EMUES

Virtual queue system integration and optimization of an existing web platform

Bachelor's Thesis

By:

Jonas Holmström

Industrial Electrical Engineering
Faculty of Engineering, LTH, Lund University
SE-221 00 Lund, Sweden

LTH Ingenjörshögskolan vid Campus Helsingborg
Lunds universitet
Box 882
251 08 Helsingborg

LTH School of Engineering
Lund University
Box 882
SE-251 08 Helsingborg
Sweden

Tryckt i Sverige
Avd. för Industriell Elektroteknik och Automation
Lunds universitet
Lund 2016

Abstract

When a website has critical points in time when the amount of visitors increases significantly within a very short time frame, it is extremely important for the underlying system to be prepared to properly handle these large amount of requests. In Emues' case it's about more than just keeping the services running, but also to improve on the user experience while purchasing tickets for events. The objective of this bachelor thesis is to analyze the current system architecture and provide an extendable starting point where all of these issues have been improved upon and also to develop a virtual queue system that can help ease the burden on the system that comes from a sudden surge of visitors. The live system was successfully analyzed and the bottle-necks were found. A virtual queue system was also successfully implemented and tested with satisfying results. The queue automatically activates whenever a pre-determined load breakpoint is reached. The queue system was implemented using mainly Node, Redis and related open source technologies. A few suggestions for how Emues can improve upon their current system's hosting architecture were made.

Keywords: System load, Queue, Open source, Node, Redis, Automated

Sammanfattning

När en webbplats har kritiska tidpunkter då antalet besökare ökar väsentligt inom en mycket kort tid, är det oerhört viktigt för det underliggande systemet att vara beredd på att korrekt hantera dessa stora mängder av förfrågningar. I Emues fall handlar detta om mer än att bara hålla igång alla tjänster som körs, men också för att förbättra användarupplevelsen för användare som samtidigt köper biljetter till samma evenemang. Syftet med detta examensarbete är att analysera den nuvarande systemarkitekturen och ge en utbyggbar utgångspunkt där alla dessa områden har förbättrats och även att utveckla ett virtuellt kösystem som kan bidra till att lätta bördan på systemet som kommer från en plötslig våg av besökare. Det nuvarande systemet analyserades framgångsrikt och flaskhalsar hittades. Ett virtuellt kösystem som lätt kan integreras utvecklades och testades med ett tillfredsställande resultat. Kön aktiveras automatiskt när en förutbestämd brytningspunkt av belastning nås. Kösystemet utvecklades främst i Node, Redis och tillhörande teknologier med öppen källkod. Några förslag på hur Emues kan förbättra deras nuvarande systems arkitektur presenterades.

Nyckelord: Systembelastning, Kö, Öppen källkod, Node, Redis, Automatiserad

Acknowledgments

This Bachelor's thesis would not exist without the support and guidance of a few people listed below.

First of all, I wish to thank Magnus Jönsson for helping me find the subject of this thesis and also for providing me with his input in regards to the developed system's functionality.

Second, I want to give a big thank you to Emues' founder and CEO Markus Wiklander for his commitment to allowing me to carry out the project for Emues.

I also want to thank my supervisor Christian Nyberg and examiner Mats Lilja for their guidance.

Lastly, I want to thank the people at Load Impact for sponsoring me with their testing service which was a huge part of me being able to carry out this project.

Jonas Holmström

List of contents

Preface

In this thesis work, Jonas Holmström has been the only author and system developer with input from project supervisor Magnus Jönsson.

1. Introduction

In order to help the reading of this report, this introduction attempts to provide the reader with the base knowledge needed to understand each step and each decision that has been taken to carry out the full project. This first chapter contains the project's background, purpose, scope, the objectives that were set and a couple of questions that are important for the success of the project. In addition to this, there is also a part about the organization of this report and finally there is a short discussion about the source criticism used throughout the sources and citations.

1.1. Background and purpose

Emues is a company that primarily runs the concert platform Emues.com. It's a platform developed for users, artists and venues where anyone can suggest and market events and concerts. They have created a complete marketplace for concerts which includes both crowdfunded concerts and also the traditional ticket sale approach. While the platform at the current point in time is still in its growing stages, the community already consists of organizers, venues, artists and concert goers.

Emues currently has a ticket sales system in place, but the aspect of system performance and load handling is not something that has yet been taken into serious consideration. System load can increase dramatically during short time intervals when, for example, tickets to popular events become available. There is also no solution in place for how to help prevent concurrency problems during ticket sale processes that includes steps like seat reservation.

By using modern and open source technologies, Emues wants to find ways they can optimize and enhance the current system and identify future bottlenecks and areas where load problems can and will occur. This project was taken on with the objective to attempt to solve these problems, while keeping the user experience in mind.

Specifically, they want part of the solution to be by implementing a queue system for the platform.

1.2. Objectives

The main objective of the project is to improve the current system's performance and to provide Emues with a good foundation for how they can continue to improve their platform in this regard. Ticket sales of events with up to 15 000 max visitors have to be able to be marketed on the site while still allowing the system to continue operating without any serious interruptions or crashes. A modular virtual queue system will be developed that will help alleviate high loads on the main platform. The complete combined solution should be able to handle 10 000 virtual users which are spawned using a load tester. To reach these objectives a couple of tasks have to be carried out:

1.2.1. Analysis of existing platform

In order to be able to measure any improvements made by the work in this project, the underlying architecture of the system and its current performance needs to be known beforehand. Load tests of the current platform will be carried out using existing tools and their test results recorded. These results will be analyzed and presented in an easy to read format.

1.2.2. Development of a modular virtual queue system

A virtual queue system that is able to give a few users at a time access to certain parts of the platform will be developed. It should be made in a modular way so that it can be applied on the live system without any major modifications to the existing source code. It should also be easily extended and improved upon after the project has been carried out and delivered.

1.2.3. Architecture modification suggestions

The project should propose any additional changes to the architecture of the existing system that can improve the performance of the system as a whole. If time allows, the proposed changes will be implemented in a test environment identical to the live system and load tests will be carried out to measure the improvements over the existing solution.

1.3. Problem specification

A few additional points were raised as questions at issue. These are some core problems that will be faced during the project and answers to these questions must be found in order to achieve the best possible end result.

- The virtual queue will be passive/disabled most of the time and then automatically activate whenever it's needed. Which metrics are needed to be monitored to determine when the queue should activate itself?

- Some parts of the web platform could be more popular than others e.g. a ticket sales page for a large concert compared to a random user's profile page. How can less popular pages be excluded from the virtual queue?

- The virtual queue system should handle all incoming traffic, including API calls made from other external platforms such as Android or iOS apps. How can this be accomplished?

- What are some ways to enhance the user experience despite being stuck waiting in the queue?

- Which different queue techniques and methods will have to be considered? FIFO (First-in, First-out), LIFO (Last-in, First-out), Priority Queue etc.

1.4. Project scopes

Because of the time constraints put on the project, it's important that the scope is properly defined. Solutions explored in regards to the system architecture improvements will be constrained to use the same backend technologies already in use. Development of the virtual queue system will be primarily made using the Node.js technology and will only be tested while hosted in a Linux environment, more specifically on a droplet from the cloud server provider DigitalOcean (http://www.digitalocean.com).

