popular Connect framework. It provides a robust set of features for building single and multi-page web applications. It's main competitor on the Node platform is Geddy, another framework that is very similar to Ruby on Rails. Both of these frameworks are built to encourage an MVC architecture on the server side.

(3.4.2) Database

Since the fully functional application was going to be storing data, potential databases formed a significant part of the research process. Node.js has modules designed to interact with a wide-array of databases.

The most popular choices seemed to be CouchDB and MongoDB, both NoSQL databases. Other options included Redis (an in-memory key-value data store suitable for applications with rapidly changing data) and MySQL (the most popular relational database used on the web). Strauch (2009, p.3) described how NoSQL databases are designed with scaling in mind and one of the NoSQL movement's motives is the “avoidance of expensive object-relational mapping.” This fit in with the goals of the project so MySQL and Redis were dismissed as potential candidates.

(3.4.3) Testing Framework

Testing was the final area where a framework was deemed necessary to reduce the workload and enhance the final quality of the web application. There are many testing frameworks available for Node.js, though any candidate needed to be capable of easily testing asynchronous code. Overall, Mocha, Node-Unit and Vows were all frameworks that met those requirements, and, after extensive research into them, they seemed so similar that they needed to be tested individually. In the end, Mocha was chosen, specifically for how easy it makes asynchronous testing, though Node-Unit was definitely easier in terms of writing tests.

Here is the sample test in Vows:

```
vows.describe('Person').addBatch({
    'Check Name': {
        topic: person.name,
        'is joe': function(topic) {
            assert.equal(topic, 'Joe');
        }
    }
});
```

This is the same test in Node-Unit:

```
exports.checkName = function(test) {
    test.equal(person.name, 'Joe');
    test.done();
};
```

And finally, this is the same test in Mocha:

```
describe('Person', function() {
    it('should return Joe', function() {
        assert.equal('Joe', person.name);
    });
});
```

Chapter 4

Architecture and Design

(4.1) User Stories

As explained on their website (Microsoft Developer Network, 2013), Microsoft have defined the usage of user stories as communicating “functionality that is of value to the end user of the product of system.” The following is the collection of user stories that were provided at the start of each iteration throughout development:

As a user who is not logged in on the home page, I want to see registered companies and their job postings.
As a user who is not logged in, I want to be able to view a list of all job postings by each company.
As a user who is not logged in, I want to be able to log in on any page of the web application.
As a user who is not logged in on an employer's page, I want to see their publicly viewable information and their job postings.
As a user who is not logged in on a user's page, I want to see their publicly viewable information.

As a user who is not logged in on the register page, I don't want to be able to submit the form until all required information has been entered correctly.

As a user who is not logged in, I want to be able to view a list of all jobs by category.

As a user who is logged in on the home page, I want to have easy access to view and edit my profile page.

As a user who is logged in on a “job posting” page, I want to be able to apply for that job without having to complete any additional forms.

As a user who is logged in, I want to be able to edit any information that could potentially change over time.

As a user who is logged in, I want sensitive data like my email address, password and phone number to be hidden from other users and employers.

As a user who is logged in, I want to receive an “are you sure” message before being able to delete my profile.

As a user who is logged in, I want to be able to log out in a single click.

As a user, I don't want my password to be stored in plain text.

As an employer, I want the ability to log in or register on the same page, separate from the jobseeker's log in and register page.

As an employer who is logged in, I want the ability to add a job posting from my profile page.

As an employer who is logged in, I want sensitive data like my email address, password and phone number to be hidden from other users and employers.

As an employer who is logged in, I want to be able to delete a job posting from my profile page.

As an employer who is logged in on a “job posting” page, I want to see a list of candidates who have applied to that job with links to each respective profile.

As an employer who is logged in, I want to be able to log out in a single click.

As an employer who is logged in, I want to receive an “are you sure” message before deleting my profile.

(4.2) Development Process

A number of factors played into which development process was chosen for this project. Most popular processes were taken into consideration but the process needed to be a good fit for the type of project. Since the project is split into two parts – the fully functional web application and the load testing – a development process was only required for the former. The load testing would only be undertaken after the web application was completed.

A variation of multiple agile software development methods was ultimately chosen; namely Extreme Programming (XP) and Feature Driven Development (FDD) which are both similar to begin with (Khramtchenko, 2004 p.18), though XP is probably more adaptable for a single developer, despite including methods like pair programming and peer reviews.

Development of the web application would be optimized using short development cycles since it was being built using unfamiliar software, being developed by a single developer, and had several equally necessary features. Both XP and FDD advocate short iterations with FDD obviously putting the focus on a specific feature. Iterations were kept down to a maximum time of one week, with any features that needed longer to develop being broken down into multiple smaller features.

Documentation was not overly critical to the project since the web application's purpose was to demonstrate Node's suitability as a web development framework, as well as contributing to the load testing. Nevertheless, due to the short iteration times for each feature, several user-stories were created at the start of each cycle, as used in XP (Khramtchenko, 2004 p.2).

A Test Driven Development process was not employed and instead all unit tests were performed during the third stage of each iteration. In the situation were a test failed, you would return to the coding phase, fix the issue, test and repeat until a pass was achieved. This enabled rapid development and avoided spending a lot of time at the start of each iteration writing tests that needed to fail.
Prior to testing, the user-stories were utilized to gain an insight into the design and functionality of each feature. At this stage, the coding phase began with client-side code being written before business logic was implemented.

