A JavaScript Framework Comparison Based on Benchmarking Software Metrics and Environment Configuration

Jefferson Ferreira

A JavaScript framework comparison based on benchmarking software metrics and environment configuration

Jefferson Ferreira

BSc, Computer Science, Universidade Veiga de Almeida, 2012

A dissertation submitted in partial fulfilment of the requirements of
Dublin Institute of Technology for the degree of
M.Sc. in Computing (Advanced Software Development)

January 2018

I certify that this dissertation which I now submit for examination for the award of MSc in Computing (Knowledge Management), is entirely my own work and has not been taken from the work of others save and to the extent that such work has been cited and acknowledged within the test of my work.

This dissertation was prepared according to the regulations for postgraduate study of the Dublin Institute of Technology and has not been submitted in whole or part for an award in any other Institute or University.

The work reported on in this dissertation conforms to the principles and requirements of the Institute's guidelines for ethics in research.

Jefferson Luiz da Conceição Ferreira

Signed: ___________________________________

Date: **03 January 2018**

ABSTRACT

JavaScript is a client-side programming language that can be used in multi-platform applications. It controls HTML and CSS to manipulate page behaviours and is widely used in most websites over the internet. JavaScript frameworks are structures made to help web developers build web applications faster by offering features that enhance the user interaction with the web page. An increasing number of JavaScript frameworks have been released in recent years in the market to help front-end developers build applications in a shorter space of time. Decision makers in software companies have been struggling to determine which frameworks are best suited for a specific project. This work investigates the actual state-of-the-art of JavaScript framework comparison, and it proposes metrics and methods that could help developers when choosing a JavaScript framework. In this work, a benchmark framework executes tasks to test the efficiency of three JavaScript frameworks (AngularJS, Aurelia, and Ember). The research shows the impact of the environment (CPU usage and network connectivity) on JavaScript frameworks.

Keywords: *JavaScript, JavaScript framework comparison, performance testing, benchmarking, test environment, JavaScript framework adoption*

ACKNOWLEDGEMENTS

I would like to express my sincere thanks to my supervisor Dr John Gilligan who was extremely patient, attentive with me; always suggesting improvements that challenged my knowledge and enhanced my confidence in my technical skills.

I would also like to thank my family, especially my mother who was always supportive in my decisions and gave a lot of strength to pursue this degree.

Thanks to all my friends who were always there for me in every difficult and stressful moment that I went through during the course.

TABLE OF CONTENTS

TABLE OF FIGURES

TABLE OF TABLES

1 INTRODUCTION

This project intends to use the JavaScript framework (JSF) artefact to determine metrics and environment settings in a performance comparison. Choosing the ‘right’ JSF has become a big challenge among front-end developers and this work address the issues that most developers face when choosing a software for their projects. This work uses benchmarking as a method for assessing JSFs. This research extends Mariano’s (2017) work which investigated the role of benchmarking in JavaScript frameworks. He defined benchmarking as an appropriate method for assessing JSFs.
JavaScript frameworks differ from each other in number of features they provide, community support, architecture, and size. Metrics related to memory and security can also me added to the comparison of JSFs. This work aims to determine the influence of the configuration of the environment on the performance of JSFs in JavaScript applications. This research will explore the execution of JavaScript applications in different Operating Systems and different networks to assess the affects of the environment.

1.1 Background

JavaScript is a widely-known programming language which can be used in multi-platform applications (Mariano, 2017) and just like other languages such as Java and Python, JavaScript has been improved and is gaining more relevance on the web. JavaScript is still one of the most popular programming languages in 2017 as the tendency for applications to be transferred to web platforms only increases (Gizas, Christodoulou, & Papatheodorou, 2012a).
As expected, a number of plug-ins, frameworks and libraries were created to work with JavaScript, to facilitate the use of this language in everyday tasks of the web developer. In recent years, dozens of JavaScript frameworks have been released on the market to help these front-end developers build applications quickly (Gizas et al., 2012a). Most of these frameworks are open source, and some of them have stood out to become a fundamental part of several projects, due to their functionalities, being integrated with great tools (Graziotin & Abrahamsson, 2013).
However, choosing the ‘right’ tools is not an easy task. Developers tend to seek out better and faster solutions which raises the question: Which framework is the ‘best fit’

for a project (O. Hauge, T. Osterlie, C. F. Sorensen, & M. Gerea, 2009)? Studies have been conducted to test the performance of JavaScript applications (Ratanaworabhan, Livshits, & Zorn, 2010) and the results obtained have significantly contributed to this field of research, but as the technology advances over time, these studies become out-dated.

JavaScript framework comparison is a growing area (Ratanaworabhan et al., 2010), and this field of research has gaps that need to be filled. Thus, the performance and integrity of JavaScript applications using JavaScript frameworks are the primary motivation for this research project.

1.2 Research Project/problem

Developers face a challenge when choosing the right JavaScript framework because of the extensive variety of tools and frameworks available. Methods of evaluation and validation of software (P. Miguel, Mauricio, & Rodríguez, 2014) have been developed during the past few years to measure the quality of Open Source Software (Barkmann, Lincke, & Löwe, 2009). Decision makers in companies face challenges when choosing a particular open source software (OSS); these challenges can relate to to product selection (e.g. too much choice, lack of time to evaluate the product and product version), product support (e.g. documentation, community, and maintenance of the product) which includes uncertainty about the product’s future and dependency for future support (Stol & Ali Babar, 2010). Other challenges are integration and architecture (backward compatibility issues, need for modification, component and architecture incompatibilities) as well as migration and legal issues (complex licensing and lack of precise business model) (Stol & Ali Babar, 2010). The main concerns when choosing an OSS are the frequent changes that come from the fast-growing nature of the marketplace, lack of standards to assess and describe those OSSs as well as product reputation (Ayala et al., 2009).

Experiment has been conducted to demonstrate available tools for in-browser network performance measurement in different browsers and platforms (Gizas, Christodoulou, & Papatheodorou, 2012b). This research showed the importance of proper methodology and accurate tools when measuring network and software performance. The information obtained through this research will help decision makers choose the right tool when developing software for any potential projects (Horký, Libič,

Steinhauser, & Tůma, 2015). JavaScript has been evolving, and several JSFs have been released in the past years to help developers achieve faster and better results for the development of an application (Graziotin & Abrahamsson, 2013). Each JavaScript framework claims to provide unique benefits and advantages over its competitors. Developers need to be aware of the features that each framework offers and performance gains when using a specific framework.
Therefore, this research will attempt to answer the following research question:

Does the environment configuration affect the performance of JavaScript frameworks in JavaScript applications?

1.3 Research Objectives

This research aims to determine metrics and methods of evaluation that best suit a JavaScrpit framework comparison. The results of this research will enhance the decision-making for developers and researchers when choosing a JavaScript framework by using a different performance evaluation. Performance in the context of this research is the execution time of a given task. Therefore, time is the primary object of this measurement. Environmental factors impact performance results, e.g. concurrent processes and network stability. This project explores factors that could alter the results of performance in JavaScript frameworks.
The results of each experiment will give an insight to how this analysis should be conducted and which metrics are essential for analysis when external influence acts over these frameworks. JSFs are client-side applications which means results may differ from one JSF to another, depending on the environment in which the application is being executed. This research aims to establish a controlled environment where only the critical processes are being executed and competing for the CPU usage.
The objectives of this research are:

1. To investigate the actual state-of-the-art of JavaScript frameworks and more specifically JavaScript framework comparison.
2. To define metrics and environmental standards for the experiment to be conducted.
3. To configure the environment where the experiment will be conducted according to the chosen defined standards.

4. To develop an experiment in compliance with the chosen metrics to evaluate the selected JavaScript frameworks.
5. To document findings and the results of the evaluation process from the experiment.
6. To critically analyse the results giving an overall view of the performance obtained from each JSF
7. To suggest improvements to this project through recommendations for future research in this area.

1.4 Research Methodologies

Secondary research (desk research) will be carried out as part of this thesis to summarise and synthesise existing researches done in this area. Given the structured and data-driven approach to the experiment, the following step is to conduct quantitative research based on the data collected from it. This is a quantitative research where systematic empirical investigation and statistical analysis of the data collected will be carried out.

1.5 Scope and Limitations

This research aims to compare JavaScript frameworks and use software performance metrics to generate data for this comparison. The literature review showed a wide variety of JavaScript frameworks available for developers; only three frameworks were selected for this comparison due to time constraints. The frameworks are AngularJS, Aurelia, and Ember.

The literature review also revealed different types of performance metrics which include spike testing and stress testing. Benchmarking was the method chosen for measuring performance in this project as Mariano’s (2017) findings have proven that it is the most suitable approach to take when measuring performance in JavaScript frameworks.

The benchmark tool used in this research will assess the execution time of JavaScript in different JavaScript frameworks. This application requirement only involves the use of the browser ‘Google Chrome’. Future implementations of this application will integrate other browsers such as Mozilla Firefox and Microsoft Edge. Therefore, the results of this research are collected from one browser.

JSFs are client-side applications, and this research aims to compare the execution of JavaScript applications in different environments using different JSFs. The benchmarks will run on different operating systems on the same machine. Two operating systems were selected for this comparison (Windows and Linux) because both systems can use the same machine without interfering with each other's performance.

The comparison also aims to compute the execution time of database operations in a cloud-based web server such as the AmazonAWS and Microsoft Azure. Google Cloud Platform was the cloud-based server used for this research because it offers a structure to host Node.js based application and relational database management system (RDBMS) servers. MySQL is the RDBMS chosen for running this experiment.

1.6 Document Outline

The dissertation is structured as follows:

- Chapter 2 is a research of the literature already conducted in the field of JavaScript framework comparison. This section presents the actual state-of-the-art of this research field with its limitations and areas to be explored. The literature review also presents a brief history and definitions of benchmarking and JavaScript frameworks.
- Chapter 3 contains the design and methodology of this experiment. This section describes the benchmark design and its development.
- Chapter 4 contains the description of the software used as well as the environment specification and configurations to conduct the experiment.
- Chapter 5 illustrates the results and analysis of the experiment. It also presents a discussion and interpretation of these results and findings.
- Chapter 6 summarises the research and concludes with an overview of the work done throughout the project. It also suggests future work and recommendations.

2 LITERATURE REVIEW

2.1 Introduction

As the aim of this project is to evaluate comparison results from different JavaScript frameworks using benchmarking metrics, this chapter gives a brief introduction to JavaScript and JavaScript Frameworks. The history of JSFs and a feature comparison between them are also described in this chapter. The actual state-of-the-art of benchmarking JavaScript frameworks is also discussed in this chapter with a brief presentation of benchmarking and its importance for decision makers of software projects. Benchmarking metrics and the importance of a precise clock to make speed comparisons are also discussed in this chapter. This chapter also discusses the difference between Virtual Machines (VMs) and Dual-Boot with an impact overview on choosing one of those methodologies.

2.2 JavaScript and JavaScript frameworks

JavaScript is the most used programming language for front-end web developers (Mariano, 2017). This programming language has an undeniable popularity given its overwhelming use across most modern websites and all modern web browsers (Gizas et al., 2012b). One of the critical features of the JavaScript language is that modern web browsers contain an inbuilt interpreter for JavaScript codes that can parse and execute the language (Mariano, 2017). In other words, JavaScript allows complex applications to have direct access to the browser events and Document Object Model (DOM) objects. DOM is a large hierarchical object with several elements forming a tree. In the case of browsers, it is possible to find the elements in the browser itself and in the accessed page (Mariano, 2017) and JavaScript manipulates the DOM.

Front-end web development, also known as client-side development, is the idea of designing and creating the user interface and its interactions (Souders, 2008). A front-end developer is a person responsible for creating the user interface. They are responsible for the application usability, and user experience (UX) is the main concern of this specialist (Souders, 2008).

The main set of web development technologies includes HyperText Markup language (HTML) to specify the web pages' content, Cascading Style Sheets (CSS) to specify web pages' presentation, and finally JavaScript to specify web pages' actions (S. Oney & B. Myers, 2009).

Web browsers turn the pages encoded in HTML and CSS into an understandable 'document' to the user (S. Oney & B. Myers, 2009). Modern web browsers provide an inbuilt interpreter of JavaScript language. This is not exclusive to desktops and laptops. However, this technology has expanded to a number of other devices such as game consoles, tablets, and smartphones (Mariano, 2017).

JavaScript code can be executed in different web browsers and different machines. These environmental factors can cause difficulties when assessing an enhanced performance of JavaScript (Gizas et al., 2012b).

JavaScript frameworks have now become essential tools for an agile development of web applications (P. Saxena et al., 2010). They serve as a structure for creating single page apps, enabling developers to care less about code structure, maintenance. Developers can focus on building sophisticated components and rich interfaces with the aid of these structures. The advantages of using JavaScript frameworks are their efficiency, security and low cost. The most popular JavaScript frameworks are open-source (Mariano, 2017) and the following sections discuss in detail the frameworks AngularJS, Ember and Aurelia.

2.2.1 AngularJS

As the definition of JavaScript frameworks states, AngularJS[1] is not a library but a framework that aids developers with the challenges related to the creation of single-page applications (SPA's). In other words, AngularJS allows the developer to decorate an HTML page with a special markup that synchronises with JavaScript. This separation of concern isolates the application logic from the application views.

Many frameworks available in the market are created and maintained by an open source community (Jain, Mangal, & Mehta, 2015). However, Angular is built and maintained by Google Engineers. Google developed and released AngularJS in 2010.

AngularJS is not the first attempt of Google to release a JavaScript framework tool (Jain et al., 2015). They developed an extensive Web Toolkit, which compiles Java

[1] https://angularjs.org/

down to JavaScript and it was applied in one of their products called Wave (Jain et al., 2015). However, the rise of HTML5, CSS3, and JavaScript as a triad for front-end and back-end solutions for web development, they abandoned the project as they realized that web applications could not be written purely in Java (Jain et al., 2015). The main advantages of AngularJS are the markup in DOM, data as Plain Old JavaScript Objects (POJO), and dependency injection for modules.

Templates in some JavaScript frameworks are implemented like:

- template with markup -> framework template engine -> HTML -> DOM

AngularJS uses the following approach:

- HTML with Angular markup -> DOM -> Angular template engine

AngularJS skips the template pattern by including markup straight to HTML and evaluates the markup only after HTML has been loaded into the DOM. The main advantage of this approach is the integration with existing apps since evaluation starts only after the page is completely loaded.

Digest cycle, or dirty checking, is the process that maintains the view and the data in sync. The framework continually checks all the values in the scope searching for any changes to automatically update the model. AngularJS creates a watchlist, and it will walk down the list searching for any changes in the model.

2.2.2 Ember

Ember[2] is the oldest JavaScript framework used for this comparison project. The history started in 2007 when ember was part of the SproutCore MVC framework. In December 2011, it was renamed to Ember.js to avoid confusion between the application framework and the widget library of SproutCore 1.0. Ember is an open-source JavaScript framework, based on the Model–View–Control (MVC) pattern and just like Aurelia and Angular, it allows developers to create SPA with its own methodology.

In contrast with AngularJS and Aurelia, Ember targets ambitious web applications with a set of features that emphasize scalability. Ember was designed for creating a web page with multiple ajax requests and user interface modifications. Projects like these are usually complicated to maintain, especially when there is more than one person working on the project.

2 https://www.emberjs.com/

Therefore, if there is a need for CRUD actions - create, read, update and delete - on the page, and it is necessary to improve performance, avoiding reloading with each action, the MVC pattern, adopted by Ember, makes the process easier.
Ember relies heavily on the convention over configuration paradigm. In most cases, the framework will automatically generate the modules needed for the application to operate correctly. These modules are loaded by memory without explicitly having to instantiate any class.
Ember uses Handlebars[3] by default to build templates. Handlebars is a JavaScript template system to develop semantic templates. It intends to separate the 'view' from the business logic.
Models[4] are responsible for controlling application data. They are entirely independent of the user interface (UI), but they are required by it. Upon updating, the model notifies the observers, which translates this into the UI.
Controllers[5] are responsible for representing a model in a template and for storing properties that will not be saved on the server. In other words, it manipulates the data in the model, so the view can be changed.

2.2.3 Aurelia

Aurelia[6] is the newest framework used in this project. It was released on July 2016, and it is defined as a platform for building SPAs, based on top of open source web technologies. A collection of modern JavaScript modules provided by Aurelia turns it into a collection of feature-oriented modules. These modules include dependency injection, binding, templating, and more.
The most significant stand out from this framework is the way that it performs the data binding. Aurelia uses unidirectional data flow by default by pushing data from the model into the view via DOM-batching mechanism. In other words, the changes that will affect the DOM will be stored in a queue to be then executed altogether. Also, the syntax is relatively simple and self-explanatory.

[3] http://handlebarsjs.com/

[4] https://guides.emberjs.com/v2.13.0/models/

[5] https://guides.emberjs.com/v2.13.0/controllers/

[6] http://aurelia.io/

2.2.4 JavaScript frameworks comparison

Features are necessary to compare JSFs because they highlight the framework which has an essential function in relation to any project. Table 2.1 shows a comparison between AngularJS, Ember, and Aurelia. It is important to note these are the essential features found in every JavaScript framework.

1. Data-Binding:

   A mechanism allowing the connection between the HTML tags and the defined data in JavaScript. Once the connection is created, the interface element will be updated whenever the objects in the script are changed. This relationship is called one-way binding, and when it occurs in both directions, it is called two-way binding.
2. Dependency Injection:

   A design pattern used when it is necessary to decouple different specific of a system. In this solution, the dependencies between the modules are not defined programmatically but by the configuration of a container. This container injects in each component its declared dependencies.
3. Directives:

   In the simplest way, they are marks on the DOM element (e.g. attribute, CSS class) that tell AngularJS's HTML compiler ($compile) to attach a specific behaviour to that DOM element. They also transform the DOM element and its children.
4. Controller:

   The class contains business logic behind the application to manipulate the model with functions and values.
5. Scope/Model:

   The stored data to be used in the application. It manipulates the data sent from the view.
6. Template:

   The data is presented in this layer. It is the view of the MVC model, and the entire user interface resides in the templates.

7. Routing:

   The URL routing to the application. The router maps the current URL to one or more route handlers. A route handler can render a template, load a model or redirect to a new route.
8. Third-Party add-ons:

   They are the extra functions created in order to help developers save time when developing in a specific framework. (e.g. table with filtering)
9. Structure:

   Model-View-Controller (MVC) pattern makes possible to divide the project into distinct layers which provides separation of concern. MVC standard isolates business rules from the user interface. It is possible to have multiple user interfaces which may be modified without any need to change business rules.

   The Model–View–ViewModel (MVVM) aims to establish a clear separation of responsibilities in the application, maintaining a façade between the Object Model (data) and View which is the interface, where the user interacts.

Features

Metric	Angular	Ember	Aurelia
Size	39.5 kb	90 kb	500 kb
Version	v1.6.3	v1.1.5	2.16.2
Data-Binding	One-Way Two-Way	One-Way Two-Way	One-Time One-Way Two-Way
Dependency Injection	YES	YES	YES
Controller	YES	YES	YES
Scope/Model	YES	YES	YES
Services	YES	YES	YES
Directives	YES	NO	NO
Templates/View	YES	YES	YES
Routing	YES	YES	YES
Structure	MVC MVVM	MVC	MVC
Third-Party addons	2,112 (ngmodules)	4340 (emberaddons)	-

Table 2.1 JavaScript framework feature comparison

Community

Community metrics are essential to verify the amount of support which a given framework will provide to developers. Regarding product choice, community metrics are incredibly relevant to understand the current state of software in relation to its maturity in the marketplace (P. Miguel et al., 2014).

All the chosen frameworks are open-source, and they have their code available in the GitHub, a web-based Distributed Version Control System.

Every repository in GitHub has an option to follow a specific project by clicking the 'star' button. This will allow the user to track and find similar projects. GitHub also offers a forum where developers can report bugs or suggest improvements to the software. Developers can use this function to track previous problems and its solutions in the project.

The number of YouTube videos determines the amount of teaching material that a JavaScript framework may provide.

Metric	Angular	Ember	Aurelia
GitHub stars	57,707	18,527	10,307
Open Issues	533	259	73
Closed Issues	8,169	5,190	550
GitHub Contributors	1,602	695	90
YouTube Results	193,000	11,600	7,050

Table 2.2 JavaScript framework community comparison (December 2017)

2.2.5 System processes

In the simplest terms, a process is an executing program. A program is a passive entity, such as a collection of files stored on a disk and contains a series of instructions to be executed (Tanenbaum, 2009). A program becomes a process when it is loaded into memory, and it may contain one or more processes associated with it (Tanenbaum, 2009).

A process is not only a program code, but also includes the current activity of the program, the process stack, a data section, and the heap (Silberschatz, Galvin, & Gagne, 2014).