2. Technical background

Many different technologies were used for developing the virtual queue system. This chapter will give a brief introduction to some of the various tools used throughout the project.

2.1. Node.js

Node.js was created by an American developer named Ryan Dahl. It was first presented at the conference JSConf EU year 2009. [1] Since its release there have been several versions and new updates are regularly being pushed out. As of today the current latest version is 6.2.1, while the current LTS version is 4.4.5. [2] For this project the latest version available at the time was always used.

Node.js is a JavaScript runtime built on Chrome's V8 JavaScript engine. [3] Simply put, this means that it provides you with a way to run JavaScript code on the server side as opposed to the usual way of running JavaScript which is in a web browser like Google Chrome or Mozilla Firefox. In Node.js it is primarily used for writing very fast and highly scalable web servers, especially for applications with some form of real-time components.

2.1.1. Example Node.js web server

The following code snippet is an example of one of the simplest web servers you can create in Node.js: [4]

```
const http = require('http');
const hostname = '127.0.0.1';
const port = 3000;

const server = http.createServer((req, res) => {
    res.statusCode = 200;
    res.setHeader('Content-Type', 'text/plain');
    res.end('Hello World\n');
});

server.listen(port, hostname, () => {
    console.log(`Server is running!`);
});
```

2.1.2. Express.js

While the above web server works, several frameworks have been introduced to help developers to more easily create advanced web applications. One of the most popular frameworks for Node.js, as seen on the amount of stars it has on Github, is called Express. Express is unopinionated which means it does not force you to develop your application in a certain manner but allows you to do it however you like instead. It does not do a whole lot on its own, but there are several key modules that allow you to work with all different kinds of aspects to web applications such as sessions, cookies, middlewares, template rendering, error handling and so forth. [5]

2.2. Redis

“Redis is an open source (BSD licensed), in-memory data structure store, used as database, cache and message broker.” [6]

Redis is a background service you can run on your server that gives you access to a very simple data store. Unless configured differently, Redis saves all data in memory which results in ultra-fast response times and less reliant on disk I/O. Redis supports a number of different data structures such as:

- Binary-safe strings
- Lists
- Sets
- Sorted sets
- Hashes
- Bit arrays/Bitmaps
- HyperLogLogs

The simplest example of how you can use a Redis store is with simple key-value string pairs, like in this very simple example:

```
“myKey” : “myValue”
```

Keys can contain any binary sequence, which means it can be anything from a simple string to the contents of an image file. This allows for immense flexibility, but it also puts more responsibility on the user to create its own schema and structure to follow.

In addition to being a data store, Redis can also be used as a message broker implementing the publish/subscribe (pub/sub) pattern. Messages can be published to channels without any knowledge of its receivers. Redis then automatically relays all messages to the subscribers of those channels. This can be very useful

for certain types of applications that require this type of information flow.

In order to interact with the Redis server, they provide you with a set of commands that you can use. Some examples of commands are ZADD to add or update members of a sorted set, ZRANGE to retrieve a range of members from a sorted set or FLUSHDB to remove all keys from the currently selected database. [7]

2.3. Reverse proxy

A reverse proxy can be described as an invisible middle man between clients from the internet and the web server they are trying to reach. By blocking public connections to the web server and only allowing the reverse proxy to connect to it, one can effectively hide the public existence of the web server and it's characteristics. When a client connects to the reverse proxy, the information is forwarded to the web server which then responds back to the proxy which in turn forwards the response to the client. The use cases for reverse proxies are many, here are some examples:

- Used as another layer of security.
- Add authentication to a backend service.
- Act as a load balancer
- Route endpoints to their respective internal services.

The following figure shows a simple flowchart of how it could be set up to work:

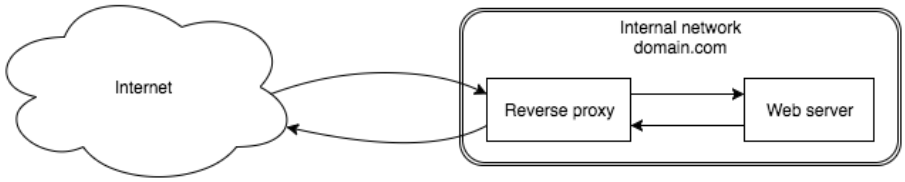


Figure 1. An example of a reverse proxy flow chart.

2.4. NGINX [engine x]

NGINX is an HTTP and reverse proxy server, a mail proxy server, and a generic TCP/UDP proxy server, originally written by Igor Sysoev. It is one of the most used web servers today and is used by large companies and organizations such as Netflix and Wordpress.com. [8]

3. Analysis

This chapter describes the different methods that were considered to be used for different parts of the project.

3.1. Test environment

In order to prove that the result of the project fulfills the objectives, there needs to be a way to reliably test the stability of a given system. A baseline has to be set, which is the user load that the existing system can handle before response times become unacceptable or even worse, the server crashes completely.

The first thing to consider is how to carry out these tests. Blasting the live server with virtual users until it crashes is not that great of an option. Instead a cloned environment of the live system will be created. Then any test can be carried out without affecting the performance of the live website.

To establish the baseline, a method to load test a system on demand must also be found. It turns out that simulating a high load of user requests is not that simple and requires some special tools. Two options were found and taken into consideration during the project.

3.1.1. Apache Jmeter

The first alternative to be considered was to use an application called Apache Jmeter. Unfortunately, running heavy tests with a lot of virtual users requires a very powerful machine. Luckily, if you have the resources. you can also use distributed tests using this tool, which means you can have a cluster of machines running the application as slaves, while one master machine instructs the slaves which actions they should perform. Result data is then collected, aggregated and displayed in the master machine. [9]

3.1.2. Load testing as a service

The second approach is to use a service that exists for just this purpose. An example of such a service is Load Impact (http://www.loadimpact.com). In their graphical interface you can set up tests which they then carry out for you and report any data you're interested in with nicely formatted graphs. The drawback of using services like these is that they are quite expensive and often targeted towards larger corporations. Load Impact were generous enough to offer their services for free to be used within this project since the tests would be made with an academic purpose. This was therefore the clear choice of load testing method that was chosen to be used. The only negative with this method is that the standard plan that was provided only supports a maximum of 1000 concurrent virtual users, but it will still give a good idea of how the system performs.

3.2. Development environment

The development environment will be kept as light-weight as possible. Most of the development will be done in the text editor SublimeText 3, since this is what I am already familiar with. Several plugins are used for things like modern syntax highlighting and auto-completion.

To keep the development environment as close to the production environment as possible, a tool called Vagrant was used to run and manage a virtual machine via VirtualBox. This allows you to run your code very quickly on a virtual machine hosted on the machine you are developing on that is almost identical to what the production server will look like. This saves a lot of time since you don't have to continuously update an external server with your changes every time you need to test the system.

4. Methodology

4.1. Development method

Being a single developer on this project, a full-fledged methodology such as Scrum is not ideal. Instead, a board similar to a Kanban board was used, however a bit more simplified. All tasks were created on the fly and those that were left to be carried out were put on the board and then moved to the right as they progressed throughout the project. For this a service called Trello (https://trello.com/) was used. For continuous communication with Emues, a channel on Slack was used.

For version control a simple Git repository hosted at Bitbucket (https://bitbucket.org/) was used.