Overall, the process is very similar to FDD but lacks the emphasis on complete documentation for each feature. Both FDD and XP have multiple phases and methods that focus on teamwork. Obviously, since this project was completed alone, these phases and methods had to be either altered or ignored.

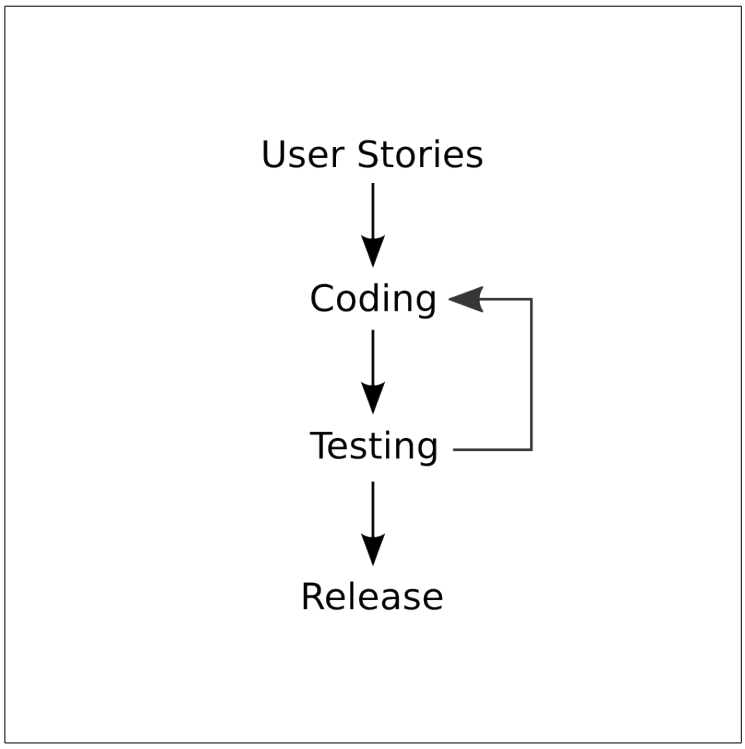

Figure 4.1: Software Development Process

(4.3) Project Architecture

This section explains the architecture of the fully functional web application built for this project and avoids referencing the architecture of the smaller applications that will be used for load testing later. The architecture for these smaller applications will be covered in the Testing section.

Most web applications have an n-tiered architecture as opposed to many traditional applications which many only consist of one tier. This web application consisted of three tiers:

1. Presentation (the web browser)
2. Application (Node.js server and functionality)
3. Storage (MongoDB database)

The benefit of using the Express framework in the development of my application was being able to utilize it's MVC architecture to separate business logic from the user interface. This style was followed explicitly and it greatly simplified development.

(4.3.1) Presentation

When a user sends a request to a URI, the Node server responds by sending back the relevant data, rendering inside of a HTML template. Two template engines were used for the presentation tier, Jade and Stylus, developed by VisionMedia and LearnBoost respectively. Jade minimizes the amount of HTML you have to write while also allowing JavaScript code to be embedded into the file. This enables you to customize how you use the data sent in a response, since it is all done in JavaScript.

Stylus is a dynamic variant of CSS which reduces the amount of CSS you

have to write. It is compiled into valid CSS by the Node.js server. This was not essential to the project but, since it was very similar in style and purpose to Jade, it was incorporated.

(4.3.2) Application

Only two models were required for the web application: a user and an employer. These two were actually Mongoose schemas, which will be explained in the description of the database tier. The application essentially consisted of three major elements:

1. Users created profiles that displayed their information, including work experience and educational background.
2. Employers created profiles that displayed their information, including a list of job positions available.
3. Jobs were created by employers. They were stored as an array in the employer schema, and users were able to apply for these positions.

The original goal was to base employers and users off the same model, but as the project progressed their differences were more obvious, so two separate models were used which actually simplified the “jobs” element of the web application.

Node creates a server itself, which means you have to implement the majority of it's functionality. To correctly respond to requests to various URIs, a router was created which would map a request to the relevant controller function. This function would then be responsible for reading the request header and body and providing the relevant response. Here is an example of what the router file looked like:

```
app.get('/users/login', users.logIn)
app.post('/users/new', users.create)
app.put('/users/:username', users.update)
app.del('/users/:username', users.delete)
```

This router followed a RESTful style architecture by making use of the four main HTTP request methods. This was supported on Node through the use of a middleware provided by the Express framework. The methods passed as a second parameter are callbacks, which ensures the Node server does not get stuck waiting for a response. Instead, when the callback function is invoked from the controller, then the response is put in an event queue which will be handled by the server.