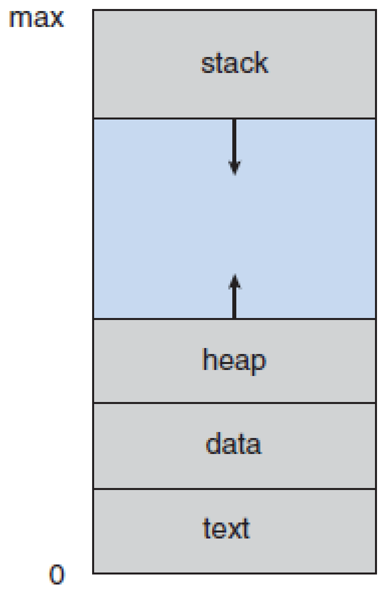


Figure 2.1 Process in memory

Note: Retrieved from *"Operating System Concepts Essentials"* (p. 106), by Silberschatz, A., Galvin, P. B., & Gagne, G. (2014), John Wiley & Sons, Inc.

Stack contains temporary data such as function parameters, return addresses, and local variables. *Heap* is a memory dynamically allocated during runtime process. *Data Section* contains global variables. *Text* is the program source (Silberschatz et al., 2014).

A thread is a basic unit of CPU utilization to which the Operating System allocates processor time and consists of a thread ID, a program counter, a register set, and a stack (Silberschatz et al., 2014). A thread shares the code and the data section with other threads in the same process as well as other operating system's resources, such as open files and signals (Silberschatz et al., 2014). A process can contain one (single-threaded) or more (multi-threaded) threads (Silberschatz et al., 2014). For instance, a web browser might have one thread display images or text while another thread retrieves data from the network.

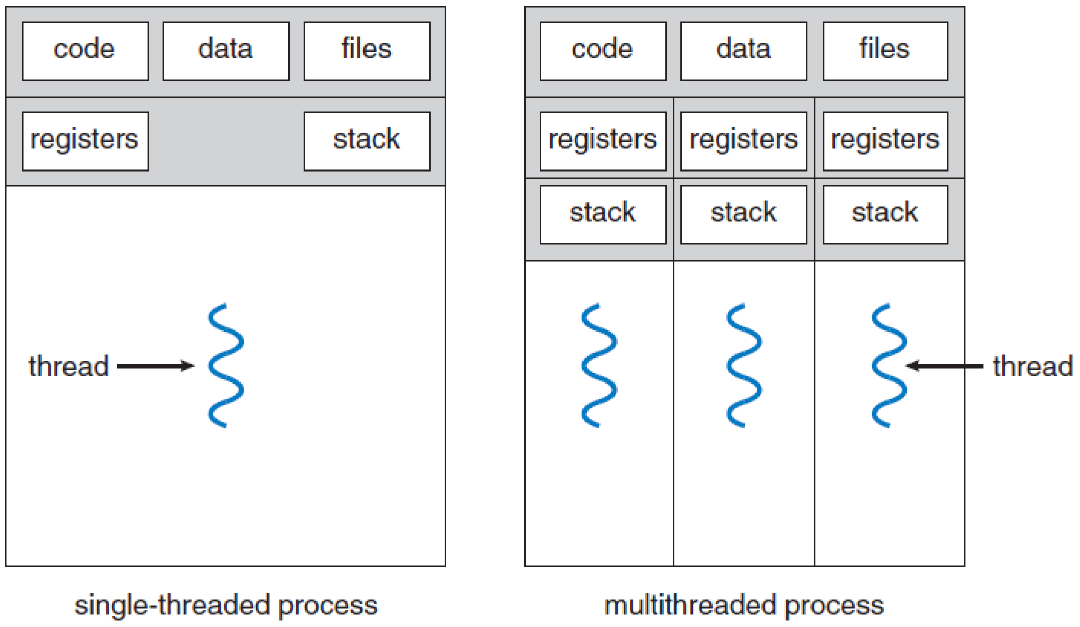


Figure 2.2 Single-threaded and multithreaded processes

Note: Retrieved from *"Operating System Concepts Essentials"* (p. 164), by Silberschatz, A., Galvin, P. B., & Gagne, G. (2014), John Wiley & Sons, Inc.

A thread pool is a set of pre-instantiated threads ready for use and is usually in an idle state (Silberschatz et al., 2014). The process of creating a thread is resource-intensive and monitoring all the active ones is not an easy task. The thread pool helps to reduce the number of application threads and minimizes the CPU effort by avoiding the creation and destruction of several threads.

Essential Process in Windows

The following list contains the critical processes that Windows uses to run in initial configurations.

Process	Name	Description
csrss.exe	Client/Server Runtime Subsystem	User-mode side of the Win32 subsystem. Provides the capability for applications to use the Windows API.
dwm.exe	Desktop Window Manager	The compositing manager introduced in Windows Vista that handles compositing and manages special effects on-screen objects in a graphical user interface
System (ntoskrnl.exe)	NT Kernel & System	The Windows kernel image. Provides the kernel and executive layers of the kernel architecture, and is responsible for services such as hardware virtualization, process and memory management, etc.
services.exe	Service Control Manager	Service Control Manager (SCM) is a

	(SCM)	particular system process under the Windows NT family of operating systems, which starts, stops and interacts with Windows service processes.
explorer.exe	Windows Explorer	Provides an interface for accessing the file systems, launching applications, and performing common tasks such as viewing and printing pictures
dllhost.exe	COM Surrogate	COM stands for Component Object Model. This is an interface Microsoft introduced back in 1993 that allows developers to create “COM objects” using a variety of different programming languages. Essentially, these COM objects plug into other applications and extend them.
RuntimeBroker.exe	Runtime Broker	It is used to determine whether universal apps you got from the Windows Store–which were called Metro apps in Windows 8–are declaring all of their permissions, like being able to access your location or microphone.
sihost.exe	Shell Infrastructure Host	It’s responsible for presenting universal apps in a windowed interface. It also handles several graphical elements of the interface, like Start menu and taskbar transparency and the new visuals for your notification area flyouts–clock, calendar, and so on.
svchost.exe	Host Process for Windows Services	
fontdrvhost.exe	User-mode Font Driver Host	
winlogon.exe	Windows Logon Application	This process performs a variety of critical tasks related to the Windows sign-in process.
lsass.exe	Local Security Authority Process	Local Security Authority Subsystem Service (LSASS) is a process in Microsoft Windows operating systems that is responsible for enforcing the security policy on the system. It verifies users logging on to a Windows computer or server, handles password changes, and creates access tokens.
wininit.exe	Windows Initialization Process	It initializes the user-mode scheduling infrastructure.
smss.exe	Session Manager Subsystem	It is executed during the startup process of those operating systems (it is the first user-mode process started by the kernel). It creates environment variables and starts the kernel and user modes of the Win32 subsystem.

Table 2.3 Process in Windows

Services

A Windows service operates in the background as a computer program. It is similar in concept to a Unix daemon.

Name	Process associated	Description
BrokerInfrastructure	svchost.exe (DcomLaunch)	Background Tasks Infrastructure Service
CoreMessagingRegistrar	svchost.exe (LocalServiceNoNetwork)	CoreMessaging
CryptSvc	svchost.exe (NetworkService)	Cryptographic Services
DcomLaunch	svchost.exe (DcomLaunch)	DCOM Server Process Launcher
EventLog	svchost.exe (LocalServiceNetworkRestricted)	Windows Event Log
LSM	svchost.exe (DcomLaunch)	Local Session Manager
PlugPlay	svchost.exe (DcomLaunch)	Plug and Play
Power	svchost.exe (DcomLaunch)	Power
ProfSvc	svchost.exe (netsvcs)	User Profile Service
RpcEptMapper	svchost.exe (RPCSS)	RPC Endpoint Mapper
RpcSs	svchost.exe (RPCSS)	Remote Procedure Call (RPC)
StateRepository	svchost.exe (appmodel)	State Repository Service
SystemEventsBroker	svchost.exe (DcomLaunch)	System Events Broker
tiledatamodelsvc	svchost.exe (appmodel)	Tile Data model server

Table 2.4 Services in Windows

2.2.6 Asynchronous programming explained

Asynchronous programming is a challenging activity which confuses and presents difficulties for many. In the most imperative languages, such as C# and Visual Basic, the execution of methods (functions, procedures) is sequenced (Cristian, 1996). In other words, once a control thread starts executing a particular method, it will be working with this task until the method execution has been completed. Sometimes the thread is executing statements in different methods, but this is part of the execution of the main method (Cristian, 1996). The thread will never do something that was not requested by its own method.

Sometimes this synchronicity is a problem because the method might be waiting for a long task to be completed, e.g. a download or calculation performed on a different thread. In these cases, the thread gets completely blocked doing nothing. Synchronous behaviour creates a bad user experience as the interface is locked/frozen whenever the user attempts to perform a time-consuming operation (Okur, Hartveld, Dig, & Deursen, 2014).

An asynchronous method (creation of a thread) will immediately be returned, and the program will perform other operations while the calling method completes its work (Okur et al., 2014). The behaviour of the asynchronous method differs from the synchronous because the asynchronous method creates a thread separately and this thread starts to be executed immediately. However, the control is instantly returned to the thread that called it, while the other thread continues to be executed.

In general, asynchronous programming makes sense if it is necessary to create an application with an intensive interface in which the user experience is the primary concern. In this case, an asynchronous call allows the user interface (UI) to continue responding and does not stay frozen (Okur et al., 2014). The second scenario would be a complex computational work or a very time-consuming task, and the user still needs to interact with the UI while those computations are being executed in the background (Okur et al., 2014).

Asynchronous programming is one of the main advantages of the JavaScript language especially because JavaScript runs on a single thread (Klein & Spector, 2007). If there is only one thread to run the code, this code should avoid blocking the thread as much as possible. Therefore, delayed operations such as HTTP requests and disk access, or a database, are typically executed asynchronously (Klein & Spector, 2007).

Although JavaScript is executed by a single thread, it does not mean the language engine and its host application only use one thread (S. Tilkov & S. Vinoski, 2010). For example, if a Node.js application requests disk access, Node may use another thread to perform such access (S. Tilkov & S. Vinoski, 2010). But the code that requests this access and the callback code that handles the result is executed on that single thread dedicated to JavaScript code. This single thread executes an event loop (S. Tilkov & S. Vinoski, 2010).

Event loop is basically an infinite loop in which each iteration verifies the existence of a new event (Richards, Gal, Eich, & Vitek, 2011). In Node.js, the EventEmitter[7] is the module responsible for issuing events

[7] https://nodejs.org/api/events.html#events_class_eventemitter

When a given code issues an event, the EventEmitter sends it to a queue to be executed by the Event-loop. The Event-loop returns the result in a callback (S. Tilkov & S. Vinoski, 2010). Such callback is usually executed through a listen function.

```
var http = require('http');

var server = http.createServer(function(request, response){
response.writeHead(200, {"Content-Type": "text/html"});
response.write("<h1>Hello World!</h1>");
response.end();
});

server.listen(3000, function() {
console.log("Server running! Listening to port 3000");
});
```

In the example above, the event loop will work with only two listening events: http.createServer() and server.listen(). In http.createServer(), this event callback will always be executed every time a user accesses the URL from the server. In this case, http://localhost:3000. This event will run more frequently through the Event-loop because it will be added to the queue each time the server receives a new request. The Server.listen() event will be executed only once by the Event-loop because this event occurs when the server starts. In this example, it is started by port 3000.

2.2.7 Promises explained

JavaScript is a single-threaded language, which means that every event called is executed independently, one after another (Richards et al., 2011). Functions cannot run at the same time, and JavaScript shares a thread with many other functions carried by the browser (Richards et al., 2011).

```
function readJSONSync(filename) {
  return JSON.parse(fs.readFileSync(filename, 'utf8'));
}
```

In this example, the function readJSONSync will freeze the application until all the data is loaded. One way to avoid this is called Callback functions.

Callback is a function that is passed to another function as a parameter, and then the callback function is executed inside of the other function (Kambona, Boix, & De Meuter, 2013). A callback function is essentially a pattern often referenced as callback pattern (Geiger, George, Hahn, Jubeh, & Zündorf, 2010). The cascade execution should guarantee that the second function is executed after the first one.

```
function readJSON(filename, callback){
  fs.readFile(filename, 'utf8', function (err, res){
    if (err) return callback(err);
    callback(null, JSON.parse(res));
  });
}
```

However, this is not always true as the events might delay between one and another. The implementation above does not include handling errors, which are very likely to happen when dealing with time and asynchronous functions.

```
function readJSON(filename, callback){
  fs.readFile(filename, 'utf8', function (err, res){
    if (err) return callback(err);
    try {
      res = JSON.parse(res);
    } catch (ex) {
      return callback(ex);
    }
    callback(null, res);
  });
}
```

Events are great for dealing with the same object multiple times. However, when it comes to handling the asynchronous call, events may not be the best way to deal with async success/failure (Richards et al., 2011).

Using these callbacks in a large-scale application may lead to a mix of collocated code fragments that are hard to understand (Kambona et al., 2013). It may also force the programmer to create complex flows to pass callbacks around in order to use their delayed values.

One way to handle this problem is to use Promises. The term was coined by Daniel Friedman and D. Wise (1978), is defined as a proxy object that represents an unknown result that is yet to be computed.

Promises are primarily the result of an asynchronous operation (Kambona et al., 2013). Instead of immediately returning the value of the method, it will return a promise that should be fulfilled or reject after at some point in the future. In other words, they are event listeners that can only succeed or fail once. Promises have three states:

Pending: Initial state, neither Fulfilled nor Rejected

Fulfilled: Operation succeed

Rejected: Operation failed

A pending promise can either be fulfilled with a value or reject with an error. When either of this happens, the method .then() will handle in case of success or failure.

```
function readJSON(filename){
  return readFile(filename, 'utf8').then(JSON.parse);
}
```

Data can also be manipulated within the method .then(). In the example above, the response is transformed into a JSON object. Promises also allow chaining through this method. It is possible to append more .then() methods in order to create a queue of responses. Each response will only be satisfied after the previous one.
Asynchronous JavaScript, AJAX, and HTML5 technologies provide developers with essential tools to create complete responsive JavaScript applications (P. Saxena et al., 2010). Promises play a significant role when it comes to JavaScript and Restful applications (Kambona et al., 2013). Each JavaScript framework implements an HTTP client and promises in different ways. However, they are based on the same concept of RESTful applications and asynchronous functions.

2.3 Performance evaluation

Performance testing is a broad activity, which can address many risks, take many forms, and provide a wide range of value to an organization (Ramos & Valente, 2014). It is essential to understand the different types of performance testing to reduce risk and cost, and it is also important to know when to apply the appropriate test over a given project (Ramos & Valente, 2014).
In order to apply different types of tests in a performance test, the following key points need to be considered:

- The goals of the performance test
- The context of the performance test (e.g. resources involved)

Performance tests consist of testing a system for its performance requirements (Denaro, Polini, & Emmerich, 2004) such as:

- Latency: the time between a request and response of operation.

- Throughput: the number of operations that the system is able to complete in a period of time.
- Scalability: the number of concurrent users the system can handle;
- Use of machine resources: such as memory and processing

Although a complete and ideal performance test depends on the existence of a fully integrated and functional system, performance tests are often applied in all steps throughout the process, in the context through which it will work.

2.3.1 Benchmarking

Benchmarking is a technique used to measure the performance of a system or one of its components (Vokolos & Weyuker, 1998). More formally it can be understood that a performance test is the result of the execution of a computer program or a set of programs on a machine, with the aim of estimating the performance of a specific element and being able to compare the results with similar machines (Vokolos & Weyuker, 1998). In the field of computers, a performance test could be performed on any of its components, be it the CPU, RAM, graphics card, etc. (J. L. Henning, 2000). It can also be specifically directed to a function within a component, such as the floating-point unit of the CPU, or even to other programs (J. L. Henning, 2000).

Benchmarks submit computational systems to load tests, which are executed through programs, exercising an appropriate set of instructions that generate loads in the system, used as a method of comparing performance among various subsystems (SPEC[8]). Benchmarks are evaluation measures which perform a defined set of workload operations to produce a result, according to the metrics defined by the algorithm or benchmark software (Alves, Ypma, & Visser, 2010).

Load testing is used to evaluate the operating limits of a system according to variable workloads (Draheim, Grundy, Hosking, Lutteroth, & Weber, 2006). The system's behaviour during the execution of the test helps to determine maximum operating capacity and bottlenecks in the system (B. M. Subraya & S. V. Subrahmanya, 2000). In general, measurements are taken based on the data transfer rate of the workload and the response time (Vokolos & Weyuker, 1998). There are cases where the load test maintains the workload, but the system configuration varies (Draheim et al., 2006). In

[8] https://www.spec.org/web2009/

these cases, the environment where the system is configured could influence the system’s performance.

2.3.2 Other performance evaluations types

Performance testing also covers different aspects of computational systems such as a system’s resources and data volume. These load tests are the stress testing, volume testing and spike testing.

Stress testing is a type of reliability test designed to evaluate how the system responds under abnormal conditions (Krishnamurthy, Rolia, & Majumdar, 2006). System stress can cover extreme workloads, insufficient memory, unavailable hardware and services, or limited shared resources (Krishnamurthy et al., 2006). The test should put the application under stress to verify that the software can operate normally under heavy processing load (B. M. Subraya & S. V. Subrahmanya, 2000). Often, the requirements define the expected processing load, for instance, one thousand hits per hour or one hundred transactions per minute. These numbers should be used as parameters at the time of the stress test run.

Volume testing tests the amount of data that a system can manage, the purpose of this test is to determine the system's ability to handle the volume of data specified in its requirements (Teitel, 1981). In general, this type of test uses large amounts of data, which is used to determine the limits at which the system fails (Teitel, 1981). Moreover, they are usually used in identifying the maximum load or volume of data that the system can manage in a time period (Teitel, 1981).

Spike testing aims to analyse the behaviour of a Web system under an atypical condition of high load for a specific period of time (Menasce & Almeida, 2001). In general, systems can handle gradual increases of the load. However, serious problems can arise during a sudden increase in load (Menasce & Almeida, 2001). For example, a given system can correctly support a load growth of one to five users per minute in a ten-minute interval, reaching between ten to fifty users. This same system may not support an abrupt increase of ten to twenty users per minute, in the same ten-minute interval, which could reach between one hundred to two hundred users per minute. Problems related to the connection with the web server or the database could arise due to the unexpected workload. Web systems can experience these sudden spikes of charge during a special event, such as an advertising marketing campaign or a new product release (Menasce & Almeida, 2001).

2.3.3 Benchmarking JavaScript frameworks

Mariano (2017) conducted research on JavaScript frameworks (JSF) using benchmarking software metrics. The research investigated whether computer and software benchmark metrics are appropriate for the comparison of JavaScript frameworks. Three JSFs were selected based on their popularity among the development community. They are AngularJS, BackboneJS and React. The number of frameworks was limited to three due to time constraints. However, the results gathered from the experiment contributed to the research in this field.

Another significant contribution to this field of research is a paper from Gizas, Christodoulou, and Papatheodorou (2012a) where they evaluated the most popular JSFs in that period. The researchers focused on the quality and performance of the JSFs, and they contributed with software quality metrics and performance measures that inspired Mariano’s work and consequently this project. They compared ExtJS, Dojo, jQuery, MooTools, Prototype, and YUI. Their results revealed that some points in the code needed to be improved and they suggested carrying on the research in different platforms such as the mobile platform.

Graziotin and Abrahamsson (2013) proposed an improvement in the Gizas et al. study by suggesting an implementation of a benchmark framework based on the TodoMVC project. The aim of their research was to provide reliable data based on the performance of those frameworks when executing different tasks. Their proposal led to the creation of the TodoMVC benchmark project which was used by Mariano in his research.

Ratanaworabhan, Livshits, and Zorn (2010) discussed the research limitations in the field of JavaScript frameworks. They evaluated the JavaScript behaviour in commercial websites and compared with benchmark suits such as SunSpider and V8. They measured function, code and event handlers. They concluded that those benchmarks could not represent real-life situations as common behaviours native to real websites were not included in the benchmarks such as event-driven execution, instruction mix similarity, cold-code dominance, and the prevalence of short functions.

Pano, Graziotin, and Abrahamsson (2016) conducted a survey to investigate the deterministic factors that lead developers to adopt JavaScript frameworks. The interview questions were designed based on performance expectancy, effort expectancy, social influence, facilitating conditions and price value.

Performance expectancy is how much individuals believe that a system will help them to achieve their results faster. The metrics for this factor are performance and size. *Effort expectancy* is the degree of difficulty of software. In other words, it is how software can be straightforward and easy to learn. The metrics for this factor are flexibility, complexity and understandability.

Social influence is the degree of influence people have over the decision makers. Social influence metrics include competitor analysis, collegial advice, community size and community responsiveness.

Facilitation conditions are the individual beliefs of how the software is well supported and maintained. The metrics for this factor are suitability, updates, modularity, isolation and extensibility.

Price value is the cost of adopting the specific software. Most JavaScript frameworks are open-source and free. However, participants often mentioned the cost of adopting a JSF.

Factor	Metrics
performance expectancy	Performance; size
effort expectancy	Automatization; learnability; extensibility; complexity; understandability
Social influence	competitor analysis; collegial advice; community size; community; responsiveness
facilitating conditions	Suitability; updates; modularity; isolation; extensibility
Price value	cost

Table 2.5 Influence factors on choosing JavaScript Frameworks

The interview grouped eighteen decision makers regarding the JavaScript framework selection, divided by their role in the project. These stakeholders are the customer, developer, team, and team leader.

