NODE JS PERFORMANCE TESTING

A Senior Project

presented to

the Faculty of the Liberal Arts and Engineering Studies Department

California Polytechnic State University, San Luis Obispo

In Partial Fulfillment

of the Requirements for the Degree

Bachelor of Art

by

Massimo Siboldi

June, 2014

© 2014 Massimo Siboldi

Terms

Data: "Information processed or stored by a computer. This information may be in the form of text documents, images, audio clips, software programs, or other types of data."[4]

Program: A series of instructions for a computer to carry out. This can include playing a song, requesting a web page, and checking the time.

Server: A computer that sends data over the internet.

Backend: A general term for servers and programs that are executed on servers.

Client: A computer used to access data from a server.

Frontend: A general term for clients and programs that are executed on clients. In the context of web development, these programs are sent to the client's browser by the server.

Request: A message sent from the client.

Response: Data sent to the client by the server.

HTTP: Hypertext Transfer Protocol. This allows computers to communicate over the internet through uniform syntax.

Throughput: The average number of requests handled by a server per second.

Response Time: This is the time taken for a client to fully receive a response..

Maximum User Load: This is the number of clients that can be using a server simultaneously.

Stress Testing: Simulating a large number of clients using a server.

Load Testing: Simulating an expected number of clients using a server.

Soak Testing: Simulating an expected number of clients using a server over a long period of time, with a goal to uncover memory leaks and other unexpected problems that may occur.

Uptime: This is the time a server continuously runs before reaching an undesirable state, such as crashing, slowing, or producing errors.

JavaScript: A programming language used by browsers to provide dynamic functionality to websites.

R: A program used for data analysis.

JMeter: A program used for simulating a number of clients using a server.

CPU: Central Processing Unit, found on all computers to execute programs.

RAM: Random Access Memory, found on computers to store data temporarily. Doing this is faster than saving it to a hard drive.

Memory Leak: Am error in a program that results in the RAM being taken up by unnecessary data. Over time, this causes a reduction in uptime due to crashes and unexpected errors.

CSV: Stands for Comma Separated Value, a text file containing spreadsheet information with columns delimited by commas and rows delimited by new lines.

Node JS: A server side JavaScript implementation, gaining popularity due to programmers' familiarity with JavaScript, low system requirements, and simplicity since its release in 2009.

DIAGNOSING AND AMENDING PERFORMANCE ISSUES IN NODE JS

Web applications must be functional. Developers work to accomplish this goal by taking complex problems and breaking them into smaller, manageable chunks. They solve these problems on organizing and sending data, displaying images and buttons, etc. The question is: How can we make this work? In general, if a program produces the desired output for given inputs, it is functional.

However, there is more to apps than functionality. Web apps can be pleasant or a pain to use. They can be unresponsive or responsive. The goal of any software is to be used. In considering the non-functional aspects of a web app, developers can create something that not only works, but does so with ease. Reliable, speedy, beautiful websites are a joy to use.

SPEED

The internet has brought us many great things, and one of them is connectivity speeds. This is a benefit for the free spread of information, and not a benefit for the urgent sense of instant satisfaction everyone has now. In competing with other websites and satisfying their customers, companies should strive to have their websites be as fast as possible. The fast sites get the attention, users, and profit. It doesn’t matter if a site is functionally useful; consistency and speed make or break websites.

75% of users, in a study by Akamai in 2006, reported they wouldn’t wait for a website taking more than 4 seconds to load. [13]

Music streaming applications suffer the same fate as any other site: They must be fast and reliable or lose the race for customers and their satisfaction. Songs and content should be as immediate as possible.

RELIABILITY

Unreliable websites lose customers. 99.9% uptime is desired in web applications, so that customers have something stable to go to. Random inconveniences are hard to deal with. Imagine a website that loaded most, but not all, of the time. If it were an email client, or something else you rely on, it would be frustrating to not have access to something you thought was constant. Frequent downtimes lead to customers never coming back.

To make websites reliable, they have to hold a certain amount of users for an indefinite period of time. Assessing the maximum user rate is important here. You can't just make a web app fast and hope that it can serve a lot of people.

Usually, servers crash when memory leaks occur and the program crashes. To test for this, you use a soak test[18]. Soak tests emulate many users on a website and report back when the server goes down. Hopefully, this will never happen. When it does, I will analyze the logs of what happened and will fix the problems that come up.

MUSIC PUTTY

I am the lead developer for Music Putty, and will be doing performance tests for Music Putty's server. Music Putty is a platform for emerging musicians and their fans. For listeners, Music Putty provides a way to listen to ad-free music of your favorite musicians. For musicians, Music Putty provides a platform for receiving crowdfunding, music sales, and a

community for them to gain supporters and fans. A central aspect of Music Putty is its music service, where listeners can add songs to playlists or use the radio service to discover new local bands. Eventually, we will have the ability to find local shows of people on Music Putty.

IN ADDRESSING THE CONCERNS OF SPEED AND RELIABILITY

I will attempt to solve the problems of testing for flaws, assessing code / server setup, and developing and implementing potential solutions. I will focus on improving the speed and reliability of web applications to compete with or surpass that of other companies.

I will be assessing Music Putty's server technology and implementing solutions to bottlenecks in speed and reliability. This involves first running diagnostic tests. Then assessing these results, and lastly, through research and code examination, implementing improvements to current server side builds.

In a startup environment, there are limited resources to accomplish much-needed goals. Without money, it is difficult to hire professional web developers who are willing to spend the time to develop a platform. Many times founders must do roles outside of what they're used to. In Music Putty, the CEO Arash Namvar is also designing web pages, testing code, signing up bands, and writing legal documents such as our terms of use. This being said, teams are usually small and dedicated to the cause.

Although my primary responsibilities include programming the frontend and backend of Music Putty's web application, I also have to do various kinds of tests that would usually be deferred to others. I am performing performance tests on Music Putty's Node JS backend server, implementing improvements, and reporting the results of these tests. My deliverable is

an analysis on Node JS performance improvement strategies, as well as the scripts and files used to test and analyze the server.