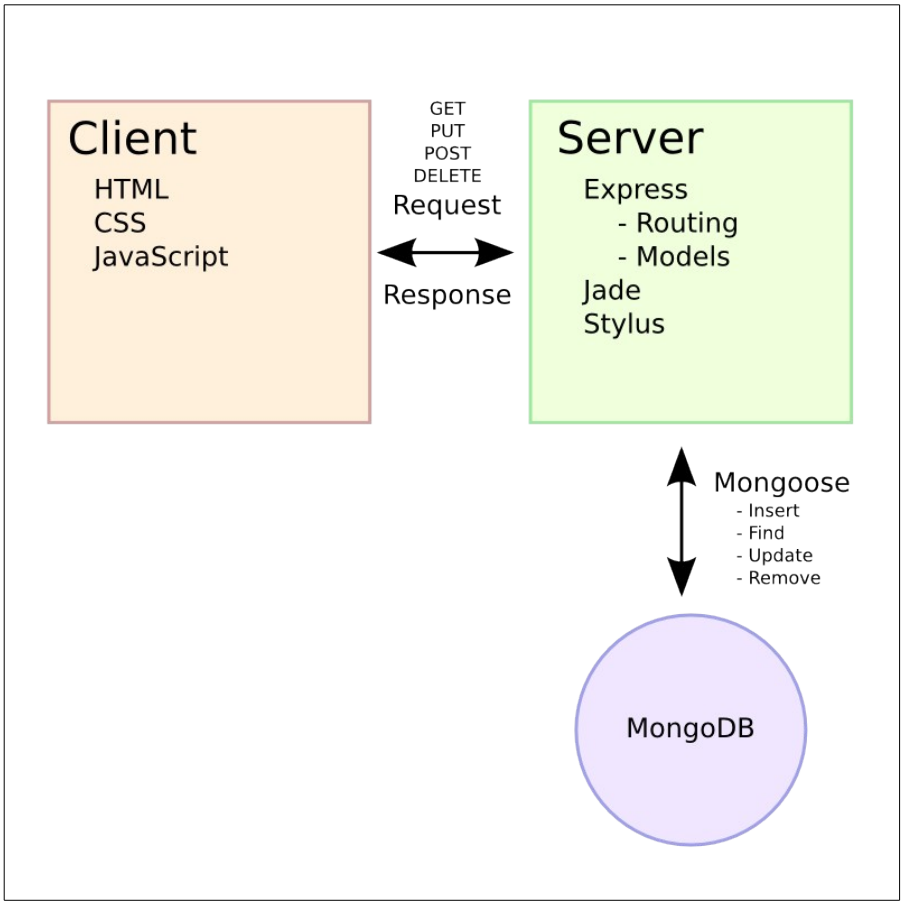

Figure 4.2: Project Architecture

(4.3.3) Storage

For the application's database, MongoDB was used. It is a NoSQL database that stores data in JSON format. The reason MongoDB was chosen was that it bears a slightly more familiar resemblance to MySQL in comparison to CouchDB, mainly due to it assigning each document in a collection an ID. It is also more suitable for larger applications with frequently changing data that require dynamic queries.

MongoDB is a database that consists of collections. When compared to MySQL, a collection would be similar to a table. Collections in turn contain documents, which would be similar to a row in MySQL. Since MongoDB uses dynamic schemas, there is no need for you to specify the structure of a document before inserting it into a collection.

To connect to my MongoDB database from my web application, the Mongoose module was required, which provides a lot of functionality for interacting with the database in JavaScript. According to the Mongoose documentation (2013), it provides a type of element, called a “schema”, which maps to a MongoDB collection. On top of this, the module incorporates “models”, which are constructors compiled from schema definitions. As with normal constructors in JavaScript, these models can be instantiated. An instance of a model represents a MongoDB document, which can be saved and retrieved from the database.

Chapter 5

Implementation and Testing

(5.1) Implementation

As outlined in the previous section, a variation of both the XP and FDD processes was used in the development of the jobs board application. Each iteration focused on a specific feature (or set of smaller features) with code being written before tests were carried out. In addition to this, each iteration also included fixing a number of bugs and issues from the previous releases.

The choice of application made it easy to prioritize features. Table 5.1 details the key parts of each release, sorted by the earliest version number:

Version	Features
0.1	Server fully operational and able to serve static files.
0.2	User schema built and related functionality implemented.
0.3	Server able to retrieve user data from database and display to the client.
0.4	Router configured allowing server to render different views for each URI.
0.5	Employer schema built and related functionality implemented.
0.6	Jobs added to employer schema and are now able to be created by employers.
0.7	User, Employer and Job views created and rendering.
0.8	Clients can now register employers and users using the HTML forms.
0.9	Users and Employers can now log in and sessions are working.
1.0	Sessions now stored in database. Users and Employers can edit or delete profiles.

Table 5.1: Major releases in development phase

Table 5.1 shows the ten major releases, which are accordingly numbered with values at one decimal place. There were 16 releases in total, with the other 6 being minor releases that mainly included a number of bug fixes. Version 1.0 was the final release of the working product, where all user-stories had their cases satisfied and no major bugs were evident. Keeping the development process focused on features made it a lot easier to notice bugs and identify any important aspects that may not have been tested. During each iteration, a number of user-stories were provided which were all relevant to the feature being implemented. The necessary functionality was then developed, sticking to Node's asynchronous programming model while also trying to keep code organized and readable. Once the code had been written, testing of all major functions was carried out. Any failed tests resulted in a return to the coding phase, where the issue was identified and resolved. Once all tests had passed, the iteration reached it's release phase, where all updated files were committed in GIT and usually pushed to GitHub as well.