The research concluded that metrics related to size and performance in terms of execution time play an essential role for decision makers in software projects. Although these metrics are important for practitioners, they are not sufficient to determine the degree of influence on the decision makers of the projects. The research findings demonstrated that programmers spend some time studying the framework documentation to find examples of simple tasks or hints for achieving advanced functionalities.

The results also revealed that framework maturity is an important factor as the framework age also influences the decision-making of practitioners. Framework modularity also influences the framework choice as the code modification process can be quickly done without affecting other areas of the application. Developers like frameworks that can achieve basic and advanced functionalities in their core versions without the necessity of including third-party add-ons.

2.3.4 Benchmarking databases

A database can be defined as a large structured set of persistent data (Dietrich, Brown, Cortes-Rello, & Wunderlin, 1992). In other words, a database is an organised store of data that can be accessible by its element's name. A Database Management System (DBMS) is a software program designed to create, store, update, and manage databases (Dietrich et al., 1992). DBMS software enables applications and end-users to access the same data. It provides a mechanism for creating, retrieving, updating, and deleting data from databases and is also responsible for maintaining data integrity as well as access control and recovery mechanisms (Dietrich et al., 1992).

The performance of DBMS is a crucial factor in determining the adoption of these systems by a company or service. The price of a DBMS is also an essential factor as companies always seek for an optimized return of investment. The decision-making of these companies is based on data collected from performance measures of DBMS's from different vendors. Database benchmarking is the process that provides the necessary data to aid companies in making such decisions.

Database performance benchmark has a long history, and it has been evolving and adapting throughout time. The DebitCredit benchmark is a transactional benchmark created to measure the performance of transactions in a DBMS. The DebitCredit benchmark is one example of the earlier stages of benchmarking databases. The results provided for this benchmark were not reliable as some publishers were able to alter critical requirements of this benchmark to improve performance results. It was necessary to create benchmark standards for DBMS's as research in this field was not well developed. Transactional Processing Performance Council (TPC) is a non-profit

corporation created to establish industry standards for transaction processing and database benchmarks.
TPC modified the CreditDebit benchmark by adding and establishing standards for benchmarking databases. They called TPC benchmark A (TPC-A). TPC-A document a series of guidelines to measure performance and price of DMBS. TPC started to establish performance measures for various types of DBMS's and scenarios that had been applied to these systems. The council enumerates a series of benchmarks that measure transactions in databases such as TPC-C, TPC-W and TPC-H (May, Kossmann, Kaufmann, & Fischer, 2013). DBMS's have been evolving over time. Therefore, it is essential that the benchmark field also updates and adapts to the new ways of storing and retrieving data.
TPC has been struggling to keep up with the rapid changes that occur within the DBMS industry (Nambiar & Poess, 2013). There has been a swift decline in publications because traditional databases and system vendors are being bought out by other companies, reducing TCP's membership (Nambiar & Poess, 2013). In an attempt to keep the council relevant, TPC separated their benchmark into two categories. They are called TPC Enterprise Benchmarks and TPC Express Benchmarks (Nambiar & Poess, 2013).
The first category contains the traditional set of benchmarks with an extensive specification of those benchmarks. TPC Enterprise Benchmarks are expensive and hard to maintain, however, they have a rigorous set of tests and checks that guarantee the quality of the benchmark (e.g. Ensure ACID properties) (Nambiar & Poess, 2013).
Express benchmarks are based on predefined, executable kits that can be rapidly deployed and measured. The Express category contains a new set of benchmarks mainly focusing on Big Data Systems and Cloud Platforms(Nambiar & Poess, 2013).
Although there is little question about the quality of TPC's benchmark standards when they first started in the early 1990's, they still are a reference in benchmarking databases.

Transactions

Benchmarking databases are mainly focused on the transactions made in a database. A transaction is a small unit of a program that may contain small tasks within its process (Dietrich et al., 1992). A transaction can also be defined as atomic (all or nothing). The atomic concept is crucial to maintaining the consistency of the stored data in the database (Dietrich et al., 1992). ACID stands for Atomicity, Consistency, Isolation, and Durability and it determines the core properties of a transaction (Dietrich et al., 1992).

- Atomicity – The system must ensure the all-or-nothing quality of transactions. No data will be left partially updated.
- Consistency – DBMS must ensure that the data transaction is always abiding by the established rules and the data affected must be changed only in allowed ways.
- Isolation – DBMS can serve data to multiple user programs. In this case, the system must carry each transaction independently as it was the only transaction being made in the system. Lock mechanism helps to prevent two transactions being made at the same time.
- Durability – In accordance with atomicity, the DBMS must be able to hold all the changes even if the system crashes. The DBMS must provide recoverability.

In addition, when the DBMS is executing multiple transactions, it must schedule operations for the execution of concurrent transactions. This property is called serializability, and it is crucial in multiuser and distributed database environments, where various transactions are likely to be executed at the same time for the same data. In a scenario where only a single transaction is executed, serializability is not an issue.

2.3.5 TODO benchmark application

TodoMVC is a project created to help developers in choosing the right JavaScript frameworks. It aims to have an implementation of a simple todo application in all the most popular JavaScript frameworks including Angular, Backbone, React, and others.
In the application, the user can add new tasks, mark them as completed and erase them. The application also offers different views depending on how the user wants to visualise the list of tasks (All, active and completed). It provides two buttons; to complete all the remaining tasks and clean the list of all completed tasks. The user also has the option of completing or eliminating them one by one.

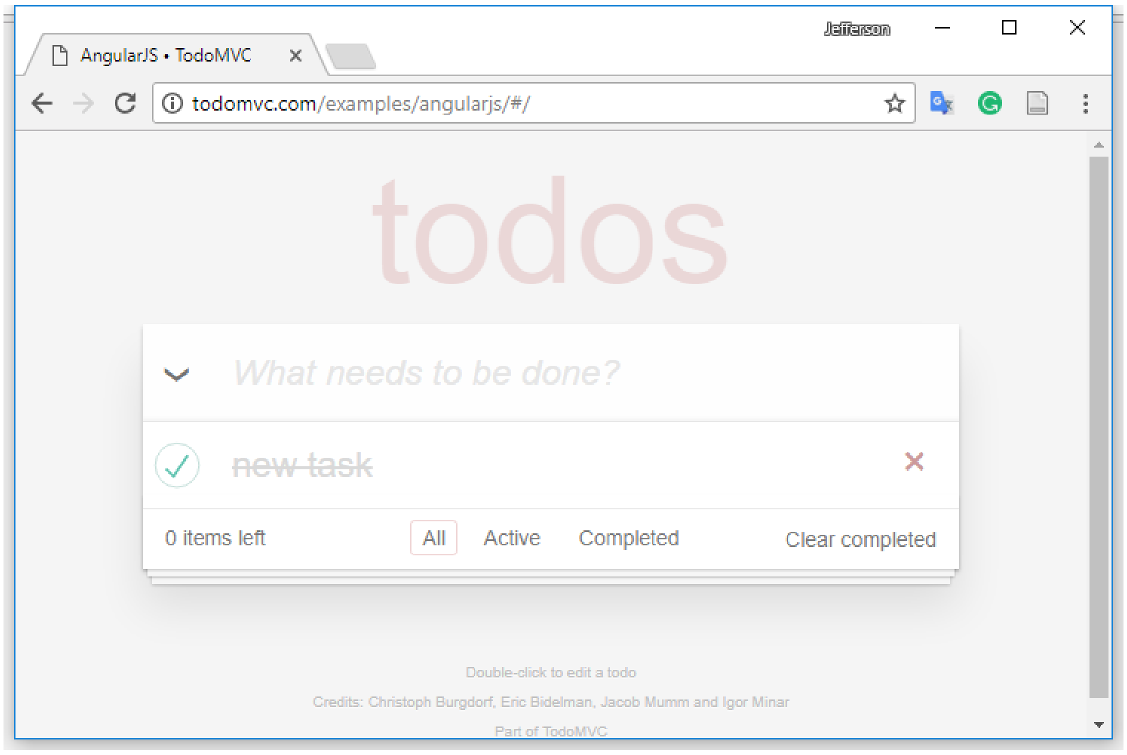

AngularJS • TodoMVC
Jefferson
todomvc.com/examples/angularjs/#/
todos
What needs to be done?
~~new task~~
0 items left
All
Active
Completed
Clear completed
Double-click to edit a todo
Credits: Christoph Burgdorf, Eric Bidelman, Jacob Mumm and Igor Minar
Part of TodoMVC

Figure 2.3 TodoMVC user interface

The TodoMVC benchmark framework uses the TodoMVC application to group a set of actions that the application executes. The benchmark framework performs three actions in the application. It adds a hundred new tasks completes all the tasks and finally erases all the tasks from the list.

The time of each set of operations is counted and stored. After executing all the operations, the benchmark framework uses the execution time of the tasks to generate a chart showing the performance of the framework.

The framework calculates the time spent between the start and end of the request. Before each step-start, the application prepares and starts the clock. The event execution takes place, and the clock stops counting at the end of the event. The total time of the execution is calculated at the end of the process.

The trigger is the emulation of a keypress event activated by the Key Code number thirteen. Every time this event happens the application activates the clock and calculates the time at the end of it.

1. Add 100 items

This is the first task the benchmark executes. It starts with a clean state where an empty to-do list is created. The function simply adds a hundred items to the list with the code:

```
newTodo.value = 'Something to do ' + i;
```

A keypress event is triggered to complete the task and end the event. The benchmark framework recodes the execution time of this process and how long the DOM objects take to be created in the browser.

This process aims to measure how long the JavaScript framework takes to execute and create objects in the browser. Speed is the main metric in this process, therefore, the faster, the better.

2. Complete 100 items

The second step of the set of tests is to complete all tasks created in the previous step. The process starts by loading the list and looping through the list simulating a click on the checkboxes.

```
var checkboxes = contentDocument.querySelectorAll('.toggle');
for (var i = 0; i < checkboxes.length; i++)
        checkboxes[i].click();
```

This process aims to measure how long the JavaScript framework takes to read the DOM objects and update them. Speed is the main metric in this process, therefore, the faster, the better.

3. Delete 100 items

The final step is to eliminate all the completed tasks. Similar to the previous step, this process starts by loading a list of tasks to be eliminated and loops on the array destroying the DOM objects marked as a completed task.

```
var deleteButtons = contentDocument.querySelectorAll('.destroy');
for (var i = 0; i < deleteButtons.length; i++)
        deleteButtons[i].click();
```

This process aims to measure how long the JavaScript framework takes to read the DOM objects and delete them. Speed is the main metric in this process, therefore, the faster, the better.

2.4 Benchmarking metrics

Lines of code (LOC) is the oldest metric for measuring the efficiency of algorithms, and it is used to measure the size of a software program by counting the number of lines in the text of the program's source code (Gizas et al., 2012b). The IEEE has standardised two methods of counting lines of code; Physical Lines of Code (SLOC) and Logical Lines of Code (LLOC) (Park, 1992). SLOC is the real number of code lines written in a programme excluding the comment lines, and LLOC is the number of executable statements such as functions and procedures in a piece of code. SLOC is usually used to predict the amount of effort that will be required to develop a program, as well as estimating programming productivity or effort, once the software is produced.

Cyclomatic complexity is a metric used to measure the complex nature of a program (T. J. McCabe, 1976). It measures the number of linearly-independent paths through a program's source code. In other words, this metric checks the logic gates that a program uses during its execution and calculates the source code complexity.

Halstead's complexity metrics were developed by Maurice Halstead as a means of determining a quantitative measure of complexity directly from the operators and operands contained in the module of a program from the source code. These metrics are often used as a maintenance metric. However, evidence shows that Halstead metrics are also useful during development to assess code quality in dense computational applications or to keep up with complexity trends.

Concerns about software quality are measured through the maintainability index which measures how easy it is to maintain the code. The maintainability index metric uses a series of formulas containing the Source Lines of Code and the Cyclomatic Complexity (Mariano, 2017). This metric also gives insights about the quality of the software.

Other time measures are the Request Time with the time between connection initialization and first response byte received from the server (Filipe, Boychenko, & Araujo, 2015). Response Time is the time between first and last response byte received from the server. Query Processing Time is the time that an HTTP request spends on the database (Filipe et al., 2015).

2.5 Performance.now (clock)

This section introduces two methods to measure the execution time of JavaScript Applications (Date.now() and Performance.now()). Accurate timestamping is a crucial need in the software benchmarking field because milliseconds may completely change the outcomes of a comparison. Software clocks are constantly being improved to meet these needs and overcome challenges related to precise clocks for measuring software performance.

A Date object is a time value representing time in milliseconds since 01 January 1970 UTC (Suh et al., 2017). In other words, this definition is sufficient to represent a precise time in milliseconds or any instant that is within approximately 285,616 years from 01 January 1970 UTC. For example, the following code ran on Sat Dec 23, 2017, 08:40:24 UTC. The result was a thirteen-digit number (1514018761800) representing the number of milliseconds from the start of that timebase, January 1970.

```
<script>
console.log(Date.now());
</script>
```

Although the data object definition is useful for showing the current calendar time, the definition is subject to both clock skew and adjustment of the system clock (A. Rajaram, Jiang Hu, & R. Mahapatra, 2006). In other words, the value returned may decrease or remain the same due to its non-monotonic nature.

```
var mark_start = Date.now();
doTask(); // Some task
var duration = Date.now() - mark_start;
```

For instance, the previous piece of code may compute a positive, negative, or zero number for the variable duration.

Sub-millisecond resolution is required to measure elapsed time accurately (e.g. using navigation time APIs, resource time, benchmarking). In tasks where precision is a crucial factor for the results, this definition may not fulfil all the requirements as it does not provide a sub-millisecond resolution and is subject to system clock skew.

The High-Resolution Time specification[9] sets a new time base with a microsecond resolution (one-thousandth of a millisecond). The specification reduces the number of bits used to represent specific number and increases readability, instead of measuring the time from January 1, 1970, UTC; this new time base measures the time from the start of document navigation, *performance.timing.navigationStart*.

The specification defines performance.now() as the alternative method of Date.now() to determine the current time in high resolution. DOMHighResTimeStamp is the alternative type for DOMTimeStamp that sets the high-resolution time value.

```
performance.now(); //13.405000000000001
Date.now(); //1514021548850
```

[9] https://www.w3.org/TR/hr-time/

The values returned from each function [above] represents the same instance in time, but they are measured from a different source. The time value from performance.now() is clearer compared to Date.now().

```
<script>
var mark_start = Date.now();
console.log("time is relative");
var duration = Date.now() - mark_start;
console.log(duration); //3
</script>
```

In the above example, the value returned when using the Date.now() function is 3.

```
<script>
var mark_start = performance.now();
console.log("time is relative");
var duration = performance.now() - mark_start;
console.log(duration); //2.1849999999999996
</script>
```

For the performance.now(), the value is 2.1849999999999996.

2.6 Virtual Machine vs Dual Boot

Many computers hold only one Operating System (OS) such as Windows, Linux or MAC. Dual-boot allows users to install and use multiple OSs in their machines enabling them to choose which OS they would like to use at the boot time (Chang, Ho, & Chang, 2014).

Having multiple OSs in one machine is hugely advantageous because users have more options for executing the same task in different ways. They may also make use of a program which is not available in another OS or another version of the same OS (e.g. Games). Users may take advantage of the higher number of applications that each OS offers without losing performance (Chang et al., 2014).

Dual-boot consists in partitioning a hard-drive or adding a different hard drive to the computer to install the second OS (Fourment & Gillings, 2008). Two OSs cannot coexist in the same partition which creates the necessity of having a bootloader that allows users to choose which OS they wish to start (Fourment & Gillings, 2008). The booting process will increase because the system will have to seek for file systems in a different partition of the hard-disk (Fourment & Gillings, 2008).

Virtual Machines (VMs) are an alternative and fast way to use two different OSs on one machine. However, VMs run on top of the existing OS which obligates them to share the same resources while they are running (Chang et al., 2014).

Virtual Machines consists of installing a virtual machine app which will host the operating system. VMs can emulate multiple systems at the same time (Chang et al., 2014). This may cause some overhead as all the systems will be sharing the same hardware resources. It is useful to simulate networks in one environment or use as an environment for security tests.

Dual-booting does not affect the performance of OSs, but it does increase the booting time (Chang et al., 2014). Some studies have addressed the issue of enhancing booting time in dual-boot machines with positive results (Kureshi, Holmes, & Liang, 2010). Booting time will not impact the results of this research as the running operating system does not share any resources with the concurrent OS. In other words, when Windows is running Linux is completely shut down and therefore, incapable of interfering with its performance and vice versa.

Although dual-boot does not affect the performance of the environment for tests, the ideal scenario would be two different machines with the same hardware configuration running different Operating Systems.

2.7 Chapter summary

This chapter provided an overview on the JavaScript frameworks and its research on the field of benchmarking. A brief description of the target frameworks for this comparison was discussed as well as a comparison of features available from each framework. Concepts of JavaScript language that were relevant for this research were explained and illustrated.

Relevant literature reviews supported the chosen metrics for the JavaScript Framework comparison. This chapter also provided a detailed description of the clock used for recording the execution time of JavaScript.

3 DESIGN AND METHODOLOGY

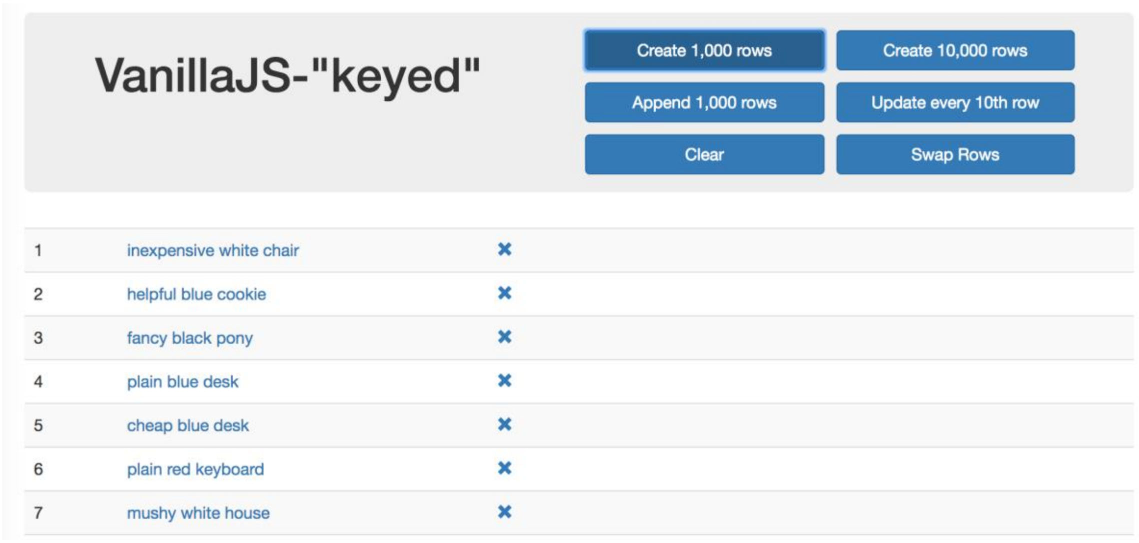
3.1 *Introduction*

In this chapter, the design and methodology of this experiment are discussed in detail. The chapter starts with an introduction to the application design and all the essential factors that the application needs to be validated such as the cloud platform and network specifications. Then, it will present a description of the analytical procedure to evaluate the data collected.

3.2 *JavaScript framework benchmark application*

The js-framework-benchmark is a benchmark application built to compare JavaScript frameworks through a set of performance tests and memory consumption. Stephan Krause created the benchmark application as a personal project in 2014, and since then the project has been growing with the aid of an online community of JavaScript developers.

The ambitious project aims to compare 30+ JavaScript frameworks performing a set of tests in those frameworks and their different variations.



1	inexpensive white chair	✖
2	helpful blue cookie	✖
3	fancy black pony	✖
4	plain blue desk	✖
5	cheap blue desk	✖
6	plain red keyboard	✖
7	mushy white house	✖

Figure 3.1 js-framework-benchmark user interface

The benchmark application focusses purely on the performance of JavaScript frameworks and their memory usage. Like TodoMVC benchmark, this framework

consists of creating, updating and deleting DOM objects but with a different approach. The focus of this framework is to test the performance of two different code implementations of the same framework and compare them to other frameworks. The application's creator defines two types of implementations to be tested. The first is the "keyed" implementation where any modifications associated with the data will be applied to the associated DOM node. In the "non-keyed" implementation there is no association between the data and the DOM node. The main difference in these two implementations is how the code behaves using two different methods of data binding.
The benchmark framework uses the Chrome WebDriver to capture all results. This driver uses the Chrome timeline to generate the performance results. It uses the same API used for the TodoMVC benchmark, the *performance.now* API.
In comparison with *Date.now()* method, the *performance.now()* provides a higher timestamp resolution, and always increases at a constant rate that is independent of the system clock, which can be adjusted or manually skewed. The W3C organisation has a formalisation for this API, and it will be discussed in section 2.5.
The benchmark functions run separately when the server is running. However, the benchmark application runs all tests and results altogether. It also prepares the environment before the test starts (e.g. cleaning the database before inserting new data). Some of the following benchmarks use the warm-up technique which is commonly used to avoid cold-start bias (J. W. Haskins & K. Skadron, 2001). They iterate five times before the evaluation starts and the execution time of each function is the difference between the start and the end of the operation.

