|  |
| --- |
| DESIGN AND IMPLEMENTATION OF A QUALITY OF SERVICE TESTING SUITE FOR CLOUD COMPUTING SERVICES |
| A Project Report  Presented to  The Faculty of Computer Science  California State University, Los Angeles |
| In Partial Fulfillment  of the Requirements for the Degree  Master of Science  in  Computer Science |
| By  Adekola Kehinde Togunloju  December 2018 |

© 2018

Adekola Kehinde Togunloju

ALL RIGHTS RESERVED

The thesis of Adekola Kehinde Togunloju is approved.

Jiang Guo, Committee Chair

Russell Abbott

Raj Pamula, Department Chair

California State University, Los Angeles

December 2018

# ABSTRACT

DESIGN AND IMPLEMENTATION OF A QUALITY OF SERVICE TESTING SUITE FOR CLOUD COMPUTING SERVICES

By

Adekola Kehinde Togunloju

This web-based system is used to measure the overall quality of web services. Factors that determine the quality of a web service are its performance, availability, accessibility, capacity and its scalability. Since most web applications utilize some form of web service in one way or the other, there is a need to have a tool to test the quality of such service, to ensure the stability of the application. The purpose of this project is to provide a tool that can be used to measure the overall quality of a web service and provide visual tools pertaining to the performance of the service. The system was designed using a Representational State Transfer (REST) architecture, to ensure easy interfacing with the user interface, and to allow for continuous integration with other implementation and services in the future.

# ACKNOWLEDGMENTS

Firstly, I would like to thank my advisor, Dr. Jiang Guo for his support and encouraging me to write a graduate thesis. Secondly, I would like to thank my sisters for their support and encouragement along the way. Finally, I would like to thank Nelson and James for their vision.

TABLE OF CONTENTS

Abstract iv

Acknowledgments v

List of Figures viii

Chapter

1. Introduction 1

Availability 1

Reliability 2

Performance 2

Scalability and Capacity 3

2. QoS Technical Background and Framework 4

HTML & CSS 4

TypeScript and Angular 5

Jersey and JAX-RS 8

MySQL 9

3. Design and Architecture 10

The Data Layer 11

Application Layer 14

Presentation Layer 16

4. Project Implementation 22

Creating a New Project 22

Managing Existing Projects 26

Managing Web Services 27

Web Projects 29

Login 31

5. Measurements 32

Availability 32

Reliability 35

Performance 37

Capacity 38

Scalability 41

6. Conclusion 45

References 46

LIST OF FIGURES

Figure

1. Sample HTML Code 5
2. Angular Interpolation Example 6
3. Model Binding 7
4. Conditional Statements 7
5. Sample TypeScript Code 7
6. Jersey Example 8
7. Sample MySQL Statement 9
8. Three Tier Architecture 10
9. Database Model for QoS Application 12
10. Application Layer Structure 15
11. Angular Module Declaration 17
12. An Angular Component 18
13. Router Outlet 18
14. Routing Module 19
15. Angular Service Example 20
16. Angular Application Structure 21
17. New Project Page 22
18. Service Configuration 23
19. Project Test Type 24
20. Review and Start Test 24
21. Test Running 25
22. Real Time Performance Graph 25
23. Existing Projects 26
24. Project Comparison 27
25. Web Services managed through WSDL Discovery 28
26. Web Services managed through WSDL Discovery 28
27. Web Projects Page 29
28. Upload Test 30
29. Download Comparison Times 30
30. Login Page 31
31. Availability Options 32
32. Get Availability (Part 1) 33
33. Get Availability (Part 2) 33
34. Options for Availability 34
35. Result for Availability 35
36. User Options for Reliability 35
37. Code Calculation Reliability 36
38. Reliability of a Web Service 36
39. Performance Code 37
40. Real Time Performance Monitor 37
41. Capacity Test Options 38
42. Capacity Code 39
43. Capacity Results Graph 40
44. Scalability Options 42
45. Scalability Code 43
46. Scalability Graph 44

# CHAPTER 1

# Introduction

Quality of Service (QoS) Suite for Cloud Computing Services is a system that used to measure the quality of web services. The back-end is designed using the Representational State Transfer (REST) architecture. The REST API, handles all the application data and processes, including threads pools, servlet contexts and the database connection. The system also has a front end, which consumes the REST API. The front end is what the user interfaces with and it’s used to provide visual data to the user in regard to which test they may be running.

Users are provided with a configuration wizard, to help them initialize tests. The test URL can be constructed using a WSDL (Web Service Definition Language) or it can be constructed manually. Also, users can save constructed URLS for use in future tests. They are saved to the database. These can also be deleted at the user’s convenience.

In the application, there are five types of tests to choose from. Each test determines what kind of analysis the user wishes to perform on the web services. The types of tests are as follows

**Availability**

This refers to the frequency a web services resources are available. This is an important factor in web services as prolonged downtime, can lead to a severe loss of revenue. Availability measures the probability of a web service being available in a specific time frame. This application computes the availability Ai as a measure of its uptime divided by the sum it’s uptime and downtime. In other words:

Ai =

**Reliability**

Reliability represents the quality of a Web service; It refers to the probability of a web service being capable of maintaining the service and service quality, without any outages or failures. A measure of reliability is the number of failures per month, or per day. This is also an important aspect in web services, as services that are not reliable could prove counter-productive to a developer’s goal. The Reliability Ri, in the application is calculated using Lusser’s equation [1]

Ri =

where 𝑙 is the failure rate and 𝑡 is mission time, such as 1 day, 1 week, 1 month, and 1

year. 𝑡 is needed to give reliability 𝑅 a time dimension for calculating results. Failure rate 𝑙 is defined as:

**Performance**

This refers to the overall response rate of the web service. Performance is the quality aspect of Web service, which is measured in terms of throughput and latency. Higher throughput and lower latency values represent good performance of a Web service. Throughput represents the number of Web service requests served at a given time period. Latency is the round-trip time between sending a request and receiving the response. The application computes the average performance of the web service by dividing the cumulative response times by the number of requests made. This can be expressed by the equation:

**Scalability and Capacity**

This is the quality aspect of a service that represents the degree it can serve a Web service request. It may be expressed as a probability measure denoting the success rate or chance of a successful web service call a point in time. There could be situations when a Web service is available but not accessible. In order to make web services more accessible, highly scalable systems need to be built. Scalability refers to the ability to consistently serve the requests despite variations in the volume of requests [2]. It measures the capacity of the web service, and how many users it can handle at a single point in time. The scalability can be represented as the performance relative to the arrival rate (increase in user count) of the system.

CHAPTER 2

QoS Technical Background and Framework

Considering the purpose of this application, it was necessary to use tools that were highly scalable, performance oriented, and efficient in the utilization of resources. The application backend was designed using a RESTful architecture. This refers to a transferring, accessing, and manipulating textual data representations in a stateless manner [3]. The framework used for implementing this was JAX-RS, implemented using Jersey. A servlet container was used to store, initialize and destroy, every application level instance, including the database instance, in order to avoid concurrency errors, and scheduling errors. The backend also utilized Threads and Thread Pools, in order to ensure the asynchronous nature of API calls. The front end was designed using the Angular.io framework, which utilizes HTML, CSS and TypeScript. This was used consume the back end and provide the user with visual data in the form of charts and tables. The database was designed and implemented using MySQL.

**HTML & CSS**

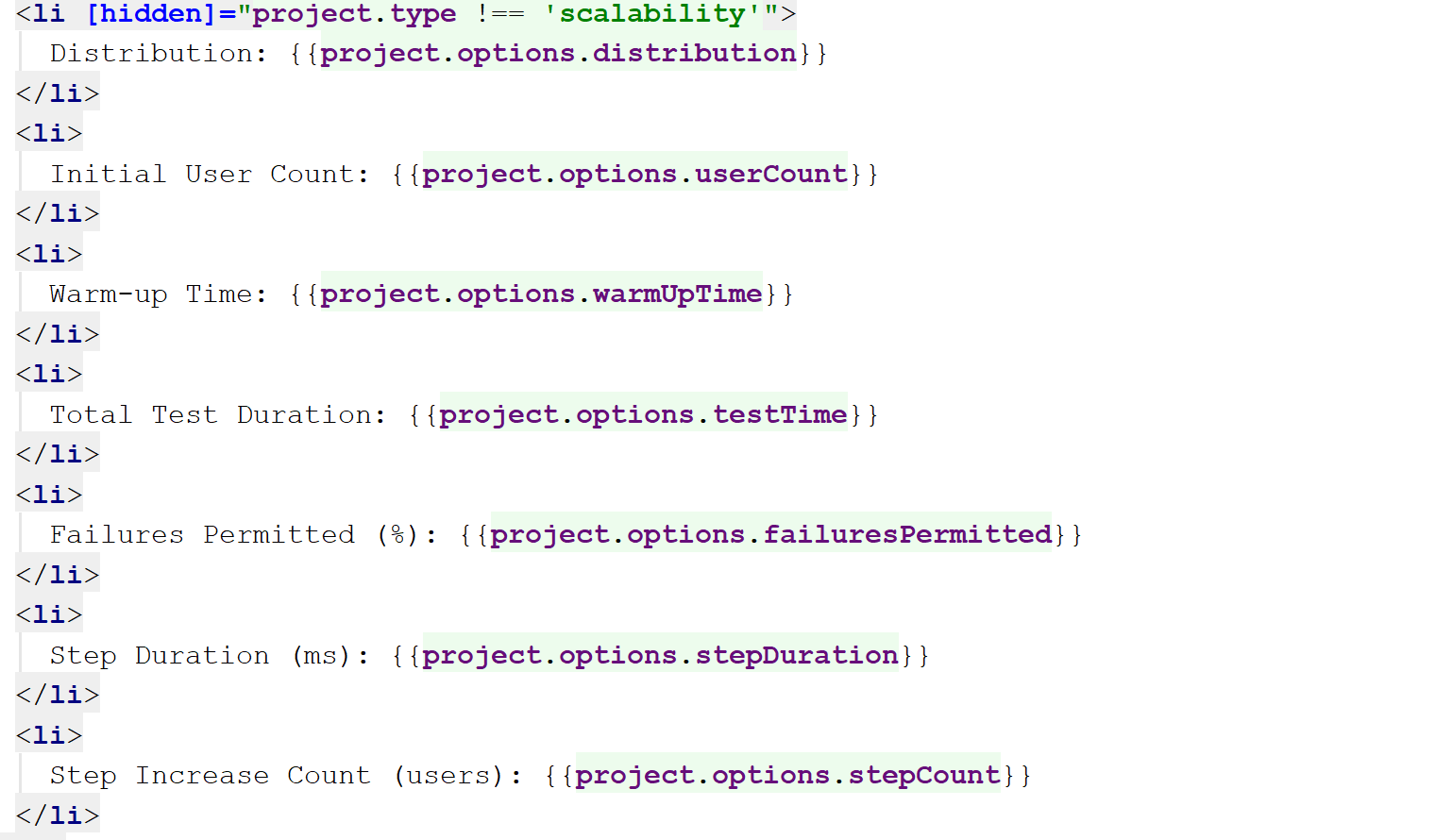
Hypertext Markup Language (HTML) is a scripting language that is supported in all existing browsers and environments. HTML defines markup that is used to define the tree structure of a web page. The markup takes the form of a starting tag being paired with closing tags. A sample HTML code is shown in Figure 1.



*Figure 1.* Sample HTML Code.

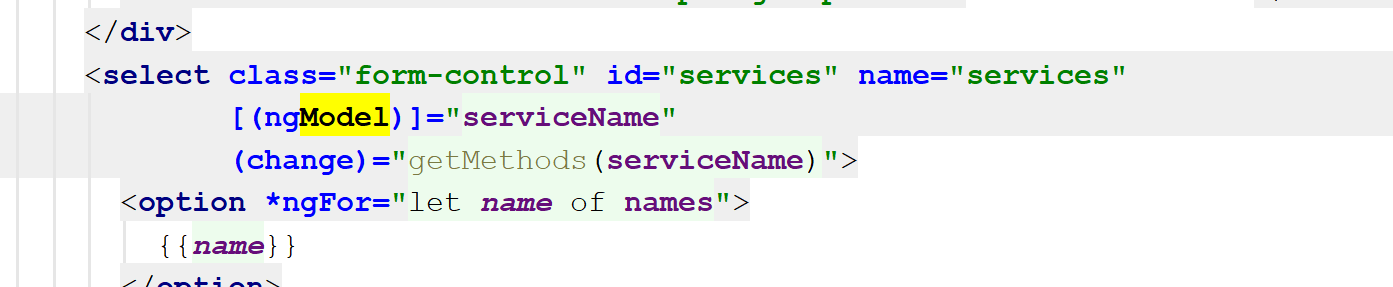
**Typescript and Angular**

Angular is a modern framework built entirely in TypeScript, and as a result, using TypeScript with Angular provides a seamless experience. TypeScript is a strict syntactical superset of JavaScript and adds optional static typing to the language. Angular follows the Model-View-Controller (MVC) pattern. A component in Angular is divided into two parts, The HTML& CSS file(s) and the TypeScript file. Angular makes use of data binding in order to add data to its html files. An example of this is interpolation. Figure 2 shows an example of interpolation in Angular

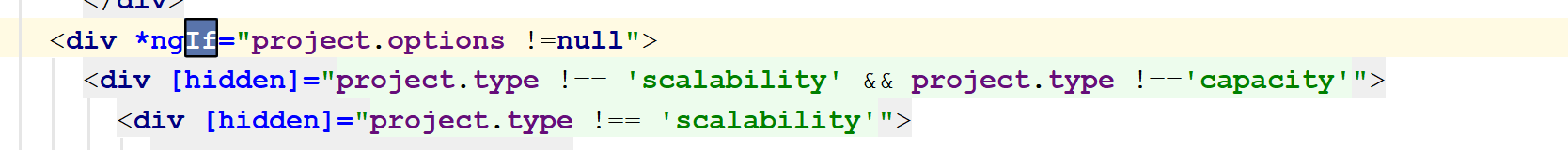


*Figure 2.* Angular Interpolation Example

Another form of Data binding is model binding. In both interpolation and model binding, variables from the TypeScript file are bound to the HTML file which allows developers to add custom logic to their application, and consequently give developers the ability to produce dynamic content. An example of model binding, and conditional statements can be seen in Figure 3 & 4. It also implements the Document Object Model (DOM) events, creating its own custom events, which is defined in the lifecycle of application. An example of a TypeScript file that implements multiple lifecycle events can be seen in Figure 5.



*Figure 3.* Model Binding



*Figure 4.* Conditional Statements



*Figure 5.* Sample TypeScript Code

**Jersey and JAX-RS**

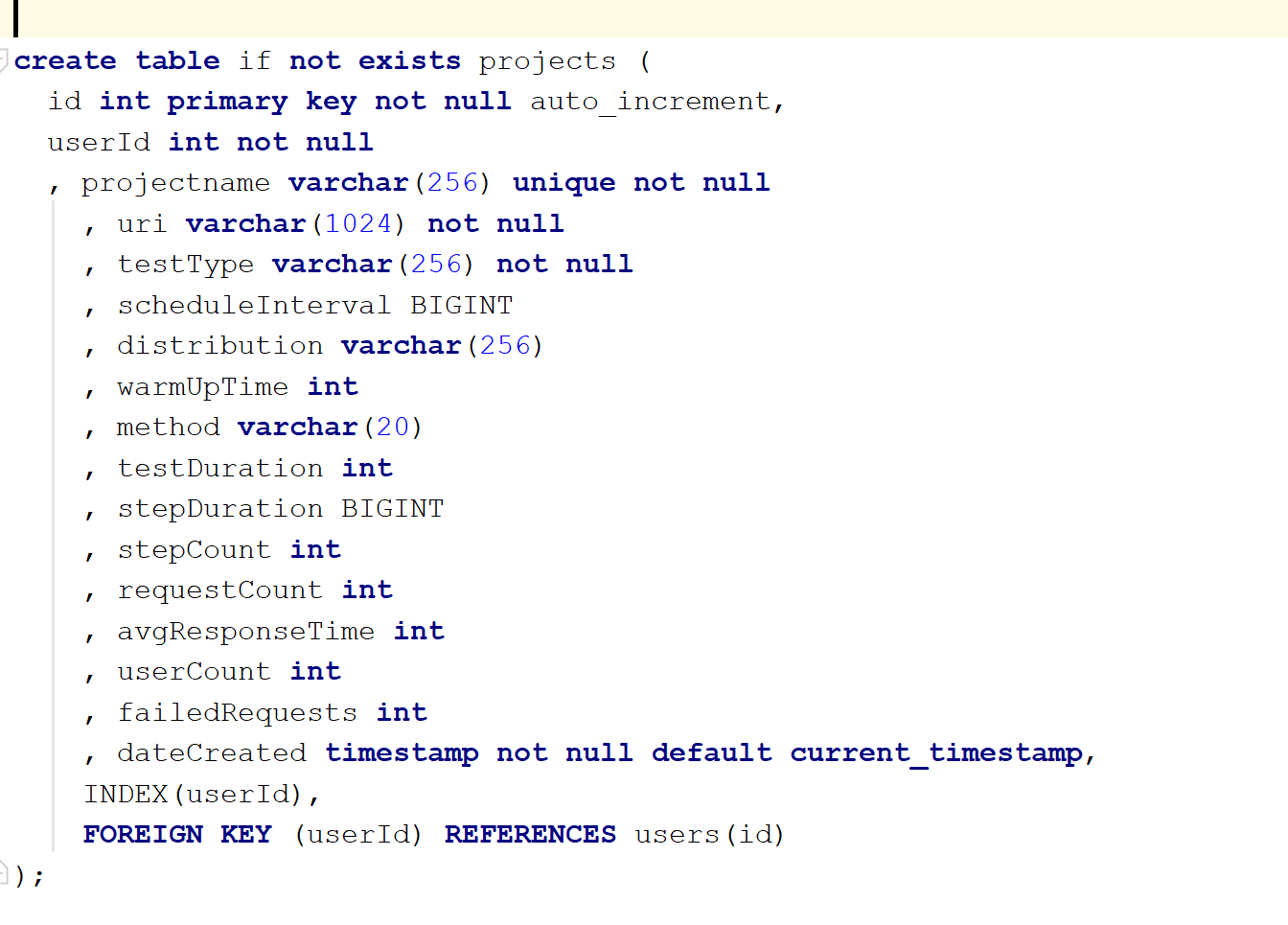
Java API for RESTful Web Services (JAX-RS) is a Java programming language API specification that provides support in creating web services using the REST architectural pattern [4]. JAX-RS uses annotations, to simplify the development and deployment of web service clients and endpoints. The annotations are used to signify HTTP request methods, URL patterns and many more. In Figure 6, a Java Servlet class built using the Jersey framework is shown. In this figure, the method “getProjects” is called whenever a GET request is made the path app-root/webprojects. The @Produces annotation tells the application, that the method will produce a JSON object. In this example, the program queries the database for a list of Web Projects, uses the GSON library to convert it into Json format, and sends it as a response.