4.2. Source criticism

Considering the tools and techniques used in this project, the references used are almost exclusively from online sources. When dealing with online sources, one has to be very meticulous when determining the credibility of those sources. All online sources used in this project were carefully researched and evaluated with the help of the following checklist: [10]

- Who is behind the material?
- Can you see who is responsible for the website?
- Who published the material?
- Was it an individual, organization, company or an institution?
- What do you know about the 'publisher'?
- Who is the originator – and what is their background?
- What is the professional level?

- Is the information documented in the form of references or similar?
- Are there other sources that are in agreement?
- What is the objective of the website/page?
- Are there special interests behind the individual/organization who wrote the website?
- Which interests can control their selection of and perspective on the material?
- Does the originator have an e-mail address or a contact address?
- When was the website/page last updated?

5. Pre-development

5.1. Planning

Before starting the development of the virtual queue system, a list of features and suggestions to different ways they could be implemented was produced. This was done through brainstorming in collaboration with the project supervisor Magnus Jönsson and Emues' Founder & CEO Markus Wiklander. Some of the discussed features are reflected in the problem specification in chapter 1.3. These features were then prioritized and corresponding tasks were created. Out of these features, a few were selected to be "must-have" while others were labelled as "nice-to-have" if time allows.

The main features were the following:

Must have

- The queue system must be able to differentiate between URLs and only queue the most visited pages.
- The queue system should be able to be activated and deactivated manually and automatically.
- The queue system must be compatible with the current platform.
- After gaining access, a user will lose access again after a certain time.

Nice to have

- A user will lose access after purchasing tickets for an event he or she was queued for.
- The queue should be able to act as a load balancer.
- Admin interface with controls and real-time statistics over the queue system.

5.2. Setting up the test environment

5.2.1. Current live architecture

The existing system architecture is implemented according to figure 2, found below.

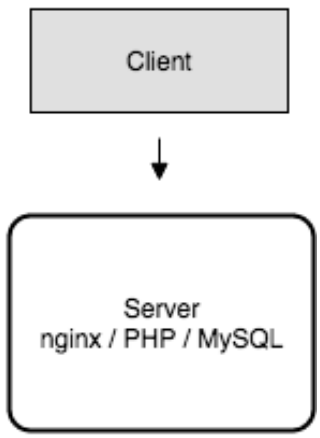


Figure 2. Existing system architecture

The live server is currently hosted as a droplet on DigitalOcean. A droplet is just DigitalOcean's definition of a virtual private server. [11] This single droplet hosts everything related to the current Emues platform such as NGINX, PHP, MySQL and also an Elasticsearch server.

5.2.2. Laravel Forge

Laravel Forge is a service for provisioning and managing cloud servers hosted on either AWS, DigitalOcean, Linode or even your own custom servers. You can also manage other related things such

as SSL certificates, domains, Supervisor, Cron jobs, Load balancing, SSH keys and horizontal scaling. A server can be linked to a branch on a Git repository and either deploy new changes manually or automatically when commits are pushed to the repository. [12]

5.2.3. Test architecture

Since it's very important that the test environment is as similar to the live environment as possible, a clone of the live droplet was created on DigitalOcean and then linked to an account on Laravel Forge. For security reasons, NGINX was configured to only allow certain users to access it. With this new droplet set up there was now a good starting point to begin initial load testing and system analysis.

6. Initial test phase

6.1. Performance testing

Performance testing is the overall name for investigating the speed, scalability and stability of a product. It consists of different types of tests. In the context of this project it's important to understand the differences between the different types of performance tests and which of them that are relevant to the scope of this project. This chapter will briefly touch on the different types of performance tests. [13]

6.1.1. Load testing

Load tests are tests which are designed to investigate how your application performs under normal to peak load conditions. This means that these tests are crucial to determine if an application can meet the performance goals that have been set as the minimum accepted level assuming you want your application to work during peak load times. Endurance testing is a certain type of load tests that are conducted to verify how an application performs during extended periods of time. [13]

6.1.2. Stress testing

Stress testing is a type of test that are conducted to investigate how an application behaves when the load conditions go beyond normal or peak levels. This testing is very useful for finding bugs or errors that only occur during higher than anticipated load levels. Spike testing is a type of stress testing that involves repeatedly pushing an application beyond its limits for short periods of time. [13]

6.1.3. Capacity testing

Capacity tests are used to investigate how many users or transactions that a system can support while still remaining within the performance goals of the application. This testing strategy is well suited for the planning of future growth. You can for example anticipate which resource increases could be needed in the future, such as network bandwidth and hardware scaling. [13]

6.1.4. Smoke test

A smoke test is a preliminary test that is often used to discover critical errors in a system. [14] It is used early in the testing phase to test basic features and to make sure that the system is not broken enough to render further testing unnecessary. It is sometimes also referred to as a confidence test and/or a sanity test. [15]

6.2. Load Impact

To satisfy all their customer's needs, Load Impact offers several advanced options for performance testing web applications. In order for the tests to provide useful information, we need to understand the configuration of the conducted tests. In this chapter the Load Impact test configuration for this project is briefly discussed as well as some of the terms used in their tests.

6.2.1. Virtual users

A virtual user is, as the name implies, the term used to describe each individual simulated user that is created to perform user scenarios.

6.2.2. User scenarios

User scenarios are scripts written in the Lua language to instruct each virtual user what it should do during the test. You can for example tell the virtual users which pages they should visit, how frequent they should visit them and make them take different actions depending on the result of other actions. This makes the tests conducted with Load Impact extremely versatile as you can instruct each virtual user to do almost anything. They provide you with a collection of functions, methods and associated documentation to access all of this functionality. Below is an example of what a basic user scenario might look like:

```
http.page_start("My page")
responses = http.request_batch({
    { "GET", "http://loadimpact.com/" },
    { "GET", "http://loadimpact.com/style1.css" },
    { "GET", "http://loadimpact.com/image1.jpg" },
    { "GET", "http://loadimpact.com/image2.jpg" }
})
http.page_end("My page")
client.sleep(math.random(20, 40))
```

The first row marks the start of a metric measurement, in this case it is named “My page”. After that, the virtual user requests the four resources listed and saves them in the responses variable. When that is finished, the script marks the end of the “My page” measurement and that result is recorded in the test result. Finally, the client sleeps for a random amount of seconds in the interval [20, 40]. In addition to being able to script your own user scenarios, there are also browser extensions that can help you record scenarios while you’re carrying them out manually in the browser.