Benchmark functions

1. Create rows

This function generates a thousand rows containing three words in each row. The application registers the time before the function starts to generate random rows and present the data afterwards. The application returns this speed metric in milliseconds.

2. Replace all rows

This function updates all the records in the table by executing the function Create Rows. The data generated will replace the existing data in the table. This benchmark has a five warm-up iteration, and the application returns this speed metric in milliseconds.

3. Partial update

This function updates the text of every tenth row. It loops through the data array generated in the create rows and adds an extra text ('!!!') at the end of each row. This benchmark has a five warm-up iteration, and the application returns this speed metric in milliseconds.

4. Select row

This function calculates the time that each JavaScript framework takes to highlight a row in response to a click on the row. This benchmark has a five warm-up iteration, and the application returns this speed metric in milliseconds.

5. Swap rows

This function swaps two rows into a thousand rows table. The execution time is the difference between the start and the end of this operation. This benchmark has a five warm-up iteration, and the application returns this speed metric in milliseconds.

6. Remove row

This function deletes one row from a thousand rows table. The execution time is the difference between the start and the end of this operation. This benchmark has a five warm-up iteration, and the application returns this speed metric in milliseconds.

7. Create many rows

This function generates ten thousand rows containing three words in each row. This benchmark has a five warm-up iteration, and the application returns this speed metric in milliseconds.

8. Append rows to large table

This function adds a thousand rows to the current table created. The benchmark runs this function after calling the `Create many rows` function which generates ten thousand rows. This benchmark has a five warm-up iteration, and the application returns this speed metric in milliseconds.

9. Clear rows

This function deletes all the rows in the table. The benchmark runs this function after calling the `Create many rows` function which generates ten thousand rows. This benchmark has a five warm-up iteration, and the application returns this speed metric in milliseconds.

10. Start-up time

This benchmark calculates the time for loading, parsing the JavaScript code and rendering the page. This benchmark has a five warm-up iteration, and the application returns this speed metric in milliseconds.

11. Ready memory

This benchmark calculates the amount of memory used after the page is loaded. Total memory usage metric displayed in megabytes.

12. Run memory

This benchmark calculates the amount of memory used after adding a thousand rows in the table page. Total memory usage metric displayed in megabytes.

13. Insert DB (database)

This function inserts a thousand rows into the database using a RESTful API. First, it generates a thousand rows containing three words in each row. The data is parsed into a JSON object and then sent in the body of the HTTP POST request. When the application gets the response, the data is presented to the user. The application registers the time before it starts to generate the random rows and after the data is presented to the user. It calculates the time difference between the start and the end of this operation. This benchmark starts by cleaning the database before starting the insertion operation.

14. Select DB (database)

This function selects a thousand rows from the database using a RESTful API. The application sends a GET request to the API which returns with a JSON object containing a thousand rows from the database. It parses and presents the data to the user. The application registers the time before it sends the request to the server and after the data is presented to the user. It calculates the time difference between the start and the end of this operation. This benchmark starts by cleaning the database and inserting a thousand rows into the database before the selection operation starts.

15. Update DB (database)

This function updates a thousand rows in the database using a RESTful API. The application sends a PUT request to the API with the data to be updated in the body of the request in a JSON format. When the application receives the response, it parses and presents the data to the user. The application registers the time before it sends the request to the server and after the data is presented to the user. It calculates the time difference between the start and the end of this operation. This benchmark starts by cleaning the database and inserting a thousand rows into the database before the update operation starts.

16. Delete DB (database)

This function deletes a thousand rows from the database using a RESTful API. The application sends a DELETE request to the API which returns an HTTP status code. When the application receives the response, it cleans all the rows from the table. The application calculates the time difference between the start and the end of this operation. This benchmark starts by cleaning the database and inserting a thousand rows into the database before the update operation starts.

3.3 RESTful API

Representational state transfer (REST) is an architecture style for creating networked applications, and it can be defined as a conceptual abstraction of the basic HTTP architecture. It was created to provide interoperability between computer systems using a network, but unlike the complex alternative technologies like SOAP or CORBA, REST is a lightweight, simple to build and maintain (Schreier, 2011).

RESTful applications use HTTP request to send data (create or update), receive data (read), and delete data. Therefore, RESTful applications follow the four essential functions of persistence storage which are: create, read, update, and delete (CRUD). RESTful applications make use of the HTTP protocol to manage security and encryption (e.g. HTTPS protocol).

REST can be implemented in any language (e.g. Java, JavaScript) and it does not depend on the platform that runs it (e.g. Linux, Windows). Its simplicity, scalability, and portability are the core advantages of using this technology to integrate systems using the Internet.

A REST request is executed through an URL. For instance, the following URL:

https://js-benchmark-185712.appspot.com/api/data

A simple request using the HTTP method GET will return data in a JSON format. The same URL can be used to execute all the CRUD operations on the resource (/data). Although this API will return a JSON object, REST can adopt different response formats like CSV (comma-separated values) or XML (eXtensible Markup Language).

XML JSON

```
<response >                              {
<id>1</1>                                "id":1
<label>pretty red table</label>          "label":"pretty red table"
</response>                              }
```

JavaScript Object Notation or JSON[10] is a simple format for structured data. It is an alternative for the complex XML notation, and it is used primarily to transfer data between a server and a web application. The concept of a JSON object is based on the key/value pairs.

- Key: A key represents an identifier of the information
- Value: It is the information that needs to be transferred

A key is always a string enclosed in quotation marks. A value can take any format, also enclosed in quotation marks (e.g. number, strings, arrays).

js-benchmark RESTful API

JavaScript frameworks help front-end developers to create fast and easy to maintain applications. The frameworks also provide HTTP methods to access information in a database through an API. In this project, a RESTful API was developed to simulate a real environment where an application could execute all the CRUD operations.

The API was developed in Node.js using a MySQL database.

Method	Default Value	Description
GET	<empty>	Return all the values in the database
POST	JSON	Insert the values in the database
PUT	JSON	Update the values in the database
DELETE	<empty>	Clear the database

Table 3.1 API specification

The database schema consists of a table with two columns.

Column	Type	Description
Id	Int(11)	It contains a unique identifier for the value stored
label	Varchar(255)	It contains the random text generated by the application

Table 3.2 Database specification

[10] https://tools.ietf.org/html/rfc4627

3.4 Google Cloud Platform

Google offers a cloud computing based platform called Google Cloud Platform (GCP). The service uses the same infrastructure that Google uses internally for its own products such as YouTube or the search engine. Google has a massive infrastructure scattered around the globe with high expertise in scalability and security measures. They compete directly with other cloud services such as Amazon Web Services and Microsoft Azure. They offer solutions for computation, storage, networking, big data, and machine learning.

The Cloud Platform resources are a set of physical assets which includes computers and HDs, and virtual resources such as Virtual Machines (VMs). All these resources are available in the Google Data Centres around the globe. Data Centres are located globally in several regions. Available regions include Central US, Western Europe, and East Asia. A region is a collection of zones. Zones are isolated from each other within the region. For example, *c* is a zone in west Europe *europe-west2-c*.

This distribution of resources provides a series of benefits including high availability and failure control due to the resource location. This distribution states some security rules of how they can be used together.

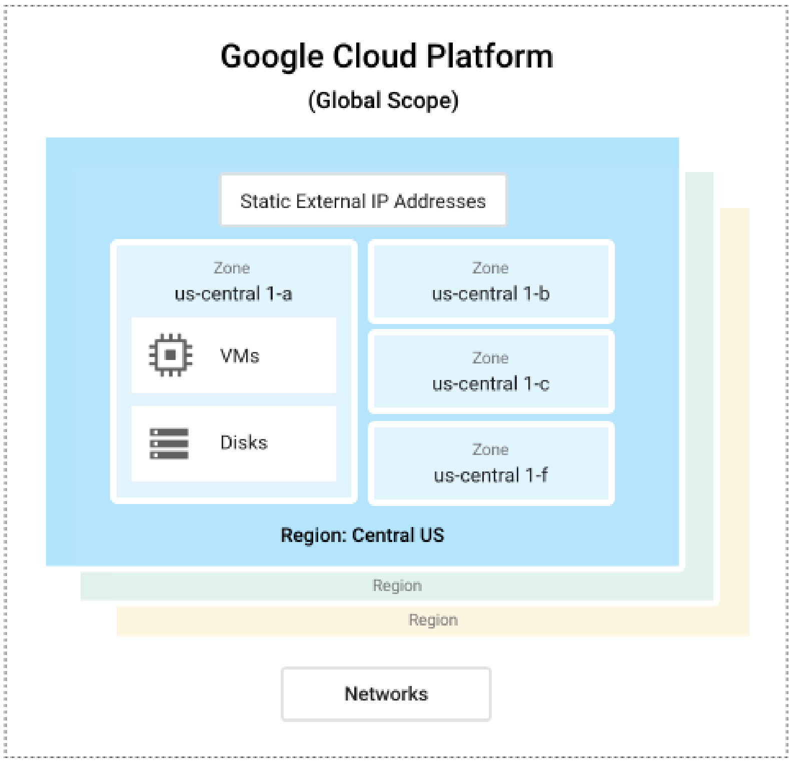


Figure 3.2 Google Cloud Platform Architecture

Note: Retrieved from *"Google Cloud Documentation"*, https://cloud.google.com/docs/overview/, accessed on 02/11/2017.

GCP offers infrastructure and software as a service that can grow according to the application necessity. In this project, europe-west2 was chosen, which is based in London, U.K.

The Google App Engine runs on Virtual Machines, and it supports a variety of languages, libraries, and frameworks such as Java, PHP, and Node.JS. GCP also provides a fully-managed database service to manage relational databases such as MySQL and PostgreSQL.

3.5 Network

From the beginning of 2000's, the internet has been experienced a considerable growth in the need for speed and stable network connections as streaming services and applications which heavily depend on the use of the internet are becoming widely used. These applications intensively use the internet, and they use a significant amount of bandwidth.

Network administrators have been tackling this issue by limiting the bandwidth of each computer connected to a network where many users compete for more data to be consumed.

Transmission rate in networks is usually measured in Mbps. Mbps or Mbit/s means megabit per second. It is a data transmission unit equivalent to 1,000 kilobits per second or 1,000,000 bits per second. For instance, streaming a VHS video quality needs 2Mbps, a DVD video quality needs 8Mbps, and HDTV video quality needs 55Mbps. The rate varies according to the quality desired. In other words, Mbps is the throughput used in serial communications and measures the number of megabits that are transferred per second.

In this project, the Dublin Institute of Technology (DIT) network infrastructure was used to run the experiment. DIT is built on Extreme switches with 100Mbit of LAN connections to the labs where the experiment was conducted. The average internet speed connection is reduced to 13Mbps maximum in the wired connection. They use a fiber connection to the backbone with a 1Gbit link. The wireless network uses Enterasys wireless AP, and it can support 254 simultaneous users. The average internet speed connection is reduced to 10Mbps maximum in the wireless connection.

Specs		**Wired**	**Wireless**
Internet Speed Connection		13Mbps	10Mbps
Hardware	Model	Extreme 7100-Series	AP 3705
	Speed	100Mbit	10Mbit

Table 3.3 Network specification

3.6 Experiment Design

In the preliminary literature review, approximately sixty different JavaScript frameworks (JSFs) entries were identified for performance evaluation. Comparing all possible JSFs would be difficult to complete within the timeframe. Therefore, the three JSFs selected are AngularJs, Ember, and Aurelia.

Lines of code (LOC) is the oldest metric for measuring the efficiency (Gizas et al., 2012b). The IEEE has standardised two methods of counting lines of code; Physical Lines of Code (SLOC) and Logical Lines of Code (LLOC) (Park, 1992). SLOC is the real number of code lines written in a programme excluding the comment lines, and LLOC is the number of executable statements such as functions and procedures in a piece of code.

Cyclomatic complexity is another metric used to measure the complex nature of a program (T. J. McCabe, 1976). It measures the number of linearly-independent paths through a program's source code. In other words, this metric checks the logic gates that a program uses during its execution and calculates the source code complexity.

Concerns about software quality are measured through the maintainability index which measures how easy it is to maintain the code. The maintainability index metric uses a series of formulas containing the Source Lines of Code and the Cyclomatic Complexity (Mariano, 2017). This metric also gives insights about the quality of the software.

The complexity-report tool[11] offers a complete analysis of size and complexity metrics such as LOC, cyclomatic complexity, and Halstead effort. This package is incorporated to the NPM which can be reached by the Node.js server. Metrics related to size and complexity of the code will be extracted using this tool, and the comparison among the chosen frameworks will be based on the numbers collected from these metrics.

Speed metrics will be assessed with the aid of the benchmark application. The execution time of JavaScript operations will be recorded and compared.

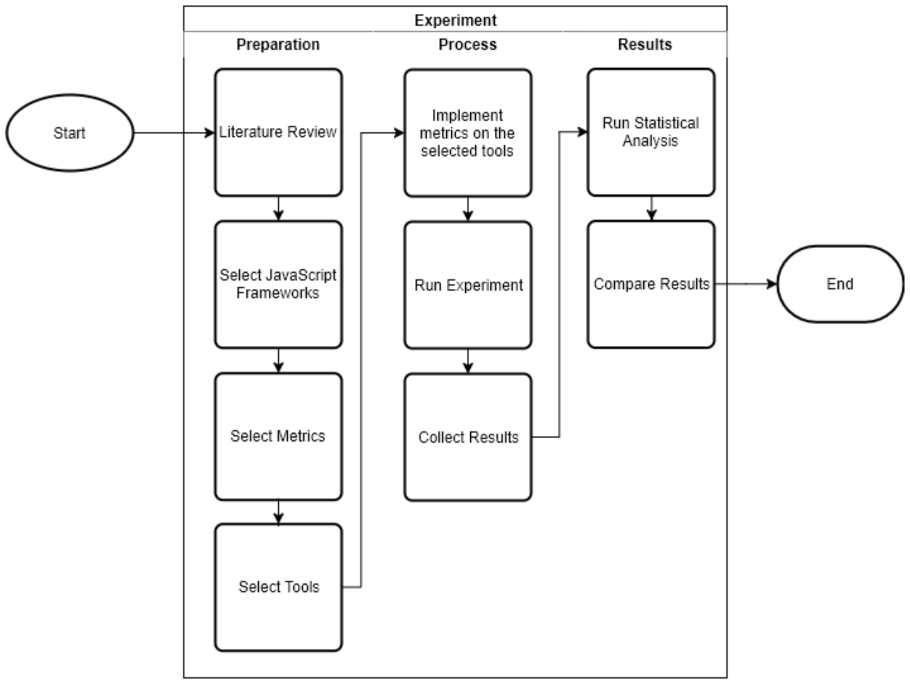


Figure 3.3 Experiment design

[11] https://github.com/escomplex/complexity-report

3.7 Chapter summary

This chapter presented an overview of the design and methodology of this research. The chapter started with the explanation of the JavaScript framework benchmark and its metrics. Then it introduced Cloud Platform which hosts the RESTful API and the application database. The network specification was also covered in this chapter.

4 IMPLEMENTATION AND RESULTS

4.1 Introduction

This chapter describes the development and structure of different software for building the benchmark application which were used to run the experiments. Aspects related to the environment, configurations, results, and problems encountered during the process of implementation will also be discussed in this chapter.

4.2 Software used

This thesis aims to compare JavaScript frameworks using benchmarking metrics. Therefore, JavaScript is the programming language used for the software development. Three JavaScript frameworks were selected for this comparison given the time constraints of this project. The JavaScript frameworks are AngularJS, Ember, and Aurelia. The versions and feature comparisons are described in Chapter 2 section 2.2.4. Visual Studio Code[12] (v. 1.19) is a source code editor developed and distributed by Microsoft, and it is supported in Windows, Linux, and MacOS. It includes support for debugging, intelligent code complementation, syntax highlighting, embedded Git control, and code refactoring. The code editor was chosen for this project due to the familiarity of the researcher to the program and its intuitive interface.

The framework was built on the Node.js platform. Node.js is built on the Google Chrome JavaScript engine to build fast and scalable network applications easily. Node.js uses a non-blocking event-driven input/output model that makes it lightweight and efficient, ideal for real-time applications with robust data exchange across distributed devices. Node.js is a server, but unlike Apache or Tomcat web servers, which are ready to build and run applications instantly, Node.js has the concept of modules that can be added to the core of Node.js. There are literally hundreds of modules to run with Node.js, and the community is very active in producing, publishing, and updating dozens of modules per day. These modules can be found in the Node Package Manager (NPM).

[12] https://code.visualstudio.com/

NPM is an online repository for publishing open source projects for Node.js. It is also a command line utility that interacts with this online repository, which helps with package installation, version management, and dependency management.

4.3 *Framework implementations*

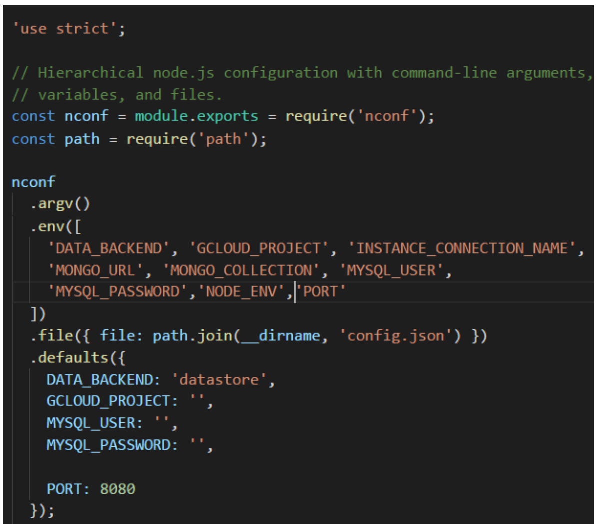
This section presents a detailed description of the framework implementations to run the performance benchmarks. Specifically, the implementation of the interaction with the cloud server. Each framework requests data from the server using their own libraries. These features are discussed in 2.2.4.

4.3.1 RESTful API

The restful API was built in node.js. The application was built with the Express API[13] which is a framework for creating routes, middleware and other functions. Express API handles the requests by associating the URL and the HTTP method to an action in the application.