*Figure 6:* Jersey Example

**MySQL**

Structured Query Language (SQL) is the standard language for relational database management systems. MySQL is the most popular Open Source SQL database management system. It is developed, distributed, and supported by Oracle Corporation.

MySQL is used to manage a database that stores the data for a web application. It comes with a robust set of commands and constraints, which are used to properly define the data in the database. Figure 7. Shows an example of a sample MySQL statement.

*Figure 7.* Sample MySQL Statement

CHAPTER 3

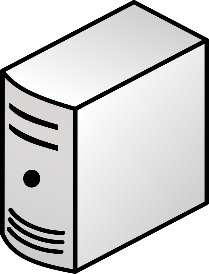
Design and Architecture

The web application is comprised of three layers; The Data Layer, The Application Layer, and The Presentation Layer. The data layer is composed of the database used to store application related data, logger, which logs the application processes and errors that may occur. The application layer contains the logic of the web application, including but not limited to the REST API end points, the various formulae and equations used and the overall flow of the application. The presentation layer is what is shown to the user. It is composed the user interface, which consumes the Application Layer and presents data to the user in a way that makes sense to them.

DBMS

Application Logic

Database



User Interface

Server

Query

Retrieve

Store

Host

Request

Response

Presentation Layer

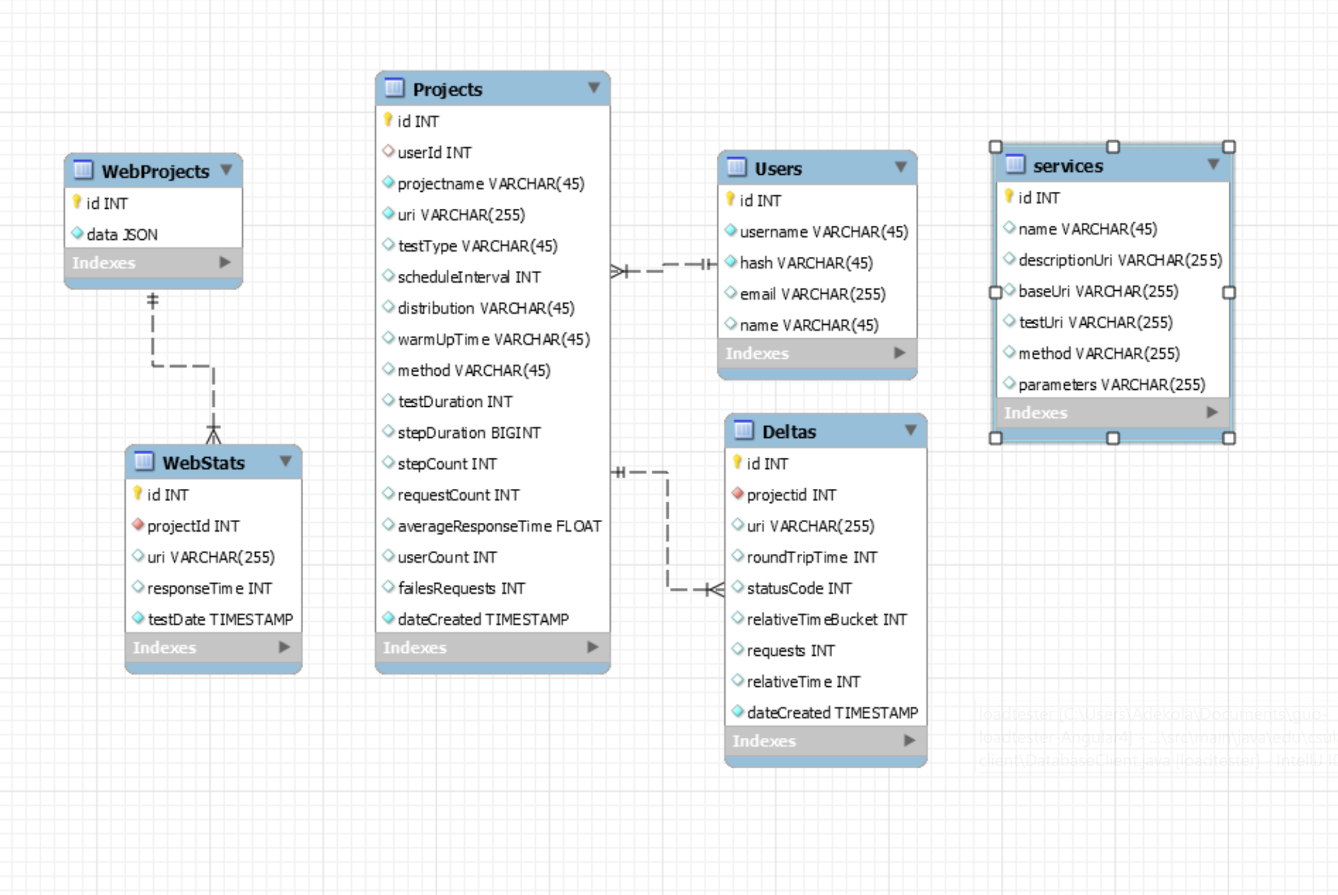
Application Layer

Data Layer

*Figure 8.* Three tier Architecture

**The Data Layer**

MySQL is used here in managing the relational database. The data is organized into relations, or tables. In this project, there are six relations: projects, deltas, and services, users webprojects and webstats. Projects store every test project that a user has created or tested. Deltas store the results of a web service response call for a project. Each time a user starts a test, the application makes multiple calls to the resource the user tests. The results of this calls are what is stored in the Deltas tables. Each row in the Deltas table is specific to a project. Services tables stores all the web service URLs that the user saved while testing. It also contains some predefined URLs, that the user can use to start a quick test. Users tables contains the list of users in the application, and their usernames, and a hash of their passwords. Webprojects tables stores all project specific to a application, like an upload application, or more generally, non-web API calls. Webstats, stores the results of each call in the webprojects table. Figure 9 shows the model for the database, created using MySQL Workbench.



*Figure 9.* Database Model for QoS Application

The projects table has a one-to-many-relationship with the deltas table. The tables are bound with the projectId column on the deltas table referencing the id column on projects table. The corresponding row to the foreign key on the deltas table is set to cascade delete and cascade update, whenever the referenced primary key on projects table is deleted or updated. The user’s table has a one-to-many relationship with the projects table. The tables are bound with the userId column on the projects table, referencing the id column on the users table. The corresponding row to the foreign key on the deltas table is set to cascade delete and cascade update, whenever the referenced primary key on users table is deleted or updated. The webprojects table has a one-to-many-relationship with the webstats table. The tables are bound with the projectId column on the webstats table referencing the id column on webprojects table. The corresponding row to the foreign key on the webstats table is set to cascade delete and cascade update, whenever the referenced primary key on webprojects table is deleted or updated

On deltas table, the column roundTripTme, indicates the time it took for the Web service to receive a request and send a response. StatusCode indicates the HTTP status code; whether the call was a success, or an error, and what kind of error it was. The column relativeTimeBucket, indicated how long it took before the application sent its first request, in the case of a performance test, or how long it was before subsequent tests, in the case of Scalability or Capacity tests. Requests indicate how many requests the application sent, before sending that request.

The projects table contains columns that enable the user to keep a history of past projects and to re-test them. Columns avgResponseTime and failedRequests lets the user see the averagePerformance of all tests, and how many times a request to that resource failed. Attributes such as testType, testDuration, stepDuration, stepCount, and requestCount provide context as to why a service performed the way it did for that

project. Attributes like userCount and distribution are specific to certain test types like scalability tests or capacity tests.

The webprojects table contain dynamic information for special webprojects like Upload tests, website response tests and comparisons. The webstats table contain information pertaining to the results of the projects run in the webprojects table. A JSON type is used to store the project data in the webprojects table to allow for dynamin attributes.

The users table contain information for each user using the application. Their passwords are stored with a hash string as a security measure against database attacks.

**The Application Layer**

The application is written in Java. The framework used in Jersey REST Web services API, and it is hosted using a Tomcat Server. When the application starts, the database instance is initialized and available in the servlet context in order to avoid concurrency exceptions where too many instances are open. Thread pools for application threads are initialized in order to optimize performance and memory management [5].

When an API call is made to the applications, the URI is parsed and matched to a pre-defined pattern in the application in order to get the method. When the method is called, the method performs its task and then returns a response back to the entity that requested the HTTP call. Methods in the application are mapped to GET, POST, PUT or DELETE requests. The specified HTTP call must also match the type of HTTP call or else a 404 error is returned as a response.

Each test that is performed in the application is asynchronous meaning, the application doesn’t wait for the test to finish executing, before performing other tasks. The application achieves this using thread. Each test runs on its own thread and its child functions such as saving data to the database also run on their own threads. Since threads are costly to create, the application manages it using thread pools. The application stores all running threads in a thread pool available in the servlet context. This enables the application to find test instances and stop them, run multiple tests at the same time while preserving memory and performance efficiency. Figure 10 shows the overall structure of the application layer

Tomcat Server

Database Context

API Call

Tests

Handlers

Consumers

Thread Pool

*Figure 10.* Application Layer Structure

The application also uses token-based authentication to verify a user. If a valid token is not sent with a request, that request is parsed by the server as a demo user. If a valid token is sent, that request is parsed, and the user details are retrieved. An authenticated user has their tests personalized to them and cannot be viewed by any unauthorized user that isn’t them.

The application follows a process for carrying out tests. When a request for a test is sent to the server, the application parses the test information such as the type of test, the resource URL, and the parameters for the test. The application uses this to create a test context, which it puts in a map in the servlet context. It starts up a test client and executes the test. While the test is being processed in the background, the application sends a response the to client indicating testing has begun. In the background, a data consumer is initialized to store data, while a local asynchronous task handler is initialized to deal with HTTP events. These are also stored in the servlet context. The application begins to make a series of HTTP calls, and the results of the HTTP call are processed by the handlers and consumed by the data consumer. The consumer saved the data to the database by using a queue; Each result is sent into a queue, while an insert call is made for each block. This is necessary in order to avoid database exceptions. When a stop request is sent to the server for a test, the test is fetched from the servlet context using its project id, and all tasks associated with that test are halted. This mechanism allows for a project to only be tested one at a time, rather than multiple times concurrently. This avoids any data corruption that may arise from such event.

**Presentation Layer**

This displays a graphical user interface for users to interact with. This is achieved with the use of HTML and CSS, creating structures and defining rules to style the application. But what achieved the successful interaction between the front-end and back-end is Angular, with Typescript.

Angular is an open-source web application platform developed by the Google team. It is a complete rewrite of AngularJS which uses JavaScript as its language. TypeScript is similar to JavaScript but with few differences. TypeScript is a fully typed language while JavaScript is not. It supports the use of Observables and other services that make web application development easier.

The root of an Angular application is used to define modules. Modules can also be defined at the sub-modules of the application, but their use will be limited to that module and its children. Angular uses module definition to bootstrap the application and inject services and libraries into the application. This also allows for the application to be compartmentalized, using only what needed, giving way for developers to produce clean scalable code. Figure 11 shows an example of a module declaration in angular.

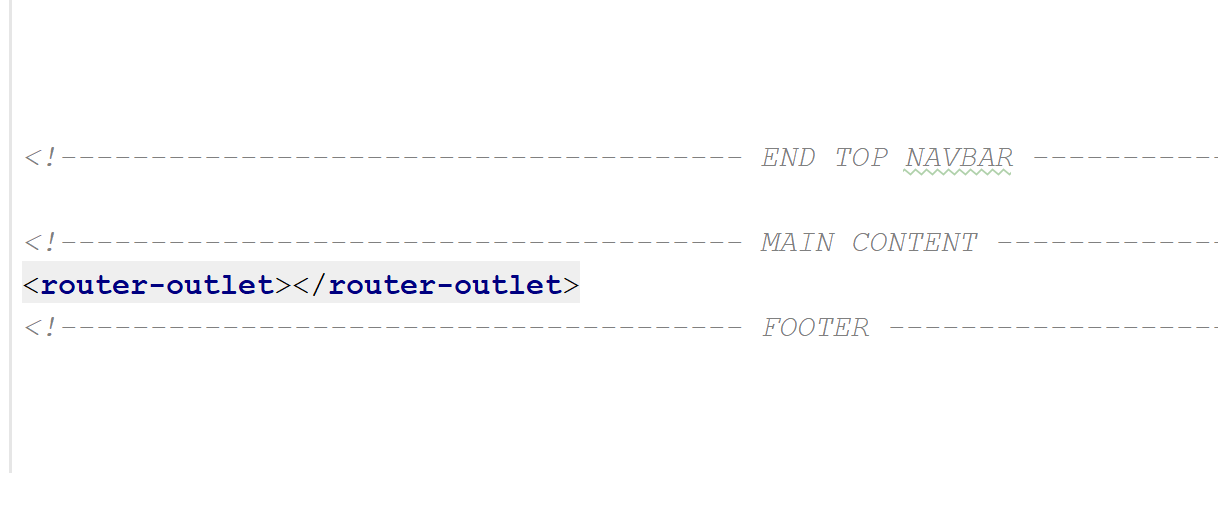


*Figure 11.* Angular Module declaration

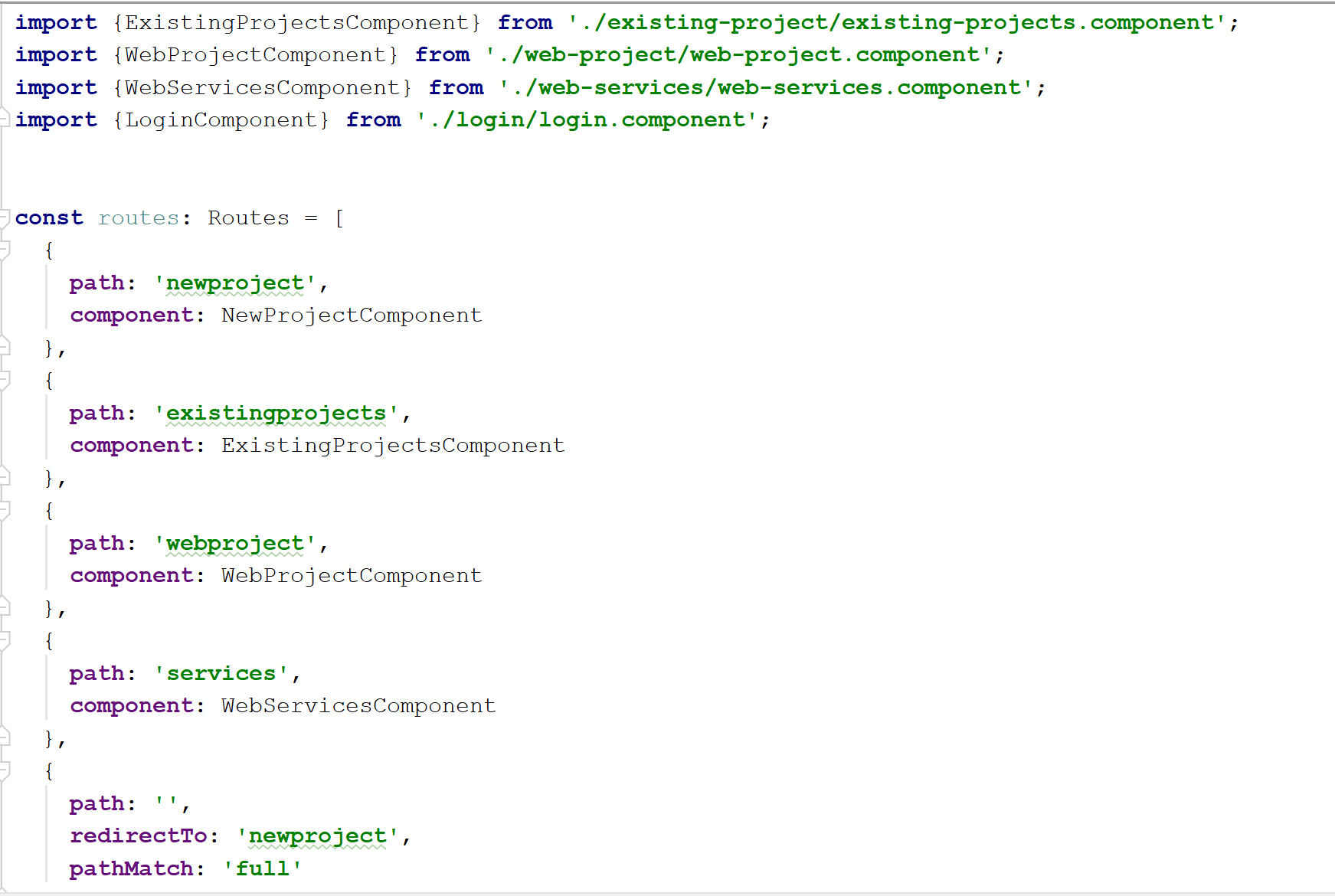
Angular uses routing to navigate between components. Components are like pages in an Angular application. It has its own data and behavior. It consists of a HTML and TypeScript file, and optionally a CSS file. Figure 12 shows an example of a component in Angular. A router is a module defined in an application. A route is specific to a component, although a component may have multiple routes. This is plugged into the application using a router outlet. A router outlet is used to load navigated routes into the application. It is defined using a custom HTML tag. Figure 13 shows an example of a router outlet. Figure 14 shows an example of a routing module.