In terms of the languages and frameworks used during development, all of which were described in the “Architecture and Design” chapter, there were very few issues encountered. The asynchronous programming model that Node enforces actually made development easier, since it encouraged you to pass back to the caller whenever possible. Any modules downloaded using NPM stuck to this model, so there were no noticeable delays whenever large amounts of work were being done.

(5.2) Testing

All testing in the development phase was done using Mocha, which is a Node testing framework that allows for asynchronous testing. As outlined previously, testing was the third stage of each iteration and all tests were designed and executed after the coding stage.

As a testing framework, Mocha was generally pleasant to work with. It was designed to be very human-readable, a trait which was replicated with the tests designed for this project. In total, 66 tests were designed throughout the entire project, aiming to ensure that all key functionality was working as specified in the user stories. The following is an example of a test used in the project:

```
describe('passwordHelpers', function() {
    describe('#validatePassword', function() {
        it('should validate and return true', function(done) {
            var hash = 's5sO5FyNus4cbc1e71f341220813ca033cb6ba3557';
            passwordHelpers.validatePassword('password', hash,
done);
    });
});
```

As can be seen from the above test, Mocha allows for very human-readable tests. The "`describe`" function allows you to describe the object or element being tested and can be nested inside of another describe function. There is also an "`it`" function, that describes what the result of the test should be. The second parameter of the "`it`" function is an anonymous function; this is where you specify your test. This function takes one parameter, the callback method to be called when the test is complete, allowing for asynchronous testing.

Chapter 6

Evaluation

(6.1) Load Test Configuration

All load tests were carried out using a Samsung Series 3 350V laptop with an Intel Core i3 CPU with two cores, 6 GB RAM, and running the Ubuntu 12.04 distribution of Linux.

JMeter, an open-source Java desktop application from the Apache Software Foundation, was used to handle the load tests. This was chosen ahead of several other similar projects, namely because of it's graphical user interface which allowed tests to be set up very quickly.

Four load tests were performed on each application, with each set of tests being identical. All tests were looped five times, with the averages calculated to garner more accurate results. The four tests were as follows:

1. 10 users (1 second ramp-up period)
2. 100 users (2 second ramp-up period)
3. 1,000 users (5 second ramp-up period)
4. 10,000 users (10 second ramp-up period)

I used relatively low ramp-up periods to simultaneously try and put stress on the application. This would be especially evident in the tests simulating larger numbers of users.

(6.2) Load Test Results

The results of these load tests are displayed and explained in this section. As detailed before, three applications were used in these tests.

Two of the applications were identical, with one written in JavaScript running on a Node server and the other written in PHP running on an Apache server. By using two identical applications, it ensured that the results from the Node server could be compared with the results from the Apache server.

The final application to undergo load tests was the full scale jobs board application running on a Node server. The purpose of testing this application was to compare it's results to the smaller Node server, as this would give an indication of how well more complex Node applications perform.

All graphs provided in this section were generated using JMeter, the load testing software used for this part of the project. All relevant data was also recorded and is provided in the following sub-sections. The most important thing to note from all graphs is the throughput of the server, since that indicates how well the server performs under each specific load test configuration. As well as throughput, the other data notably relevant to the server's performance is the average response time.

(6.2.1) Graph Legend

Throughput	Data	Average (ms)	Median (ms)	Deviation (ms)

(6.2.2) Small Node.js Application

The following diagrams (Fig. 6.1 – 6.4) display the results of load tests on the small Node application. As mentioned in the previous chapter, the tests simulated a fixed set of users sending a request to the root URI of the application and expecting a simple HTML response from the server.