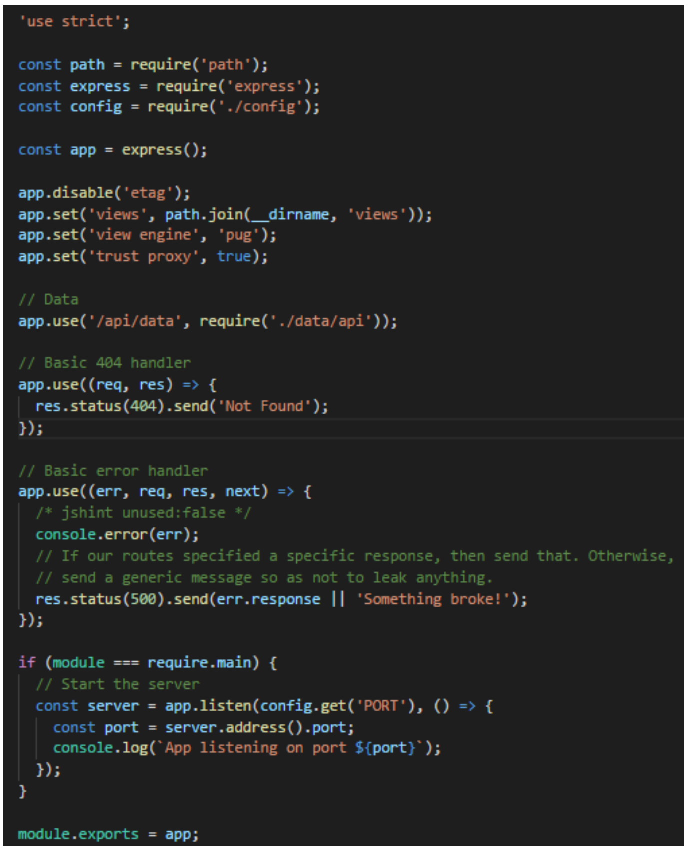
Figure 4.2 shows the application server code. The server will be listening to the configured port in the configuration file. The configuration file (Figure 4.1) contains the credential access to the Google Cloud Server and the method of accessing it. The code bellow routes the URL to the HTTP methods implemented in the api.js

```
        app.use('/api/data', require('./data/api'));
```



```
'use strict';

// Hierarchical node.js configuration with command-line arguments,
// variables, and files.
const nconf = module.exports = require('nconf');
const path = require('path');

nconf
  .argv()
  .env([
    'DATA_BACKEND', 'GCLOUD_PROJECT', 'INSTANCE_CONNECTION_NAME',
    'MONGO_URL', 'MONGO_COLLECTION', 'MYSQL_USER',
    'MYSQL_PASSWORD','NODE_ENV','PORT'
  ])
  .file({ file: path.join(__dirname, 'config.json') })
  .defaults({
    DATA_BACKEND: 'datastore',
    GCLOUD_PROJECT: '',
    MYSQL_USER: '',
    MYSQL_PASSWORD: '',

    PORT: 8080
  });
```

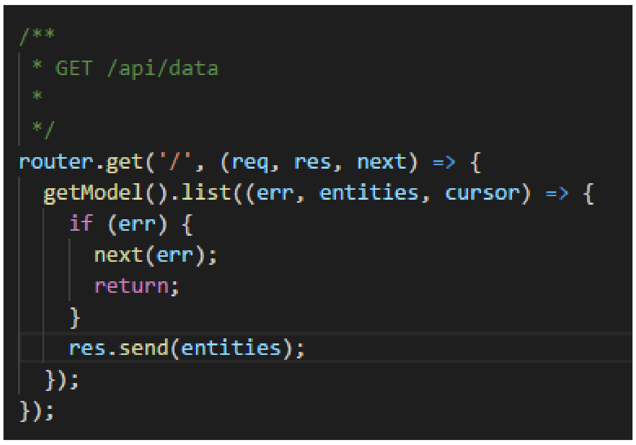
Figure 4.1 Configuration File



```
'use strict';

const path = require('path');
const express = require('express');
const config = require('./config');

const app = express();

app.disable('etag');
app.set('views', path.join(__dirname, 'views'));
app.set('view engine', 'pug');
app.set('trust proxy', true);

// Data
app.use('/api/data', require('./data/api'));

// Basic 404 handler
app.use((req, res) => {
  res.status(404).send('Not Found');
});

// Basic error handler
app.use((err, req, res, next) => {
  /* jshint unused:false */
  console.error(err);
  // If our routes specified a specific response, then send that. Otherwise,
  // send a generic message so as not to leak anything.
  res.status(500).send(err.response || 'Something broke!');
});

if (module === require.main) {
  // Start the server
  const server = app.listen(config.get('PORT'), () => {
    const port = server.address().port;
    console.log(`App listening on port ${port}`);
  });
}

module.exports = app;
```

Figure 4.2 Server for the RESTful API

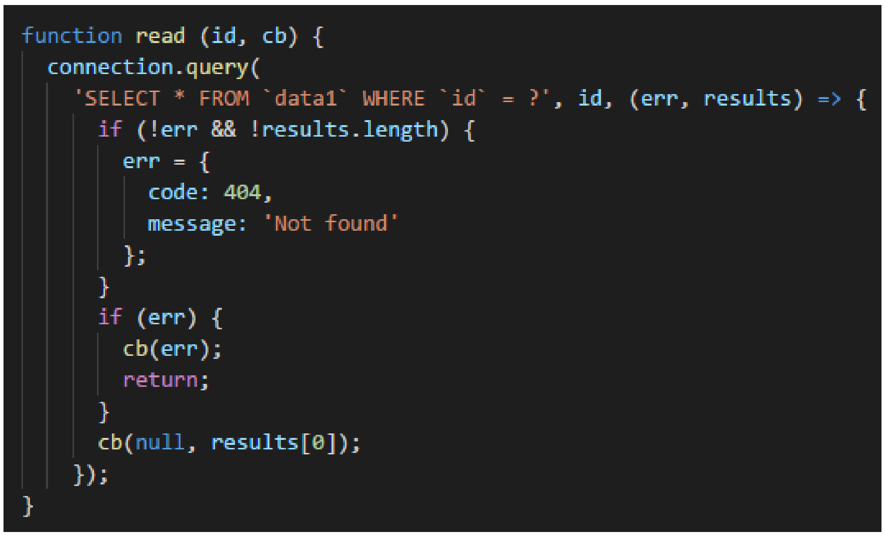
Figure 4.3 and Figure 4.4 shows the code to request data from the database. Once the application receives a GET request, the Express API identifies the route and request the model in which the database is configured. The model contains the access credentials to the database and the SQL statements to query the data. Figure 4.3 shows the code snippet in data/api,js and Figure 4.4 shows the code snippet in data/model-cloudsql.js



```
/**
 * GET /api/data
 *
 */
router.get('/', (req, res, next) => {
  getModel().list((err, entities, cursor) => {
    if (err) {
      next(err);
      return;
    }
    res.send(entities);
  });
});
```

Figure 4.3 GET method

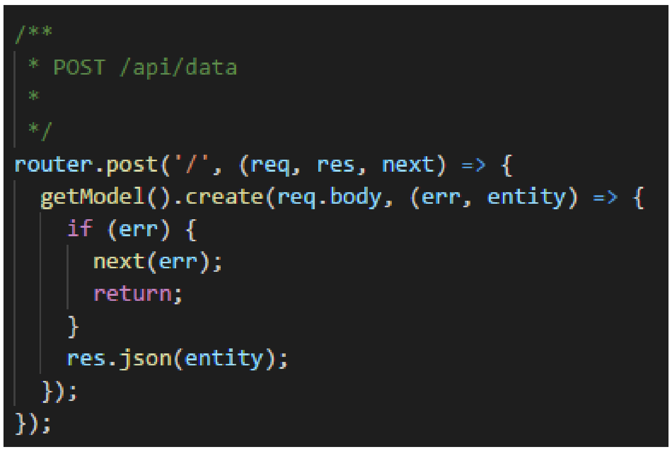
[13] https://expressjs.com/



```
function read (id, cb) {
  connection.query(
    'SELECT * FROM `data1` WHERE `id` = ?', id, (err, results) => {
      if (!err && !results.length) {
        err = {
          code: 404,
          message: 'Not found'
        };
      }
      if (err) {
        cb(err);
        return;
      }
      cb(null, results[0]);
    });
}
```

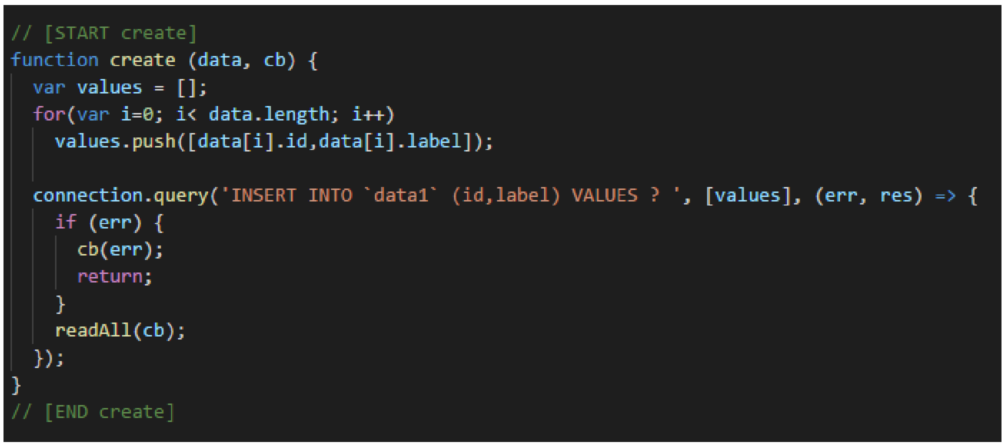
Figure 4.4 Select from the database

POST method inserts data in the database with data embedded in the request body. When the application detects a POST request, it uses the model to create an insert SQL statement using the data send in the body of the POST request. After inserting the data in the table ‘data1’, the function returns the data inserted within the response object.
Figure 4.5 shows the code snippet in data/api,js and Figure 4.6 shows the code snippet in data/model-cloudsql.js



```
/**
 * POST /api/data
 *
 */
router.post('/', (req, res, next) => {
  getModel().create(req.body, (err, entity) => {
    if (err) {
      next(err);
      return;
    }
    res.json(entity);
  });
});
```

Figure 4.5 POST method

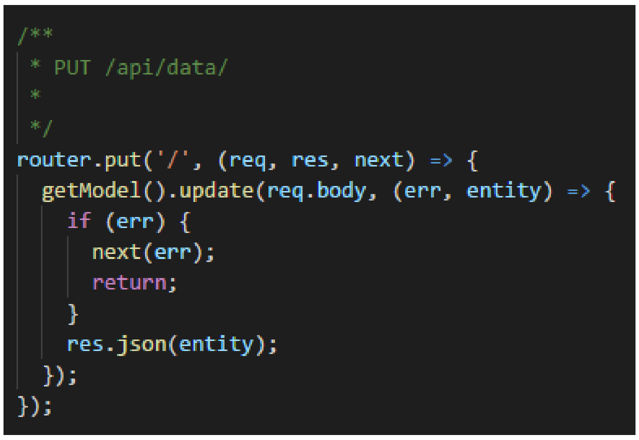


```
// [START create]
function create (data, cb) {
  var values = [];
  for(var i=0; i< data.length; i++)
    values.push([data[i].id,data[i].label]);

  connection.query('INSERT INTO `data1` (id,label) VALUES ? ', [values], (err, res) => {
    if (err) {
      cb(err);
      return;
    }
    readAll(cb);
  });
}
// [END create]
```

Figure 4.6 Insert into the database

Figure 4.7 and Figure 4.8 shows the process of updating data in the database. The code starts by detecting the PUT request and instantiating the model with the data sent in the body of the request. The database will update the rows in the table ‘data1’ by inserting the values where the primary key is duplicated. After updating the table, the function returns the data updated within the response object.

Figure 4.7 shows the code snippet in data/api,js and Figure 4.8 shows the code snippet in data/model-cloudsql.js



```
/**
 * PUT /api/data/
 *
 */
router.put('/', (req, res, next) => {
  getModel().update(req.body, (err, entity) => {
    if (err) {
      next(err);
      return;
    }
    res.json(entity);
  });
});
```

Figure 4.7 PUT method

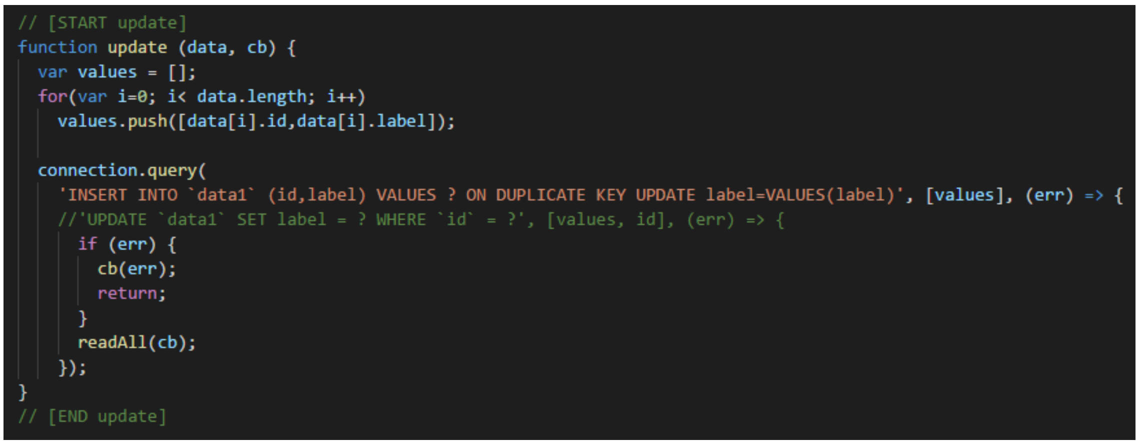

```
// [START update]
function update (data, cb) {
  var values = [];
  for(var i=0; i< data.length; i++)
    values.push([data[i].id,data[i].label]);

  connection.query(
    'INSERT INTO `data1` (id,label) VALUES ? ON DUPLICATE KEY UPDATE label=VALUES(label)', [values], (err) => {
    //'UPDATE `data1` SET label = ? WHERE `id` = ?', [values, id], (err) => {
      if (err) {
        cb(err);
        return;
      }
      readAll(cb);
    });
}
// [END update]
```

Figure 4.8 Update the database with Insert

Figure 4.9 and Figure 4.10 shows the code to delete data from the database. The operation deletes all the data in the table ‘data1’ and returns the status code 200 (OK) in the response.

Figure 4.9 shows the code snippet in data/api,js and Figure 4.10 shows the code snippet in data/model-cloudsql.js

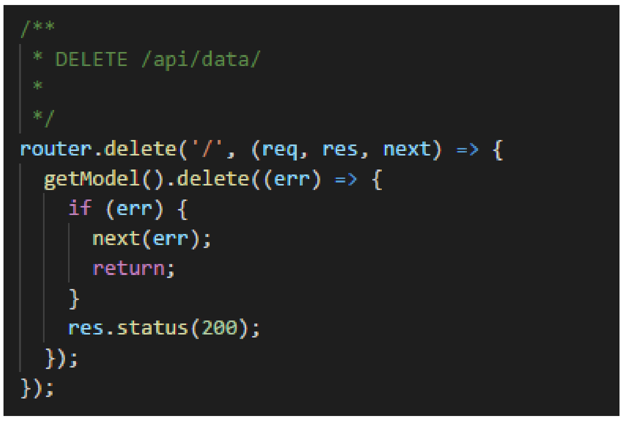

```
/**
 * DELETE /api/data/
 *
 */
router.delete('/', (req, res, next) => {
  getModel().delete((err) => {
    if (err) {
      next(err);
      return;
    }
    res.status(200);
  });
});
```

Figure 4.9 DELETE method

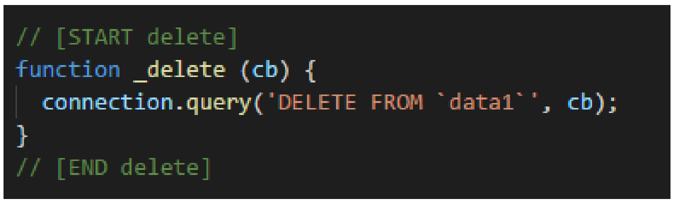
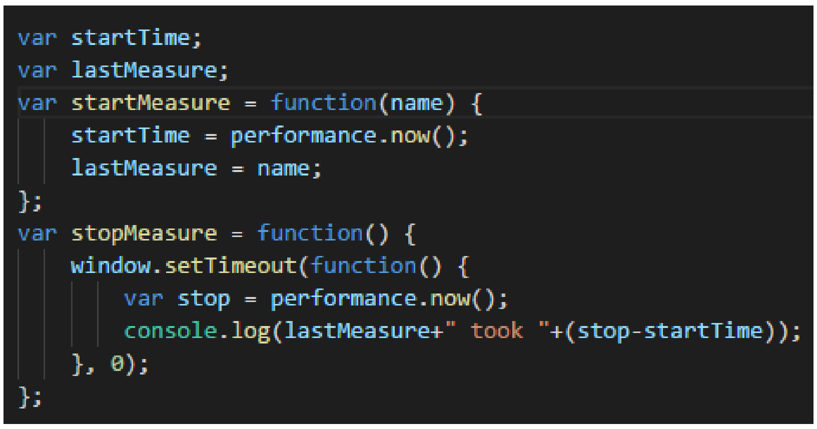

```
// [START delete]
function _delete (cb) {
  connection.query('DELETE FROM `data1`', cb);
}
// [END delete]
```

Figure 4.10 Delete from the database

4.3.2 Clock settings

This section shows the implementation of the clock which will record the execution time of the JSFs. Performance.now() provides a high-resolution timestamp in a precision needed for benchmarking processes. Section 2.2.5 describes the clock specifications in detail.

Figure 4.11 shows the clock settings for record the execution time of each functionality implemented using the JSFs selected. The function expression called *startMeasure* records the time before the execution of the functionality. The function stores the data in a global variable and records the name of the functionality being tested (e.g. InsertDB). *stopMeasure* is the function expression called when a functionality from the benchmark has finished. The function records the time that the function finished and calculate the difference between the start and end of the operation.



```
var startTime;
var lastMeasure;
var startMeasure = function(name) {
    startTime = performance.now();
    lastMeasure = name;
};
var stopMeasure = function() {
    window.setTimeout(function() {
        var stop = performance.now();
        console.log(lastMeasure+" took "+(stop-startTime));
    }, 0);
};
```

Figure 4.11 Clock implementation

The result is printed in the console, and the benchmark will use this information to calculate the final score of each function. The clock is implemented in each JSF selected.

4.3.3 Processes

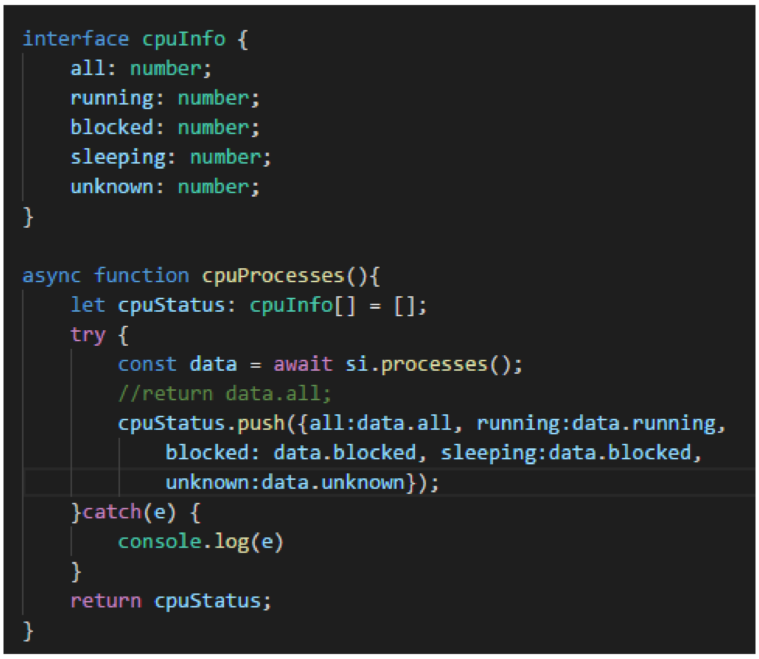
Node offers a module called systeminformation[14] which can retrieve detailed hardware, system and OS information from Windows, Linux and OSX. This module was used to retrieve CPU status and the number of running processes.

The module is initiated by the following code:

```
        const si = require('systeminformation');
```

[14] https://www.npmjs.com/package/systeminformation

Figure 4.12 shows the code snippet of retrieving all the running processes in the CPU. The function processes() returns an array of information about processes running, blocked, sleeping and unknown.