*Figure 12.* An Angular Component.



*Figure 13.* Router outlet.



*Figure 14:* Routing Module

Angular uses services to perform special tasks such as HTTP calls, an also to pass data within components. Services are singleton objects which get instantiated only once during the lifetime of an application [6]. Services are very useful for this task because, it is considered bad practice to allow Angular components to talk to each other. Services are injected into components, enabling components to have access to a single instance of a service. Due to the asynchronous nature of HTTP calls, it is often inadvisable for components to make HTTP calls. Services take care of that by making the calls and emitting them to the component using an observable. Observables provide support for passing messages between publishers and subscribers in an angular application. Observables offer significant benefits over other techniques for event handling, asynchronous programming, and handling multiple values. Observables are not executed until a consumer subscribes to it. The subscribed consumer then receives notifications until the function completes, or until they unsubscribe. It can deliver multiple values of any type literals, messages, or events, depending on the context. Figure 15 shows an example of a service.



*Figure 15.* Angular Service Example

The various parts of an Angular application come together to form a highly scalable and resourceful application, capable of handling event, routing and errors. Figure 16 shows the basic structure of the angular application.

**Root**

**App Module**

**Routes**

**Components**

**Assets**

**Config**

**Services**

**Models**

*Figure 16.* Angular Application Structure

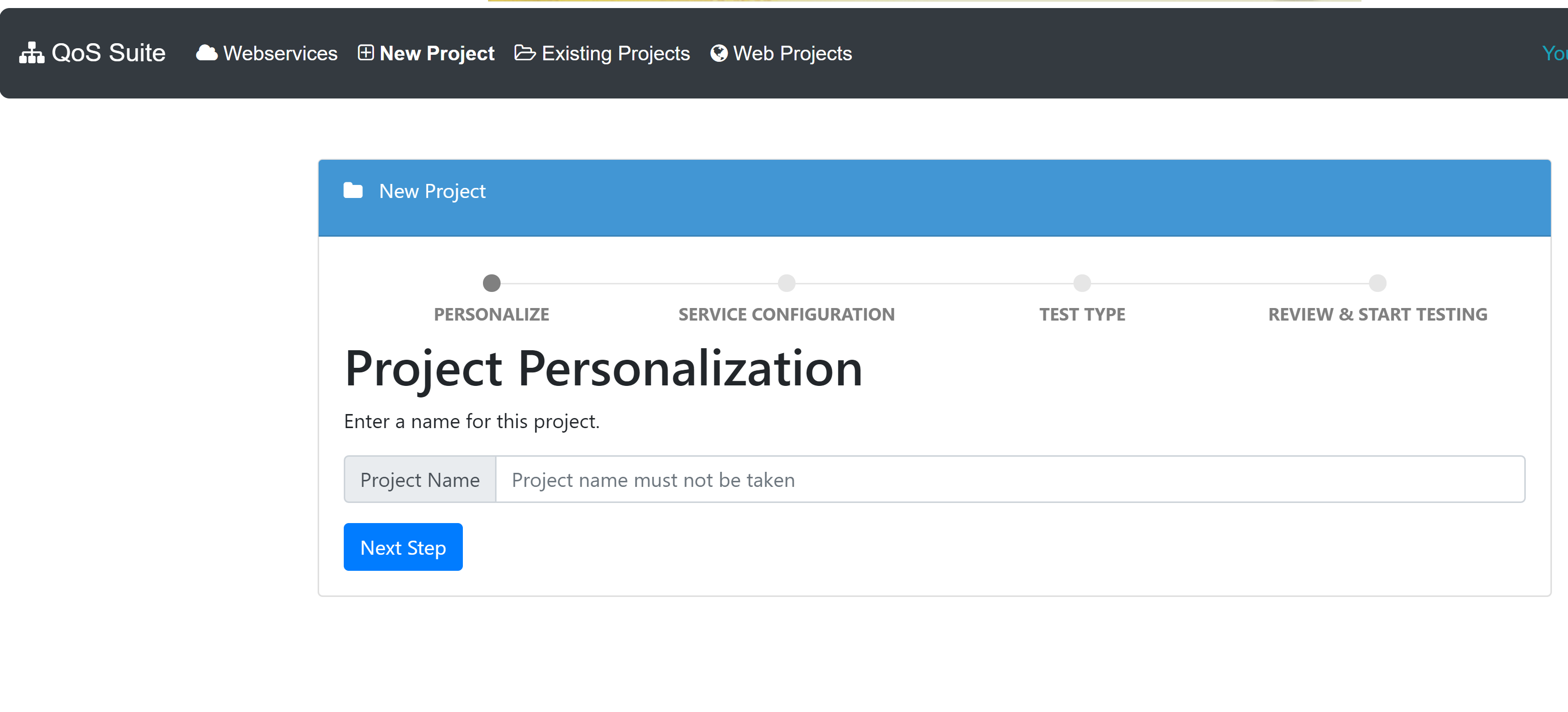
CHAPTER 4

Project Implementation

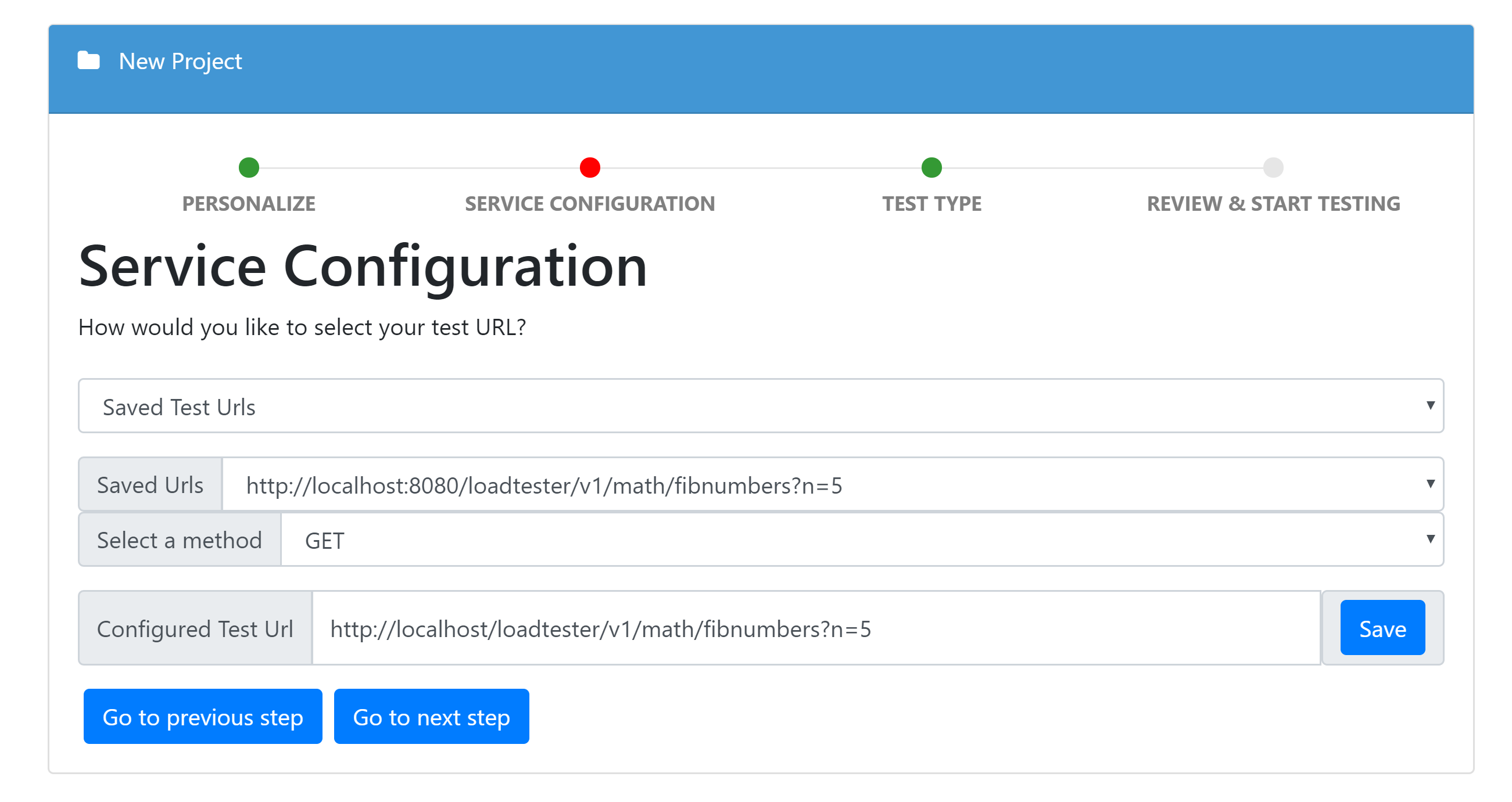
The application provides users with the capability of initializing tests from the GUI. The test results are shown with the aid of charts. External libraries like Google Charts and Chart.js were used in producing these charts.

**Creating a New Project**

Starting up the application, the first page that comes up it the new project page. This allows users to begin creating a test. Figure 17 shows the new project page.

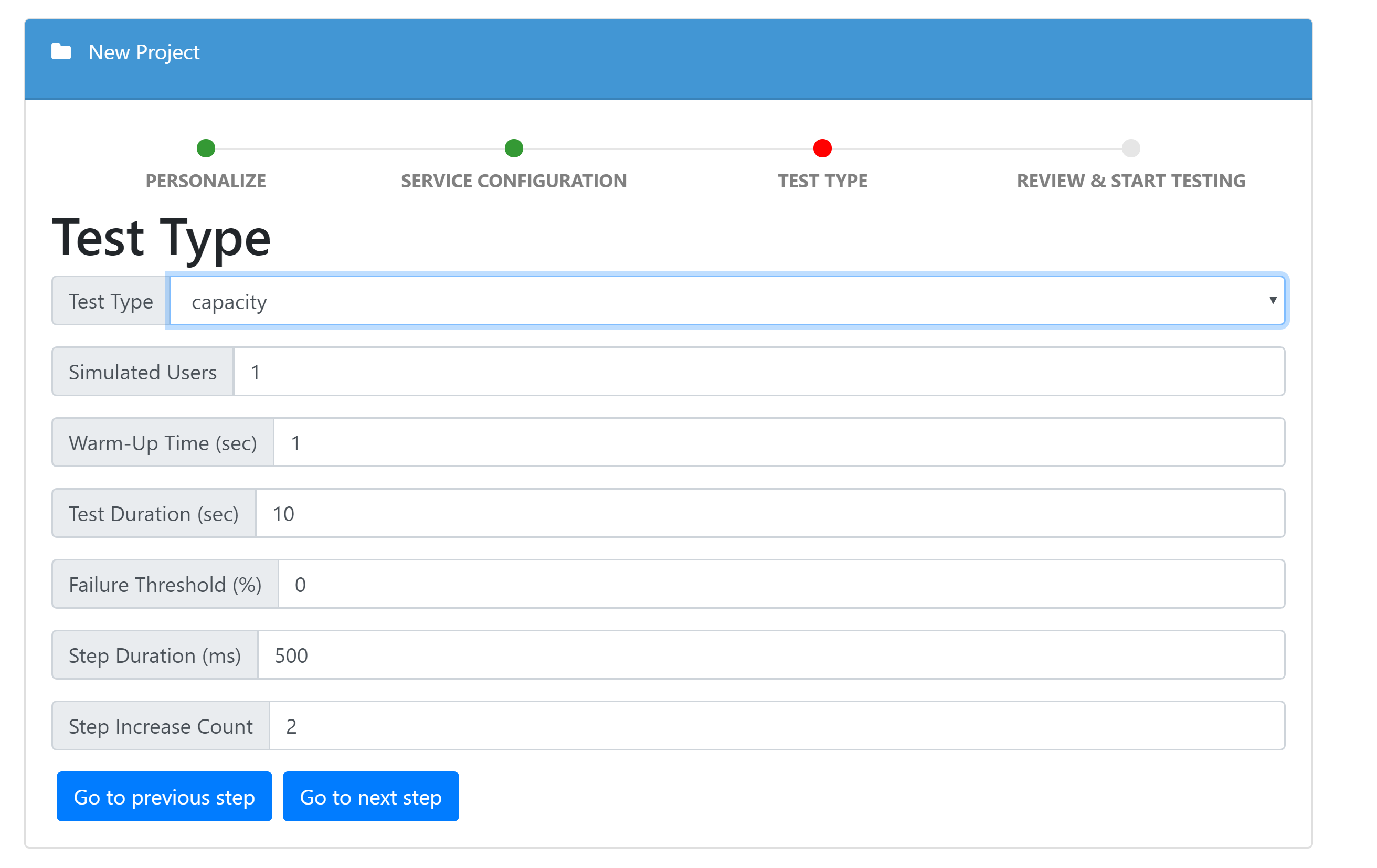


*Figure 17.* New Project Page

The application uses a wizard that allows the user to navigate back and forth. Each step has its own requirements on whether the user can move forward, or the user needs to fill in some required data. In this first page the user creates a unique project name. The Application then takes the user to configure the type of test they want to do. A figure depicting this is shown in Figure 18

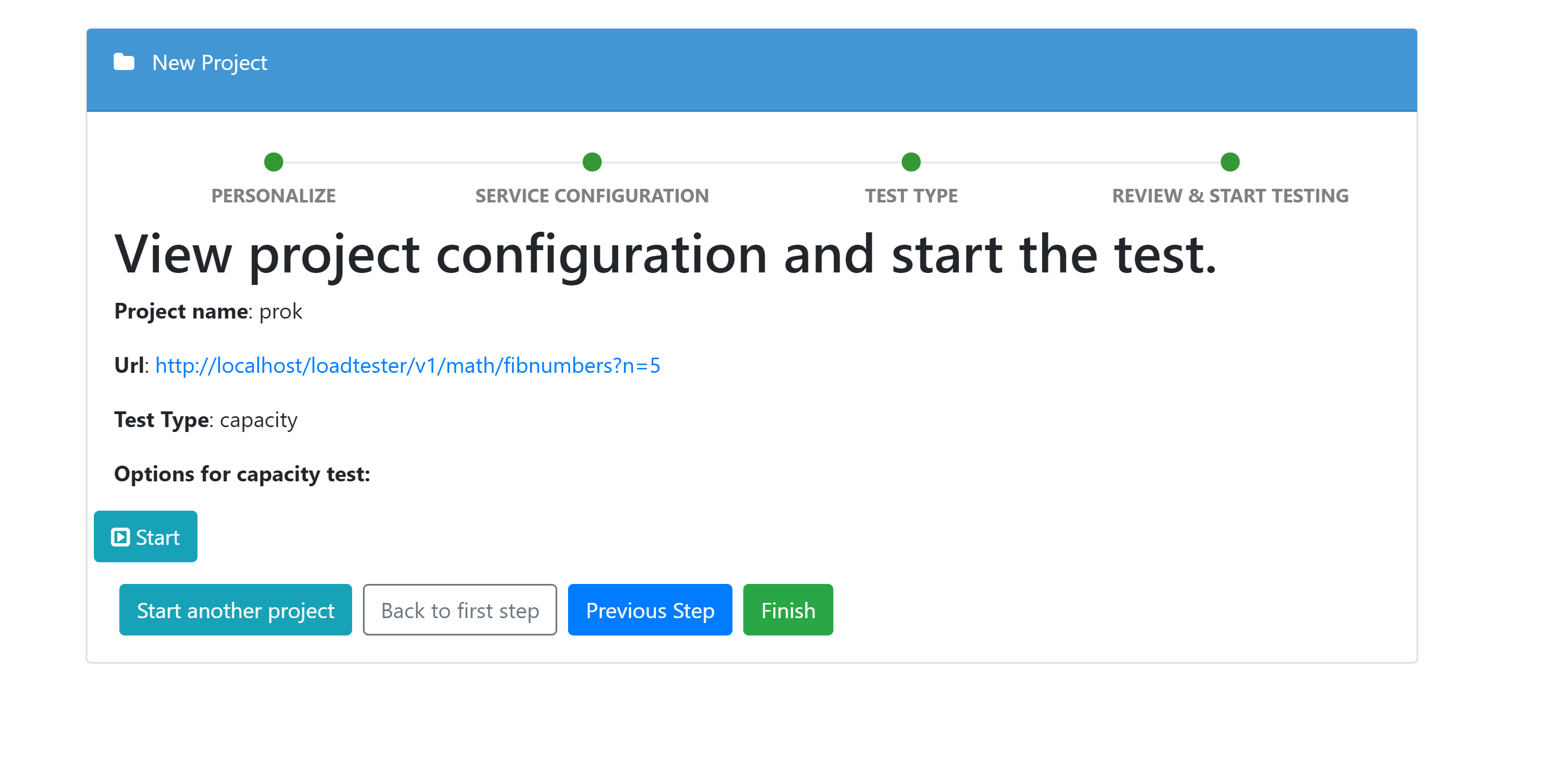
*Figure 18.* Service Configuration

Next the application asks the user what kind of test the user would like to perform. Option include scalability, performance, reliability, capacity and availability. The step is shown in Figure 19.