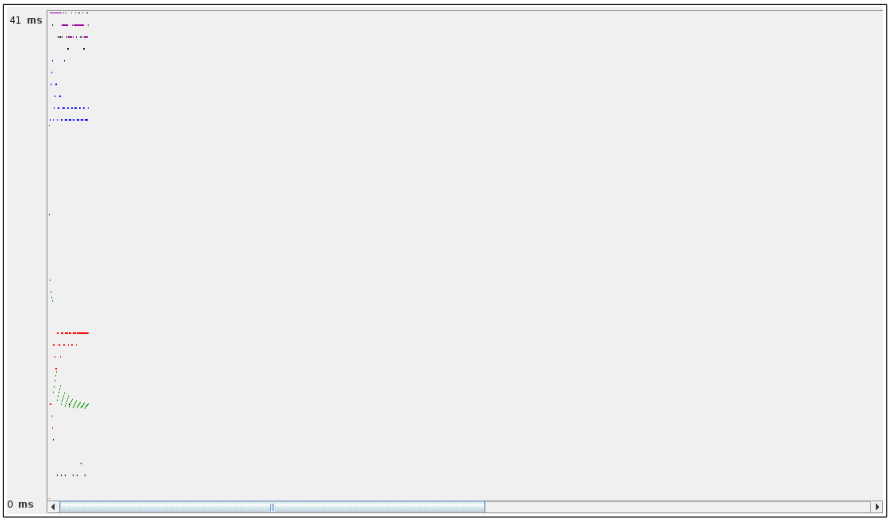

Figure 6.1: 10 users with 1 second RUP

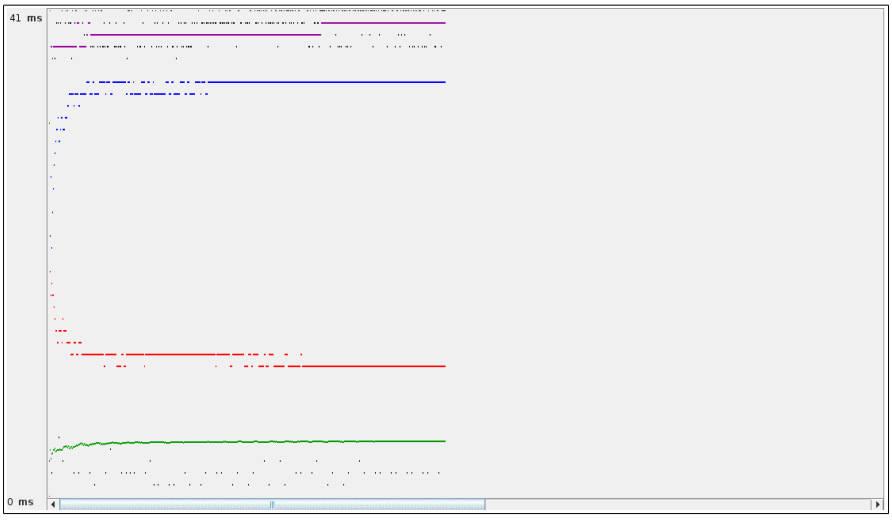

Figure 6.2: 100 users with 2 second RUP

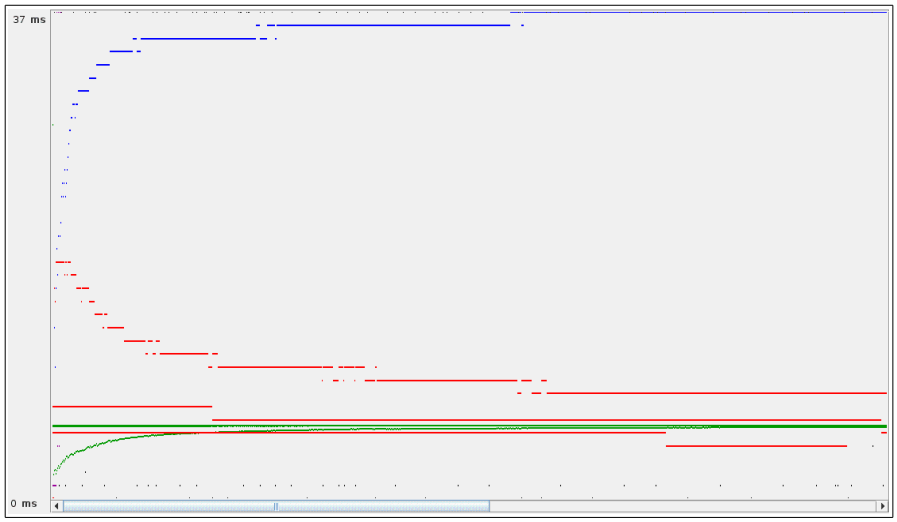

Figure 6.3: 1,000 users with 5 second RUP

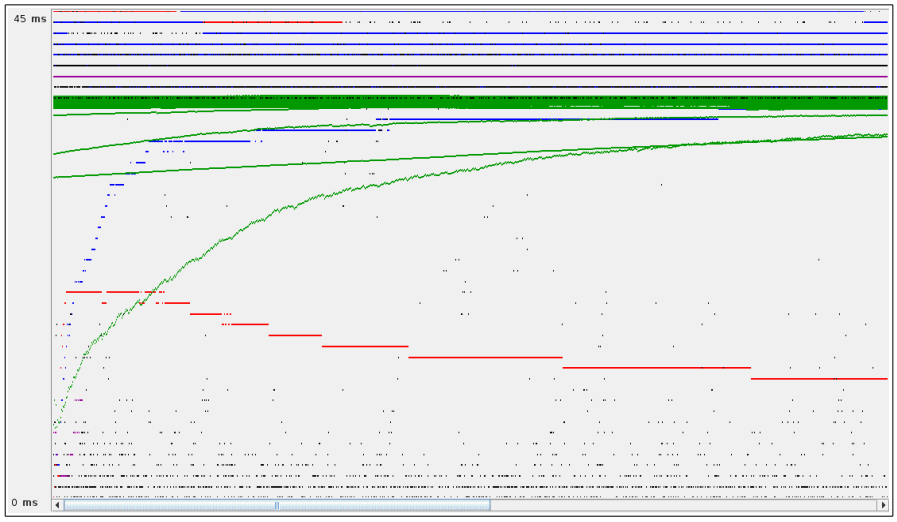

Figure 6.4: 10,000 users with 10 second RUP

It is important to note that the Y-axis in these graphs represents the response time while the X-axis represents the n-th sampler (in this case, n-th request). Table 6.1 shows the actual data from these tests, making evaluating them a little easier.