6.2.3. Project test configuration

When determining the user scenario for this project we came to the conclusion that it is sufficient to make all virtual users load the same URL since the load on the server will be the same as if they all loaded different URLs. However, it is important that the URL chosen is one that is fairly heavy for the server to generate to make sure the tests don't give unrealistic results. One page that matches this criterion is any specific event page. In the case for this project the user scenario also doesn't have to be very advanced, a simple scenario that loads the page and reports back the result is completely fine. This is the user scenario used for this initial test that will determine what the current maximum capacity is (some long URLs have been shortened for readability, this is just to give an idea of what it looked like):

```
http.page_start("Page 1")
http.request_batch({
    {"GET", "http://dev.emues.com/event/king-dude-true-moon-babel"},
})
http.request_batch({
    {"GET", "http://dev.emues.com/css/all.css?c=1"},
    {"GET", "http://dev.emues.com/js/libs.js"},
    {"GET", "http://dev.emues.com/js/app.js"},
    {"GET", "http://dev.emues.com/media/profile/avatar/0.jpg"},
    {"GET", "http://dev.emues.com/media/avatar/avatar/1.jpg"},
    {"GET", "http://dev.emues.com/media/avatar/avatar/2.jpg"},
    {"GET", "http://dev.emues.com/media/avatar/avatar/3.jpg"},
    {"GET", "http://dev.emues.com/media/avatar/avatar/4.jpg"},
    {"GET", "http://dev.emues.com/media/avatar/default/avatar.png"},
})
http.page_end("Page 1")
client.sleep(math.random(20, 40))
```

Note that all the assets and resources such as the CSS file, JavaScript file and images must be explicitly included in the script. You can't just put the site's URL in there and be satisfied with that. Fortunately, there is a fast way to generate these simple user scenarios

using their quick scenario feature where all you have to do it put in the main URL you want to test and Load Impact will find all the assets that the page requires and add those to the script automatically.

6.3. Live system smoke test

6.3.1. Test specification

Number of VUs	**Duration**
1000	5 minutes

6.3.2. Test results

Blue line is number of active VUs
Green line is average response time

Figure 3. Test results from the first load test

All static assets had an average response time of ***54.34 ms*** while the main PHP generated page had an average response time of ***29.42s***. All static files were returned successfully from the server. The main PHP route experienced a fairly high failure rate as depicted in the table below:

Total requests	Successful requests	Success ratio
4819	278	5.77%

The failures mainly consisted of the two errors:

504 Gateway Timeout and **502 Bad Gateway**

6.3.3. Test conclusion

As shown in the graph, the server becomes unstable very quickly. It only requires around 200 concurrent requests for the response time to increase exponentially and for most requests to fail completely. The reason that the response time is actually decreasing when the server load becomes higher, is because at this point almost every single request to the server returns an error. To understand what these errors mean you have to also understand how a PHP request is handled by the server. Every request that the server receives goes through the HTTP server NGINX. Depending on what is requested and how the server is configured, this request is then handled in different ways. For static resources such as images, CSS and JavaScript files, the files are simply returned by NGINX itself. But when a client requests a PHP page, it works a bit different. PHP is a server side scripting language that needs to be run and evaluated before anything can be returned from the server to the client. NGINX cannot do this on its own, but instead forwards the request to a process called PHP-FPM where FPM stands for FastCGI Process Manager. This is a process on the server that can run the PHP script and return its result to NGINX which in turn returns that result to the client.

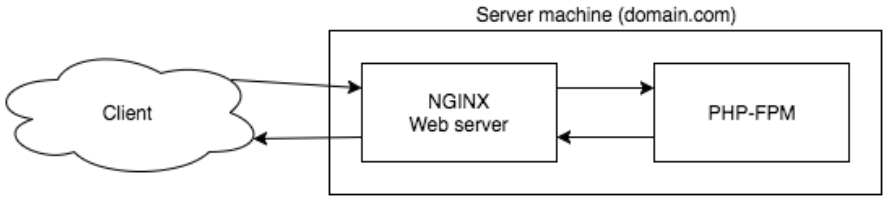


Figure 4. Basic PHP request flow

What these gateway errors essentially mean, is that the server does not have the capacity to spawn new PHP processes for all the incoming requests. NGINX does therefore not get a result back from PHP-FPM which results in NGINX returning these errors codes instead of the result that the client is expecting. This is a very simplified explanation but should give an idea of what is happening.

With these results there is a baseline for how the current system performs under high loads.

7. Virtual queue system development phase

This chapter describes the development phase of the virtual queue system. First there is some initial thoughts about the developed system and what the vision was for how it should work. Some of the steps taken during development are described and a few of the chosen implementation solutions are presented. Another already existing similar product to what was developed is also briefly compared.

7.1. Request flow overview

The first thing that was produced was a sketch of how the overall request flow of the application was supposed to work (or at least how it could work).

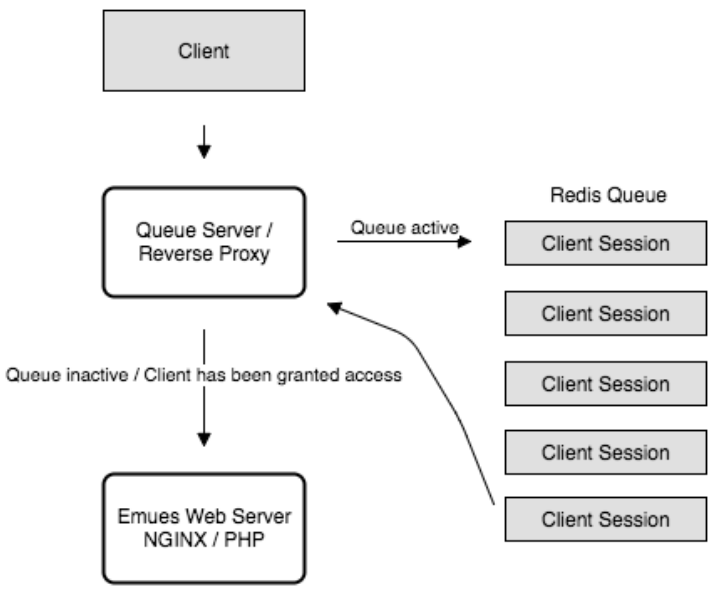


Figure 5. Request flow overview with the queue system in place.

7.2. Code style, guidelines and linting

To make sure the project stays reasonably maintainable and consistent across all files, a few rules and guidelines were made that should be followed when implementing the system.

- Use ECMAScript 2015 (ES6) features wherever possible.
- Indent using 4 spaces.
- Omit semicolons.
- Use Promises on all asynchronous operations.
- Every function and method must have a comment explaining what its purpose is.
- All variable and function names should be written in camel case (camelCase).

7.2.1. ECMAScript 2015 (ES6)

ECMAScript is a script-language standard specification which was first based on JavaScript. It defines what functionality a script language that follows it should have. ECMAScript 2015 specifically introduced additional features which can be very helpful compared to the older version of the specification. An overview over the new features introduced can at the time of writing this be found at http://es6-features.org/.

7.2.2. Promises

The usual way of handling asynchronous functions in JavaScript is to provide a callback function that is called whenever the asynchronous action is completed. See the following example:

```
myModule.asyncAction(function () {
    console.log('Async action is now completed!')
})
```

This works fine for a simple example like this. But what if you want to perform another asynchronous action whenever the first one is completed, and then another one after that? It quickly becomes very hard to follow the flow:

```
myModule.asyncAction(function () {
    myModule.secondAsyncAction(function () {
        myModule.thirdAsyncAction(function () {
            console.log('My async actions are now done!')
        })
    })
})
```

What Promises does, is it allows you to write the above code in a way that is much easier to follow:

```
myModule.asyncAction()
    .then(myModule.secondAsyncAction)
    .then(myModule.thirdAsyncAction)
    .done(() => console.log('My async actions are now done!'))
```

There are many different implementations of Promises but for this project a well-known package called Bluebird was used.