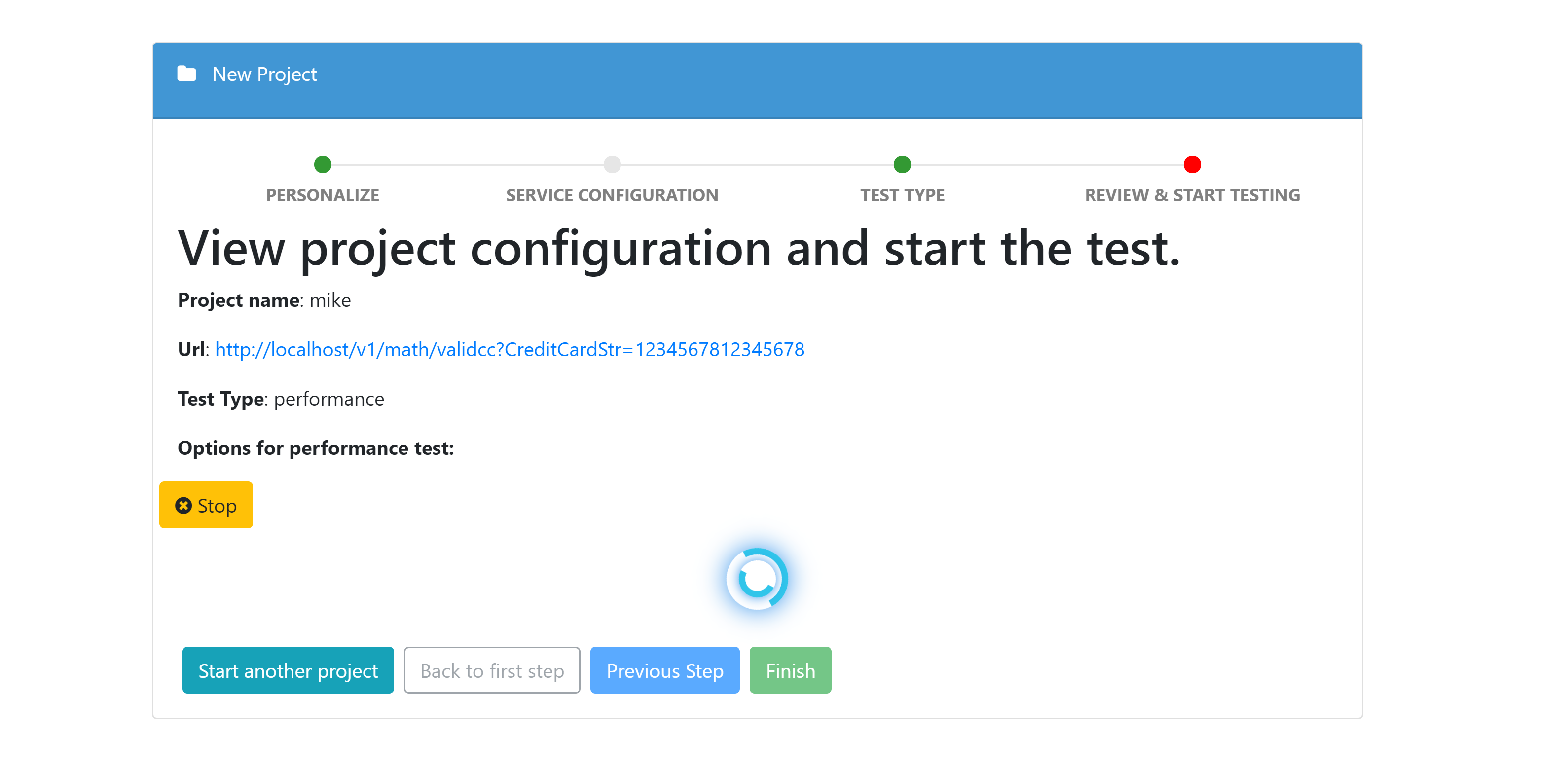


*Figure 19.* Project Test Type.

The final step is when the user reviews the information they entered and start testing. After this step, the user can stop the test, or go and test another project. Figures 20 and 21 shows the state before starting a test and after starting a test.

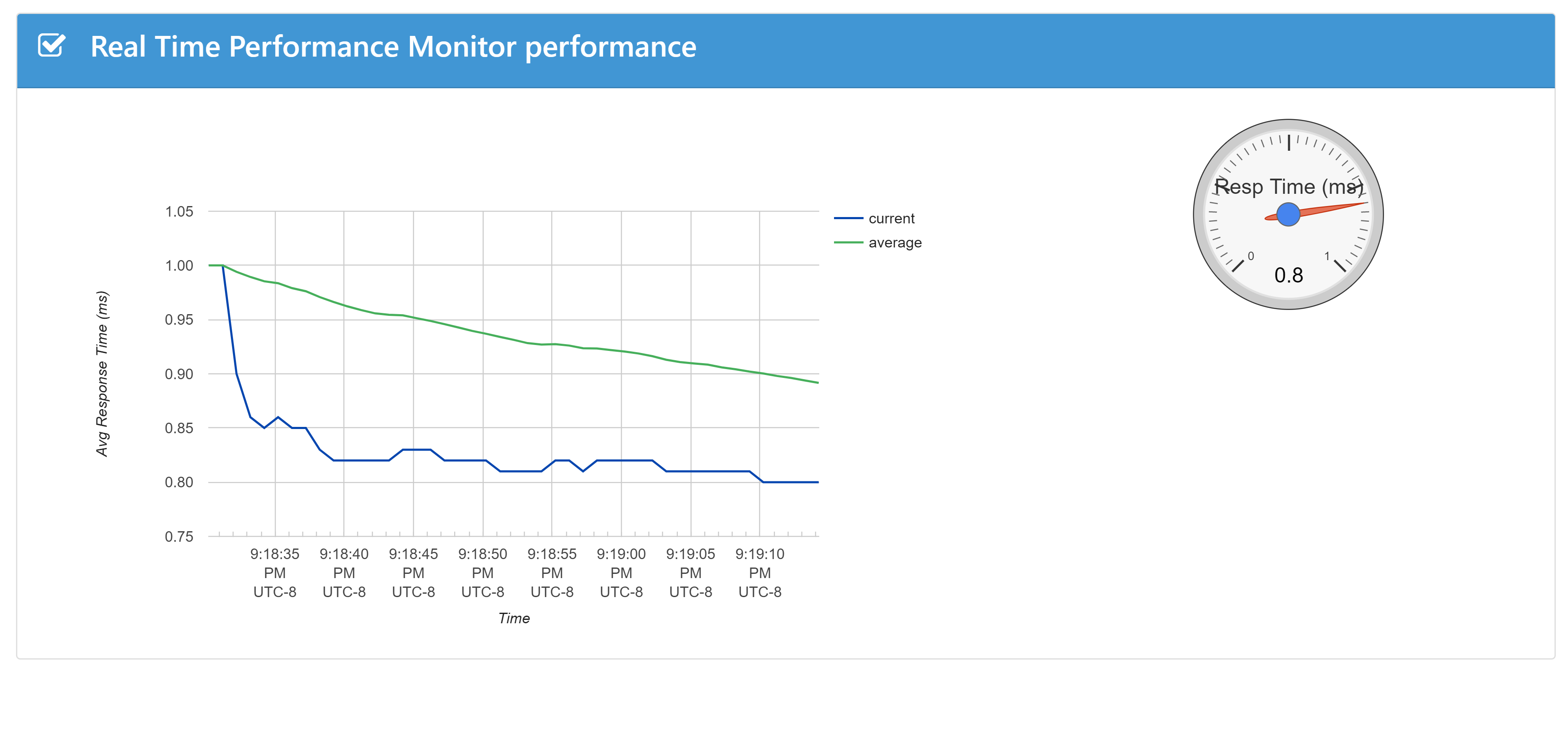


*Figure 20.* Review and Start Test



*Figure 21.* Test Running

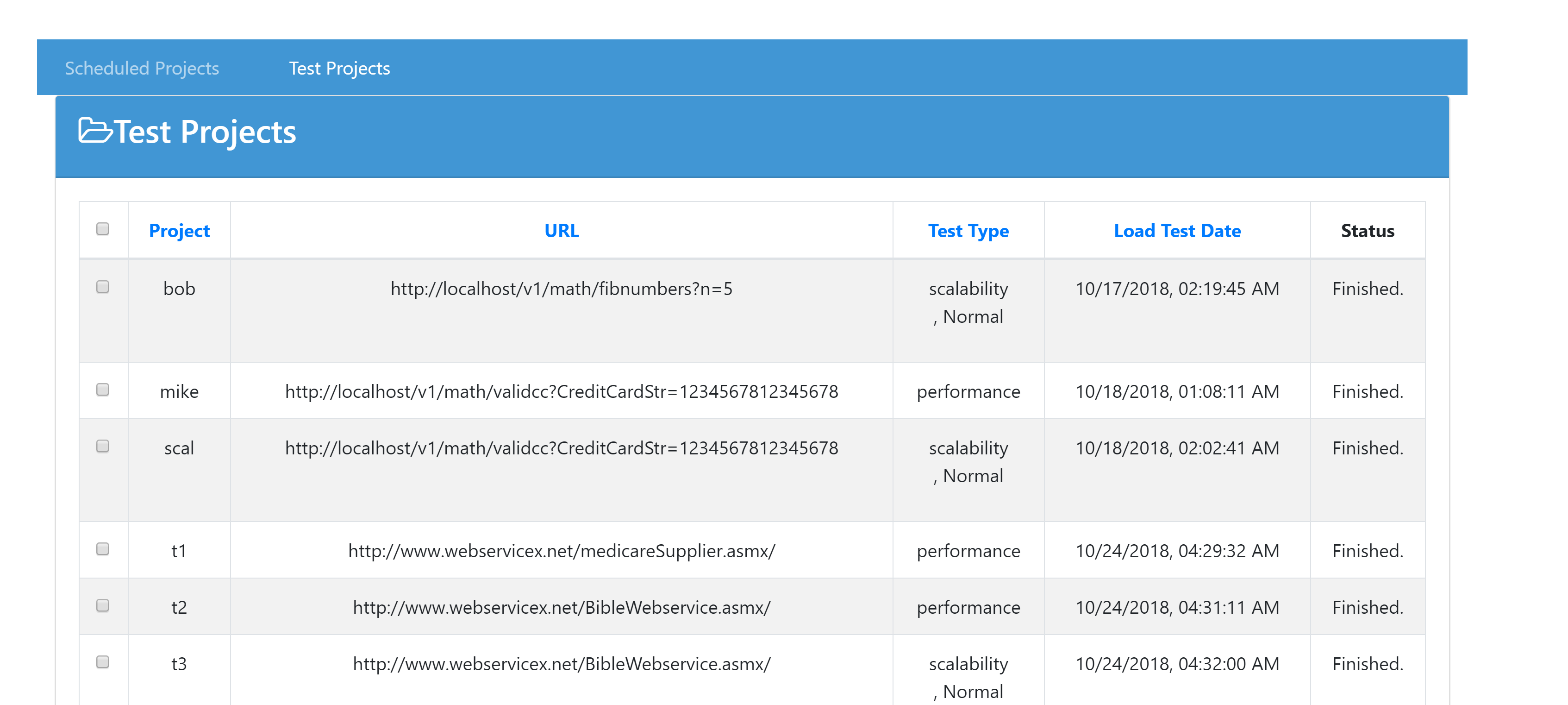
When the test is running, the interface displays a graph showing information on the results of the test. In the case of a performance test, the application displays a graph showing the real time performance of the resource, and its average performance over time. Figure 22 shows a graph depicting the real time and average performance of a resource.



*Figure 22.* Real Time Performance Graph

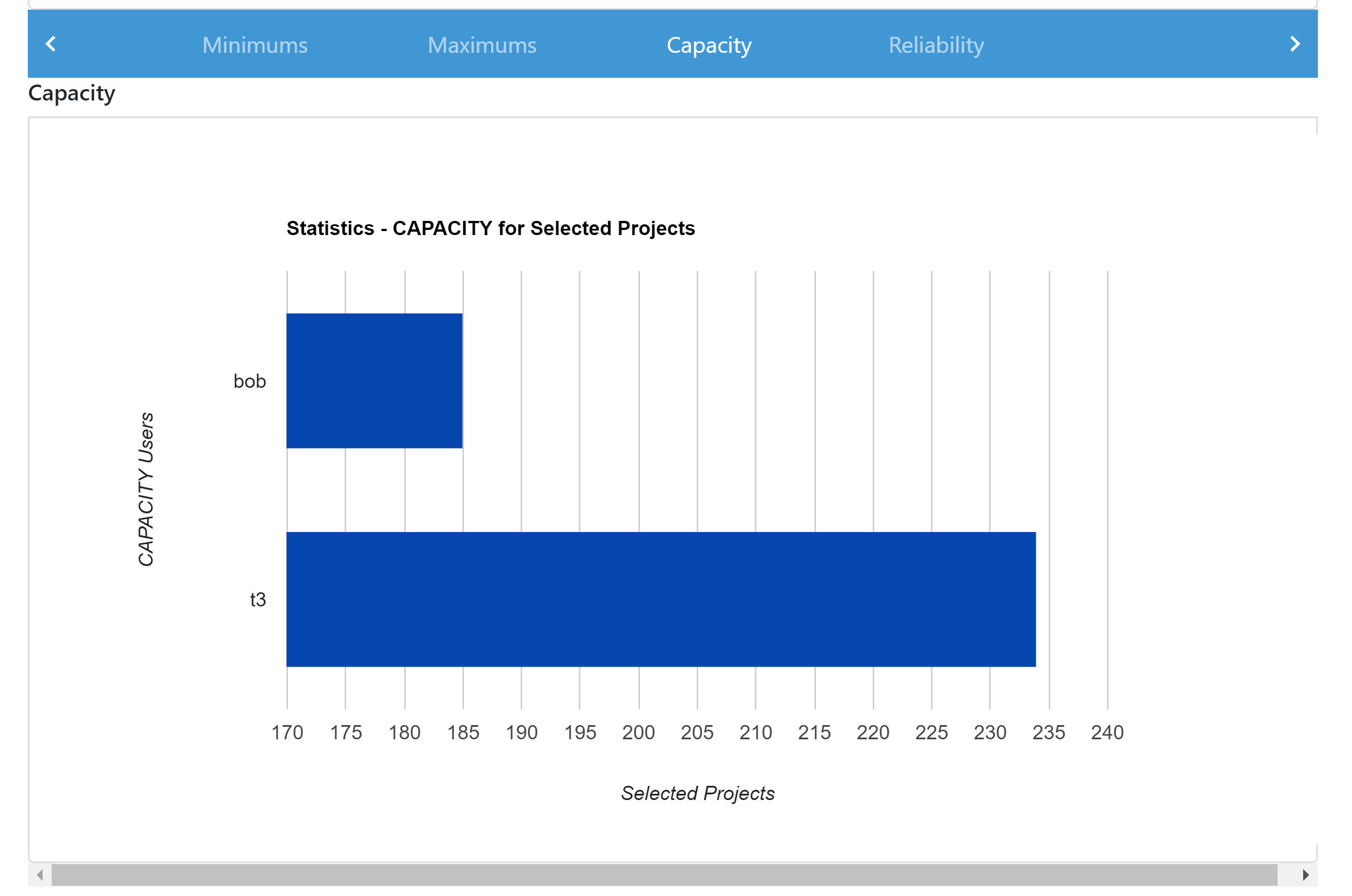
**Managing Existing Projects**

The application also allows users to view their current projects. From here, the user can perform retests of the project, or compare them, with already tested projects. Existing projects are grouped into two groups: Test projects and Scheduled projects. Scheduled projects are Availability or Reliability tests that are scheduled to run without any interruption from the users. The purpose of this tests is to determine the availability and reliability of a resource at a given time. Figure 23 depicts the page that displays existing projects



*Figure 23.* Existing Projects

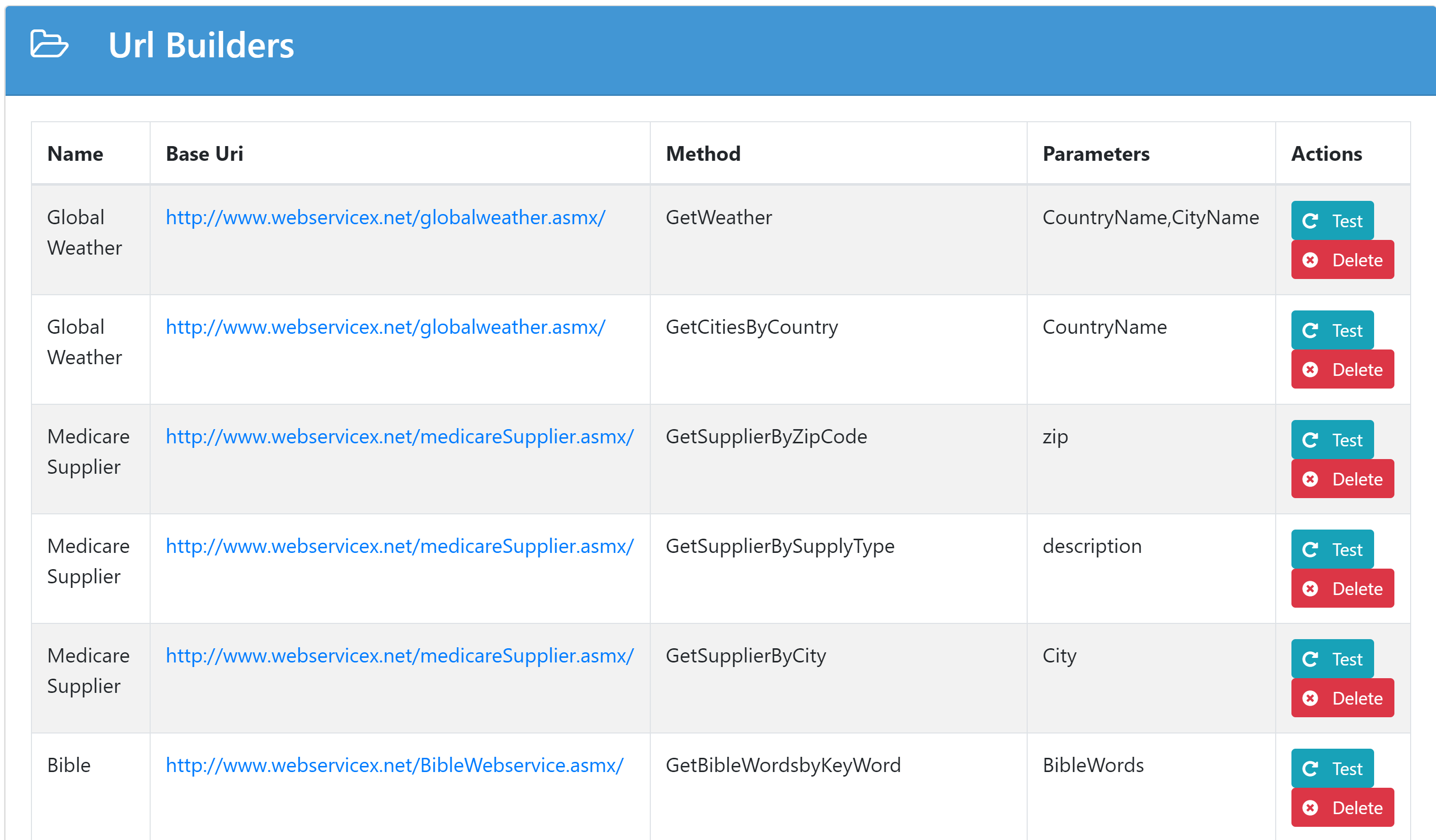
In the existing projects page, users can load the results of the test they performed. They can delete projects from the database here too. Comparison of two or more projects are usually done between projects of the same type. The comparison compares the output of the projects if there are any. The Maximum and Minimum response time, The capacity and reliability of the two projects. Figure 24 shows a comparison between two scalability projects.