MEASUREMENTS OF SUCCESS

I am testing in a local environment to avoid complications and costs that aren't necessary for the scope of this project. There are many variables that can affect the speed of a request. These include the server's CPU speed and the network strength or connection quality of a user. In that this senior project is intended to discuss the changes in node.js code that result in performance increases, not server setup or connectivity, I will be basing my success on improvements from initial tests, rather than on industry standards for professional web applications. I plan on testing for performance improvements locally to reduce uncertain factors beyond the scope of this project. Comparing a local server with a real world server to measure the success of this project is unrealistic.

There are a few measures of server performance that I will be testing for. As reported by Dr.Kumar Ramakanth[16], three quantitative measures of server performance are:

- Resource Utilization
- Response Time
- Throughput

He went on to name a few methods of approaching performance tests:

- Load Testing
- Stress Testing
- Strength Testing (also known as Soak Testing)

I will be testing Node JS for throughput, response time, and uptime. I will basing my success off a percent improvement of response time and throughput, and a goal of 12 hours uptime.

These were determined from talking with my Music Putty team and coming up with numbers we thought would be qualified as successful.

Quantitative Measurements of Success

	Failure	Improvement	Success
Response Time	0 - 10% decrease	10-50% decrease	> 50% decrease
Throughput	0 - 10% increase	10-50% increase	> 50% increase
Uptime	< 6 hours	6 - 12 hours	> 12 hours

LITERATURE REVIEW

From my research, I have found that most articles written about Node JS discuss the possibilities of the language without discussing techniques in getting an intended result. Those that do have been found to not be reputable for various reasons. There are also articles on improving server side technologies in general, these being theoretical and discussing concepts rather than real world techniques in overcoming problems revealed through performance testing. There is a noticeable lack of reputable articles that discuss performance issues and solutions for Node JS itself.

I have found few academic articles on the performance testing of specific web applications. Most have to do with the theories behind testing. For example, Dr. Kumar Ramakanth wrote an article published by the International Journal on Computer Science and Engineering (IJCSE) . In this, he defined criteria for testing web applications.[16]

He also discussed the importance of WAN simulation in testing environments. Since it is costly to perform testing on actual servers, and thusly is preferred to do tests on a LAN

environment, he recommends simulations situations in poor network reception, latency, packet loss, and bandwidth. Doing these tests will reveal problems otherwise not known to developers testing in less imperfect circumstances.

This information is of great use to the world of web developers seeking the perform these testing, but provides no access point or applicable techniques for doing these things. He goes over the importance of creating cheap but realistic systems, but offers no recommendations in this regard.

There have been articles published in journals about Node Js. Node.js: Using JavaScript to Build High-Performance Network Programs, released by IEEE, S. Tilkov explores the possibilities of Node JS in brief overview. He goes over the theoretical differences between Node JS and other server side technologies. He compares multithreading, a popular solution to handling IO requests used in commonplace languages such as PHP, to event-based asynchronous execution. [20]

He also presents a solution to utilize the server's full processing power while developed in Node JS. Using multi-node, multiple instances of one Node JS application can listen to the same port, creating an effect similar to a load balancer. He asserts that Node JS doesn't sacrifice performance in his conclusion, but doesn't include data on this. It's more of a theoretical assumption. As Node JS is an emerging technology, as is using Javascript for backend programming, many articles like these are more likely to discuss possibilities rather than solutions. Because of this hype, it is hard to accurately discuss the limitations of Node JS and overcoming them.

There are blog articles released by reputable companies. In a tech blog released by Ebay[7], Senthil Padmanabhan discussed the company's first ever departure from Java and the JVM. His team's decisions to use Node JS are similar to many articles about Node JS, like Tilkov's. Tilkov, along with the Ebay, references Node JS's ability to handle non-blocking execution and handle live communication with its servers. Where Java didn't perform how they expected it to, Node JS and server-side Javascript could take on these challenges. He noted its ease of deployment. They use a shell script that builds Node JS's packages and pushes the code to their deployment servers. As long as Node JS's interpreter is installed, servers written in the language are highly portable. I have had similar experiences running production code on my laptop and having it work flawlessly on Music Putty's servers. In general, this article goes over the decision making process of choosing a reputable language to design server-side technologies in, and notes the possibilities of Node JS, but doesn't cover issues in deployment or other problems that had to be overcome.

In a similar article written by Ryan Paul for Ars Technica about LinkedIn's mobile engineering technologies [15], Paul discusses the building of a scalable backend. He described his team's objectives as designing a technology that was "fast, easy to work with, and reliable"[15]. Upon choosing and building the application with Node JS, they found it to use less memory and offer better performance, sometimes as fast as 20 times the performance of other considered solutions. They were able to spend less money on the servers, reducing their count from 30 to 3, and could spend less time worrying about scalability. Articles like these solidify the concept of Node JS being suitable as a scalable solution for web apps, but doesn't go into detail about the implementation or problems that may occur.

There are less than reputable sources on cheaply testing server environments. Much of the available Node JS performance and scalability tests are run as blog posts, which do not have the authority of peer-reviewed documents. Aaron “Caustik” Robertson wrote a post of testing the scalability of Node JS using Amazon’s EC2 service, for example[17]. Because of this informality and the lack of academic scrutiny, he is able to freely discuss real world solutions he used to perform performance tests on servers using Node JS. Unfortunately, also as a result of this informality, he is able to perform tests without validating their integrity. We have no analysis of results of his tests, and so his concerns don’t hold as much weight as they otherwise would.

From my lack of finding a reputable article articulating the needs of performance testing and real-world techniques and recommendations on doing so on a budget, I have found this project to cover an interesting area of computer science that deals with non ideal situations that come with being a startup with limited human resources and capital. Much good has been said about the potential benefits of using Node JS, but this general optimism doesn’t help solve problems.

TECHNOLOGY USED

NODE JS

There are many ways to implement a backend server. In general, because of the bottlenecks presented through network connections, database queries, and program executions, there must be a way to keep executing code while waiting for a response. In dealing with this, many languages such as PHP and Java adopt the concept of multithreading. This is when multiple threads of execution are utilized, running concurrently on multiple cores. In this way,

threads waiting for a server response can create a new thread and continue execution until the response is received [12].