7.2.3. Code linting with ESLint

In order to follow some of the rules set in regards to code style, a linting utility called ESLint was used. It checks the source code files that you create and alerts you of any code style errors it can find. ESLint is configured using a *.eslintrc* file which contains the rules you wish to follow. This file is written in JSON format. With the help

of some plugins you can then get it integrated into most editors. The configuration file used in this project can be seen in Appendix B.

7.3. Web server implementation

The first step in the development phase was to get Express imported and integrated into the project. This was done first to have a good foundation to build upon for the rest of the development phase. A catch-all route was configured so that every single request that is received goes through the same route. This makes it easy to evaluate what the requested URL was and to request that URL specifically via the reverse proxy module.

Additional Express modules that were used were the following:

- **Express-session** was used to get access to sessions within the express route.
- **Express-handlebars** was used as the template engine.

7.3.1. Session cookies

In short, a session cookie is a way for a website to remember who a user is. Each user is given a unique session identification number that is saved as a cookie in their browser. This is required to keep track of which position that a specific user has in the queue.

7.3.2. Template engine

Basic HTML files are not very flexible when it comes to displaying dynamic content. In order to programmatically generate HTML responses one can use what is called a template engine. It usually provides a different syntax than normal HTML and it doesn't really matter which one you use. It mostly comes down to personal preference. You create templates that you can then feed with variables to make each page different depending on the values of the

variables you provide. The templates are rendered on the server and returned as pure HTML to the requesting client.

7.4. Queue implementation, phase 1

The next part to be developed was the actual queue implementation. The idea from the start was to use an in-memory data structure such as a regular array for the queue. That worked fine in the beginning, however, in order to make the system a lot more modular and scalable that idea was scrapped and instead the whole queue was implemented using Redis and its data structure SortedSet.

An entry into a SortedSet takes two values. The actual value you wish to store, and a floating point number that is that value's score. All entries that are added into a SortedSet are automatically sorted on insertion with a time complexity of O(log(N)) based on the score provided. Since all values are then already always sorted, fetching sorted entries from a SortedSet is extremely fast and efficient.

The Redis commands used can be seen in appendix A.

7.4.1. Putting users in the queue

The queue system takes advantage of the SortedSet data structure to store every user session with the session ID as the value and then the current Unix time in milliseconds as the score. This way, user sessions are always stored in the correct order in Redis' memory and the actual queue is then given almost for free. All the system has to do is add each user's session ID as they visit the site for the first time, along with the current time. Should two users have the exact same score, SortedSet falls back to sorting on the value, so they will just be sorted based on their session ID instead. This is all assuming the queue is active of course, otherwise the whole queue step is skipped and every user is granted access immediately.

The drawback of this implementation is that it becomes harder to implement special queue features such as VIP priority. You could do

it by just giving each VIP user a score of zero, in which case they will always be first in the queue, but for more advanced applications it becomes more complex.

To communicate with the Redis process from Node.js, a JavaScript module that is simply called *redis* was used. It is more or less just an interface to Redis' commands in JavaScript. This module is however not implemented using Promises which would make using it and sticking to the code guidelines a bit difficult. Thankfully, Bluebird is able to convert entire modules into being Promise based, which worked fine in this case.

A code example for adding users to the queue can be seen in appendix D.

Whenever a user tries to access a route that is currently queued, they are shown a page where they can see their current position in the queue. This page automatically refreshes to update the queue status.

7.4.2. Granting access to users

To grant access to users, a very similar approach to how the queue works was used. There is a separate Redis store that is also a SortedSet that is used for storing the session IDs of users that have access to the system. This store is referred to as the access list. When the queue system is active, a separate process is running which periodically checks if there is more room in the access list and if so, it takes as many as can fit from the front of the queue and moves them over to the access list. The score is changed from whatever Unix time that user had, to a new value which is the Unix time value of a time in the future. For example, if a user is granted access at 12:00, the time that gets set as the score could be 12:10, 10 minutes in the future. These 10 minutes are how long that user has access to the system.

At the same time, there's another process running that automatically purges the access list from entries that have expired. So at 12:10, the example user will be removed from the access list and have its access revoked. If the user visits the site again, he or she will

be put in the back of the queue and will have to wait to get access for a second time.

With this system architecture it is very easy to tweak how long users have access, either on an individual level or globally. For testing and debugging purposes, the time could be set to something low as 5-10 seconds just to rotate users quickly.

A code example for granting access to users by putting them in the access list can be seen in Appendix E.

7.5. Half-way smoke test

At this point in the development phase, it was decided that it was a good idea to make sure the performance holds up to the goals set in the objectives of the project. It would be a waste of time to continue on this development path if the performance on the current system was already an issue. An identical test to the one that was used to analyze the live system was performed on the queue system that was developed so far.

7.5.1. Test specification

Number of VUs	**Duration**
1000	5 minutes

7.5.2. Test results

Blue line is number of active VUs
Green line is average response time

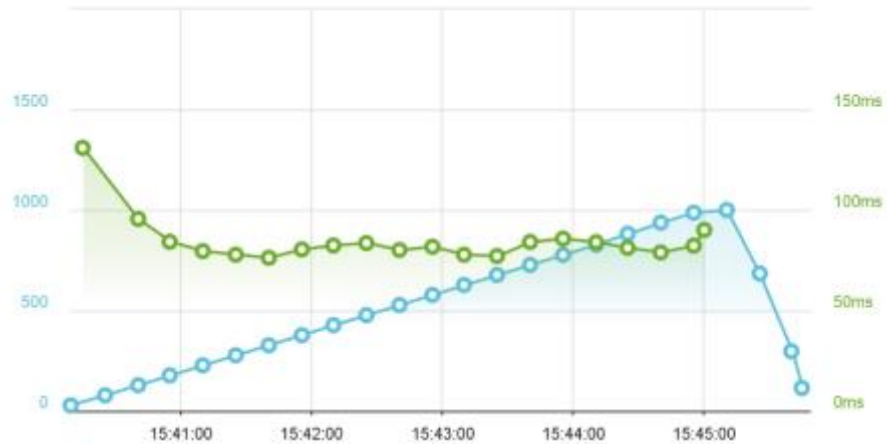


Figure 6. Results from testing the first queue implementation.

The average response time was ***84.19 ms*** and the request result rate can be seen in the table below.

Total requests	**Successful requests**	**Success ratio**
5657	5657	100%

In addition to these metrics, all requests were successfully placed in the queue and granted access whenever there was more room in the access list.

7.5.3. Test conclusion

The test results were above expectations. The response times were very fast and it was exceedingly satisfying to see that the queue implementation was working properly. This provides a good foundation for continued development of the remaining features.

7.6. Queue implementation, phase 2

7.6.1. Separating routes

One of the requirements set in the pre-development planning phase was that specific popular routes, for example the routes related to booking tickets for a big concert, should be able to be queued while other less popular routes can bypass the queue completely. This is important to make sure that the overall user experience on the website is not diminished just because certain parts of the website are more popular than others.

7.6.2. Activating the queue

In order to activate the queue, there are a couple of different metrics that can be monitored. Some alternatives are server CPU usage, memory usage or requests per second. The best solution probably should combine all of these metrics, but for this project it was decided that requests per second was a good starting point. This is because it is very easy to measure which requests per second value the server starts to perform badly at. This also makes the queue very flexible since this value can be very easily changed whenever it is needed.