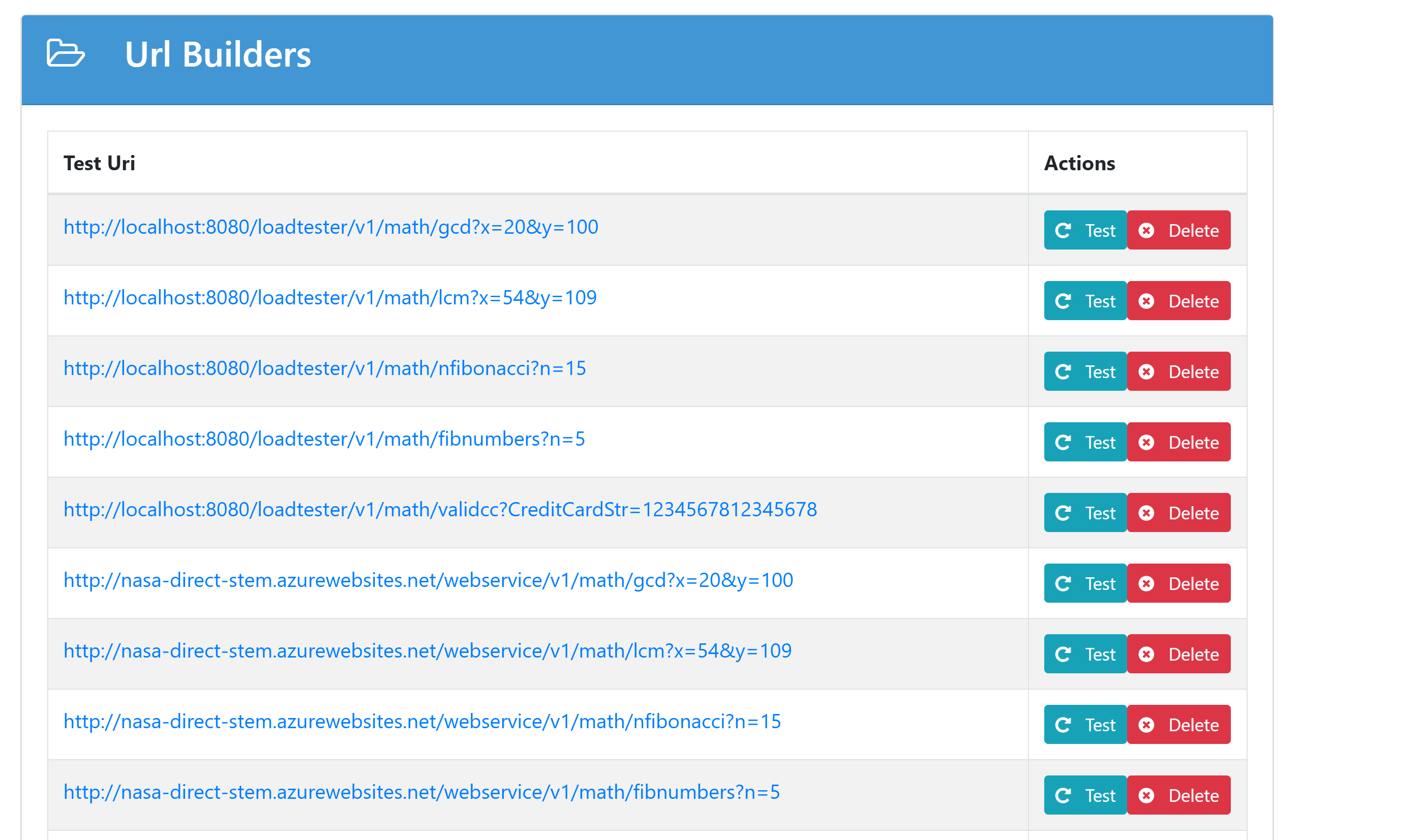
*Figure 24.* Project Comparison

**Managing Web Services**

The project also allows users to manage existing web services in the database. In the web services section, users can define new web services to be tested, and they can choose web services that have already been saved to be tested. There are two sections in this page. Web services with URL builders and web services without URL builders. Web services with URL builders manage webservices through WSDL discovery. The one without URL builders, manages webservices through a URI. Figure 25 shows the GUI for web service management via WSDL discovery. Figure 26 shows the GUI for web service management through a URI.



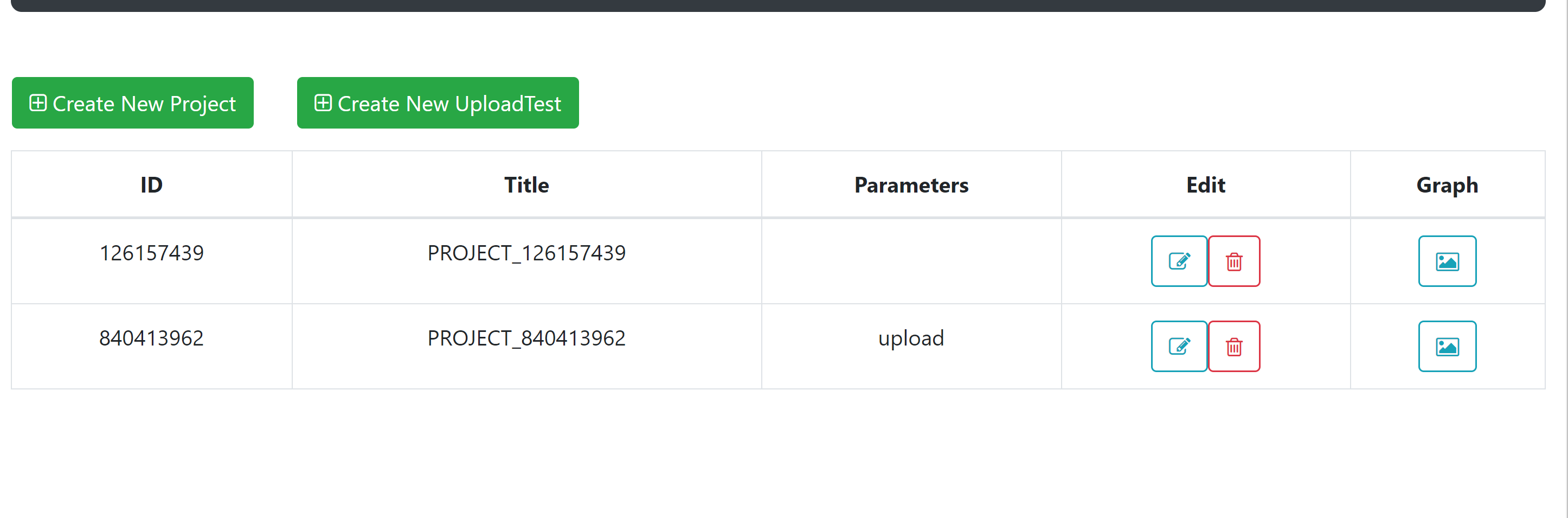
*Figure 25.* Web Services managed through WSDL Discovery



*Figure 26.* Web Services managed without WSDL discovery.

**Web Projects**

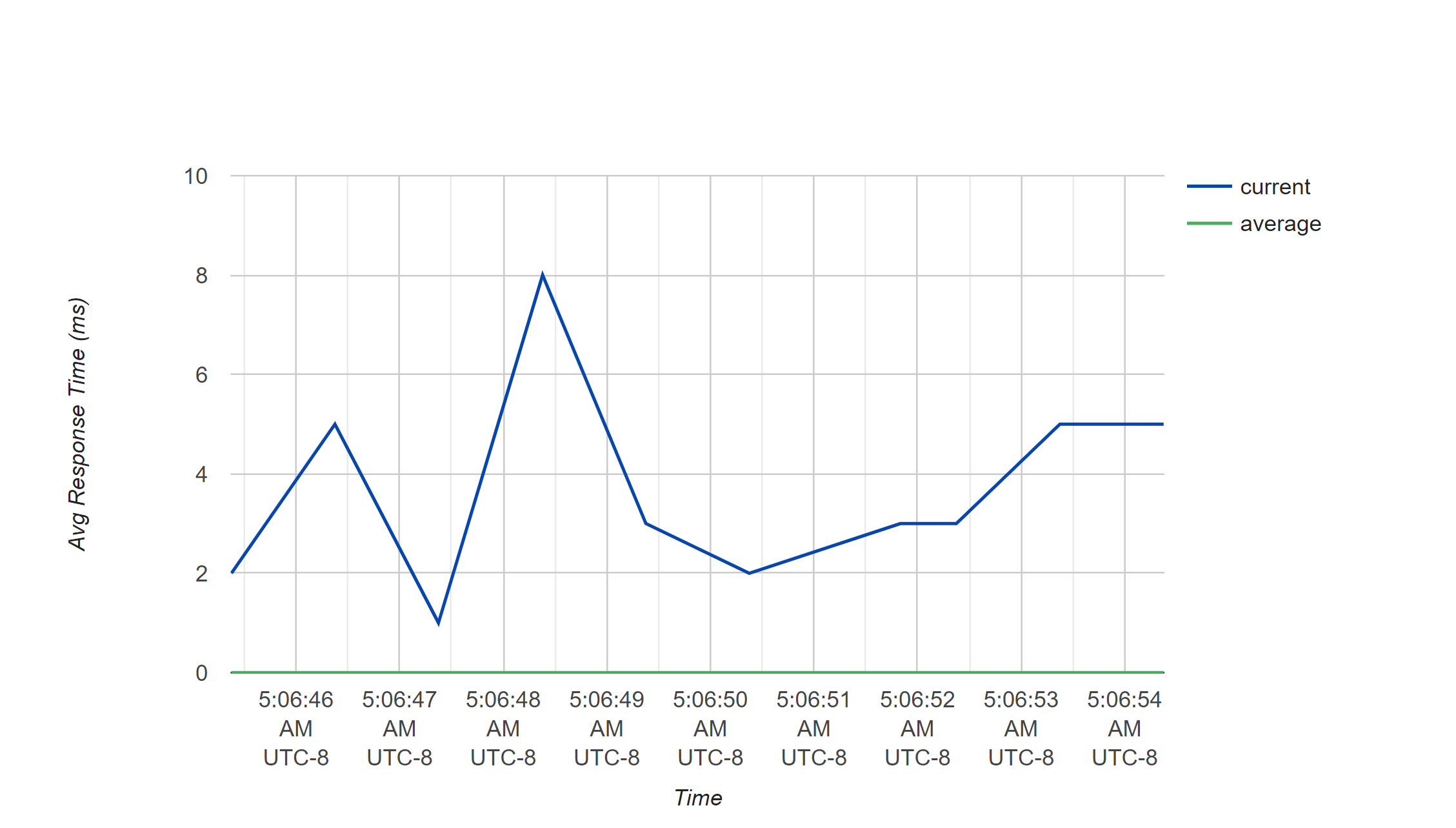
This section managed special web projects, such as upload tests, and measuring response rates between several web services or web sites. This section lists all the current web projects in the database, and it allows the user to test them and load them. The page is shown in *figure 27.*



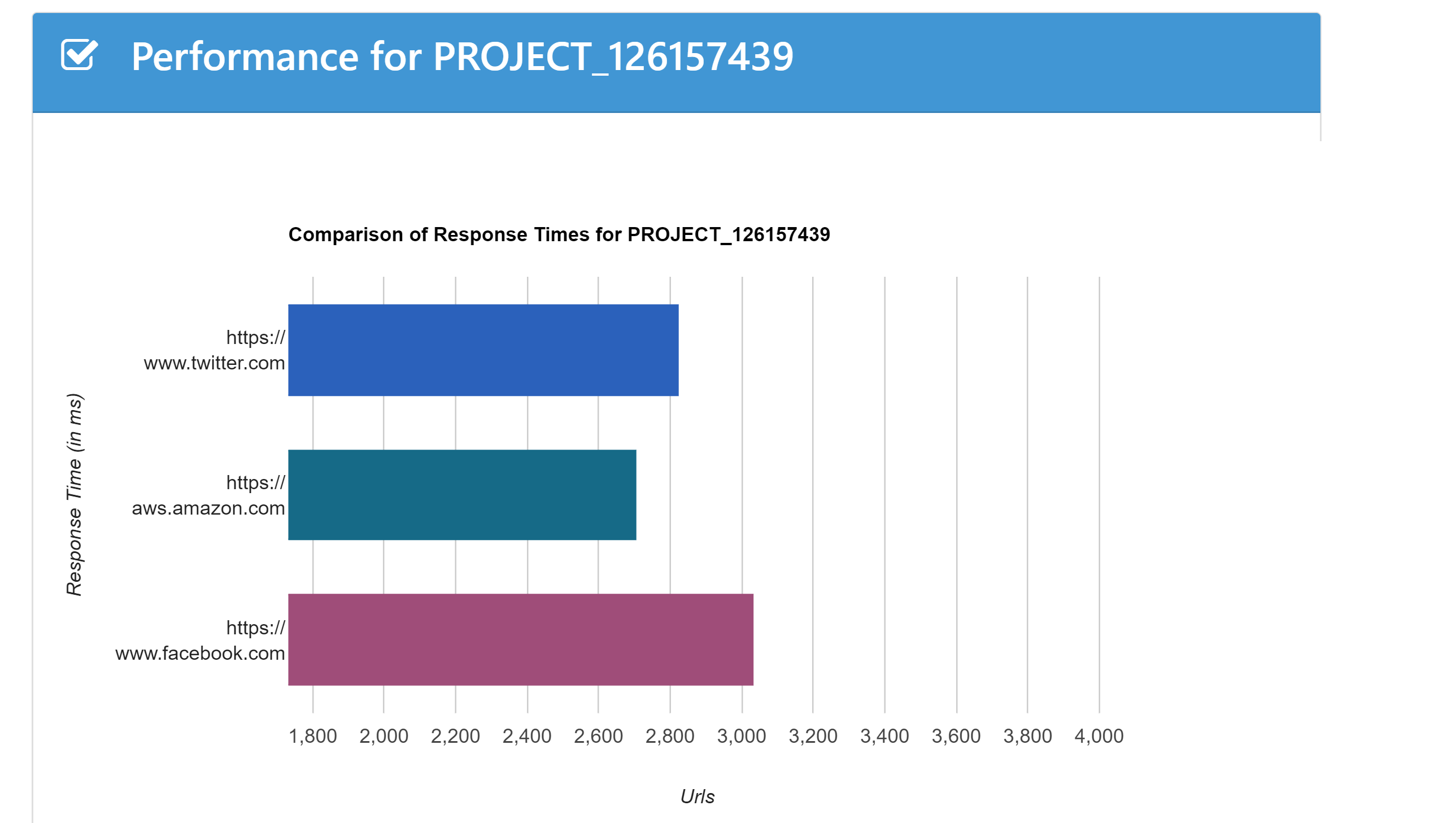
*Figure 27.* Web Projects Page.

Two kind of web projects exists. The upload test and the download test. The upload test is used to measure how long it takes for a server to upload a file of a certain size. It uses a simulated upload function to create a virtual file of a certain user specified size and uploads it multiple times, calculating how long it took for the file to get uploaded. Figure 28 below shows the results of an upload test.

The download test takes several web pages, or web services and compares the time it took for the information in URL to get downloaded. Figure 29 shows the result of a download comparison test.