The other method of dealing with this bottleneck is through what is called asynchronous execution. In one thread, a piece of code initializes an operation and waits for an event to complete it, allowing execution of other pieces of code. Javascript follows this pattern, which allows low memory usage and simpler code[5].

For Music Putty's server, I chose to use Node JS, which is a server side Javascript implementation. Due to its asynchronous nature, Javascript turns out to be ideal for starting many processes and waiting for responses before continuing. In sequential programming languages, code is executed line by line. If a request is made to another server, it waits for that response before continuing execution. This time spent waiting could be better used for executing more code, which is what Javascript does. As a server is always making requests to a database, which is a separately running program, Node JS can be speedy and do multiple things at once.

It is still a developing language, which makes it interesting to test its performance. In theory, it should be reliable and quick, but we will not know for sure until many people have used it for an extended period of time. During this time, it is necessary to question and test Node JS so that it can improve.

TESTING ENVIRONMENT AND DATA COLLECTION

There are services and programs made for doing various performance on servers. In general, services are owned by businesses and provide a variety of features, but at a high cost, with a clientele of businesses. Examples of these are LoadImpact.com and Loader.io.

Programs are usually freely distributed software that anyone can run. They are more customizable, but because of his also need more time to get set up. Also, there are considerations of how to host the instances of the program when dealing with load tests.

For testing speed and reliability, we need a load generating program. These simulate multiple users requesting information from a server. I am looking for one that could simulate conditions to test for maximum user load, average response time, average load time, and server reliability over time.

I have decided to use JMeter for these tests for a variety of reasons, most documented in an article published by the International Journal of Emerging Technology and Advanced Engineering, titled "Identification of Performance Improving Factors for Web Application by Performance Testing." It describes JMeter as an open source testing platform that can be used to carry on performance tests. It is modular and capable of handling static and dynamic requests, as well as support for cookies, headers, and constructs to process server responses. It is also capable of graphically representing results in real time, which allows for quick debugging and verification of tests during test creation.[27]

JMeter was useful in its abilities to simulate an actual user, create non ideal network situations, and simulate a web browser through using a concurrent testing system that emulates the way browsers request files linked through HTML.

I will not be explaining the details of using JMeter. You can find helpful tutorials and complete documentation from JMeter's homepage[1].

JMeter was used for all data collection. JMeter's listeners can save samples in .csv format.

SETUP

For these tests, I am using Linux, a UNIX-based open source operating system. It follows the other technologies used in that it is free, accessible, and widely used as the OS for production servers.

These tests were run on one Linux Machine running Ubuntu 14.04. JMeter and my Node JS server were run simultaneously. Due to limitations of my router and complications facing reliable LAN setup, this solution was used. Benefits to this decision is the elimination of a variety of unknown parameters affecting latency, such as WLAN signal strength, interference, and limitations regarding packet size. For some tests, an Nginx 1.6.0 was used as a static file server.

HARDWARE

Desktop Computer with:

Athlon X2 7750 (2.7GHz) CPU
4GB DDR2 500GB HDD RAM
NVIDIA GeForce 9500 GT GRAPHICS CARD

DATA ANALYSIS

LibreOffice is an open source office suite[9] which is similar to Microsoft Office. Using Calc, their spreadsheet program, I was able to calculate T distributions to statistically verify claims on decreased response time and increased throughput.

R is a scripting program tailored for data analysis. R was used to generate graphs and means of soak tests due to their large sample size (over 1 million samples per soak test), which LibreOffice couldn't handle.

IMPLEMENTATION

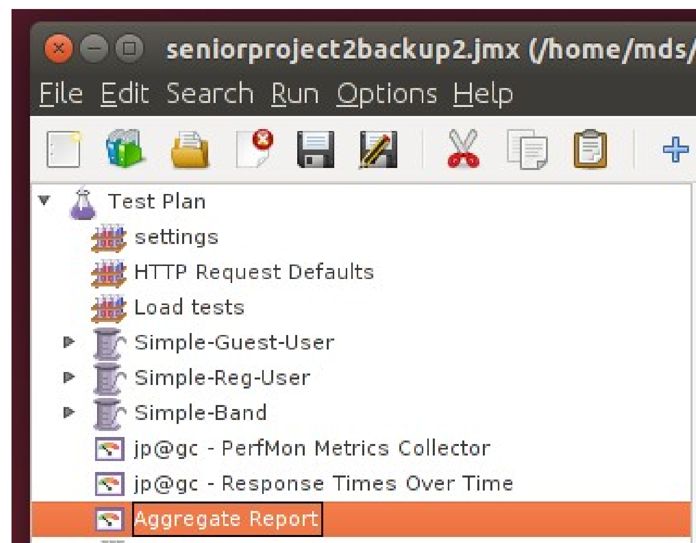
DEVELOPING TESTS

The first step in doing performance tests is to determine and create the methods used for testing. According to Nicolas Vahlas, a lead software architect for Quality & Reliability A.E., to make tests, you need to first clearly state your objectives[23]. Mine is to discover bottlenecks in a Node JS system and software, to expose them for future developers who are building back end server technologies. Without any technical training in networks, it was hard for me to find best practices, and I spent most of my time developing. I want to provide a guide for future developers to get a sense of improving network scalability.

Vahlas says "a good methodology is always to try and implement scenarios that are as close as possible to real and typical use cases of the system you are willing to test."[23] Using JMeter, I have built a system that closely emulates the behavior of users assuming various roles, such as a guest user, who navigates to the homepage, looks at it, and leaves. Another role is a registered user, who navigates to the homepage to register for or log into an account. The last role I determined for our alpha release is that of a musician, who would be registering for or logging into an account, editing their band's information, and uploading new albums to Music Putty.