7.6.3. Measuring requests per second

To measure this metric, a node module that is simply called ‘measured’ was used. Measured is an open source module that can be found on Github at https://github.com/felixge/node-measured.

This module provides something they call a Meter which can be used to measure the frequency of arbitrary events. The statistics can then be output in current rate and the rates for the last 1, 5 and 15 minutes. To avoid some spiky behavior, it was decided that the rate for the last minute should be used instead of the current rate. This metric is then monitored for each different URL that is requested so that it is possible to see which different URLs are the most popular. In this first implementation, whenever an URL reaches 20 requests per second, that route is queued up and whenever a user tries to access that URL, they will have to wait until there is room available in the access list. An example of the data-structure used for these metrics can be seen in appendix C.

7.6.4. Deploying the system

A droplet with 2 CPU cores and 2 GB RAM was chosen as the starting server that should host the system. Note that this is a droplet that costs only $20 / month and for big systems that is very cheap. Since Node is single-threaded, that means it can only utilize a single CPU core. The common way to run Node applications across all CPU cores is to run a separate instance of the application on each core. This is called a cluster. Using a process manager such as PM2 (http://pm2.keymetrics.io/) makes it very easy to launch an application in cluster mode. PM2 then automatically load balances all incoming requests to the different processes. It is important to remember that each Node process has its own scope and cannot share variables. Since this system uses Redis to store its state, this is not a problem because there is still only a single Redis process.

7.7. Final test introduction

The final system that was developed needs to be tested under higher loads than what have been done in previous tests. The account that was used for Load Impact only supports unlimited tests with 1000 maximum number of virtual users, but the objective goal was to be able to handle 10000 concurrent users. This brings us to a discussion about what concurrent users actually mean. If two systems have a certain same amount of concurrent users, that does not mean the load on the two systems is the same. What actually matters are the amount of requests that are made to the server and how much resources each request requires. For example, a system with 1000 concurrent users that only make a single request each in a certain time interval is a lot different to a system with 1000 users where each user makes 10 requests each in the same time interval. For this project the goal will be to try and reach the minimum, which is when 10000 users perform a single request each.

The tests performed so far have only been using 1000 virtual users. The way the tests work is that they ramp up the number of users slowly and in each step every active user performs the user scenario. Since there was a limit of 1000 users, the final test was made to simulate that each user acts as if it was 10 users. This was done by altering the user scenario for the test, and forced each user to make 10 requests each instead of just the one it did before. This severely increased the load generated by the test.

7.8. Final test

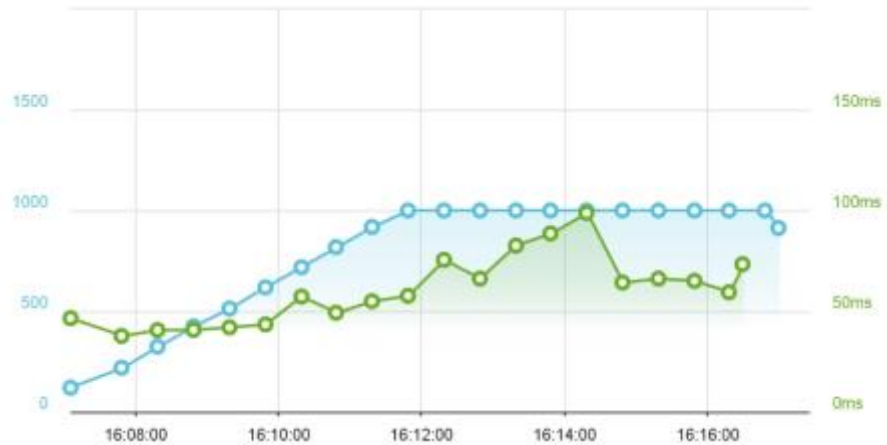
7.8.1. Test specification

Number of VUs	**Duration**
1000, simulated as 10000.	10 minutes

In addition to the much higher load, this test also ran for an extra 5 minutes with full load in order to ensure stability.

7.8.2. Test results

Blue line is number of active VUs
Green line is average response time



The average response time was ***62.43 ms*** and the request result rate can be seen in the table below.

Total requests	**Successful requests**	**Success ratio**
152994	152994	100%

7.8.3. Test conclusion

The graph shown can be a bit misleading because of the scale of the green line. Despite the line for the average response time in the graph not being completely straight, the results were very successful. Having a 100% success ratio and an average response time as low as 109.65 ms is very good considering the price of the droplet and the system being a first implementation prototype. These results and also the capacity can likely be improved considerably by increasing the server hardware to use a more powerful CPU with increased amount of cores. Then the amount of processes could be further incremented. All in all, the test was considered successful.

Some other statistics related to this test that was gathered by observation:

- **Max request rate:** 308 req/s
- **Max bandwidth usage:** ~ 9 MB/s

8. Similar product: Queue-It

A similar product that already exists on the market today is a product from a company known as "Queue-It". They provide their queue system as a SaaS and have provided their services to companies such as Toys R us, Telenor, Telia and many more. The main difference between their system and the one developed in this project is that their system is not DNS-based. This means they do not utilize proxies to redirect traffic to the end site. Their system is developed in .NET / C# as opposed to Node.js that was used in this project. In order to measure the site load on their customers site to know when the queue needs to be activated, they have developed a feature they call SafetyNet. This can be integrated into a user's site by adding a small JavaScript to their pages. This is unnecessary in this project's implementation since all traffic is routed through the queue which means the queue itself can measure the amount of traffic that goes through it.

Queue-it can be found on https://queue-it.com/.

9. System architecture improvements

This chapter discusses some of the improvements that could be made to Emues' current system architecture in order to improve performance and ensure future stability. This is just an overview over a few techniques and methods that could be used. Most of these improvements does of course come at a price, most notably increased cost and complexity. All of the suggestions are based on research and they were not discussed in-depth with Emues nor were they implemented and tested for this particular project.

9.1. Separation of concerns

Hosting all your services on a single machine works fine as long as the load on the server is relatively small. Once your average user count goes up however, it can be very CPU-intensive. One way to improve on this is to split your services up over multiple servers. It could for example be set up like in figure 7.

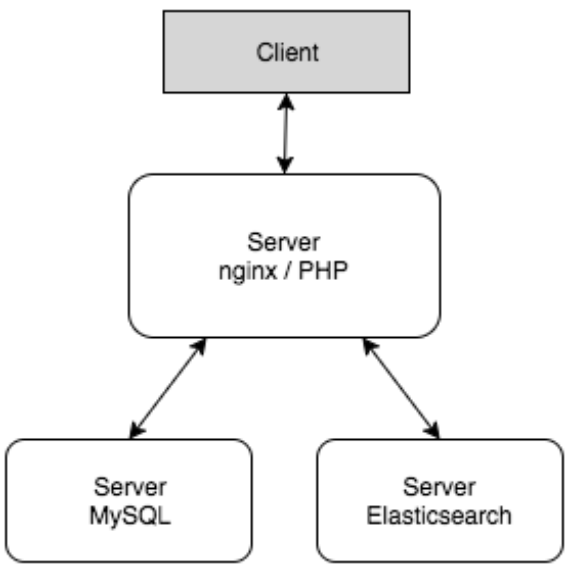


Figure 7. Server architecture with services separated.