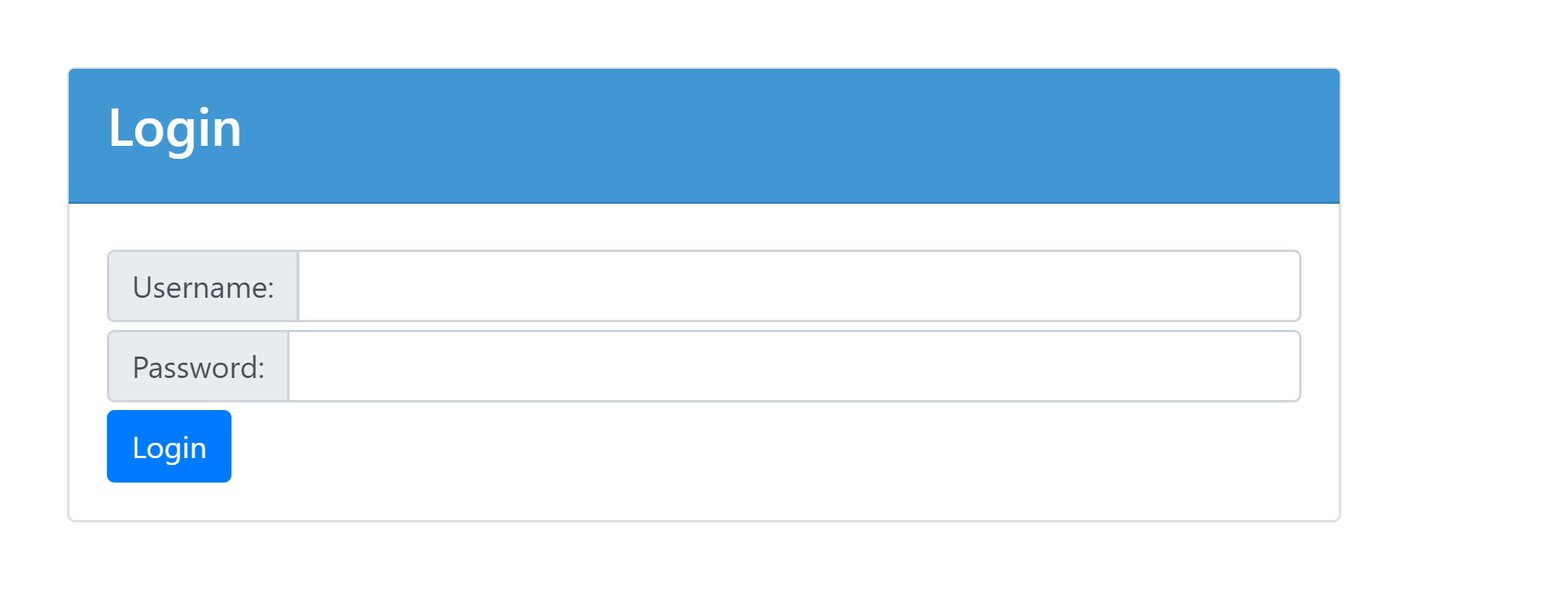
*Figure 28.* Upload Test.



*Figure 29.* Download Comparison Times.

# **Login**

The QoS applications allows users to login and monitor their web services. If a user is not logged in, the QoS application treats the user as a demo user. In that case their projects will be available to the public. Logged in users’ projects can only be viewed by them, making it more private. The login page is shown in figure 30.



*Figure 30:* Login page.

# CHAPTER FIVE

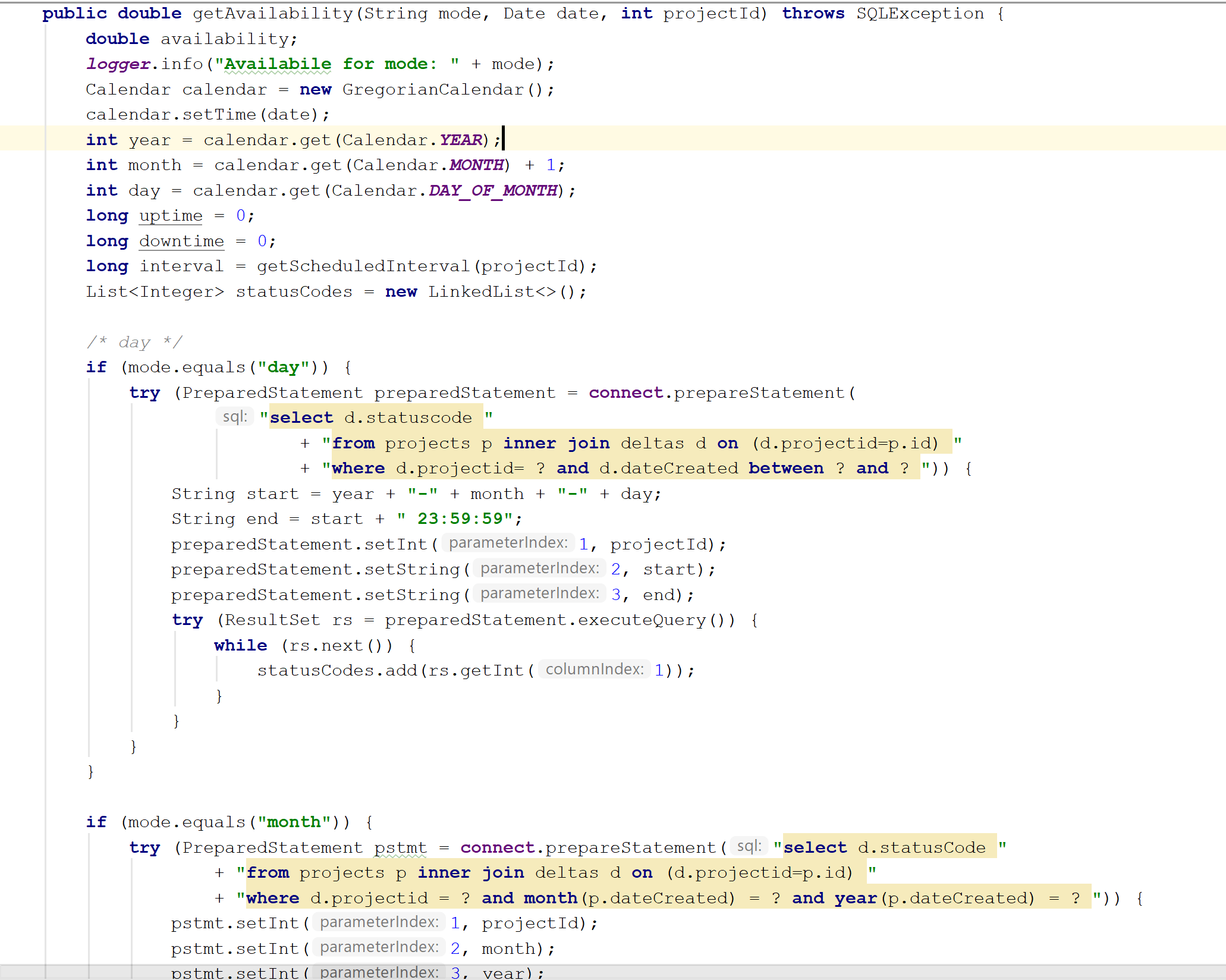
Measurements

This project measures the quality of a web service using five requirements. The availability of a service, how reliable it is i.e. reliability, the scalability of the service, the performance, and capacity.

**Availability**

Availability measures how often the service is online. It makes a series of requests to the service, and measures how many success statuses it received versus how many requests that didn’t go through. Ideally, the client should run this test everyday in order to get an accurate result. When starting the test, the user is given the options of setting up parameters for the test, such as the timeout and the time units. The graphic for the user’s options is shown in Figure 31. The code that calculates the availability for a web service is shown in Figure 32 and 33.

*Figure 31.* Availability Options

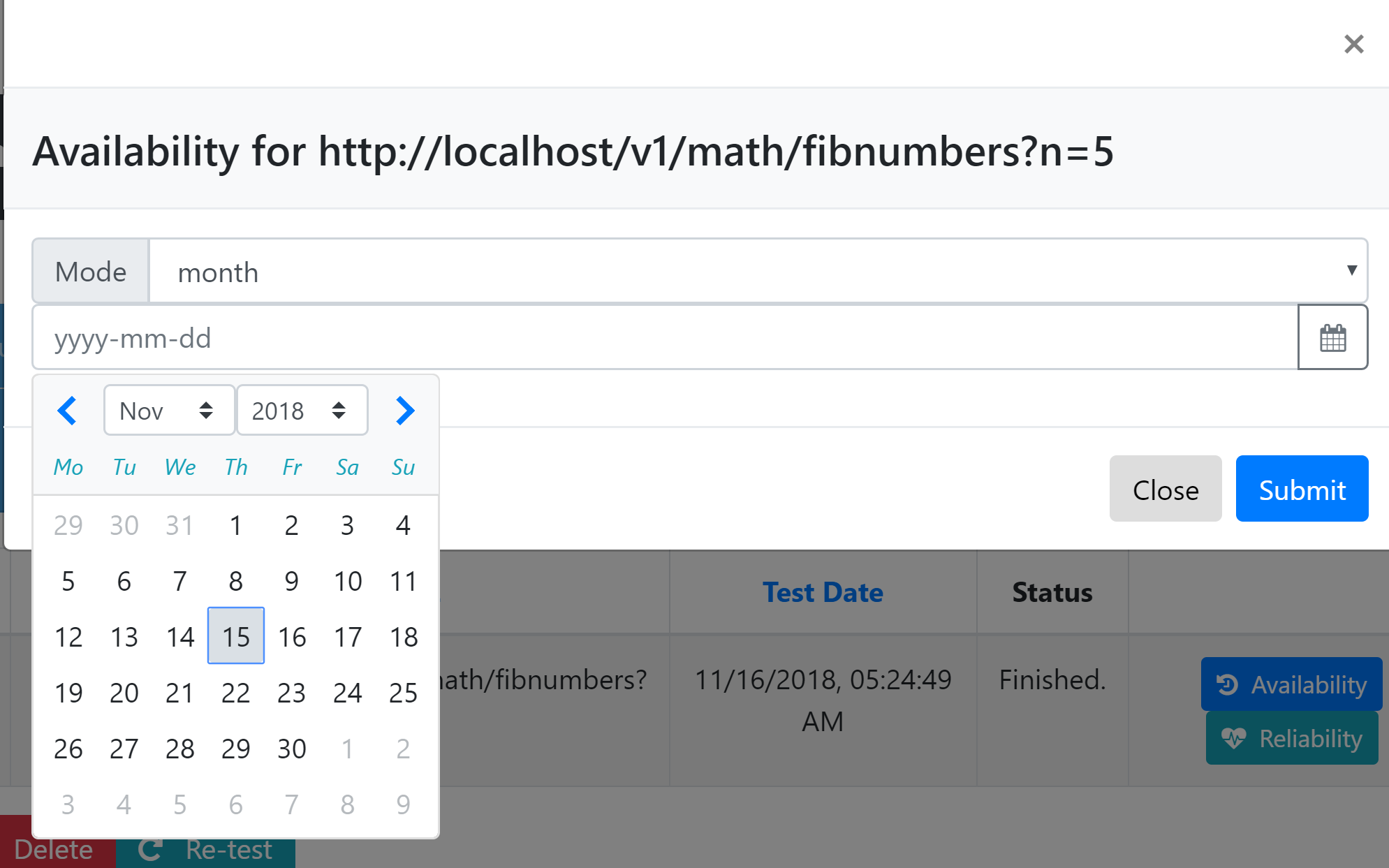


*Figure 32.* Get Availability (Part1)

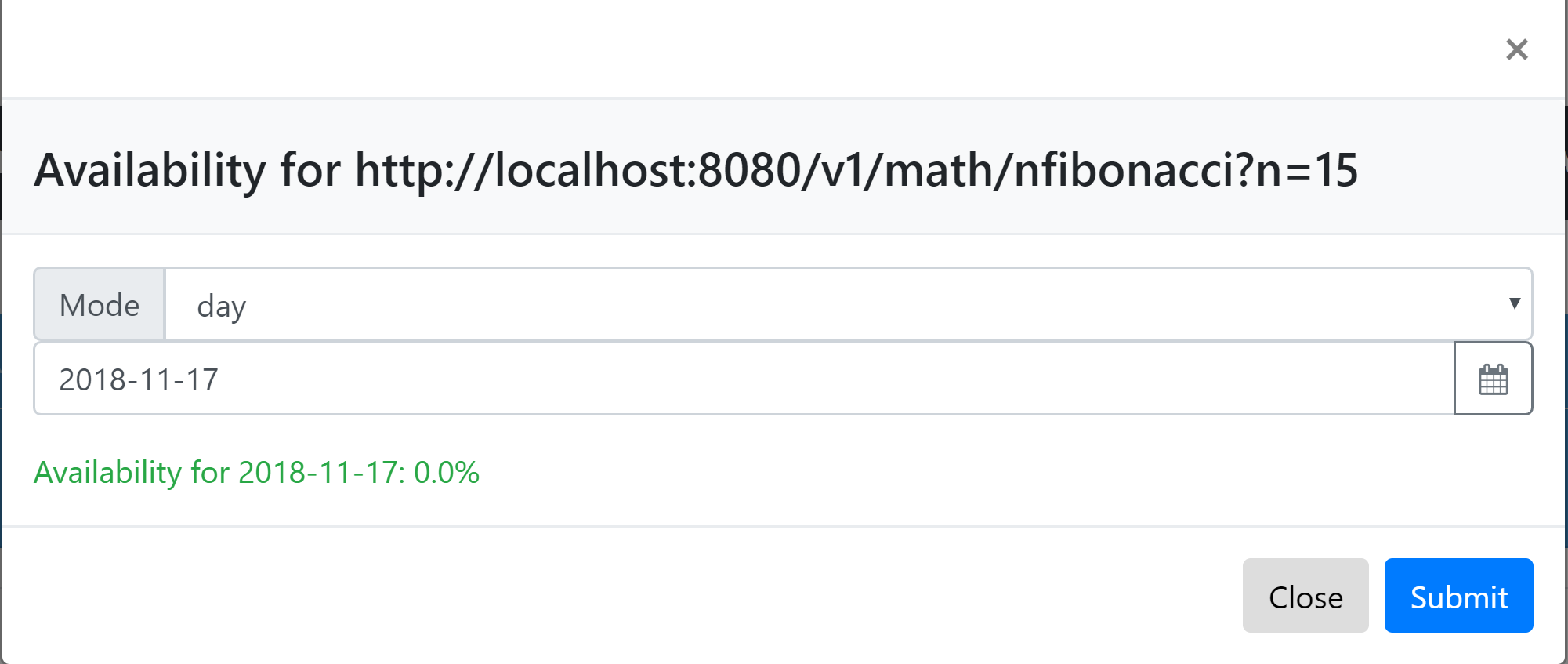


*Figure 33.* Get Availability (Part 2)

This method calculates the availability for a given year, month, or day depending on the value the user selected in the application. The user is presented with a graphic which asks what mode the user would like to test for availability. The graphic is shown in Figure 34. The method queries the database for all relevant status codes. Status codes of 200 represents a working service. All other status codes are assumed to indicate a non-functioning service. Status codes are added to a running sum of uptime or downtime accordingly. Lastly, the method computes the availability by dividing the sum of the uptimes by the sum of the uptimes and downtimes. The result the user is presented with is shown in Figure 35.