Using what are called thread groups, JMeter can concurrently run the requests associated with these roles. A thread group is a group of elements such as conditional elements, samplers, timers, settings, and listeners, which all work to emulate the requests of a type of user. For example, you can use a conditional element to randomly decide if a user will sign up for an account or log into an existing one. Samplers are used to make HTTP requests, through specifying a URL, port, body content, and files to send in a multipart form upload. Timers are

used to create a more realistic delay for user navigation. The default setting is to send requests as quickly as possible, but this is unrealistic in that it's not what regular users would do. The settings can be configured to send HTTP headers and cookies along with each request. Finally, listeners are used to the record the results of these tests. You can get the data associated with every request, an aggregate report of throughput, latency, response time, and graphs of various parameters. For my tests, I used a plugin by JMeter-Plugins to monitor the server's memory and CPU usage, as well as a plugin by the same group called "Response Times Over Timer", which compiles a graph of each response time for each sampler used. I used these plugins to get additional functionality out of JMeter.



Overview of settings, thread groups, and listeners.

Once realistic tests were produced, I went on to set the load of each thread group for each type of test. In thread groups, you can set various parameters of the test.

The “Number of Threads” parameter corresponds to the number of simultaneous connections made to the target address. It simulates a specific number of users making requests to the server. [23]

The “Ramp Up Time” is the time for JMeter to reach its specified number of threads. During this time, the theoretical number of simultaneous connections is **tn/T**, where t is current time elapsed, n is the “Number of Threads” parameter, and T is the “Ramp Up Time” parameter.[23]

The “Number of Loops” parameter corresponds to the number of times each thread will execute the specified scenario.[23]

In changing these settings, we can effectively create load, stress, and soak tests needed to evaluate the server’s reliability. To create a load test, supply an estimated number of users for some amount of time. For a stress test, you can specify a large concurrent user base with a long ramp up time, recording the number of active threads upon a server crash. For a soak test, you can supply an estimated number of concurrent users for an infinite duration, stopping the test upon a crash, or when the established soak time is reached.

In addition to these basic settings, I found it important to parse the HTML of each request to properly emulate a browser. This will fetch images, css, and javascript necessary to load a page.

You can do this by checking the “Retrieve All Embedded Files” option, and by selecting the option to use concurrent pooling.[2] This is the method regular browsers use, so in doing this you can more closely simulate real conditions [22].

JMeter allows listeners to write data to a file, allowing you to analyze it later. By checking all of the parameters you want saved, and choosing the file destination, you can open the results in Microsoft Excel or a similar spreadsheet program.

IMPLEMENTING SOLUTIONS

Once these tests have been initially completed, I implemented various changes to Music Putty's Node JS code and software setup. Most of these proved to decrease response time and increase throughput and uptime.

1. Operating System Settings

At first, load tests were crashing due to Ubuntu's maximum file limit. For each connected user, a TCP connection is made. Each TCP socket established creates a new file in Linux systems [8]. In having a high concurrent user load, enough TCP sockets, in adittion to other files, will be created to break the server. Linux stores the set maximum number of files and reveals them through the command "ulimit". Through this command, I was able to set the maximum number of files from 1000 to 1000000, reducing the possibility of reaching that limit.

Additionally, JMeter was initially running out of memory from generating the requests necessary to simulate 210 concurrent users.

```
mds@jigglesIV: ~/Documents/apache-jmeter-2.11/bin
summary +    349 in  14.3s =   24.4/s Avg:  5329 Min:   963 Max:  9360 Err:
1 (0.29%) Active: 737 Started: 2753 Finished: 2016
summary =  26178 in   369s =   70.9/s Avg:  2075 Min:     2 Max: 149613 Err:
93 (0.36%)
summary +    290 in    20s =   14.5/s Avg:  6244 Min:   962 Max: 14621 Err:
0 (0.00%) Active: 756 Started: 2800 Finished: 2044
summary =  26468 in   375s =   70.6/s Avg:  2121 Min:     2 Max: 149613 Err:
93 (0.35%)
summary +    416 in   169s =    2.5/s Avg:  8058 Min:  1119 Max: 165990 Err:
 2 (0.48%) Active: 768 Started: 2845 Finished: 2077
summary =  26884 in   381s =   70.6/s Avg:  2213 Min:     2 Max: 165990 Err:
95 (0.35%)
summary +    301 in   168s =    1.8/s Avg:  8498 Min:  1689 Max: 162778 Err:
 1 (0.33%) Active: 788 Started: 2894 Finished: 2106
summary =  27185 in   387s =   70.2/s Avg:  2283 Min:     2 Max: 165990 Err:
96 (0.35%)
Uncaught Exception java.lang.OutOfMemoryError: unable to create new native threa
d. See log file for details.#
# There is insufficient memory for the Java Runtime Environment to continue.
# Native memory allocation (malloc) failed to allocate 32756 bytes for ChunkPool
::allocate
# An error report file with more information is saved as:
# /home/mds/Documents/apache-jmeter-2.11/bin/hs_err_pid728.log
mds@jigglesIV:~/Documents/apache-jmeter-2.11/bin$
```