Test	No. Samples	Avg. (ms)	Median (ms)	Min (ms)	Max (ms)	Throughput
1	10	33	40	2	42	10.6/sec
2	100	35	40	1	46	49.3/sec
3	1000	39	40	0	58	195.3/sec
4	10000	39	39	0	2057	891.3/sec

Table 6.1: Node Server Load Testing Results

From Table 6.1, we can deduce that the average response time does not deviate too much regardless of the throughput. The maximum response time is also quite low up until the final test, when the number of requests reached 10,000. In the last test, 2.28% of the samples took longer than the desired duration to complete, though the maximum time of 2,057 ms is noticeable, but not a terrible response time at this load. The throughput is, on average, in line with the users / RUP.

When looking at the graphs, specifically Fig 6.3 and 6.4, it is obvious that Node displays it's best performance when dealing with a high number of concurrent requests. The throughput starts off quite small but continues rising as the number of requests rise, before eventually reaching it's peak and continuing at that pace until all requests have been successfully responded to.

(6.2.3) Small PHP/Apache Application

The following diagrams (Fig. 6.5 – 6.8) display the results of load tests on the small PHP/Apache application. Again, these tests simulated a fixed set of users sending a request to the root URI of the application and expecting a simple HTML response from the server.

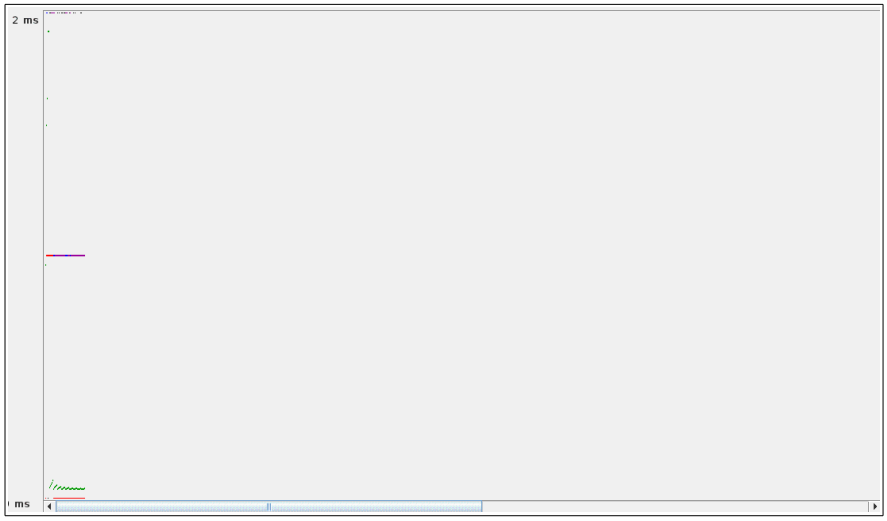

Figure 6.5: 10 users with 1 second RUP

Figure 6.6: 100 users with 2 second RUP

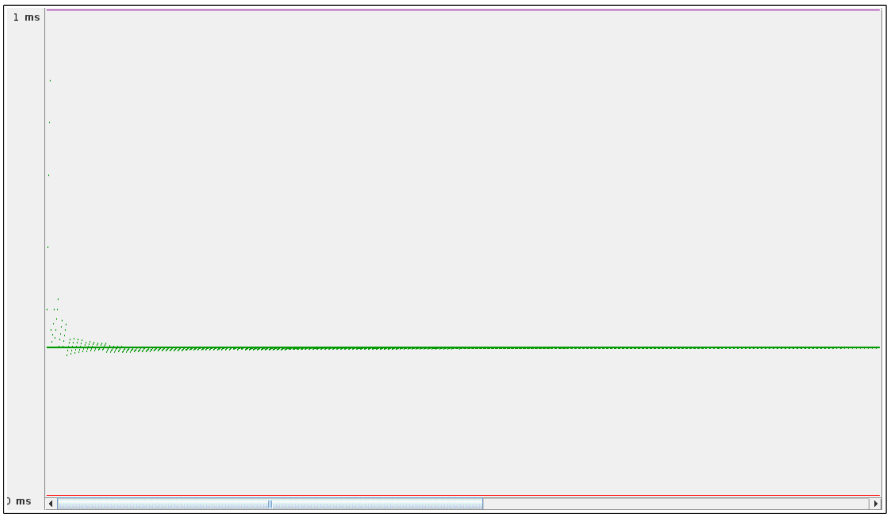

Figure 6.7: 1,000 users with 5 second RUP

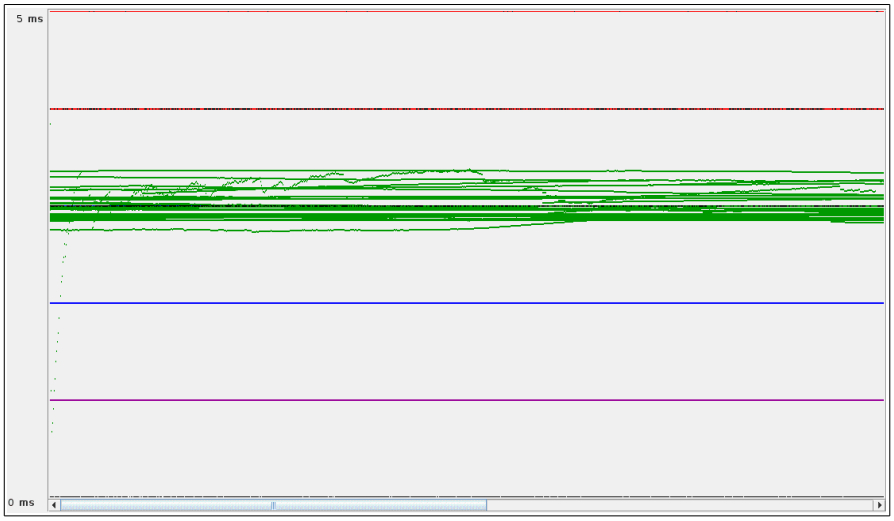

Figure 6.8: 10,000 users with 10 second RUP

What is immediately noticeable in these graphs compared to the Node server's results is that there is no noticeable easing-in period. The throughput value on the Apache server rises extremely quickly and levels out at it's peak value until all requests have been responded to. There is nothing much else of note than can be deduced from these graphs, so in Table 6.2 you can see the statistical data behind these graphs.