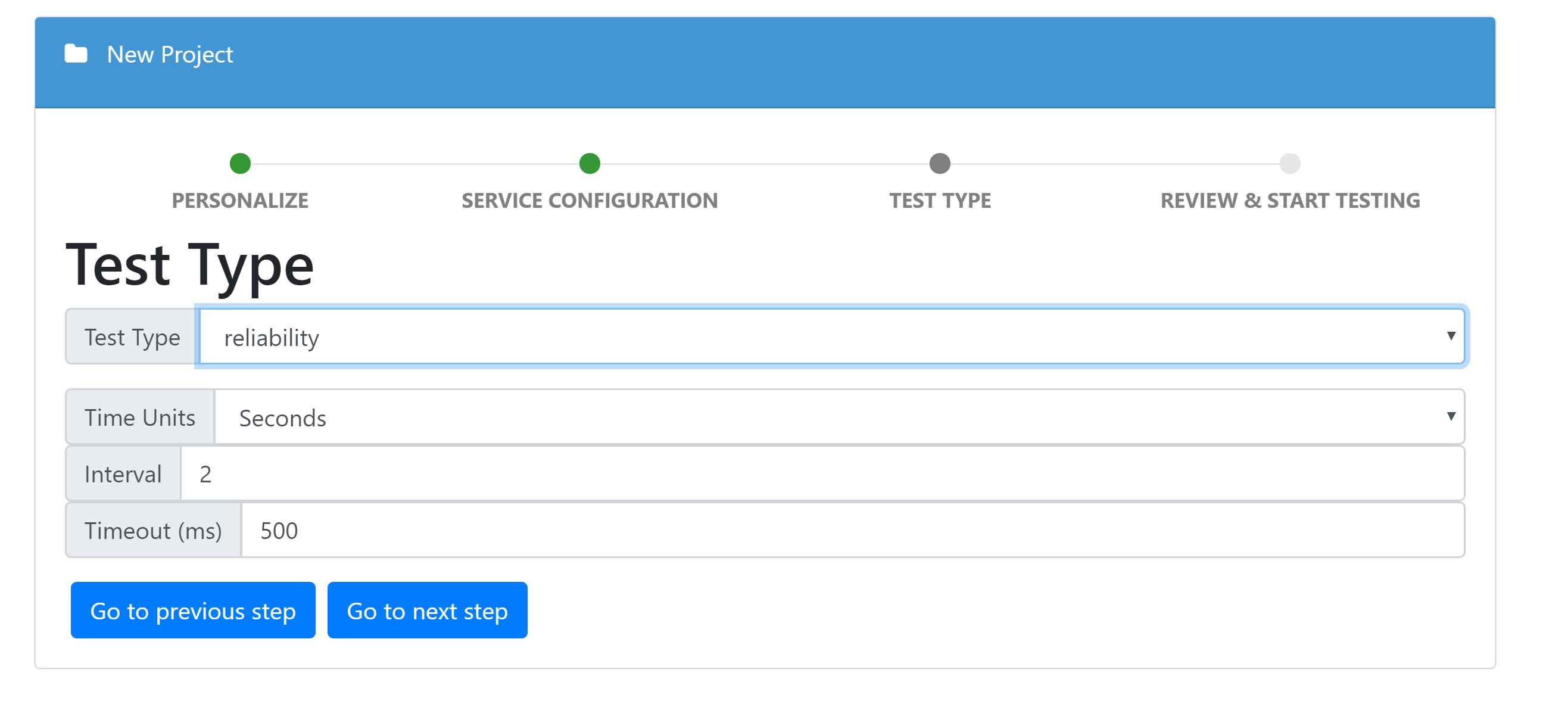
*Figure 34.* Options for Availability



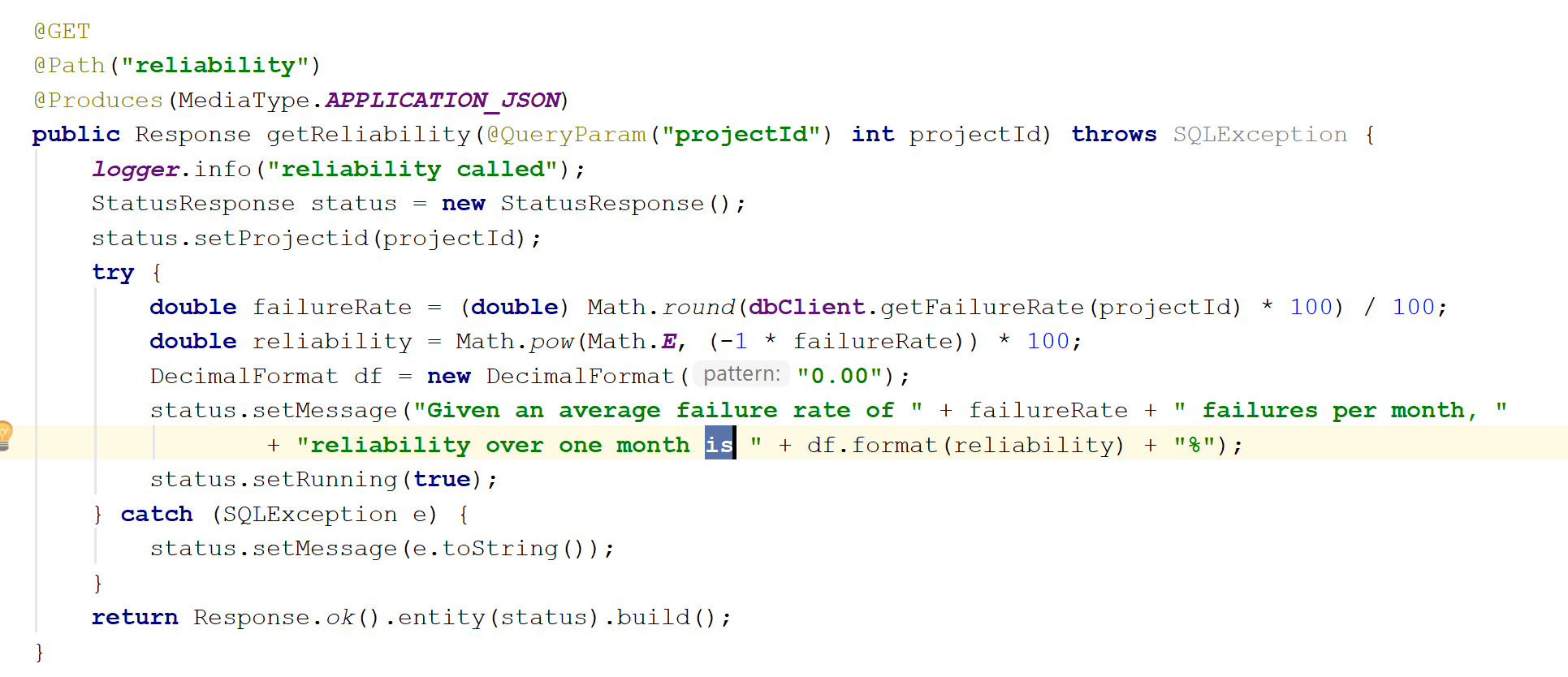
*Figure 35.* Result for Availability.

**Reliability**

This is important because is measures how likely it is a web service fails. Similar to availability, when starting the test, the user is given the options of setting up parameters for the test, such as the timeout and the time units. The graphic for the user’s options is shown in Figure 36. The code that calculates the reliability for a web service is shown in Figure 37.

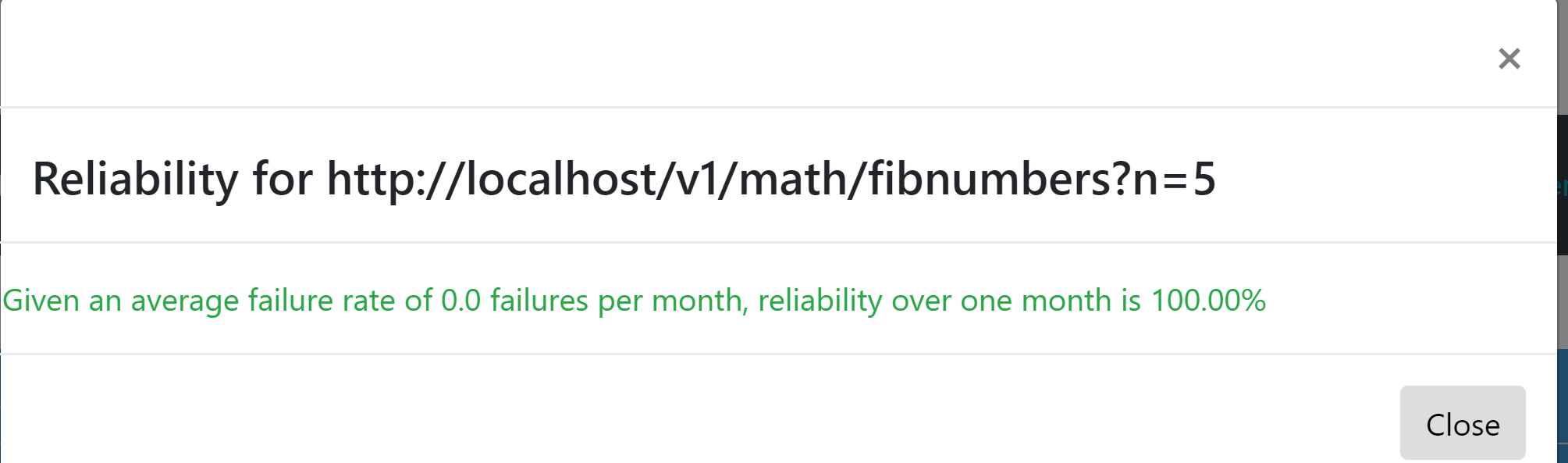


*Figure 36.* User Options for Reliability



*Figure 37.* Code Calculating Reliability

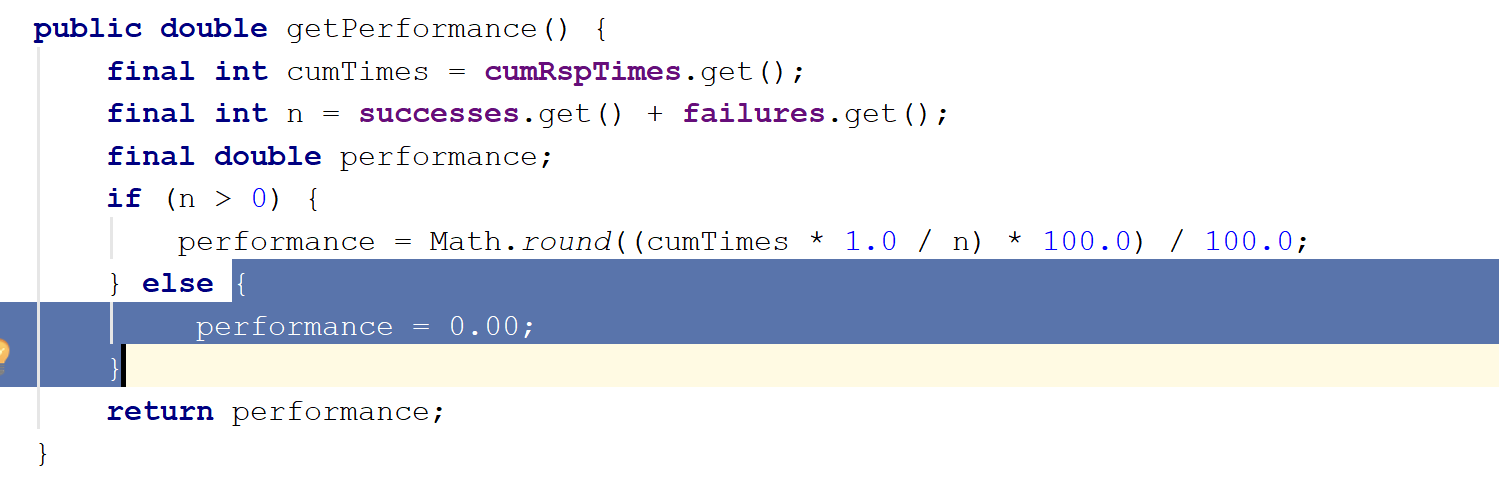
The reliability is calculated using Lusser’s equation. Reliability is always assumed over a specific time period. The method calculates the reliability over one month. This method counts the number of failures over all existing month and divides them by the number of months in order to get the rate of failure per month. It computes monthly reliability by taking the base of the natural logarithm, 𝑒, to the power of the negative value of the monthly failure rate. Figure 38 shows the interface displayed to the user when requesting for a target web service’s reliability.



*Figure 38.*  Reliability of a web service

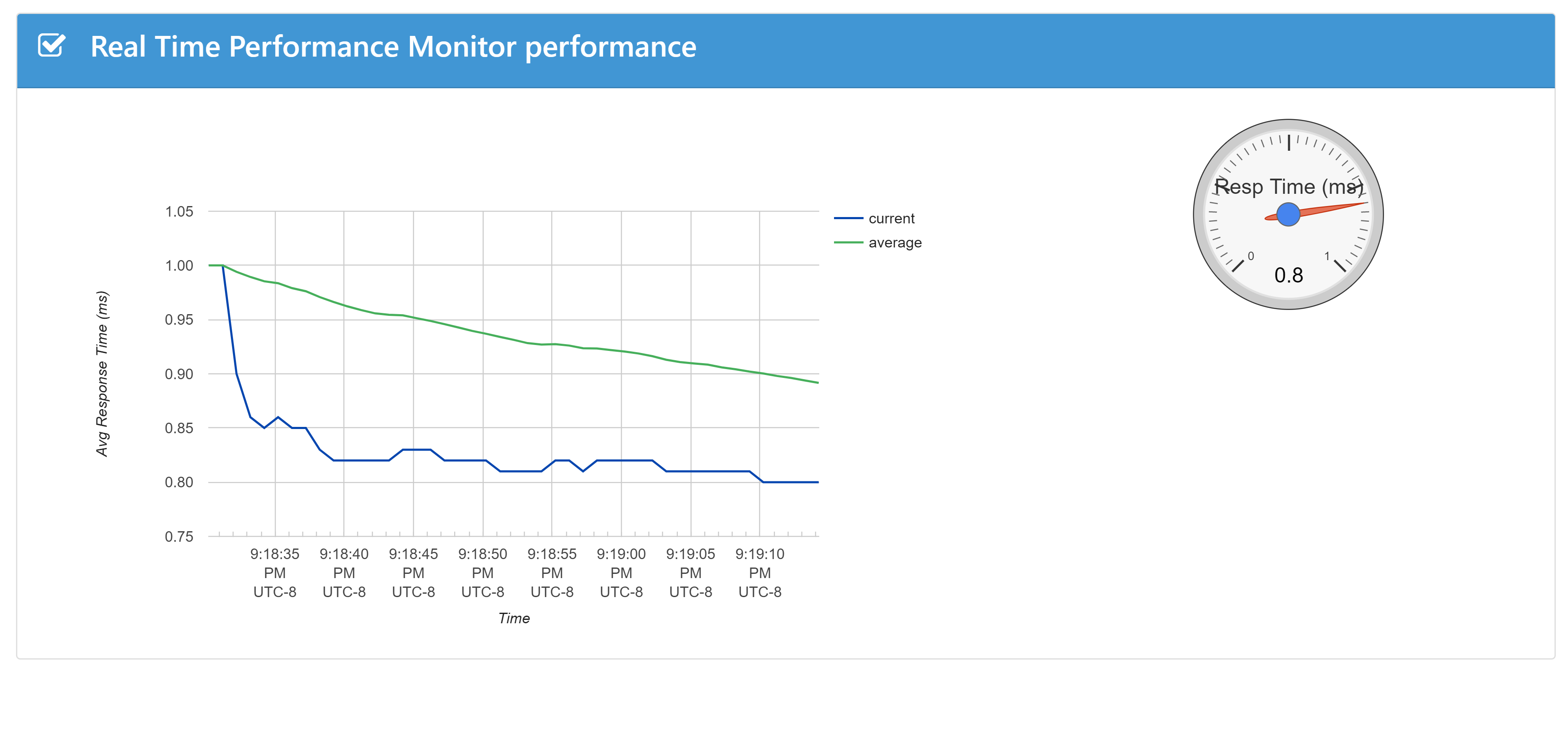
**Performance**

This measures how fast a web service responds, when a request is made to it. This is important because, a web service response time affects the smoothness and responsiveness of the application. The code showing how the response time is calculated in the application is shown in Figure 39.



*Figure 39.* Performance Code.

The method get the cumulative response times of the application, and divides it by the sum of number of success and failures, hence providing an average response time for the web service. The performance test display real time data when testing. The interface for the test is shown in Figure 40.

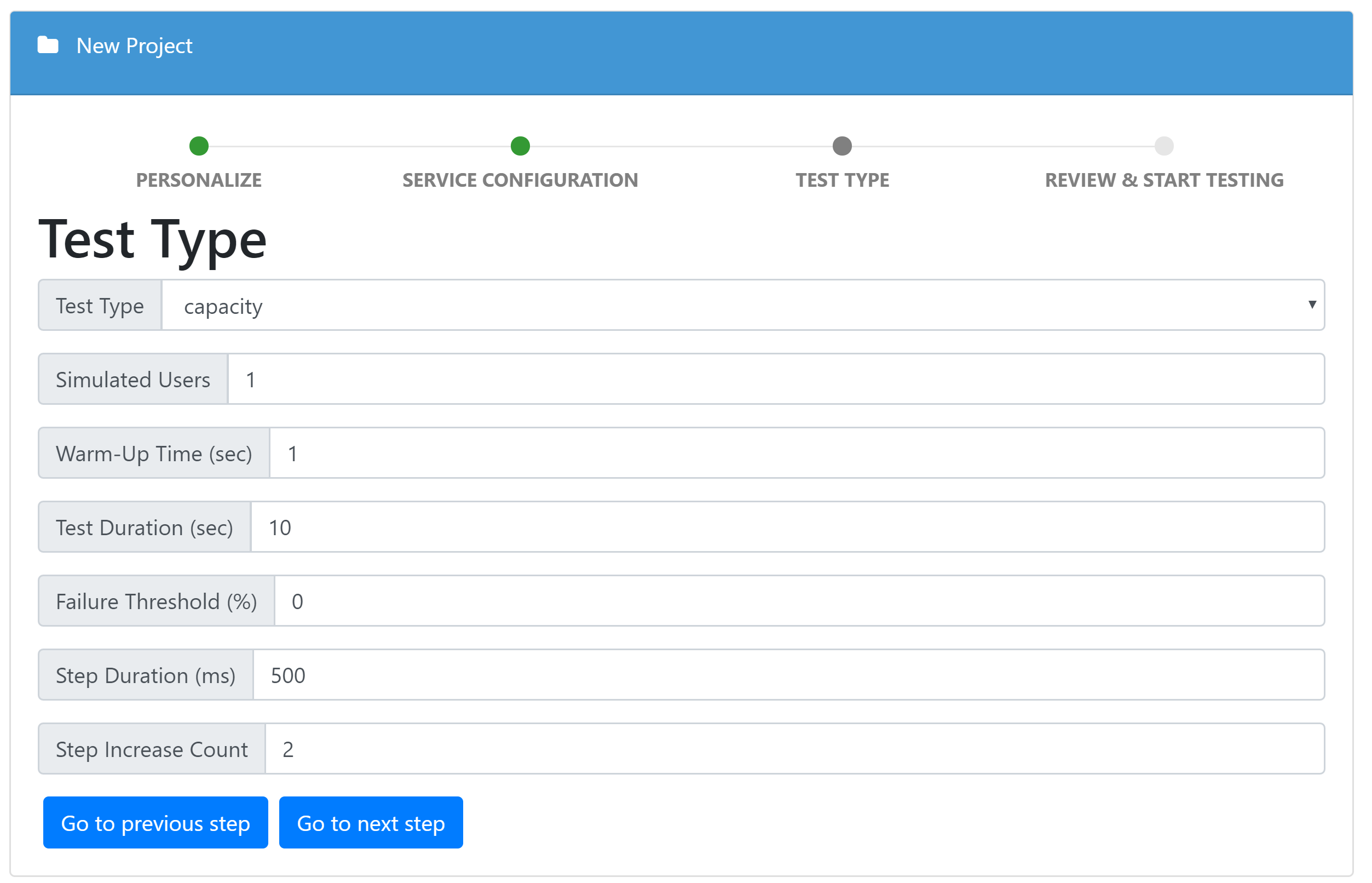


*Figure 40.* Real Time Performance Monitor

**Capacity**

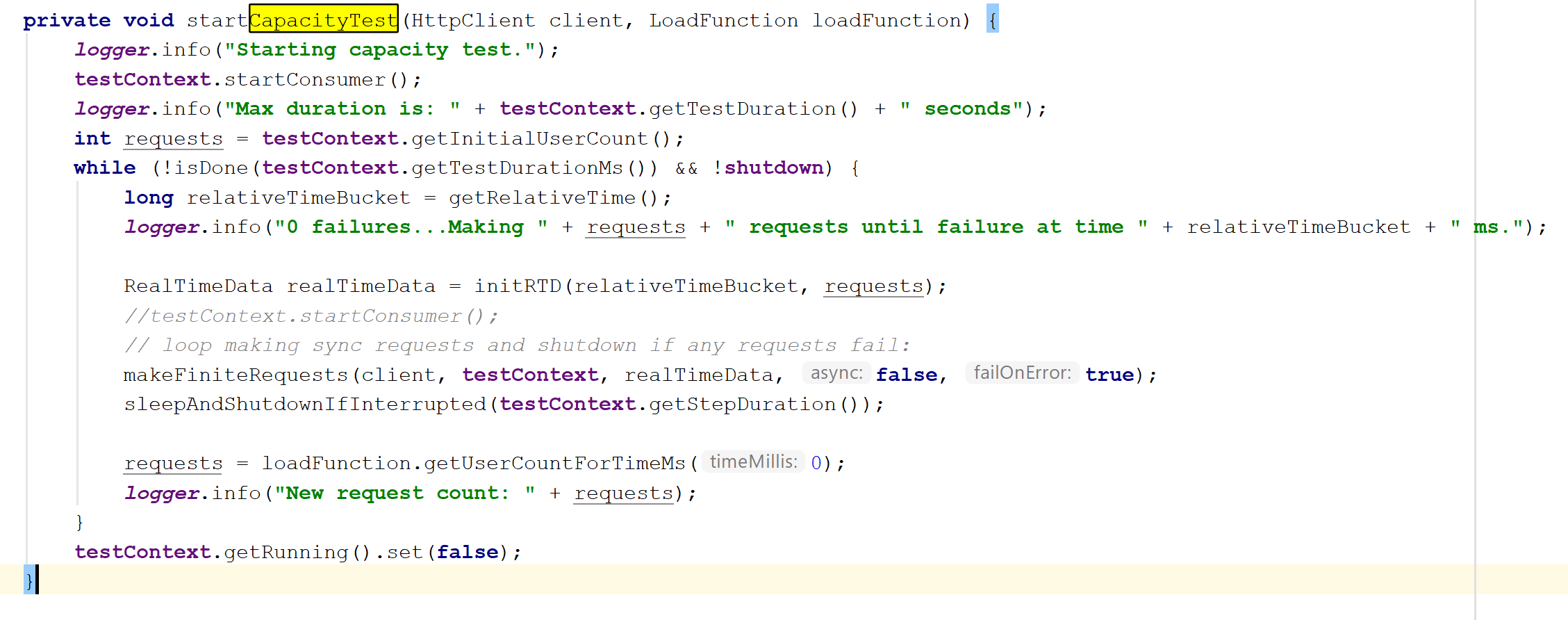
This measures how well a web service can handle multiple requests from several users. The motivation for measuring the capacity of a system is that web services have a limit on the number of requests they may process for a specific duration of time. By measuring the capacity of a system, we can determine the maximum throughput for a

system over a given period of time. The options for a capacity test are shown in Figure 41.