JMeter OutOfMemory Error

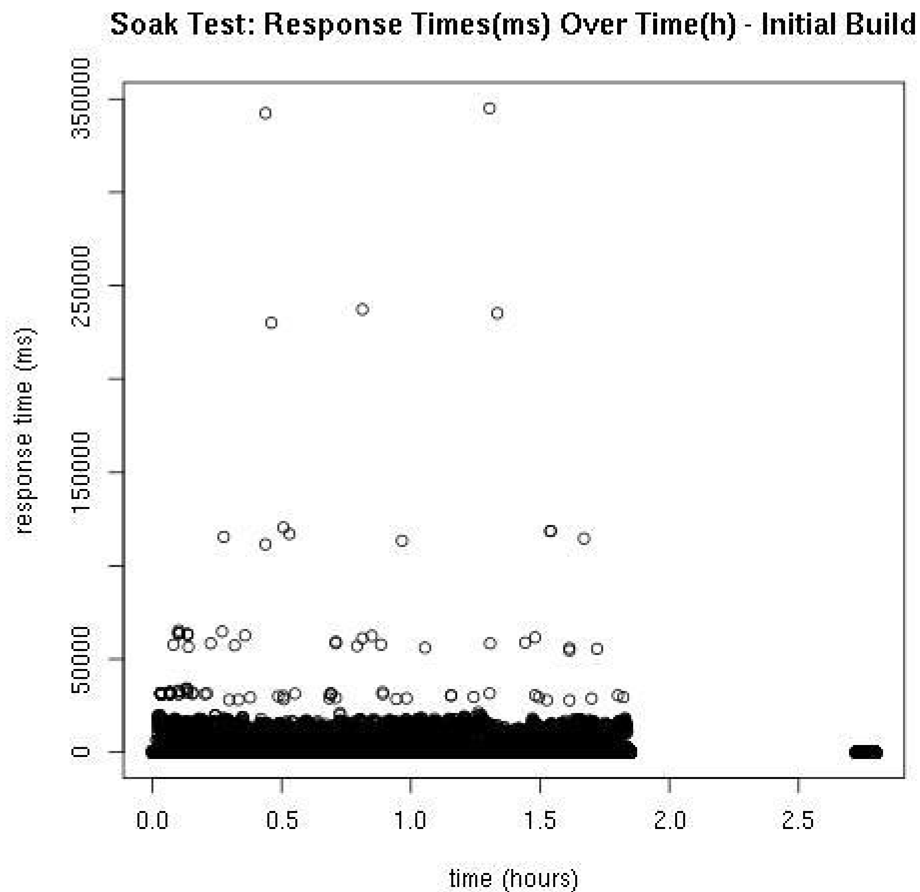
To fix this, I set Java's maximum heap size to 2GB through JMeter's configuration file. It defaults to 512MB, which isn't enough for larger, more complicated tests.

2. Debugging and Increasing Reliability

Performance testing led me to discover bugs in the codebase that adversely affect the server when under a regular expected load over a substantial period of time. These bugs would have otherwise gone unrecognized.

The initial soak tests, with 210 concurrent users over a substantial period of time (>1 hour) was initially crashing after 1.8 hours. This was due to an ENOSPC error. This error is thrown when there isn't enough space on the destination drive[25]. It was discovered that Music Putty's server wasn't correctly handling the deletion of temporary files. Upon fixing this by

deleting the temporary files as they were moved to their intended destination, the server was able to last at least 6 hours.



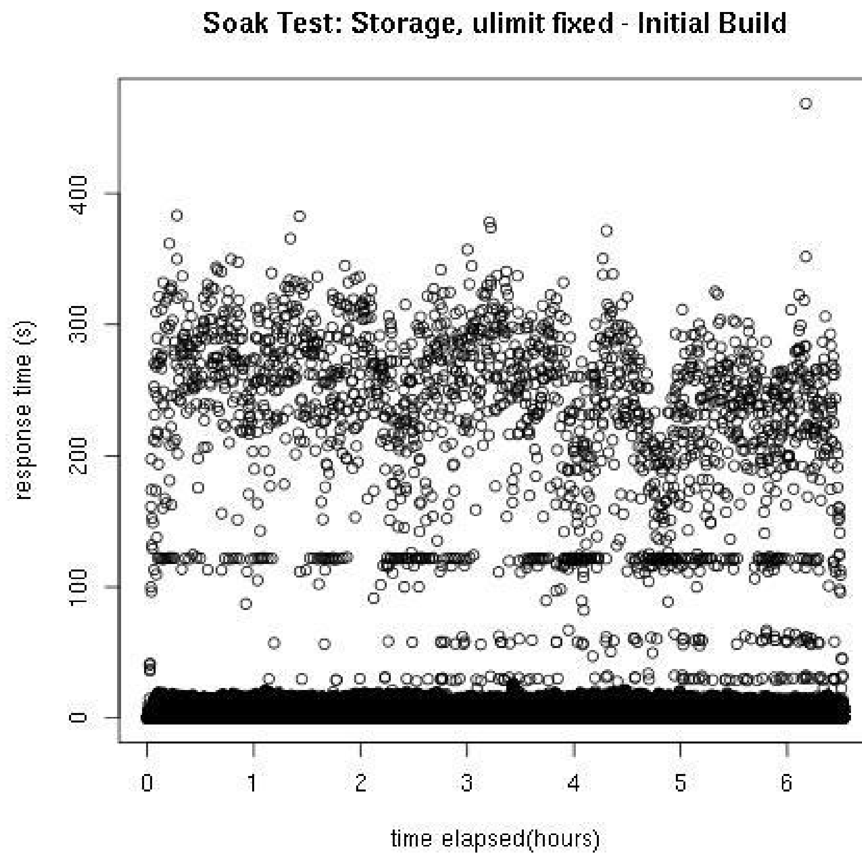
Soak test, showing crash after 1.8 hours due to ENOSPC error.

Another problem that was discovered was a 10-20% error on each band request past sign up. Through using JMeter's Result Tree listener, which records all attempted requests and their responses, I traced the error back to the band thread groups. During the band's account creation, there was a function was using global variables to relate users to their bands. These variables changes as users were created simultaneously. A table exists in our database called

band_member, where the member's ID and band's ID is inserted. This table is later used to authenticate a user trying to modify a band or upload an album.

The global variable was storing the band's ID, and through handling more than one band creation by the time an insert statement was executed, the band's ID belonged to a different member, causing subsequent authentication errors.

In this soak test, the effects of this bug are prominent. The high density of response times at 120,000 ms is due to the timeout of the improperly handled errors that relied on the user being a member of a band. Through the addition of error-handling code, I limited the possibility of these timeouts occurring in future bugs.



Soak test, with 120 second timeouts, long song upload time.