9.2. Load balancing

Separating your services into dedicated servers is a good starting point for improving a system's performance. Another step that can be taken is to have multiple servers whose purpose is the same, and then you spread out the work evenly between them. This is often referred to as horizontal scaling and can be done by using a load balancer. NGINX has load balancing built in and can be set up very easily. If we apply load balancing to the architecture shown in figure 7, we can achieve a setup like the one shown in figure 8.

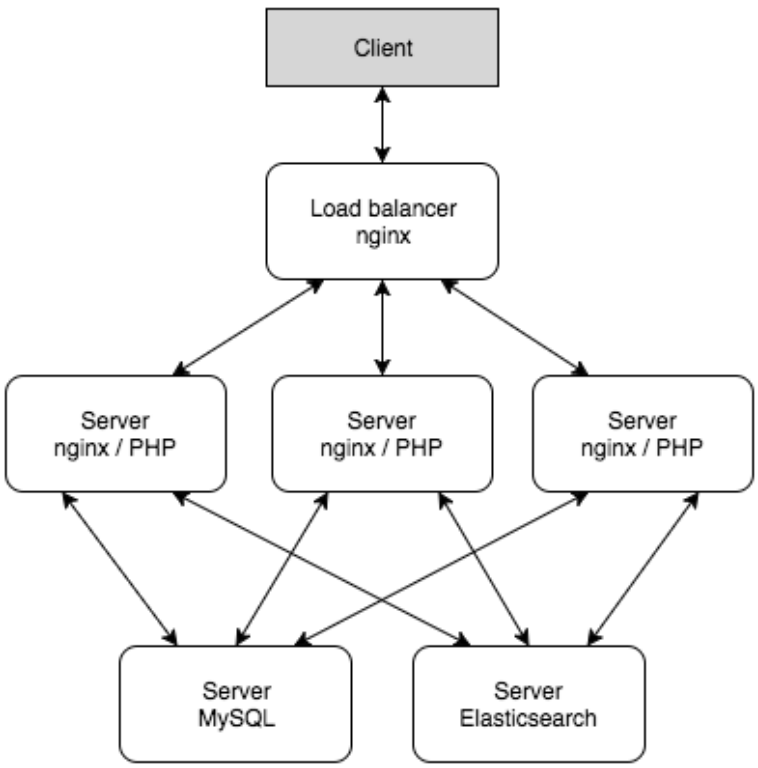


Figure 8. Server architecture with load balancing.

Certain services such as AWS provides very simple to set up solutions for load balancing so that you don't have to do all this configuration on your own. In AWS' case it's called Elastic Load Balancing. [16]

One thing you have to keep in mind when load balancing servers like this is if your website supports uploading of files, for example if users of the website can upload their own pictures. Usually when having this feature, the uploaded file is stored on the server that is processing the request. If you have load balanced servers however, they all need to have access to these uploaded files. If only the server that processes the request has the file, then that's the only server with access to that file. The easiest way to solve this is to have a dedicated server for file resources where all uploaded files are stored.

9.3. Database replication

Just as the web servers, the database server can also be load balanced. The problem with databases however, is that you need to make sure that the data between them is synchronized and consistent. Doing this yourself can be quite difficult but there are services out there that can do it for you. One such service is also provided by Amazon in AWS RDS (Relational Database Services). There you can set up read replicas that are automatically synchronized with the main source database. [17] [18]

9.4. Automated scaling

The methods described in chapter 9.2 and 9.3 are often combined with automated scaling. Automated scaling means that the amount of servers that are load balanced can be increased and decreased depending on the current load on the system. By doing this, your system can support virtually unlimited amount of users while still keeping the amount of servers down when the load is small.

10. Results

In the initial testing phase, the live system was successfully investigated and tested to find its current breaking point. The errors found were analyzed to figure out what the root of the problems were. Continuing into the development phase of the virtual queue system, all of the "must-have" requirements that were created in the planning phase were successfully implemented. Unfortunately, due to time constraints, none of the "nice-to-have" features were developed. Aside from the hosting solution from DigitalOcean, the system only makes use of technologies that are open source and available to everyone at no cost. The code guidelines that were set up were followed and the virtual queue system is very easy to integrate with the current live system that is being used at Emues.com. During and after development, the queue system was thoroughly tested to ensure that there were no apparent errors, performance issues or stability concerns when trying to meet the objectives that were set up in the start of the project.

11. Conclusions

The first problem introduced in the problem specification was about how the queue could automatically activate depending on certain metrics. The metric that was chosen was the amount of requests per second averaged over the last minute. This proved to be a good enough starting point, even though other metrics could also have been used. By measuring this value for each individual route in the Emues application, the problem of only queueing popular routes was also solved. Because of this, certain routes like API calls can also be queued individually since all traffic passes through the queue system.

To make sure the user is aware of what's going on, their current position in the queue is shown to them on the queue page. This is to make sure they have some sort of idea about when they will be let through. This should make the user experience bearable, even though being stuck in a queue is not very entertaining. Other means to improve the user experience could be to try and calculate how much time a user has left to wait, based on how fast the queue has been moving.

The only queue type that seemed appropriate for this kind of system was a regular FIFO queue. The only difference is that we are able to put users at the front of the queue so that VIP customers don't have to wait.

In addition to the developed solution being easily integrated with the intended web platform at Emues.com, the virtual queue system should also be easily integrated with any other web platform that is hosted on a single server. Considering the limited amount of similar solutions on the market, this product therefore has the potential to become commercially viable if further development is carried out. It is very easily adaptable to any system size; all you need to know is the breakpoint for when the current system cannot handle any more requests.

Using a system like this, it is possible to preserve a good user experience, even when a web platform has a user amount that is beyond its limits. Being greeted by a queue that displays your position and an estimated wait time is not particularly fun for a user, however, it certainly beats just retrieving an error message and have no idea when or if you will ever be able to access the resource.

The biggest drawback I can think of is that the system currently separates all routes from each other. What this means is for example that the route for viewing an event may be queued, while the route for booking tickets for that event is not. Users can then bypass the queue by manually going to the URL for ticket booking, while users who doesn't know about this gets stuck in the queue. Ideally all routes that belong together, like everything related to a specific event, should be grouped up and queued together.

Another drawback is that it is not that straight-forward to set up as it requires you to set up an entirely new server, deploy the queue system to it and redirect your domains to the new queue server. Setting up the server alone takes some time since you need to configure all the different services that are required for the queue system to function properly. This is not an issue if you are familiar with managing a Linux server however. There are also plenty of guides and tools that can help you with this.

12. Future Work

The currently developed system should very much be viewed as a proof of concept and a prototype. There is an endless amount of features that could be implemented to enhance the currently developed system. For starters, all of the "nice-to-have" features that were left out due to time constraints would increase the quality a lot. Many other things could also be done to increase the user experience for the queue administrators. Having a user interface where routes can be included or excluded from the queue manually, changing the breakpoints on the fly, grouping certain routes together, showing statistics and more.

Then of course there is code refactoring and most likely also a lot of optimizations that could be done to increase the performance of the system. More research could be done into different load balancing and automated scaling strategies and these could be tested and compared in practice. The whole server management part is a rabbit hole that never ends.

Bibliography

[1] R. Dahl, "Ryan Dahl: Original Node.js presentation," 8 11 2009. [Online]. Available: https://www.youtube.com/watch?v=ztspvPYybIY. [Accessed 7 6 2016].