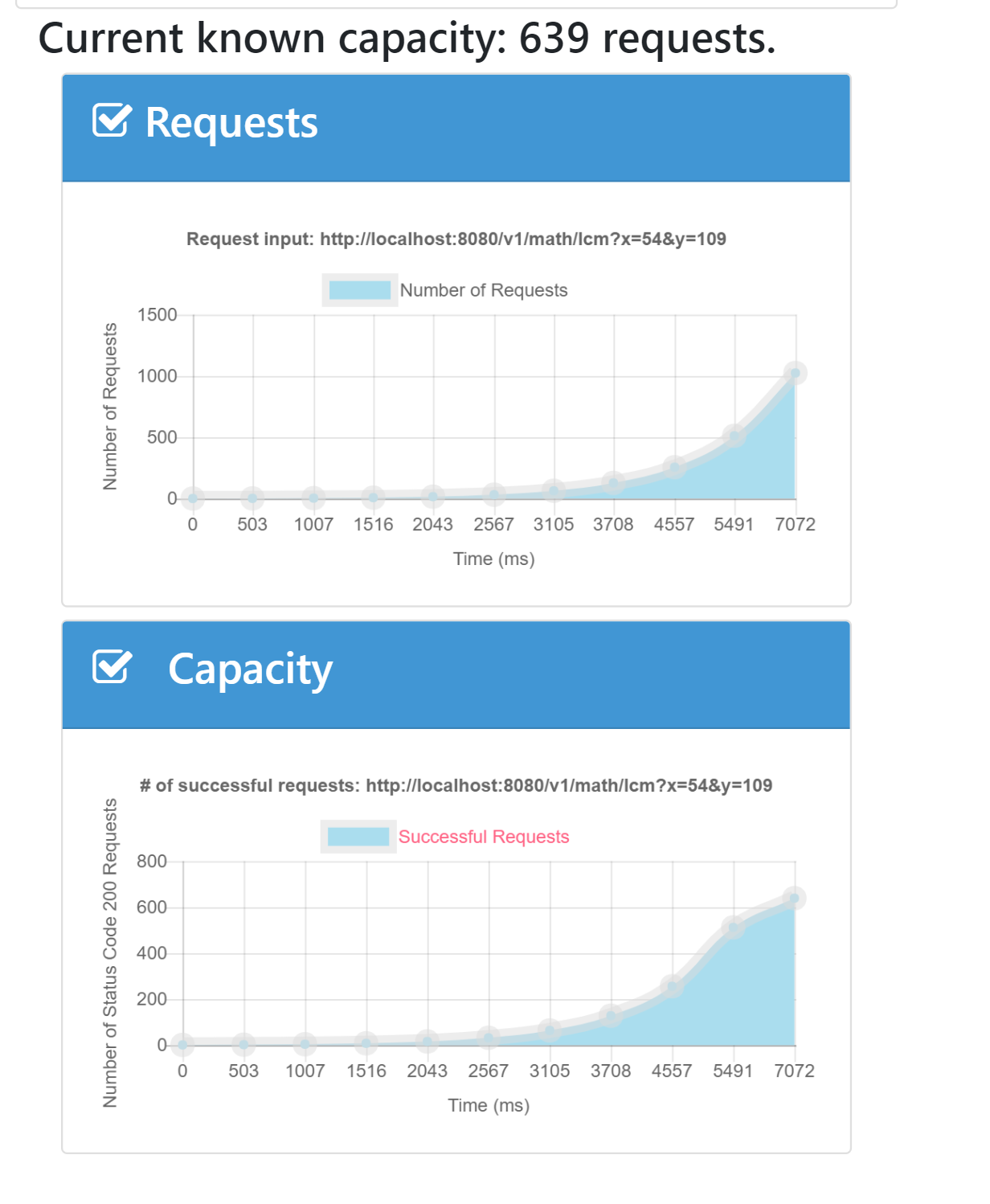
*Figure 41.* Capacity Test Options

The simulated users option stands for the number of users that the test will simulate in order to test the capacity. The warm up time indicate how many tests calls the application will make before testing. Test duration signals the length of time the test will take place. The step duration and step increase count indicates how long it will take to increase the number of requests and how many requests to decrease it by. The code that calculates the capacity is shown in Figure 42.



*Figure 42.* Capacity Code.

The method calculates the capacity by continuously making several requests and compiling the response time, and the time of requests, until the simulated number of calls have been made or the web service shuts down. The test gives back data in real time. The interface for the real time data is shown in Figure 43.



*Figure 43.* Capacity Results Graph

**Scalability**

This measures the change in the performance of a web service as the load of request increases. It is the ability to add capacity to a system so the system can support additional load from existing users or from an increased user-base. In order to measure this phenomenon, the application uses three load functions, a Normal distribution, Poisson distribution, and a Uniform distribution.

A Poisson distribution is a probability distribution resulting from a Poisson experiment [7]. This measures the average number of successes within a given region. The formula for a Poisson experiment is given below.

where *x* is the actual number of successes that result from the experiment, and *e* is approximately equal to 2.71828.

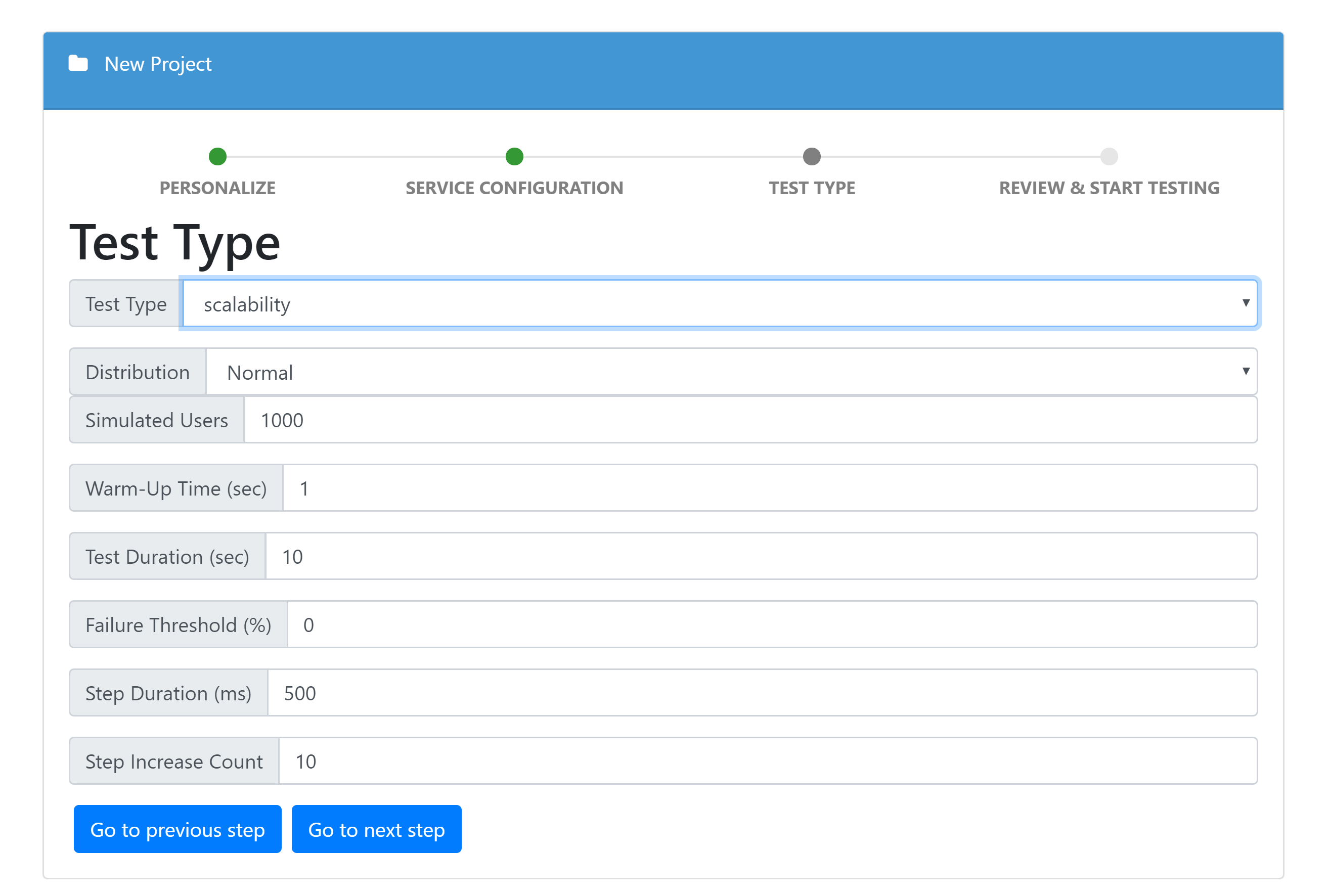
A normal distribution refers to a family of continuous probability distributions described by the normal equation [8]. It is defined by the following equation.

where *X* is a normal random variable, μ is the mean, σ is the standard deviation, π is approximately 3.14159, and *e* is approximately 2.71828.

A uniform distribution refers to a probability distribution for which all the values that a random variable can take on occur with equal probability [9]. The equation for a uniform distribution is described below

Where X is a random variable that can take on k different values.

The user can choose between these three distributions in the application. The graphical interface for scalability options is shown in Figure 44.

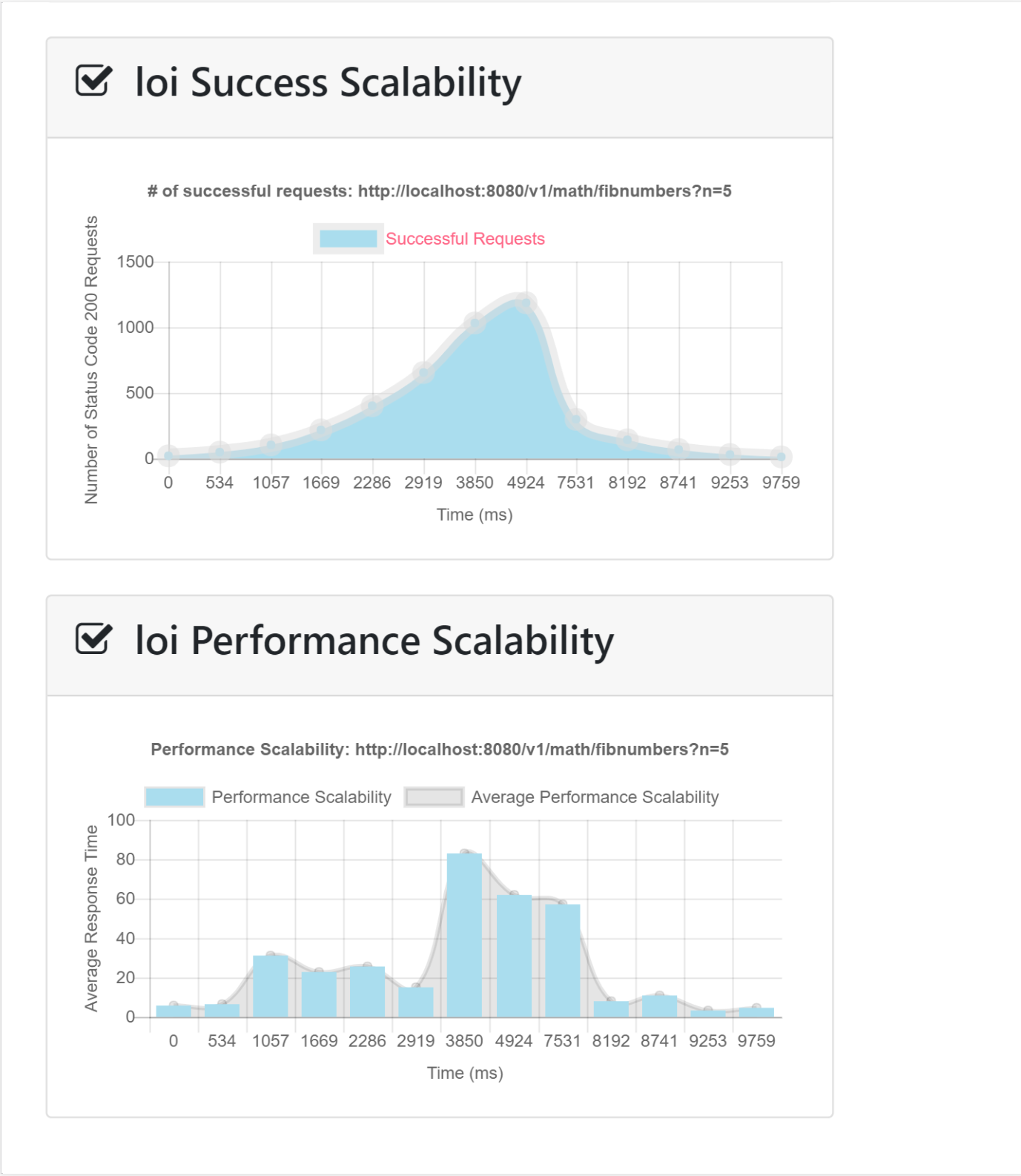


*Figure 44.* Scalability Options

Similar to a capacity test, the simulated user stands for number of users that are simulated in calling the web service. The warm-up time indicated the time it would take to begin testing. The test duration describes how long the test will run for, Failure threshold, indicated how many times a service call can fail before the test is terminated. The step duration and step increase count indicates the interval between increasing the number of calls per user, and how many calls per user the application increases. The code for calculating scalability is shown in Figure 45.

*Figure 45.* Scalability Code

The load function is passed in this function. Java interfaces for Poisson, Uniform and Normal distributions are used to calculate the average requests to be made, given the time the service call started, and the performance and successes are calculated given the time bucket, and the requests from the probability distribution. The data is given back to the user in real time. The performance scalability which is the performance of the service for each call, and the success scalability, which measures the number of successful requests per service call. The interface is shown in Figure 46.



*Figure 46.* Scalability Graph

CHAPTER SIX

Conclusion

The design and implementation of a Quality of Service application for cloud computing services is essential for measuring the performance of web applications and web services. This allows developers to test their applications and come up with informed ways to improve them and make them more scalable. The application also allows users to organize their services and applications, and test them simultaneously, in order to get a more wholesome view of the quality of service of their applications.

# REFERENCES

1. R. Collins, “Lusser's Law,” *The American Spectator*, 14-July-2003. [Online]. Available: <https://spectator.org/51313_lussers-law>. [Accessed: 02-Nov-2018].
2. A. Man et al., “Understanding quality of service for Web services,” *IBM,* 01-Jan-2002. [Online]. Available: <https://www.ibm.com/developerworks/library/ws-quality/index.html>. [Accessed: 02-Nov-2018].
3. B. Sletten et al., “Foundations of a RESTful Architecture,” *DZone*, [Online]. Available: <https://dzone.com/refcardz/rest-foundations-restful>. [Accessed: 07-Nov-2018].
4. H. Terek, “Building REST Web Services with Jersey,” *Programmer’s Café,* 03-Oct-2017*.* [Online]. Available:<https://www.programmergate.com/building-rest-web-service-using-jersey>. [Accessed: 08-Nov-2018].
5. B. Goetz, “Thread Pools and Work Queues,” *IBM,* 01-July-2002. [Online]. Available: <https://www.ibm.com/developerworks/library/j-jtp0730/index.html>. [Accessed: 09-Nov-2018].
6. D. Sharad, “What Is a Service in Angular and Why Should You Use it?” *DZone,* 02-Feb-2018. [Online]. Available: <https://dzone.com/articles/what-is-a-service-in-angular-js-why-to-use-it>. [Accessed: 17-Nov-2018]
7. “Poisson Distribution,” *Stat Trek,* [Online]. Available: <https://stattrek.com/probability-distributions/poisson.aspx>. [Accessed: 20-Nov-2018]
8. “Normal Distribution,” *Stat Trek,* [Online]. Available: <https://stattrek.com/probability-distributions/normal.aspx>. [Accessed: 20-Nov-2018]
9. “Uniform Distribution,” *Stat Trek,* [Online]. Available: <https://stattrek.com/statistics/dictionary.aspx?definition=uniform_distribution>. [Accessed: 20-Nov-2018]