In addition to these errors, the mean response time was 9758 ms. The mean response time for the "home page" and "sign up page" requests took an average of around 17800 ms.

```
> mean(agg$X181[agg$make.an.account == "nav to the home"])
[1] 17985.73
> mean(agg$X181[agg$make.an.account == "nav to signup"])
[1] 17756.17
>
```

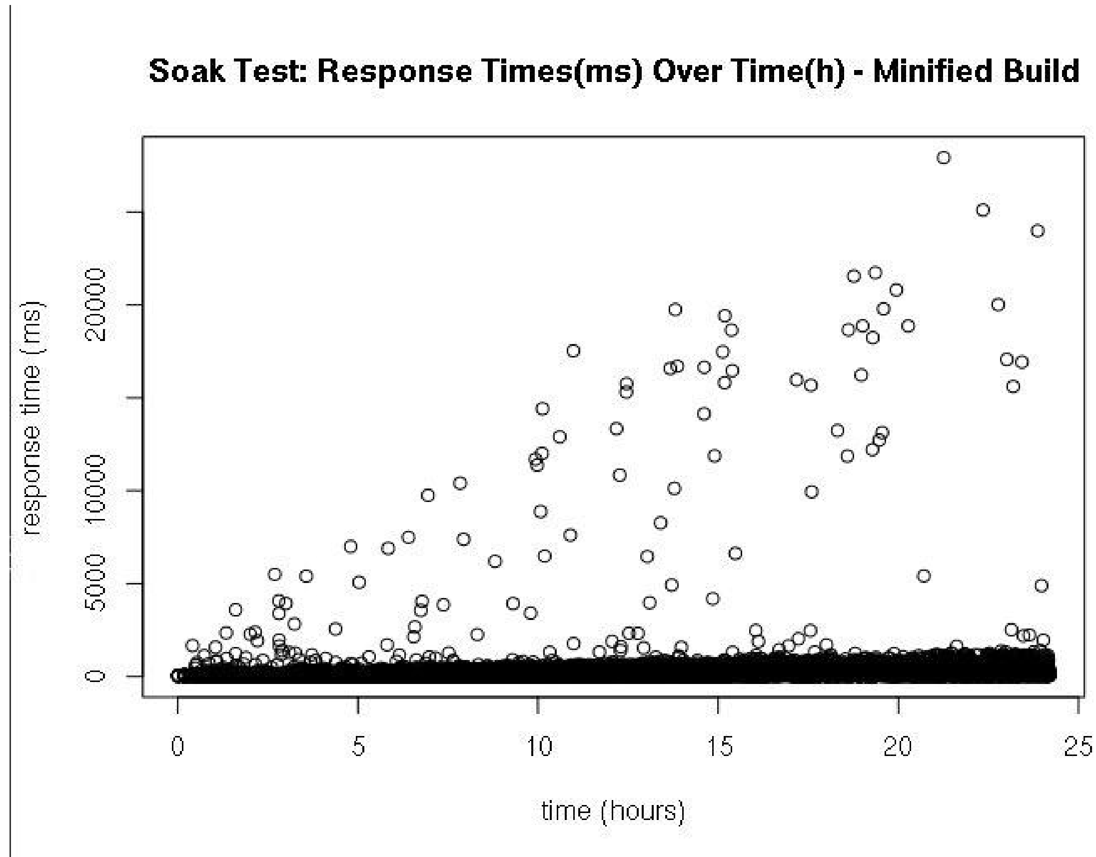
output from R, analyzing initial soak tests

Other requests, such as getting a band's basic information, were faster.

```
> mean(agg$X181[agg$make.an.account == 'get band info'])
[1] 1738.563
> mean(agg$X181[agg$make.an.account == 'get genre list'])
[1] 1930.506
> mean(agg$X181[agg$make.an.account == 'get manageBand partials'])
[1] 4157.634
> mean(agg$X181[agg$make.an.account == 'get addAlbum partials'])
[1] 4391.274
>
```

output from R, analyzing specific request types from initial soak tests

This was determined to be because of the number of files loaded due to the 'nav to the home' and 'nav to signup' requests. The resolution of this is discussed in "Minifying Files". Upon fixing these errors, the percentage of request errors fell below 0.5% and the average response time decreased substantially.



Soak test after debugging code and minifying scripts.

3. Minifying Files

The next technique applied was putting served script files through a process called minification. This is when function names are shortened to single letter names, and white space is eliminated. Although this adversely affects the readability of the code, it is usually a good idea because it reduces the amount of data sent. [6]

Minification also concatenates, or combines the script files into one large file. Concatenating files reduces the total overhead time by reducing the number of requests to make.[24] I used node-minify[19], which utilizes Google's Closure Compiler, or GCC, to minify and concatenate the files.

To test the throughput before and after minification, JMeter was run with 300 concurrent threads, making as many requests as possible for five minutes. This was repeated five times with the Initial and Minified builds.

Throughput Tests: Initial (1) against Minified (2)

alpha	0.05	X1 – x2	60.2185316
h0 statement	U1 = u2	o1	0.158276204
h1 statement	U1 != u2	o2	1.238838724
x1 (req/s)	17.21220155	O1 + o2	1.397114928
x2 (req/s)	77.43073316	sqrt(o1+o2)	1.181996162
s1 (req/s)	0.79138102	t	50.94646964
s2 (req/s)	6.19419362	df	5.005678253
N1 (#tests)	5	p	5.5073E-008
N2 (#tests)	5	result	h0 rejected

StressTests, average mean of initial tests against the average mean of minified tests.

Using Student's T Test [21], it was analyzed that, with >95% certainty, the throughput of Music Putty's home page increased. Initially, there was a sample mean (x) of 17.21 requests per second. After minifying the code, the sample mean increased to 77.4 requests per second.

To test response time, an estimated number of concurrent users during Music Putty's beta was determined to be 210 users. The team expected 100 registered users, 100 anonymous guests, and 10 bands to be using Music Putty simultaneously, at most, during the private beta. Once this was determined, load tests lasting approximately 15 minutes were executed.

Load Tests: Initial (1) against Minified (2)			
alpha	0.05	X1 – x2	771.3186998
h0 statement	U1 = u2	o1	0.16350733
h1 statement	U1 != u2	o2	0.013921407
x1 (ms)	837.4668539	O1 + o2	0.177428737
x2 (ms)	66.14815409	sqrt(o1+o2)	0.421222907
s1 (ms)	2037.46484	t	1831.141393
s2 (ms)	173.4607308	df	14566.46992
N1 (# samples)	12461	p	0
N2 (# samples)	12460	result	h0 rejected

Load Tests, average mean of initial response time against the average mean of minified response time.

I found with >95% probability that the population mean(u) of minified response times is significantly less than the population mean of the response times of initial tests.