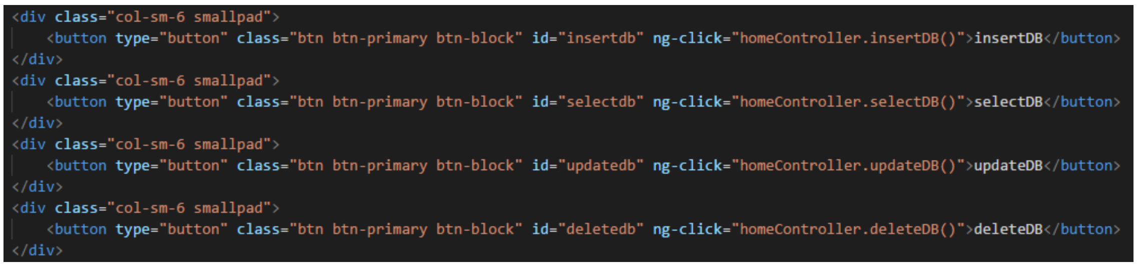


```
interface cpuInfo {
    all: number;
    running: number;
    blocked: number;
    sleeping: number;
    unknown: number;
}

async function cpuProcesses(){
    let cpuStatus: cpuInfo[] = [];
    try {
        const data = await si.processes();
        //return data.all;
        cpuStatus.push({all:data.all, running:data.running,
            blocked: data.blocked, sleeping:data.blocked,
            unknown:data.unknown});
    }catch(e) {
        console.log(e)
    }
    return cpuStatus;
}
```

Figure 4.12 Code snippet of the System Information module

4.3.4 AngularJS

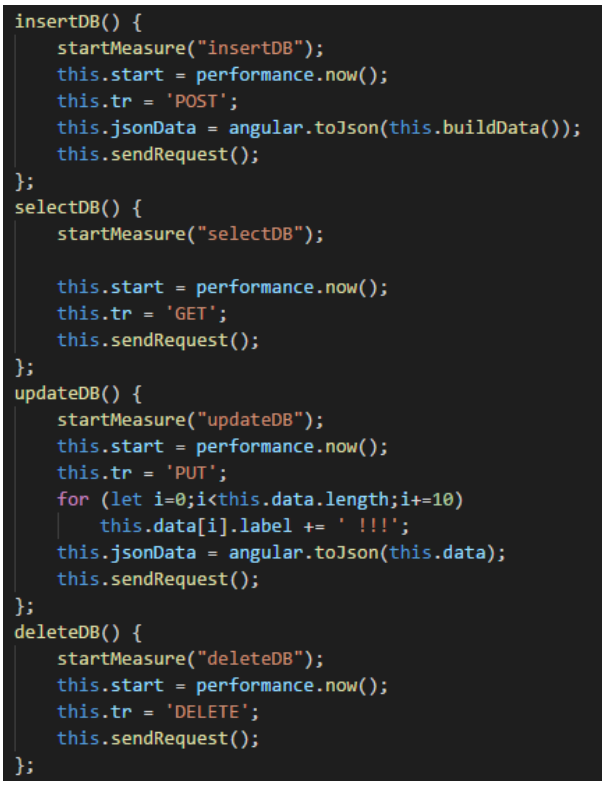
AngularJS works with the concept of directives and the ng-click works as a tag in the button element of the HTML. The ng-click directive allows the developer to create a custom behaviour when an element is clicked, similar to the on-click event of HTML. The directive calls a method in the controller of the application. Figure 4.13 shows the code for the CRUD operations.



```
<div class="col-sm-6 smallpad">
    <button type="button" class="btn btn-primary btn-block" id="insertdb" ng-click="homeController.insertDB()">insertDB</button>
</div>
<div class="col-sm-6 smallpad">
    <button type="button" class="btn btn-primary btn-block" id="selectdb" ng-click="homeController.selectDB()">selectDB</button>
</div>
<div class="col-sm-6 smallpad">
    <button type="button" class="btn btn-primary btn-block" id="updatedb" ng-click="homeController.updateDB()">updateDB</button>
</div>
<div class="col-sm-6 smallpad">
    <button type="button" class="btn btn-primary btn-block" id="deletedb" ng-click="homeController.deleteDB()">deleteDB</button>
</div>
```

Figure 4.13 AngularJS template

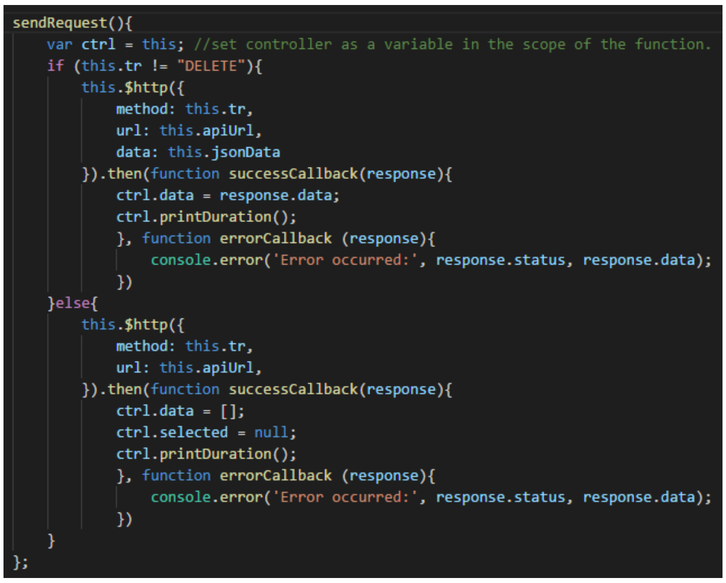
Figure 4.14 shows the code snippet of the database function in AngularJS. Each database function starts by calling the function *startMeasure* already discussed in 4.3.2. The insertDB() function generates a thousand rows with random data to be inserted in the database.



```
insertDB() {
    startMeasure("insertDB");
    this.start = performance.now();
    this.tr = 'POST';
    this.jsonData = angular.toJson(this.buildData());
    this.sendRequest();
};
selectDB() {
    startMeasure("selectDB");

    this.start = performance.now();
    this.tr = 'GET';
    this.sendRequest();
};
updateDB() {
    startMeasure("updateDB");
    this.start = performance.now();
    this.tr = 'PUT';
    for (let i=0;i<this.data.length;i+=10)
        this.data[i].label += ' !!!';
    this.jsonData = angular.toJson(this.data);
    this.sendRequest();
};
deleteDB() {
    startMeasure("deleteDB");
    this.start = performance.now();
    this.tr = 'DELETE';
    this.sendRequest();
};
```

Figure 4.14 Database functions in AngularJS

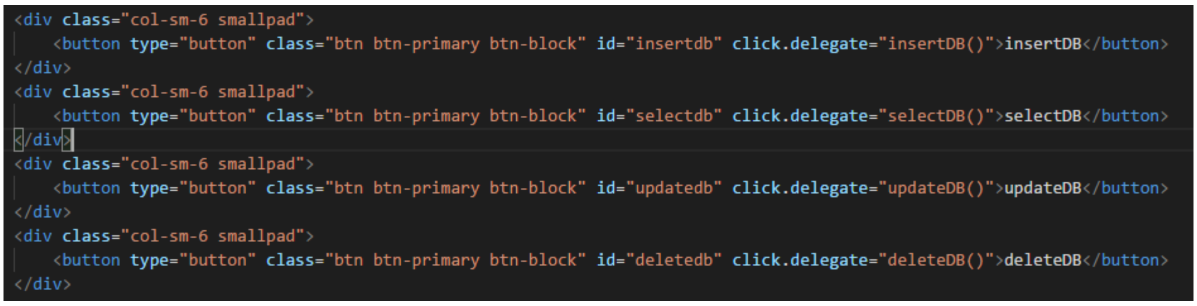
Figure 4.15 shows how AngularJS calls the RESTful API. The clock stops only after the data has been returned to the user.



```
sendRequest(){
    var ctrl = this; //set controller as a variable in the scope of the function.
    if (this.tr != "DELETE"){
        this.$http({
            method: this.tr,
            url: this.apiUrl,
            data: this.jsonData
        }).then(function successCallback(response){
            ctrl.data = response.data;
            ctrl.printDuration();
            }, function errorCallback (response){
                console.error('Error occurred:', response.status, response.data);
            })
    }else{
        this.$http({
            method: this.tr,
            url: this.apiUrl,
        }).then(function successCallback(response){
            ctrl.data = [];
            ctrl.selected = null;
            ctrl.printDuration();
            }, function errorCallback (response){
                console.error('Error occurred:', response.status, response.data);
            })
    }
};
```

Figure 4.15 Calling the RESTful API in AngularJS

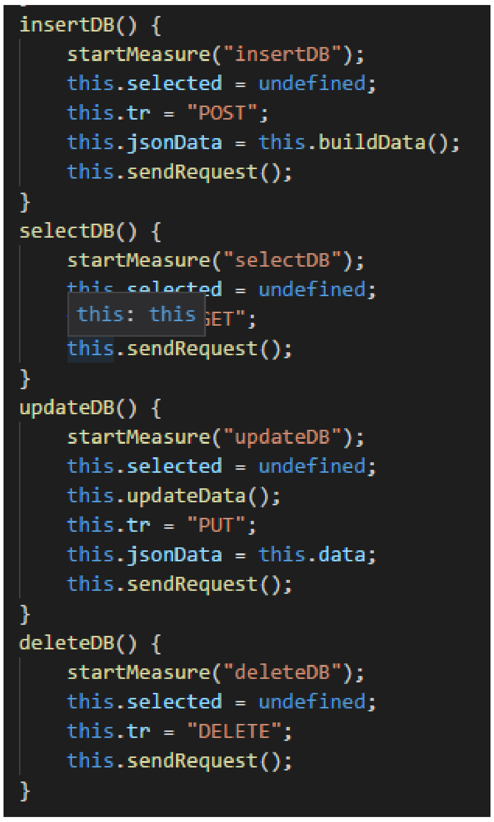
4.3.5 Aurelia



```
<div class="col-sm-6 smallpad">
    <button type="button" class="btn btn-primary btn-block" id="insertdb" click.delegate="insertDB()">insertDB</button>
</div>
<div class="col-sm-6 smallpad">
    <button type="button" class="btn btn-primary btn-block" id="selectdb" click.delegate="selectDB()">selectDB</button>
</div>
<div class="col-sm-6 smallpad">
    <button type="button" class="btn btn-primary btn-block" id="updatedb" click.delegate="updateDB()">updateDB</button>
</div>
<div class="col-sm-6 smallpad">
    <button type="button" class="btn btn-primary btn-block" id="deletedb" click.delegate="deleteDB()">deleteDB</button>
</div>
```

Figure 4.16 Aurelia template

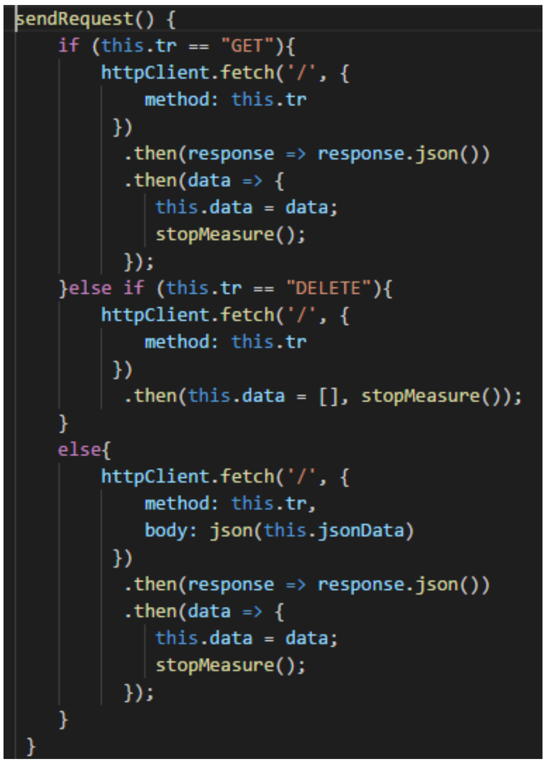
Figure 4.17 shows the code snippet of the database function in AngularJS. Each database function starts by calling the function *startMeasure* already discussed in 4.3.2.



```
insertDB() {
    startMeasure("insertDB");
    this.selected = undefined;
    this.tr = "POST";
    this.jsonData = this.buildData();
    this.sendRequest();
}
selectDB() {
    startMeasure("selectDB");
    this.selected = undefined;
    this: this ET";
    this.sendRequest();
}
updateDB() {
    startMeasure("updateDB");
    this.selected = undefined;
    this.updateData();
    this.tr = "PUT";
    this.jsonData = this.data;
    this.sendRequest();
}
deleteDB() {
    startMeasure("deleteDB");
    this.selected = undefined;
    this.tr = "DELETE";
    this.sendRequest();
}
```

Figure 4.17 Database functions in Aurelia

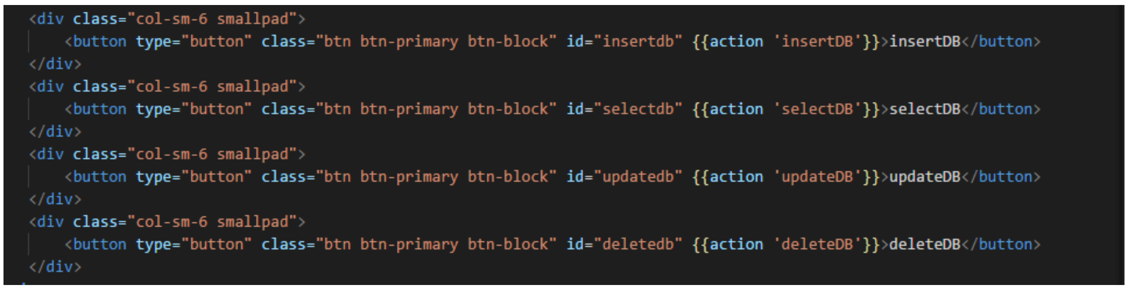
Figure 4.18 shows how AngularJS calls the RESTful API. The clock stops only after the data has been returned to the user.



```
sendRequest() {
    if (this.tr == "GET"){
        httpClient.fetch('/', {
            method: this.tr
        })
         .then(response => response.json())
         .then(data => {
            this.data = data;
            stopMeasure();
         });
    }else if (this.tr == "DELETE"){
        httpClient.fetch('/', {
            method: this.tr
        })
         .then(this.data = [], stopMeasure());
    }
    else{
        httpClient.fetch('/', {
            method: this.tr,
            body: json(this.jsonData)
        })
         .then(response => response.json())
         .then(data => {
            this.data = data;
            stopMeasure();
         });
    }
}
```

Figure 4.18 Calling the RESTful API in Aurelia

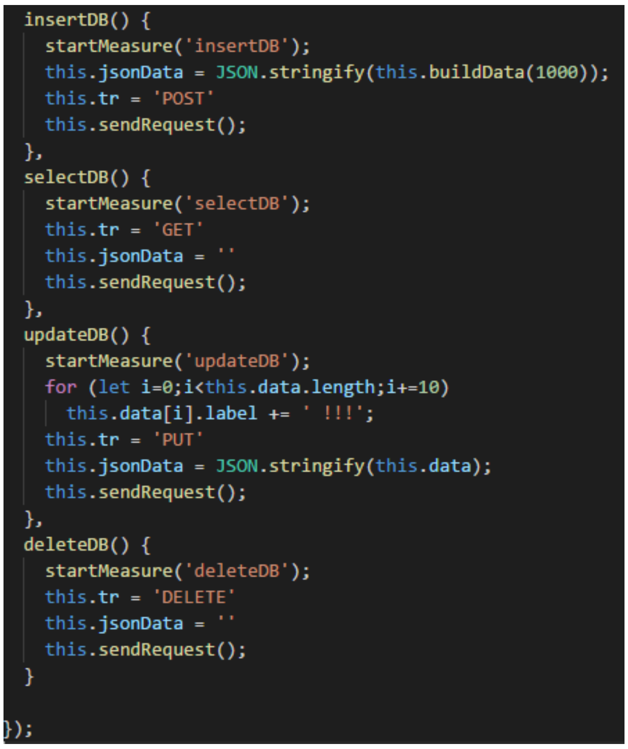
4.3.6 Ember



```
<div class="col-sm-6 smallpad">
    <button type="button" class="btn btn-primary btn-block" id="insertdb" {{action 'insertDB'}}>insertDB</button>
</div>
<div class="col-sm-6 smallpad">
    <button type="button" class="btn btn-primary btn-block" id="selectdb" {{action 'selectDB'}}>selectDB</button>
</div>
<div class="col-sm-6 smallpad">
    <button type="button" class="btn btn-primary btn-block" id="updatedb" {{action 'updateDB'}}>updateDB</button>
</div>
<div class="col-sm-6 smallpad">
    <button type="button" class="btn btn-primary btn-block" id="deletedb" {{action 'deleteDB'}}>deleteDB</button>
</div>
```

Figure 4.19 Ember template

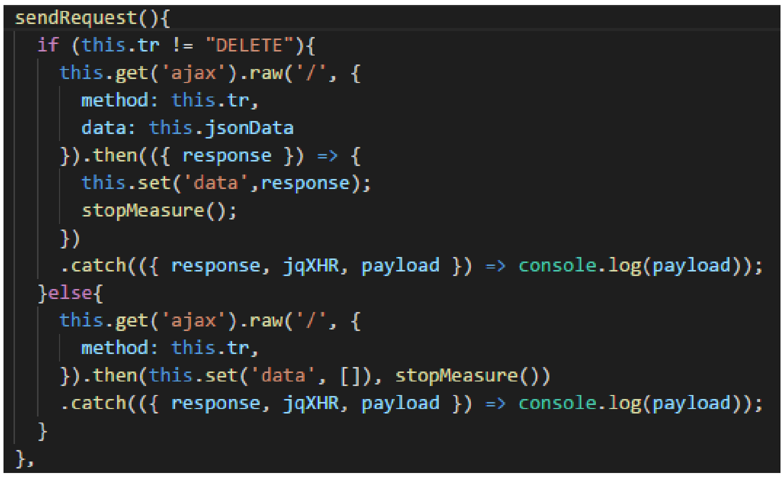
Figure 4.20 shows the code snippet of the database function in AngularJS. Each database function starts by calling the function *startMeasure* already discussed in 4.3.2.



```
insertDB() {
  startMeasure('insertDB');
  this.jsonData = JSON.stringify(this.buildData(1000));
  this.tr = 'POST'
  this.sendRequest();
},
selectDB() {
  startMeasure('selectDB');
  this.tr = 'GET'
  this.jsonData = ''
  this.sendRequest();
},
updateDB() {
  startMeasure('updateDB');
  for (let i=0;i<this.data.length;i+=10)
    this.data[i].label += ' !!!';
  this.tr = 'PUT'
  this.jsonData = JSON.stringify(this.data);
  this.sendRequest();
},
deleteDB() {
  startMeasure('deleteDB');
  this.tr = 'DELETE'
  this.jsonData = ''
  this.sendRequest();
}

});
```

Figure 4.20 Database functions in AngularJS

Figure 4.21 shows how AngularJS calls the RESTful API. The clock stops only after the data has been returned to the user.



```
sendRequest(){
  if (this.tr != "DELETE"){
    this.get('ajax').raw('/', {
      method: this.tr,
      data: this.jsonData
    }).then(({ response }) => {
      this.set('data',response);
      stopMeasure();
    })
    .catch(({ response, jqXHR, payload }) => console.log(payload));
  }else{
    this.get('ajax').raw('/', {
      method: this.tr,
    }).then(this.set('data', []), stopMeasure())
    .catch(({ response, jqXHR, payload }) => console.log(payload));
  }
},
```

Figure 4.21 Calling the RESTful API in Ember

4.4 Environment configuration

The environment for running the benchmarks in windows was configured as section 2.2.5. In this case, Windows was left in its basic configuration with a minimum amount of processes running concurrently. The aim of this step was to determine the influence of concurrent processes in the operating system when running the benchmarks.

In the first run of tests, the number of processes was reduced to 67 processes running concurrently. During the second test run, the 156 processes running by default at system start were running. The impact of the number of processes running concurrently with the benchmark application will be discussed in detail in Chapter 5.

	Windows	**Linux**
Name	Microsoft Windows 10 Pro	Ubuntu 16.04.3 LTS
OS Version	10.0.16299 N/A Build 16299	Linux 4.10.0-38-generic
System Locale	en-ie;English (Ireland)	en_IE.UTF-8
System Model	HP Pavilion dm4 Notebook PC	
System Type	x64-based PC	
Network Cards	Intel(R) Centrino(R) Wireless-N 1030	
	Qualcomm Atheros AR8151 PCI-E Gigabit Ethernet Controller (NDIS 6.30)	
Processor	Intel Core I5	
	Intel64 Family 6 Model 42 Stepping 7 GenuineIntel ~2301 Mhz	
Node version	8.1.4	8.8.1
NPM version	5.3.0	5.4.2
Java version	1.8.0_151	1.8.0_151
Javac version	1.8.0_131	1.8.0_131
Google Chrome version	63.0.3239.108 (Official Build) (64-bit)	63.0.3239.108 (Official Build) (64-bit)

Table 4.1 System's specification and configuration

4.5 Results

The data collected after running the benchmark and the complexity evaluation tools are displayed in Appendix A.

The next chapter will present an evaluation and discussion of the results with some insights over the findings.

4.6 Chapter summary

This chapter described the implementations of the new functionalities in the JavaScript framework benchmark. These functionalities include the access to a database, an implementation of a RESTful API, and the CPU status during the benchmark execution. This chapter also described the environment in which the tests were conducted including hardware and software specifications.

5 ANALYSIS, EVALUATION AND DISCUSSION

5.1 *Introduction*

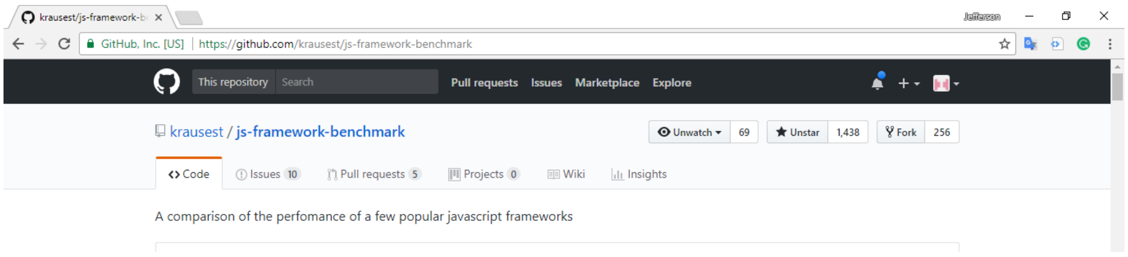
This chapter describes the analysis and evaluation process of the JavaScript frameworks. The chapter begins with the benchmark configuration and the environment preparation for running the performance tests. The first part of the experimentation and evaluation solely focuses on the performance and execution time of JavaScript frameworks. The second part of this evaluation refers to the benchmark metrics already discussed in previous literature such as Lines of Code and Halstead effort. This chapter ends with a brief discussion on the findings highlighting the strengths and weaknesses of this research.

5.2 *Experimentation*

This section describes the performance experiment related to the execution time of JavaScript frameworks. The original application is an on-going project hosted on GitHub, and it has an active community working to improve and add more relevant metrics to the benchmark framework. This project contains more than thirty JavaScript Frameworks implemented including AngularJS, Ember and Aurelia. These are the chosen JSF for this experiment.