Test	No. Samples	Avg. (ms)	Median (ms)	Min (ms)	Max (ms)	Throughput
1	10	1	1	0	4	11.1/sec
2	100	0	1	0	8	50.4/sec
3	1000	0	1	0	23	199.1/sec
4	10000	2	1	0	425	747.3/sec

Table 6.2: Apache Server Load Testing Results

These results are actually quite surprising, since the Apache server more than matches the Node server until just after the 1,000 user mark. Up until that point the Apache server yields slightly better results. The major bonus with this server is that the average time to complete a sample is almost zero milliseconds (both zero value results were actually rounded down from < 0.5).

However, as the number of concurrent users rises, the Apache server begins to hold at around 750 requests per second. Meanwhile, the Node server continues on and eventually evens out at almost 900 requests per second. This is quite a large difference per second, but when calculated per minute it shows that Node can handle over 8,640 more requests than Apache when dealing at this specific load rate.

Further testing revealed that with 20,000 simulated users and a ramp-up period of 20 seconds, the Node server's throughput stayed consistent at just under 910 requests per second while the Apache server eventually returned an error message and ceased processing requests.

(6.2.4) Jobs-Board Application (Node)

The final part of the load testing involved carrying out the same tests on the full-scale Node application. The following graphs (Fig 6.9 and 6.10) show how this application handled these tests.

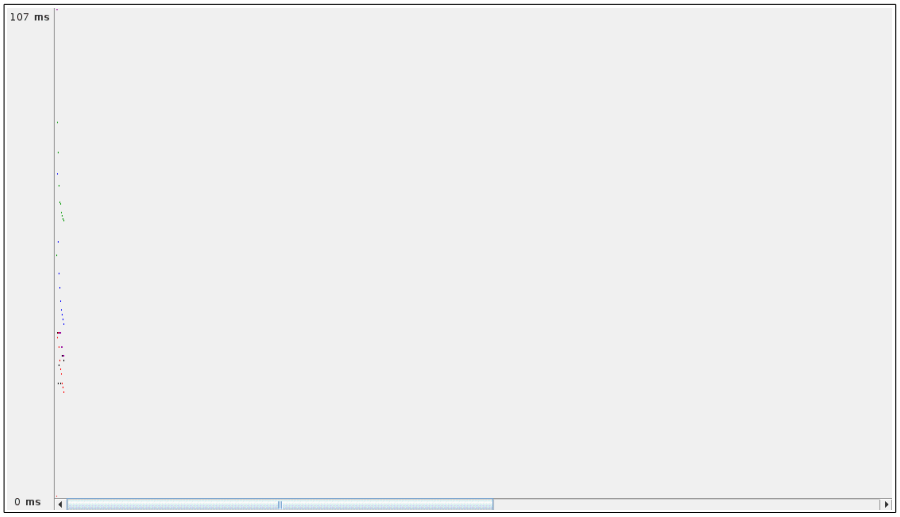

Figure 6.9: 10 users with 1 second RUP

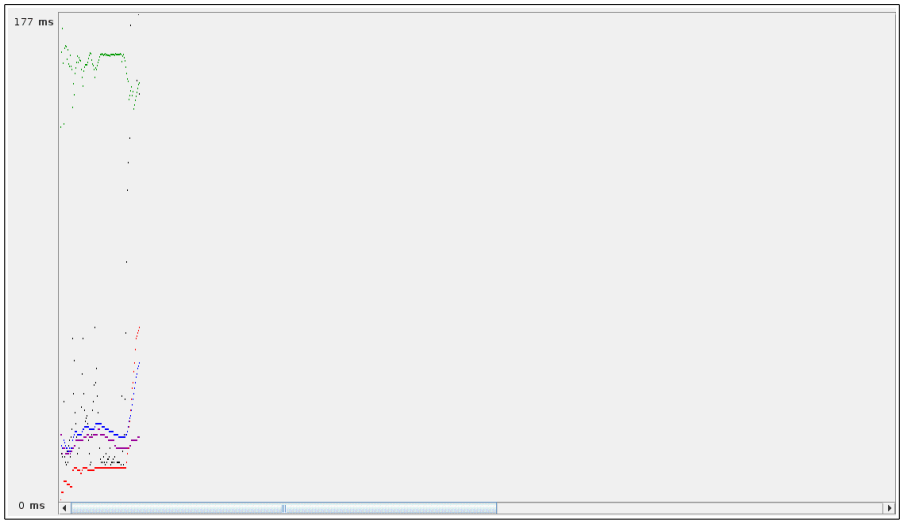

Figure 6.10: 100 users with 2 second RUP

Test	No. Samples	Avg. (ms)	Median (ms)	Min (ms)	Max (ms)	Throughput
1	10	38	31	25	107	10.7/sec
2	100	50	23	12	273	46.6/sec