4. Using a Static File Server

Node JS can be used as a simple HTTP server[20]. Dedicated, lightweight HTTP servers such as Nginx can potentially be used in place to Node JS to handle static requests, such as sending images, stylesheets, and scripts to the client. Node JS can then be dedicated to handling dynamic requests that require user validation, database interaction, and dynamically generated content.

This actually was inspired by an answer on StackOverflow, a site dedicated to answering computer science problems. A user named "m33lky" suggested Nginx as a solution for hosting static files, itself passing dynamic requests to Node JS[10]. Nginx is widely used HTTP server software focused on speed and simplicity[26].

I set Nginx to listen on another port than my Node JS application, and used Nginx as a proxy to the app when it doesn't catch a request for a static source.

Upon installing and setting up Nginx to handle all static file requests, I ran tests with my application's Minified build and compared these with the results from the original Minified tests.

```
16
17 http {
18     include       mime.types;
19     default_type  application/octet-stream;
20     client_max_body_size 50m;
21
22     sendfile        on;
23     keepalive_timeout  65;
24
25     server {
26         listen       3003;
27         server_name  localhost;
28
29         #charset koi8-r;
30
31         #access_log  logs/host.access.log  main;
32
33         location / {
34            proxy_pass http://localhost:3002;
35         }
36
37         location /javascripts/ {
38            root /home/mds/web/express/app/public;
39         }
40
41         location /angular/ {
42            root /home/mds/web/express/app/public;
43         }
44
45         location /img/ {
46            root /home/mds/web/express/app/public;
47         }
48
49         location /sylesheets/ {
50            root /home/mds/web/express/app/public;
51         }
```

Nginx configuration file

The same methods were used in these tests that were used for validating the minification of files in the previous section.

Load Tests: Minified (1) against Nginx-Static (2)			
alpha	0.05	X1 – x2	125.8360353
h0 statement	U1 = u2	o1	0.01573433
h1 statement	U1 != u2	o2	0.045577182
x1 (ms)	77.84775281	O1 + o2	0.061311512
x2 (ms)	203.6837881	sqrt(o1+o2)	0.247611616
s1 (ms)	196.0497578	t	508.1992412
s2 (ms)	567.8916857	df	20145.24512
N1 (# samples)	12460	p	0
N2 (# samples)	12460	result	h0 rejected

StressTests, average mean of initial tests against the average mean of minified tests.

Throughput Tests: Nginx-Static (1) against Minified (2)			
alpha	0.05	X1 – x2	70.03616723
h0 statement	U1 = u2	o1	0.054646975
h1 statement	U1 != u2	o2	1.238838724
x1 (req/s)	147.4669004	O1 + o2	1.293485698
x2 (req/s)	77.43073316	sqrt(o1+o2)	1.137315127
s1 (req/s)	0.109293949	t	61.58026527
s2 (req/s)	6.19419362	df	4.326996665
N1 (# tests)	2	p	4.1651E-007
N2 (# tests)	5	result	h0 rejected

Load Tests, average mean of initial response time against the average mean of minified response time.

It was determined with >95% probability that using the Nginx HTTP server along with Node JS on the same machine raised the response time and throughput. Remember, a higher response time is an undesirable quality. This is likely due to competing resources on the host

computer. It is useful to note that Nginx, when used on the same machine as a Node JS server, performed much worse at scaling. When running the Initial build with Nginx serving the static files, it frequently timed out at 120 seconds.

In implementing these fixes to our production server, I have decided to leave out implementing the Nginx server and proxy. At this time, we feel that it's more important to have a low response time than a high throughput. However, if this changes, we will know a possible method of improving the server.

OVERVIEW OF RESULTS

	Test	Initial	OS Setup	Debugging Code	Minifying Code	Results	Static File Server (rejected)
Throughput - homepage (req/s)	Stress	17	-	77		77 req/s	147.5
Uptime (hrs)	Soak	1.8	6+	6+	24+	lasts 24+ h	-
Response Time (ms)	Soak	9758	-	672.45		672.45 ms	-
Response Time (ms)	Load	837	-	66		66 ms	203

Key:
Requirements not met. Best result. Requirements met. Requirements partially met.

All of the requirements (50% increase in throughput, 50% decrease in response time, 12+ uptime) have been met.

FUTURE WORK

First of all, I will implement these changes to our production server. Knowing the benefits of the various implementations I have executed during this project, I believe Music Putty can greatly benefit. Over the next few weeks, I will be implementing these changes to production.

Additionally, I will explore the possibilities of testing for a maximum user load. This was attempted for this project, but was abandoned due to the testing environment set up. The host machine, due to memory limitations, could not produce a load with JMeter that could break the server. Therefore, a maximum user load could not be determined. A maximum user load is the number of users a server can serve while maintaining stability.

Doing online performance testing is another next step. Now that the server code itself is improved, I can explore the possibilities of improving our deployment setup. I plan on performance testing Music Putty's server using a cloud-based service such as Amazon's AWS. This costs more and has a higher initial setup, but is closer to our EC2 server than doing tests on a local machine.

SOCIETAL IMPACT / CONCLUSION

This senior project can be used as a resource for web developers with limited resources and experience wanting to test their Node JS server. I have detailed the usefulness of testing to improve current code, and explored the possibilities of changing a server's setup to increase its performance. These techniques can be applied to a variety of projects.

This senior project also ties into Music Putty's success. We feel that we are making a positive contribution to the world of music through empowering musicians and freeing them from record labels, while simultaneously providing listeners with free independent musician streaming. We have been talking about raising money to support rock artists in countries that condemn free expression through music, and in doing these kinds of activities hope to improve musicians' lives.

Sources Cited

1. *Apache JMeter - Apache JMeter™*. Apache Software Foundation, n.d. Web. 11 June 2014. <http://jmeter.apache.org/>.

2. "Apache JMeter - User's Manual: Component Reference." *Apache JMeter - User's Manual: Component Reference*. Apache Software Foundation, n.d. Web. 11 June 2014. <http://jmeter.apache.org/usermanual/component_reference.html#HTTP_Request>.