[2] Node.js Foundation, "Previous releases | Node.js," [Online]. Available: https://nodejs.org/en/download/releases/. [Accessed 7 6 2016].

[3] Node.js Foundation, "Node.js," [Online]. Available: https://nodejs.org/en/. [Accessed 7 6 2016].

[4] Node.js Foundation, "About | Node.js," [Online]. Available: https://nodejs.org/en/about/. [Accessed 7 6 2016].

[5] Express, 14 7 2016. [Online]. Available: http://expressjs.com/.

[6] Redis, "Redis," [Online]. Available: http://redis.io/topics/introduction. [Accessed 7 6 2016].

[7] Redis, "Command reference - Redis," [Online]. Available: http://redis.io/commands. [Accessed 7 6 2016].

[8] NGINX, "NGINX," [Online]. Available: http://nginx.org/en/. [Accessed 7 6 2016].

[9] Apache, "Apache Jmeter disributed testing step by step," [Online]. Available: http://jmeter.apache.org/usermanual/jmeter_distributed_testing_step_by_step.pdf. [Accessed 7 6 2016].

[10] Syddansk Universitet, [Online]. Available: http://www.sdu.dk/en/Information_til/Studerende_ved_SDU/Vejledning/studieteknik/Informationssoegning/Kilde_kritik_internet. [Accessed 7 6 2016].

[11] DigitalOcean, "DigitalOcean," [Online]. Available: https://www.digitalocean.com/help/technical/general/. [Accessed 14 7 2016].

[12] Laravel Forge, Laravel, [Online]. Available: https://forge.laravel.com/features. [Accessed 14 7 2016].

[13] J.D. Meier, Carlos Farre, Prashant Bansode, Scott Barber, and Dennis Rea, "Performance Testing Guidance for Web Applications," Microsoft Corporation, [Online]. Available: https://msdn.microsoft.com/en-us/library/bb924357.aspx. [Accessed 1 6 2016].

[14] J. Rashka and J. Paul, Automated Software Testing Introduction, Management, and Performance, Addison-Wesley, 1999, pp. 43-44.

[15] International Software Testing Qualifications Board, "Standard Glossary of Terms used in Software Testing," 4 7 2014. [Online]. Available: https://www.astqb.org/documents/ISTQB_glossary_of_testing_terms_2.4.pdf. [Accessed 20 6 2016].

[16] Amazon Web Services, "Elastic Load Balancing," [Online]. Available: https://aws.amazon.com/elasticloadbalancing/. [Accessed 16 7 2016].

[17] Amazon Web Services, "Relational Database Services," [Online]. Available: https://aws.amazon.com/rds/. [Accessed 16 7 2016].

[18] Amazon Web Services, "RDS Read Replicas," [Online]. Available: https://aws.amazon.com/rds/details/read-replicas/. [Accessed 16 7 2016].

Terminology

Term or Acronym	Description
Droplet	A virtual private server hosted at DigitalOcean.
LTS	Long Term Support
Git	A popular version control system.
Github	Community site for developers where you can host public and private Git repositories.
HTTP	Hypertext Transfer Protocol
TCP	Transmission Control Protocol
UDP	User Datagram Protocol
Vagrant	A tool that helps manage and configure virtual machines.
VirtualBox	Application used to run virtual machines on a computer.
URL	Uniform Resource Locator. A string that references a resource on the internet.
PHP	PHP: Hypertext Preprocessor. Scripting language most commonly used for web development.
MySQL	The world's most popular open source database software.
Elasticsearch	A distributed search and analytics engine.
AWS	Amazon Web Services
SSL	Secure Sockets Layer

SSH	Secure Shell
Cron	Job scheduler for Unix-based operating systems.
JSON	JavaScript Object Notation
Cookie	A small piece of data that a website can save in the user's browser.

Appendix A: Redis SortedSet commands

These are the Redis commands used by the virtual queue system

ZADD key score value	Adds all specified scores and values to the set stored at key.
ZREM key value	Removes the specified values from the set stored at key.
ZRANGE key start stop	Fetches the range of values from the set stored at key specified by start and stop. To get the first 10 for example, you set start to 0 and stop to 9.
ZCARD key	Returns the amount of values in the set stored at key.
ZRANK key member	Returns the members position in the set stored at key.
ZREMRANGEBYSCORE key min max	Removes all elements from the set stored at key with scores between min and max.
ZSCORE key member	Returns the score of member in the set stored at key.

Appendix B: .eslintrc file

```
{
  "globals": {},
  plugins: ["html"],
  parserOptions: {
    ecmaVersion: 6,
    sourceType: "module"
  },
  "env": {
    "browser": true,
    "node": true,
    "es6": true
  },
  "rules": {
    "strict": 0,
    "no-underscore-dangle": 0,
    "quotes": [2, "single", {
        "avoidEscape": true,
        "allowTemplateLiterals": true
    }],
    "camelcase": [2, {"properties": "always"}],
    "semi": [2, "never"],
    "no-new": 0,
    "curly": 0
  }
}
```

Appendix C. Metrics data structure

```
{
    total: {
        mean: 0.043340388041086975,
        count: 2,
        currentRate: 0,
        '1MinuteRate': 0.021746301622157318,
        '5MinuteRate': 0.0060904037198363445,
        '15MinuteRate': 0.0021556500149239864
    },
    '/': {
        mean: 0.021670411832295615,
        count: 1,
        currentRate: 0,
        '1MinuteRate': 0.008210113258315466,
        '5MinuteRate': 0.0028930685235779457,
        '15MinuteRate': 0.0010598629204630321
    },
    '/event/event-alias': {
        mean: 0.07406813772133873,
        count: 1,
        currentRate: 0,
        '1MinuteRate': 0.014712537947741848,
        '5MinuteRate': 0.0032510706679223385,
        '15MinuteRate': 0.0011018917421948629
    }
}
```

Appendix D. Code example: Adding to queue

```
/**
 * Adds one or more session IDs to the queue.
 * @param  {string} sessionID
 * @return {Promise}
 */
addToQueue(sessionIDs) {
    debug('Adding to queue: '.green + sessionIDs)

    const args = [this.queue]
    const score = new Date().getTime()
    if (Array.isArray(sessionIDs)) {
        sessionIDs.forEach(value => {
            args.push(score)
            args.push(value)
        })
    } else {
        args.push(score)
        args.push(sessionIDs)
    }

    return this.redisClient.zaddAsync(args)
}
```

Appendix E. Code example: Granting access

```
/**
 * Grants access to the max number of sessions.
 * @return {Promise}
 */
grantAccessToMax() {
    return this.redisClient.zcardAsync(this.access)
        .then(res => {
            if (res >= this.accessMax)
                return Promise.reject('Access list is full')

            return this.getSessionsFromQueue(this.accessMax - parseInt(res))
        })
        .then(res => {
            if (!res.length)
                return Promise.resolve(false)

            debug('These sessions are being granted access: ' + res)

            return this.removeFromQueue(res)
                .then(() => this.grantAccess(res))
                .catch(console.log.bind(console))
        })
        .then(res => {
            if (res) {
                debug('Access list has been filled.')
            }
        })
        .catch(res => debug(res))
}
```