Table 6.3: Jobs Board Server Load Testing Results

Unfortunately, the application could not reliably handle the same tests for 1,000 and 10,000 users and response times were sometimes upwards of five seconds. One of the main reasons for this was the lack of a caching module in the application. So many requests for the same data from the database was causing a bottleneck. The other reason was the extremely small ramp-up periods. These large numbers of concurrent requests were adequately handled by the two smaller application's servers, but this server had too much work to do to quickly respond to so many requests. One way of circumventing this issue would have been to use a memory object caching system like Memcached, which would have taken a significant load off the database by caching the most frequent query results in memory.

As can be seen from the graphs and table above, the application actually performed almost identically to the smaller Node application, with the average response time for 10 users being the same. The throughput was only slightly less than the smaller application, though that was to be expected since this application is much more complex.

(6.3) Conclusions

The end-result from this thesis work was a full-scale jobs board web application built using Node.js. The Express framework allowed for rapid development and displayed the power given to the developer when using the Node software system. Data storage for the application was handled using MongoDB, which allowed for dynamic querying and a highly scalable storage layer. To allow for a much more scalable application and relieve some stress on the database layer, a caching system could have been used to store the results of the most frequent queries in memory. Unit testing was carried out in the third phase of each iteration to ensure stability and relative security. Finally, load testing was carried out on this application, as well as two smaller, identical applications, one running on a Node server and the other on an Apache server.

It is easy to conclude from this thesis work that Node is very suitable for web application development. It offers minimal upfront functionality that is highly extensible, giving a large amount of control over the application and server to the developer. Using front end and server side JavaScript makes development much more straightforward, while having full control over how responses are handled helps ensure a much more satisfactory user experience.

With regards to the performance of Node web applications, the results from the load testing showed how well Node handles large amounts requests. Although MongoDB could not handle extraordinary amounts of concurrent queries, a caching system would have reduced I/O operation for the server. Even so, the load tests were aiming to test the boundaries of the applications and it was very obvious that the jobs board application's performance, being that it was much more complex, was bound to deteriorate sooner than the two smaller applications.

(6.4) Future Work

This thesis has covered the implementation of a full-scale web application using Node.js and MongoDB. There are many more areas of research that can be explored as an extension of this.

(6.4.1) Scalability

From the load test results of the jobs board application, it was inherently obvious when there were high numbers of users and low ramp-up speeds that MongoDB was struggling with that volume of queries. The database itself was stored on a single machine. Adding another machine would have enabled the usage of sharding, MongoDB's approach to scaling out. This could have reduced the stress on the database considerably. Adding a caching system to store the results of the most frequent queries could also help the application scale considerably better. Implementing these extra scaling features could be an intriguing future extension.

(6.4.2) High I/O Volumes

The jobs board application created for this thesis is a standard web application, though it does lack any features that would require significantly high amounts of I/O operations at any given time. Features like a chat client, a notification center and file reading/writing could test the I/O capabilities of Node much further, and would be a challenging future extension to the project and load tests.

(6.4.3) More Complex Load Tests

The load tests in this project simply sent a number of concurrent requests to the same URI and waited for a response. To adequately test the

performance on a larger scale, more complex tests could be carried out on the jobs board application. These tests could include simulating common user interactions like logging in, registering, posting job positions and general navigation of the various pages of the application. JMeter provides the functionality to produce these types of tests and it would be an interesting addition to the research, since it would be testing user-interaction of a more realistic manner.

Chapter 7

References

Crockford, D. (2002) JSON. Available at: http://www.json.org (Accessed 20 November 2012).

Teixeira, P. (2013) *Professional Node.js: Building JavaScript Based Scalable Software.* Indianapolis: John Wiley and Sons.

Joyent, Inc. (2012) Node.js. Available at: http://nodejs.org/ (Accessed 21 November 2012).

Tilkov, S. and Vinoski, S. (2010) 'Node.js: Using JavaScript to Build High-Performance Network Programs'. *IEEE Internet Computing*, (November/December 2010): 80 – 83

Crockford, D. (2008) *JavaScript: The Good Parts.* California: O'Reilly Media, Inc.

McCune, R. R. (2011) *Node.js Paradigms and Benchmarks.* Project Draft. University of Notre Dame.

Paudyal, U. (2011) *Scalable Web Application Using Node.js and CouchDB.* Unpublished Thesis. Uppsala University.

Strauch, C. (2011) *NoSQL Databases.* Research Paper. Stuttgart Media University.

Khramtchenko, S. (2004) *Comparing eXtreme Programming and Feature Driven Development in Academic and Regulated Environments.* B. Sc. Research Paper. Harvard University.

LearnBoost (2013) Mongoose. Available at: http://mongoosejs.com/docs/guide (Accessed 13 February 2013).