3. "Custom Plugins for Apache JMeter." *JMeter Plugins*. N.p., n.d. Web. 11 June 2014. <http://jmeter-plugins.org/>.

4. "Data." *TechTerms*. TechTerms, 2014. Web. 11 June 2014. <http://www.techterms.com/definition/data>.

5. Dewan, Prasun. "Synchronous vs Asynchronous." *COMP 242 Class Notes*. University of North Carolina, 2 Feb. 2006. Web. 11 June 2014. <http://www.cs.unc.edu/~dewan/242/s07/notes/ipc/node9.html>.

6. "Google/closure-compiler." *GitHub*. Google, 3 June 2014. Web. 11 June 2014. <https://github.com/google/closure-compiler>.

7. "How We Built EBay’s First Node.js Application." *EBay Tech Blog*. EBay, 17 May 2013. Web. 11 May 2014. <http://www.ebaytechblog.com/2013/05/17/how-we-built-ebays-first-node-js-application/#.U5iS5HXKqll>.

8. Krzyzanowski, Paul. "Introduction to Sockets Programming." *CS 417 Documents*. Rutgers, 2014. Web. 11 June 2014. <http://www.cs.rutgers.edu/~pxk/rutgers/notes/sockets/>.

9. *LibreOffice, The Do[23]cument Foundation.* 2014. Web. 11 June 2014. <http://www.libreoffice.org/>

10. M33lky. "Node.js Itself or Nginx Frontend for Serving Static Files?" *Stackoverflow*. Stack Overflow, 2 Apr. 2012. Web. 11 June 2014. <http://stackoverflow.com/questions/9967887/node-js-itself-or-nginx-frontend-for-serving-static-files>.

11. "Managing Your Files as Units." *Working with Https://pangea.stanford.edu/computing/unix/files/units.phpthe File System*. Stanford School of Earth Sciences, 3 Aug. 2004. Web. 11 June 2014. <https://pangea.stanford.edu/computing/unix/files/units.php>.

12. Martin; Roth. “Unit 12: Multithreading”. *CIS 501: Introduction to Computer Architecture.* University of Pennsylvania. Web. 10 May 2014. <http://www.cis.upenn.edu/~milom/cis501-Fall05/lectures/12_smt.pdf >.

13. Munch, Chris. "Effect of Website Speed on Users." *MunchWeb*. MunchWeb, 29 Sept. 2010. Web. 11 June 2014. <https://munchweb.com/effect-of-website-speed>.

14. Neilson, Jakob. “Response Times: The 3 Important Limits.” Nielson Norman Group: Jan 1. 1993. Web. 11 Jun. 2014. <http://www.nngroup.com/articles/response-times-3-important-limits/ >.

15. Paul, Ryan. "A Behind-the-scenes Look at LinkedIn's Mobile Engineering." *Ars Technica*. Condé Nast, 2 Oct. 2012. Web. 11 May 2014. <http://arstechnica.com/information-technology/2012/10/a-behind-the-scenes-look-at-linkedins-mobile-engineering/2/>.

16. Ramakanth, Kumar. “A Survey on Performance Testing Approaches of Web Application and Importance of WAN Simulation in Performance Testing”.

International Journal on Computer Science and Engineering. Vol 4 N.5, May 2012. Web. 11 Jun. 2014. <http://www.enggjournals.com/ijcse/doc/IJCSE12-04-05-159.pdf>.

17. Robertson, Aaron H. "Node.js Scalability Testing with EC2." *Caustiks Blog*. N.p., 6 Apr. 2012. Web. 11 June 2014. <http://blog.caustik.com/2012/04/06/node-js-scalability-testing-with-ec2/>.
18. "Soak Testing." *RPM Solutions*. RPM Solutions Pty Ltd, 4 Aug. 2004. Web. 11 June 2014. <http://www.loadtest.com.au/types_of_tests/soak_tests.htm>.
19. Srod. "Srod/node-minify." *GitHub*. Github, 19 Mar. 2014. Web. 11 June 2014. <https://github.com/srod/node-minify>.
20. Tilkov, S.; Vinoski, S., "Node.js: Using JavaScript to Build High-Performance Network Programs," *Internet Computing, IEEE* , vol.14, no.6, pp.80,83, Nov.-Dec. 2010 doi: 10.1109/MIC.2010.145
21. Uebersax, John. "Tests of 2 Sample Means". *Statistics 312.* Google Drive, 2014. Web. Accessed 11 Jun. 2014. <https://drive.google.com/file/d/164XM5UtSGfk06AXKlMn3r2WcPionVC59TfM7f4AKumRnxEzHwWkv4nMdb9EH/edit?usp=sharing>.
22. "Using Concurrent Pool Size - JMeter 2.5+." *Using Concurrent Pool Size*. BlazeMeter, n.d. Web. 11 June 2014. <http://community.blazemeter.com/knowledgebase/articles/64301-using-concurrent-pool-size-jmeter-2-5>.
23. Vahlas, Nico. "Some Thoughts on Stress Testing Web Applications with JMeter (Part 1)." *Nicolas Vahlas's Blog*. N.p., 17 Mar. 2010. Web. 11 June 2014. <http://nico.vahlas.eu/2010/03/17/some-thoughts-on-stress-testing-web-applications-with-jmeter-part-1/>.

24. Williams, Matt. "How Does Reducing JavaScript Requests & Minifying JavaScript Impact Site Performance?" *Yottaa*. Yottaa, 16 Jan. 2013. Web. 11 June 2014. <http://www.yottaa.com/blog/bid/259514/How-Does-Reducing-JavaScript-Requests-Minifying-JavaScript-Impact-Site-Performance>.

25. "Write(2)." *Linux Man Pages*. Die.net, n.d. Web. 11 June 2014. <http://linux.die.net/man/2/write>.

26. Zhu, Joshua. "Nginx Internals." *Slideshare*. Slideshare, 19 September 2009. Web. 11 June 2014. <http://www.slideshare.net/joshzhu/nginx-internals>.

27. Patel, S. (2014). Identification of Performance Improving Factors for Web Application by Performance Testing. *IJETAE*, 2(5), pp.433-436.