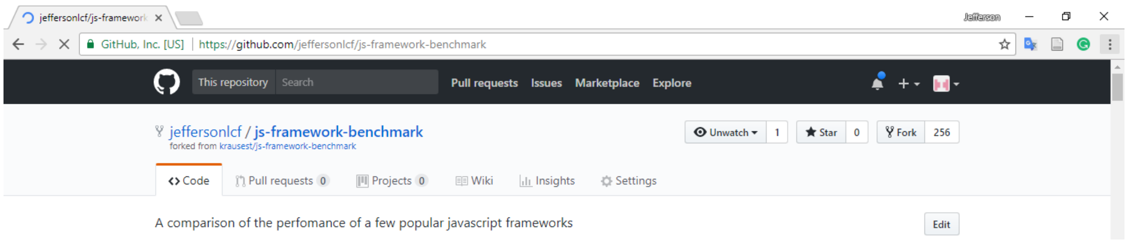
5.2.1 Preparing the experiment

GitHub hosts the js-benchmark-framework project, and this online repository offers the option Fork. Fork is a function within the GitHub platform that copies all the files in a repository to another repository making it available to edit without interfering the main project. This option allows individuals who want to contribute to the project to edit files and suggest modifications to the owner of the main project. GitHub also allows the merging of files of the main project with files of the forked project which can maintain both projects up-to-date with the changes.



A comparison of the perfomance of a few popular javascript frameworks

Figure 5.1 GitHub repository for the js-benchmark-framework



A comparison of the perfomance of a few popular javascript frameworks

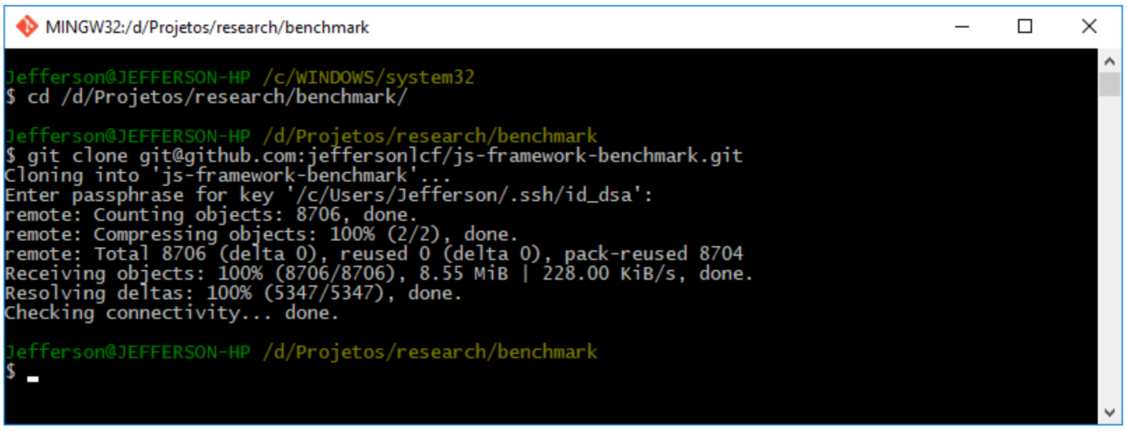
Figure 5.2 GitHub page of the forked project

After forking the main project Figure 5.1 which is under the @krauset user, the files are copied to the contributor repository Figure 5.2, which is now under the @jeffersonlcf user.

The next step is to make the repository available in the machine. In the Git Bash the following command needs to be executed:

```
git clone git@github.com:....git
```

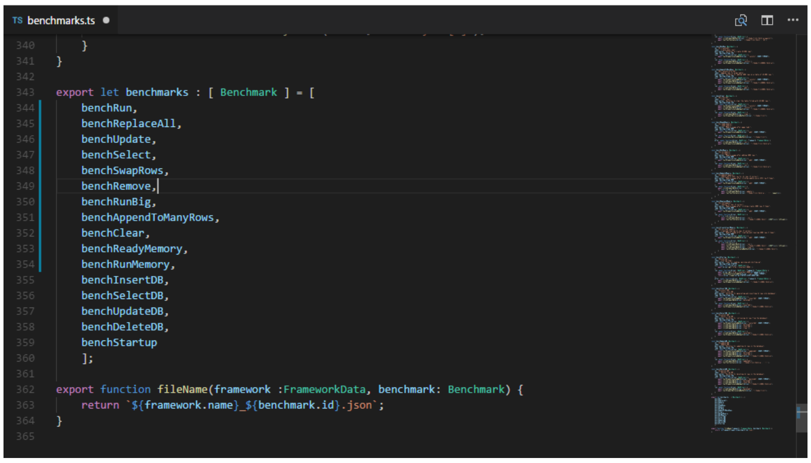
This command will download all the files to a specified directory and link with the repository for future commits of the changes.



```
Jefferson@JEFFERSON-HP /c/WINDOWS/system32
$ cd /d/Projetos/research/benchmark/

Jefferson@JEFFERSON-HP /d/Projetos/research/benchmark
$ git clone git@github.com:jeffersonlcf/js-framework-benchmark.git
Cloning into 'js-framework-benchmark'...
Enter passphrase for key '/c/Users/Jefferson/.ssh/id_dsa':
remote: Counting objects: 8706, done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 8706 (delta 0), reused 0 (delta 0), pack-reused 8704
Receiving objects: 100% (8706/8706), 8.55 MiB | 228.00 KiB/s, done.
Resolving deltas: 100% (5347/5347), done.
Checking connectivity... done.

Jefferson@JEFFERSON-HP /d/Projetos/research/benchmark
$ _
```

Figure 5.3 Cloning repository on GitHub

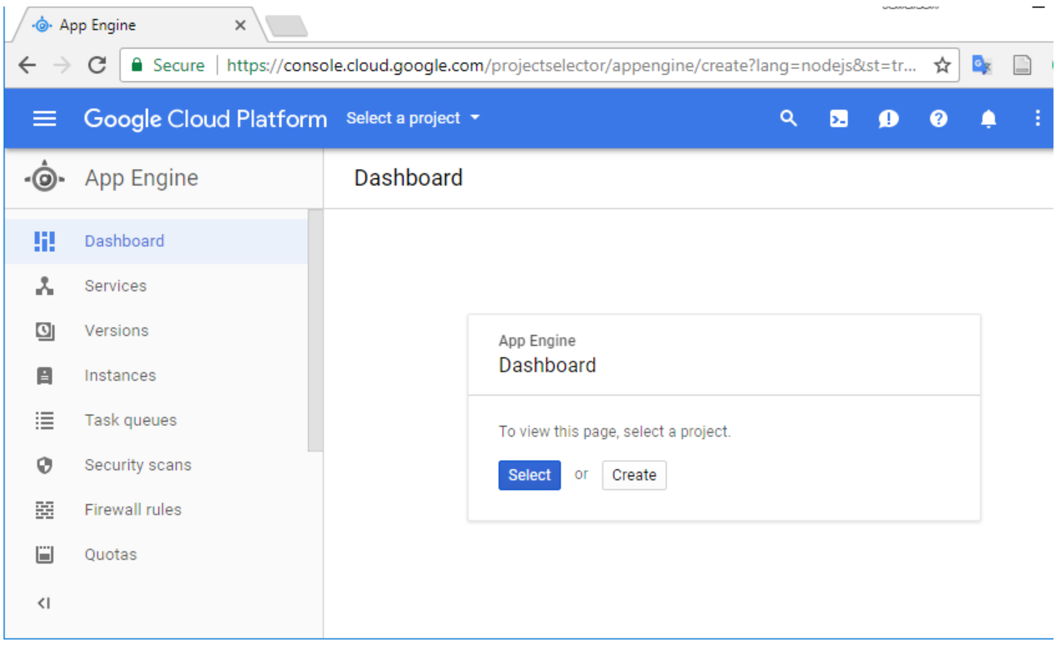
Figure 5.4 shows all the benchmarks that the framework will run. The file is in the directory \webdriver-ts\bencharks.ts



```
benchmarks.ts
    }
}

export let benchmarks : [ Benchmark ] = [
    benchRun,
    benchReplaceAll,
    benchUpdate,
    benchSelect,
    benchSwapRows,
    benchRemove,
    benchRunBig,
    benchAppendToManyRows,
    benchClear,
    benchReadyMemory,
    benchRunMemory,
    benchInsertDB,
    benchSelectDB,
    benchUpdateDB,
    benchDeleteDB,
    benchStartup
    ];

export function fileName(framework :FrameworkData, benchmark: Benchmark) {
    return `${framework.name}_${benchmark.id}.json`;
}
```

Figure 5.4 Snippet Code of benchmarks

The RESTful API runs on the Google Cloud Platform. First, it is necessary to login in to the platform and create a new project. Figure 5.5 shows the dashboard of the app engine for creating a new project. After creating a new project in the GCP, it is necessary to install the Google Cloud SDK. The SDK allows the deployment of the Node.js application in the platform.

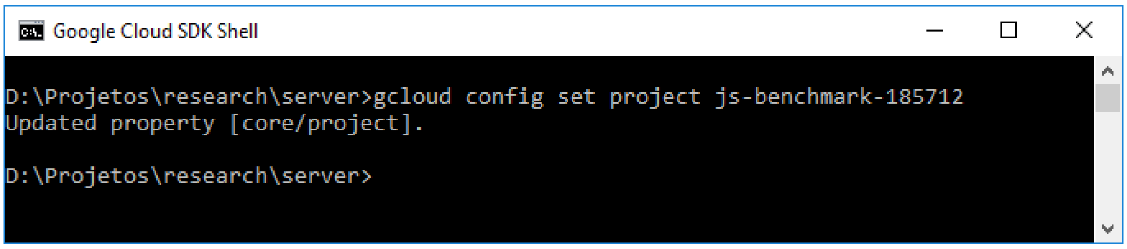


https://console.cloud.google.com/projectselector/appengine/create?lang=nodejs&st=tr...

To view this page, select a project.

Figure 5.5 Creating a new project in Google Cloud Platform

Before executing the application deployment, an environment needs to be set for the project created in the previous step. Figure 5.6 shows the command to set the working project.

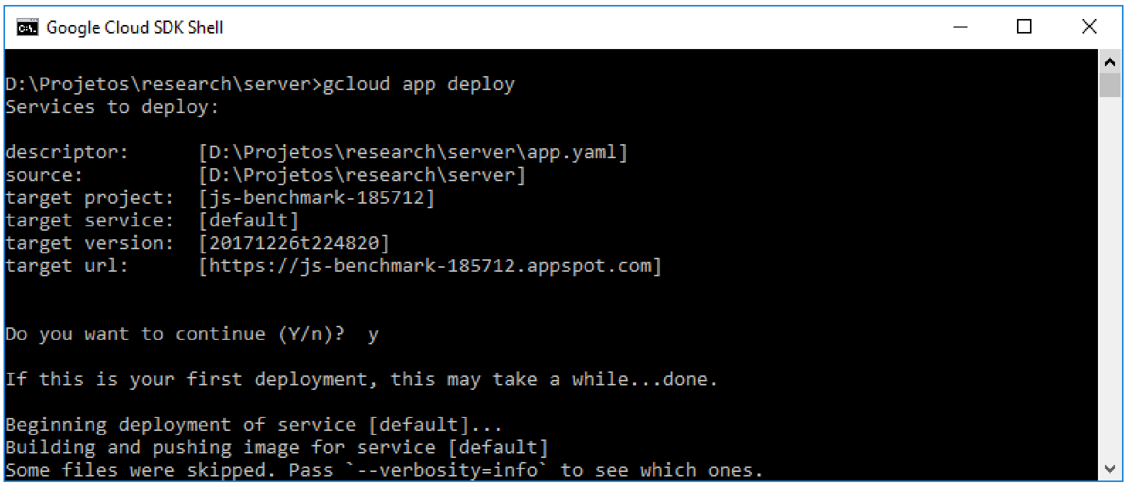


Google Cloud SDK Shell

```
D:\Projetos\research\server>gcloud config set project js-benchmark-185712
Updated property [core/project].

D:\Projetos\research\server>
```

Figure 5.6 Selecting project in Google Cloud Platform

The deploy command needs to be executed in the application folder. Figure 5.7 shows the deploy command execution. After finishing the deployment process, the application will be available in the target URL showed in Figure 5.7.

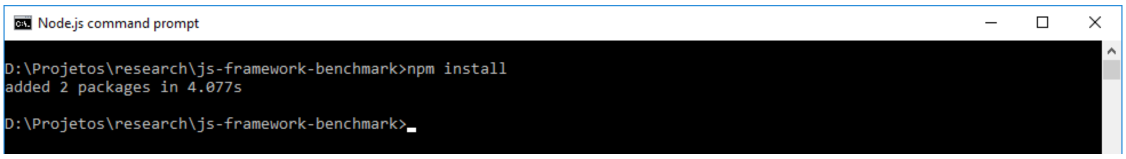


Google Cloud SDK Shell

```
D:\Projetos\research\server>gcloud app deploy
Services to deploy:

descriptor:      [D:\Projetos\research\server\app.yaml]
source:          [D:\Projetos\research\server]
target project:  [js-benchmark-185712]
target service:  [default]
target version:  [20171226t224820]
target url:      [https://js-benchmark-185712.appspot.com]


Do you want to continue (Y/n)?  y

If this is your first deployment, this may take a while...done.

Beginning deployment of service [default]...
Building and pushing image for service [default]
Some files were skipped. Pass `--verbosity=info` to see which ones.
```

Figure 5.7 Deploying an application in the Google Cloud Platform

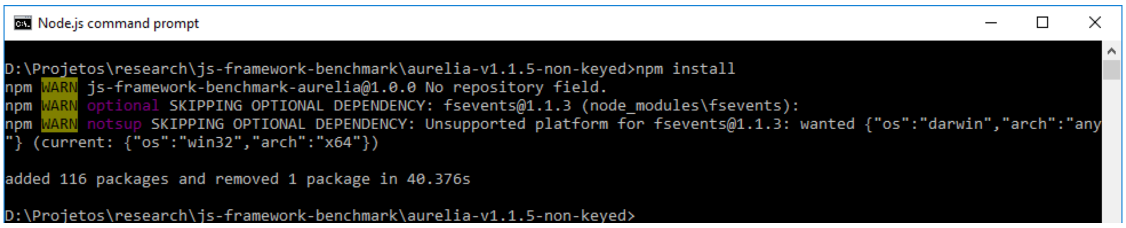
Figure 5.8 and Figure 5.9 show the dependencies installation of the benchmark application and the frameworks. It is necessary to run the command ‘npm install’ for each framework’s folder before testing it. The command ‘npm build-prod’ compiles the JSF into one or more JavaScript files.



Node.js command prompt

```
D:\Projetos\research\js-framework-benchmark>npm install
added 2 packages in 4.077s

D:\Projetos\research\js-framework-benchmark>_
```

Figure 5.8 Installing benchmark application dependencies



Node.js command prompt

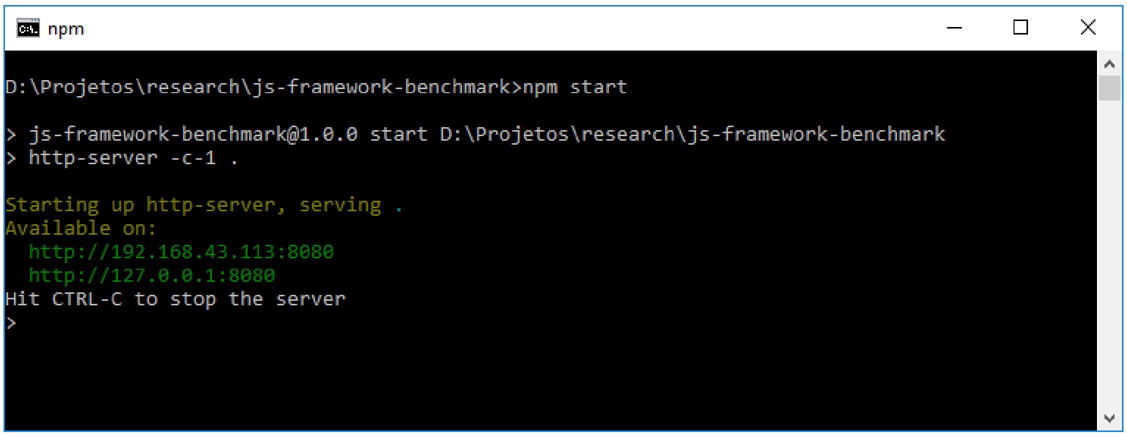
```
D:\Projetos\research\js-framework-benchmark\aurelia-v1.1.5-non-keyed>npm install
npm WARN js-framework-benchmark-aurelia@1.0.0 No repository field.
npm WARN optional SKIPPING OPTIONAL DEPENDENCY: fsevents@1.1.3 (node_modules\fsevents):
npm WARN notsup SKIPPING OPTIONAL DEPENDENCY: Unsupported platform for fsevents@1.1.3: wanted {"os":"darwin","arch":"any
"} (current: {"os":"win32","arch":"x64"})

added 116 packages and removed 1 package in 40.376s

D:\Projetos\research\js-framework-benchmark\aurelia-v1.1.5-non-keyed>
```

Figure 5.9 Installing framework dependencies

5.2.2 Running the experiment

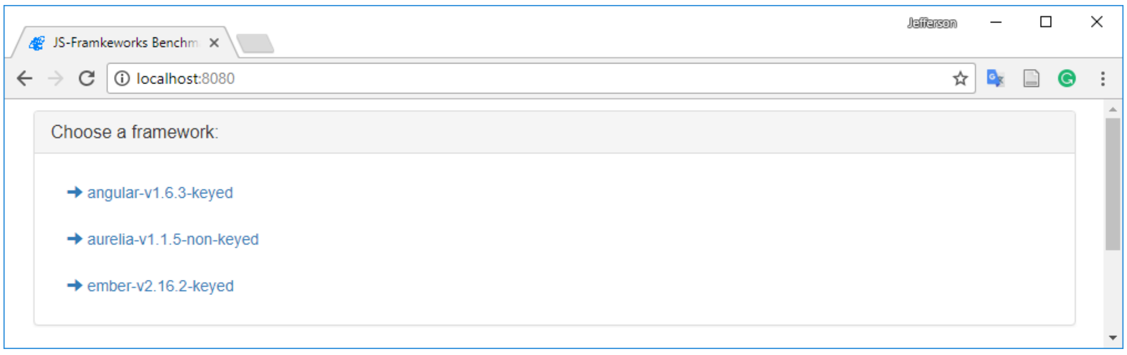
Figure 5.10 shows the console after starting the application server. The benchmark application can be accessed at the following URL: http://localhost:8080/

Figure 5.11 shows the application interface after accessing the URL above.



```
D:\Projetos\research\js-framework-benchmark>npm start

> js-framework-benchmark@1.0.0 start D:\Projetos\research\js-framework-benchmark
> http-server -c-1 .

Starting up http-server, serving .
Available on:
  http://192.168.43.113:8080
  http://127.0.0.1:8080
Hit CTRL-C to stop the server
>
```

Figure 5.10 Running the application

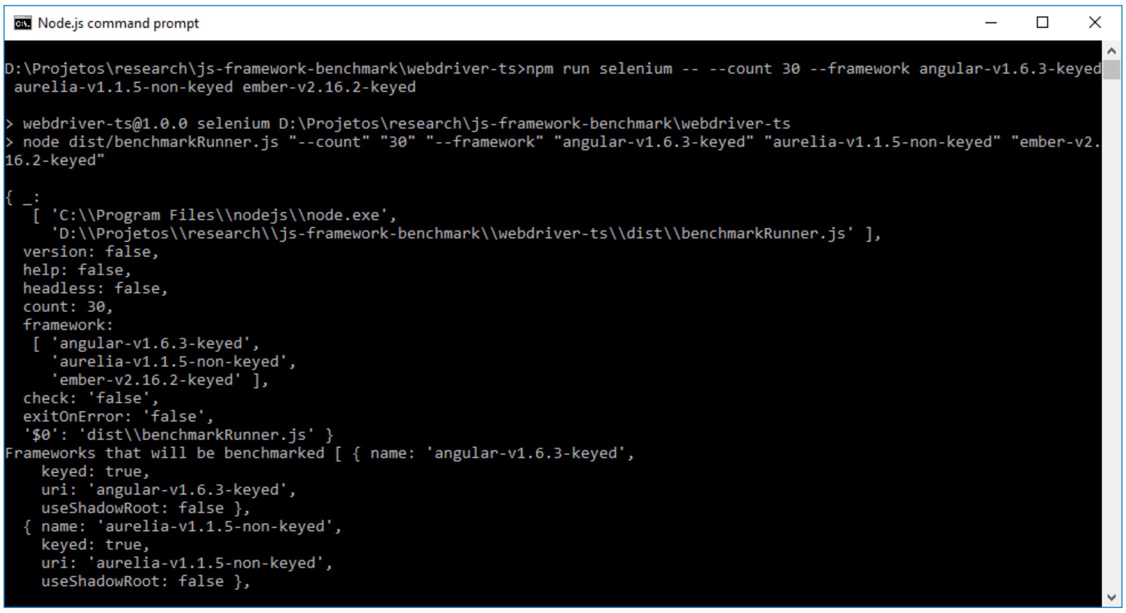


Choose a framework:

- angular-v1.6.3-keyed
- aurelia-v1.1.5-non-keyed
- ember-v2.16.2-keyed

Figure 5.11 Application Home Page

Figure 5.12 shows the console after starting to run the benchmarks. The command specifies the number of runs that each benchmark will iterate (e.g. -- count 30) and the frameworks that will be executed (e.g. --framework angular-v1.6.3-keyed). This command needs to be run from the folder \webdriver-ts.

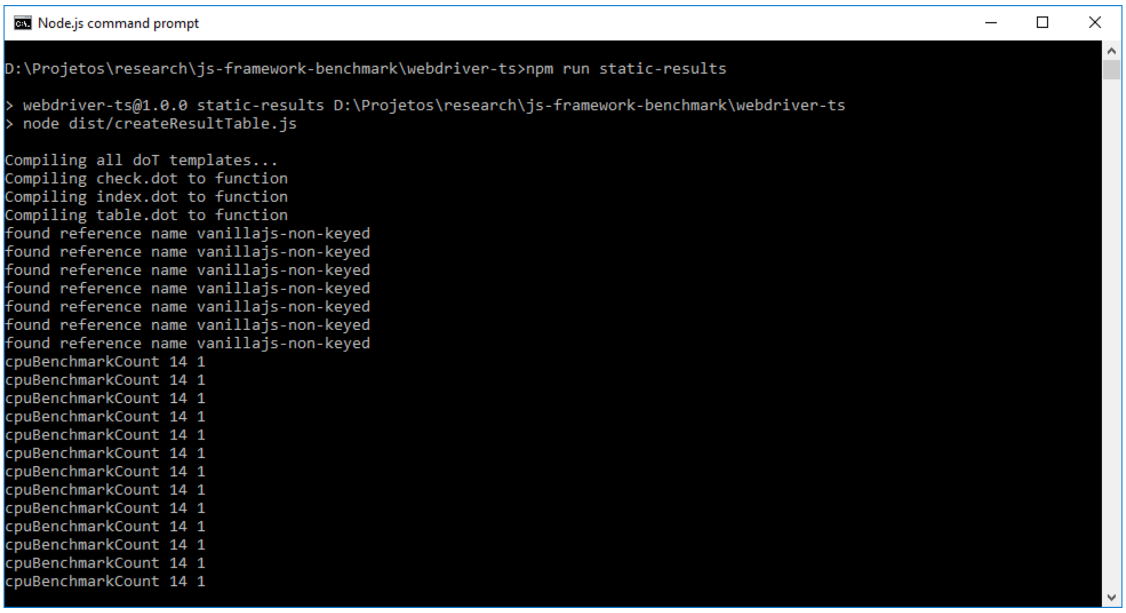


Node.js command prompt

```
D:\Projetos\research\js-framework-benchmark\webdriver-ts>npm run selenium -- --count 30 --framework angular-v1.6.3-keyed
 aurelia-v1.1.5-non-keyed ember-v2.16.2-keyed

> webdriver-ts@1.0.0 selenium D:\Projetos\research\js-framework-benchmark\webdriver-ts
> node dist/benchmarkRunner.js "--count" "30" "--framework" "angular-v1.6.3-keyed" "aurelia-v1.1.5-non-keyed" "ember-v2.
16.2-keyed"

{ _:
   [ 'C:\\Program Files\\nodejs\\node.exe',
     'D:\\Projetos\\research\\js-framework-benchmark\\webdriver-ts\\dist\\benchmarkRunner.js' ],
  version: false,
  help: false,
  headless: false,
  count: 30,
  framework:
   [ 'angular-v1.6.3-keyed',
     'aurelia-v1.1.5-non-keyed',
     'ember-v2.16.2-keyed' ],
  check: 'false',
  exitOnError: 'false',
  '$0': 'dist\\benchmarkRunner.js' }
Frameworks that will be benchmarked [ { name: 'angular-v1.6.3-keyed',
    keyed: true,
    uri: 'angular-v1.6.3-keyed',
    useShadowRoot: false },
  { name: 'aurelia-v1.1.5-non-keyed',
    keyed: true,
    uri: 'aurelia-v1.1.5-non-keyed',
    useShadowRoot: false },
```

Figure 5.12 Running benchmarks

Figure 5.13 shows the command to generate the results table. This command is run from the folder \webdriver-ts, and it can be accessed from the browser at the URL http://localhost:8080/webdriver-ts/table.html



Node.js command prompt

```
D:\Projetos\research\js-framework-benchmark\webdriver-ts>npm run static-results

> webdriver-ts@1.0.0 static-results D:\Projetos\research\js-framework-benchmark\webdriver-ts
> node dist/createResultTable.js

Compiling all doT templates...
Compiling check.dot to function
Compiling index.dot to function
Compiling table.dot to function
found reference name vanillajs-non-keyed
found reference name vanillajs-non-keyed
found reference name vanillajs-non-keyed
found reference name vanillajs-non-keyed
found reference name vanillajs-non-keyed
found reference name vanillajs-non-keyed
found reference name vanillajs-non-keyed
cpuBenchmarkCount 14 1
cpuBenchmarkCount 14 1
cpuBenchmarkCount 14 1
cpuBenchmarkCount 14 1
cpuBenchmarkCount 14 1
cpuBenchmarkCount 14 1
cpuBenchmarkCount 14 1
cpuBenchmarkCount 14 1
cpuBenchmarkCount 14 1
cpuBenchmarkCount 14 1
cpuBenchmarkCount 14 1
cpuBenchmarkCount 14 1
cpuBenchmarkCount 14 1
```

Figure 5.13 Running results table

Figure 5.14 shows a snapshot of the results table.

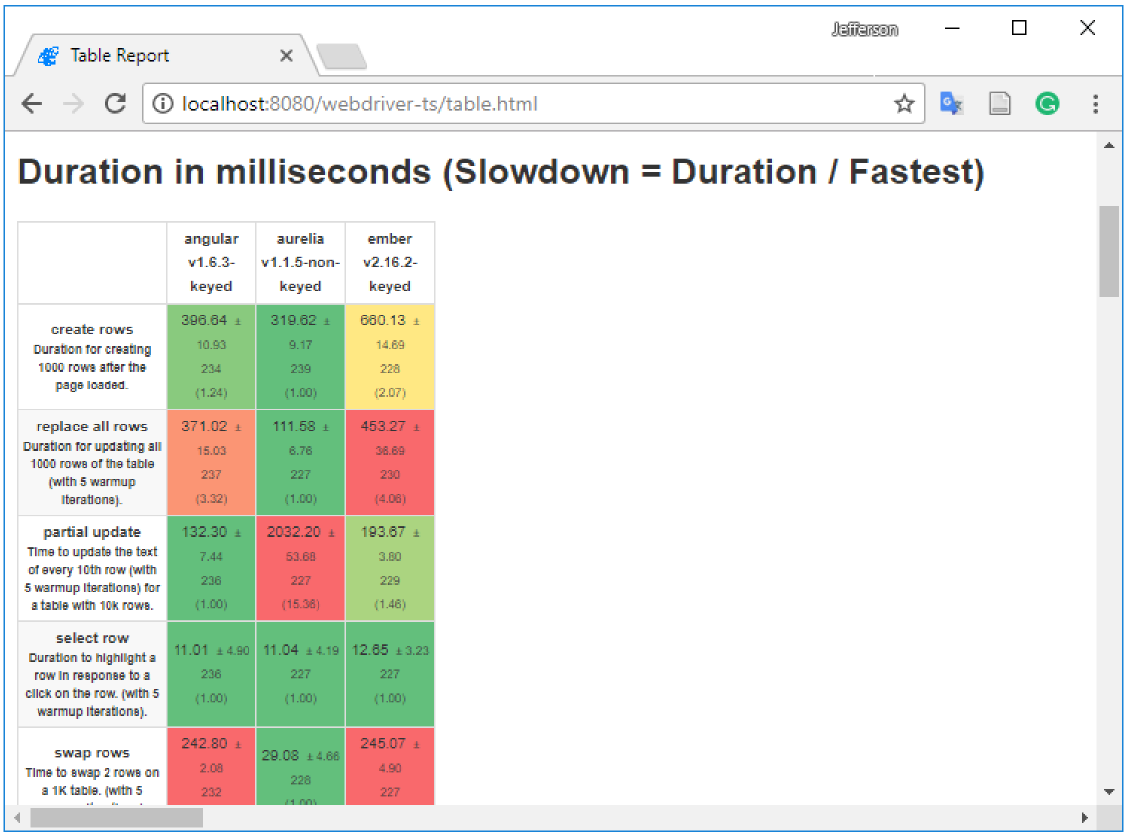


Table Report

localhost:8080/webdriver-ts/table.html

Duration in milliseconds (Slowdown = Duration / Fastest)

	angular v1.6.3-keyed	aurelia v1.1.5-non-keyed	ember v2.16.2-keyed
create rows Duration for creating 1000 rows after the page loaded.	396.64 ± 10.93 234 (1.24)	319.62 ± 9.17 239 (1.00)	660.13 ± 14.69 228 (2.07)
replace all rows Duration for updating all 1000 rows of the table (with 5 warmup iterations).	371.02 ± 15.03 237 (3.32)	111.58 ± 6.76 227 (1.00)	453.27 ± 36.69 230 (4.06)
partial update Time to update the text of every 10th row (with 5 warmup iterations) for a table with 10k rows.	132.30 ± 7.44 236 (1.00)	2032.20 ± 53.68 227 (15.36)	193.67 ± 3.80 229 (1.46)
select row Duration to highlight a row in response to a click on the row. (with 5 warmup iterations).	11.01 ± 4.90 236 (1.00)	11.04 ± 4.19 227 (1.00)	12.65 ± 3.23 227 (1.00)
swap rows Time to swap 2 rows on a 1K table. (with 5	242.80 ± 2.08 232	29.08 ± 4.66 228	245.07 ± 4.90 227

Figure 5.14 Results table

5.3 *Evaluation*

5.3.1 JavaScript framework benchmark results

A wired connection (Specification in Table 3.3) to the server, where the database is stored, was used in the first round of tests. In these tests, the benchmark application was executed in two different operating systems using the same computer. Section 2.6 explains the difference between dual-boot and virtual machine. The connection speed was 13Mbps on average. The benchmarks that use the network to execute DBMS transactions (CRUD operations) ran in the browser Google Chrome in both systems. Each benchmark ran ten times in each operating system. The average of concurrent processes in Linux Ubuntu 16 (Specification in Table 4.1) was two hundred and thirteen (213), and in Windows 10 (Specification in Table 4.1) the average of sixty-seven (67) concurrent processes were being executed during the benchmarks` run.

The concurrent processes in both systems include network processes (e.g. firewall, wireless and wired configuration), graphic user interface and so on. Table 2.3 and Table 2.4 contains a detailed description of necessary processes to run the Windows operating system.

In the Figure 5.15 to Figure 5.26, the numbers on the Y-axis, were normalised to enable data visualisation and do not depict the numbers in milliseconds of each run but an adjusted value to represent time (e.g. 0.16 = 810.70ms). Table 5.1 to Table 5.6 present a conversion of these numbers.

Value	0.16	0.14	0.12	0.10	0.08	0.06	0.04	0.02
Milliseconds	810.70	709.36	608.02	506.69	405.35	304.01	202.67	101.33

Table 5.1 Description of Y-axis values in milliseconds in Figure 5.15 and Figure 5.16

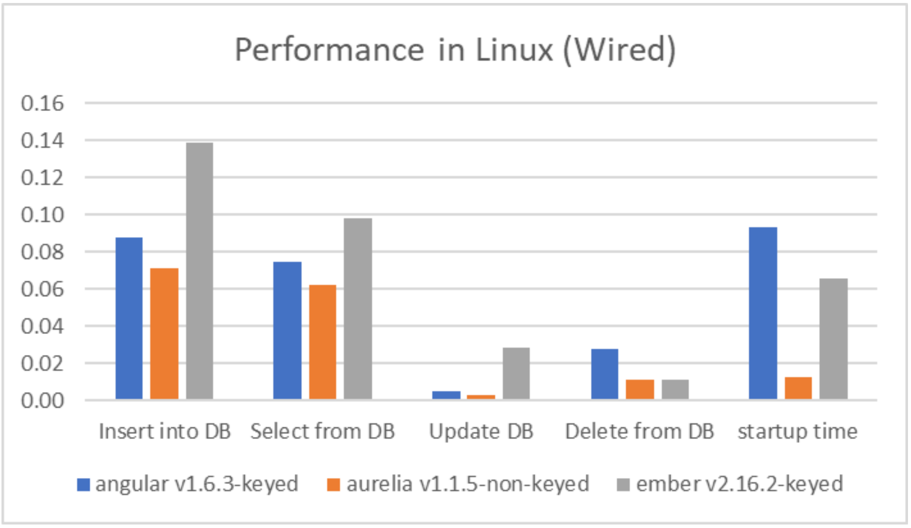


Figure 5.15 Linux performance in a wired connection (Part I)

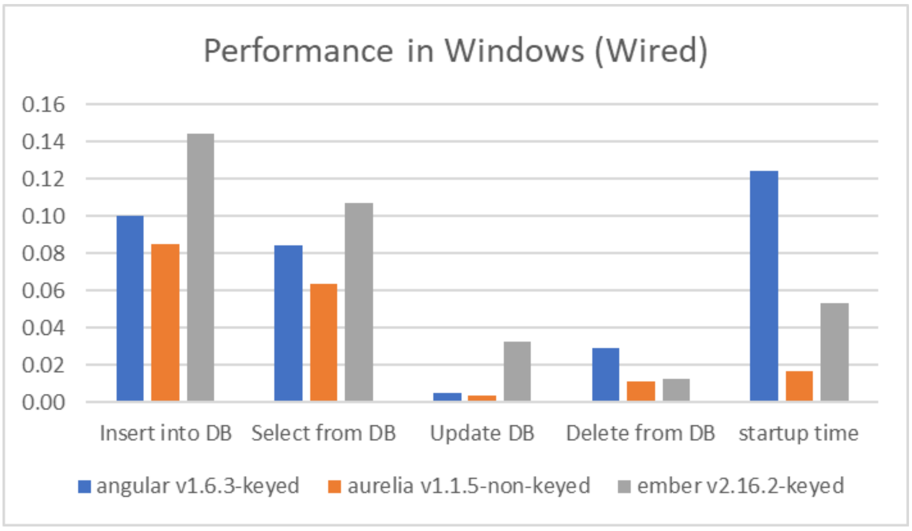


Figure 5.16 Windows performance in a wired connection (Part I)

Linux had a slightly superior performance results compared to Windows in the benchmarks above. Linux executed the database tasks in 269.78ms on average, and Windows took 294.94ms on average to execute these tasks. Database benchmarks depend on the network speed and its configurations.

Aurelia achieved better performance results compared to other frameworks. Aurelia took 368.37ms in a Linux environment to insert a thousand rows in the database while Ember took 710.8ms to insert the same amount of data. Angular took 454.4ms to execute this same task. The performance to execute this benchmark in the Windows environment slightly increased compared to the Linux environment. Aurelia took 439.12ms to execute the ‘insertDB’ task in the Windows environment; Ember took 740.53ms and Angular took 515.99ms in the Windows environment.

Ember improves its performance compared to other frameworks when executing the ‘DeleteDB’ task. Ember took 73.5ms; Angular took 154.43ms and Aurelia took 64.49ms to execute this task. Even though Angular gets a quick response from the server, it takes more time to update the DOM than the other frameworks. Angular also has the worst score in the startup time benchmark. Angular loads an entire browser environment called PhantomJS during the startup. The other frameworks only load the modules needed to run the application.

Value	1	0.80	0.60	0.40	0.20
Milliseconds	5066.91	4053.528	3040.146	2026.764	1013.382

Table 5.2 Description of Y-axis values in milliseconds in Figure 5.17 and Figure 5.18

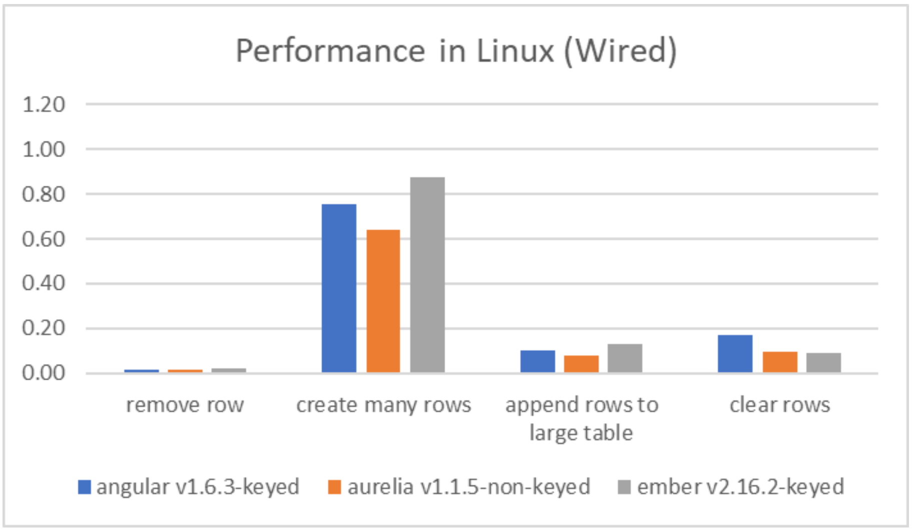


Figure 5.17 Linux performance in a wired connection (Part II)

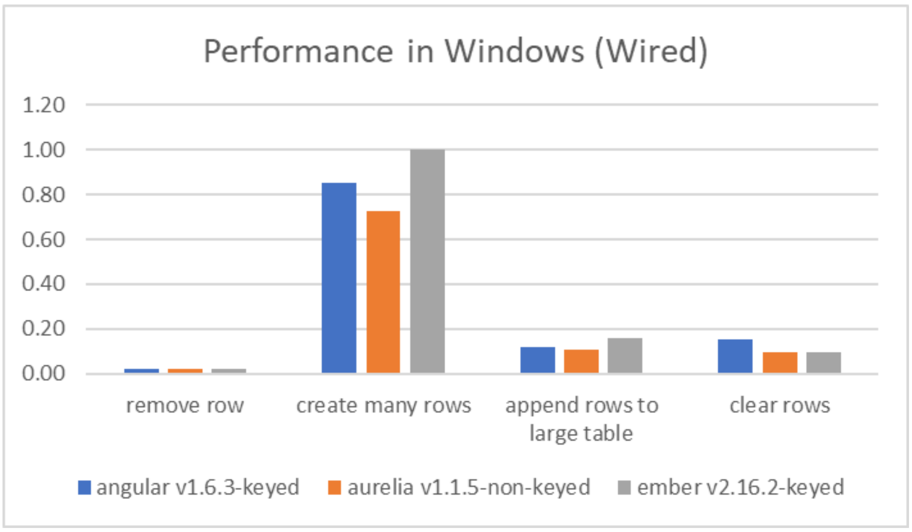


Figure 5.18 Windows performance in a wired connection (Part II)

Linux still has better performance results than Windows in these benchmarks. Both systems had similar results, and Linux ran the JavaScript frameworks slightly faster than Windows. Linux had a 1270.44ms average speed, and Windows had a 1430.41 average speed. The benchmarks ran ten times, and Aurelia performed better in every benchmark run. These benchmarks do not depend on the network speed, and they are solely based on the CPU speed. The number of concurrent processes running at the same time as the benchmark execution directly affect the results of each run. Ember has the worst results in almost every benchmark except for the 'clear rows' task. Angular achieved a slower execution time in this task with 876.ms in the Linux environment and 793.41 in the Windows environment. Aurelia achieved 492.98ms in the Linux environment and 505.99ms in the Windows environment. Ember completed the same task in 476.88ms in the Linux environment and 504.14ms in the Windows environment.

Value	0.60	0.50	0.40	0.30	0.20	0.10
Milliseconds	3040.15	2533.45	2026.76	1520.07	1013.38	506.69

Table 5.3 Description of Y-axis values in milliseconds in Figure 5.19 and Figure 5.20

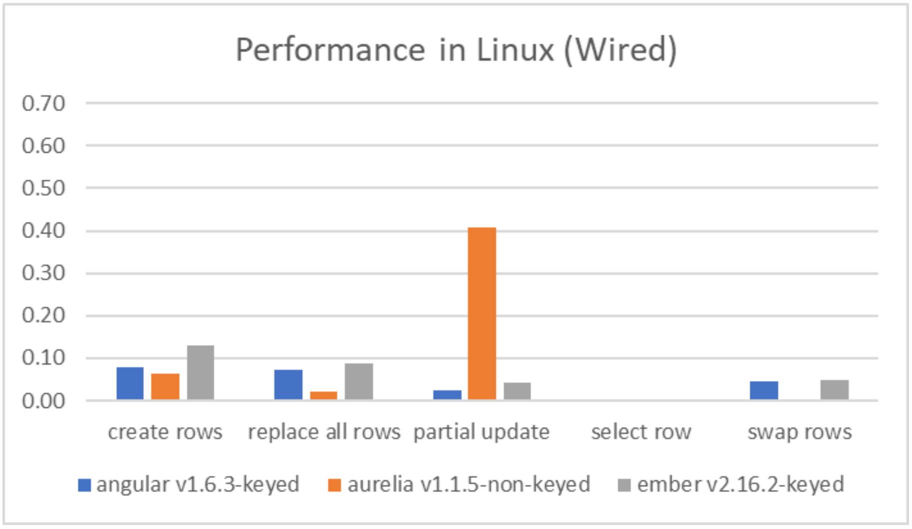


Figure 5.19 Linux performance in a wired connection (Part III)

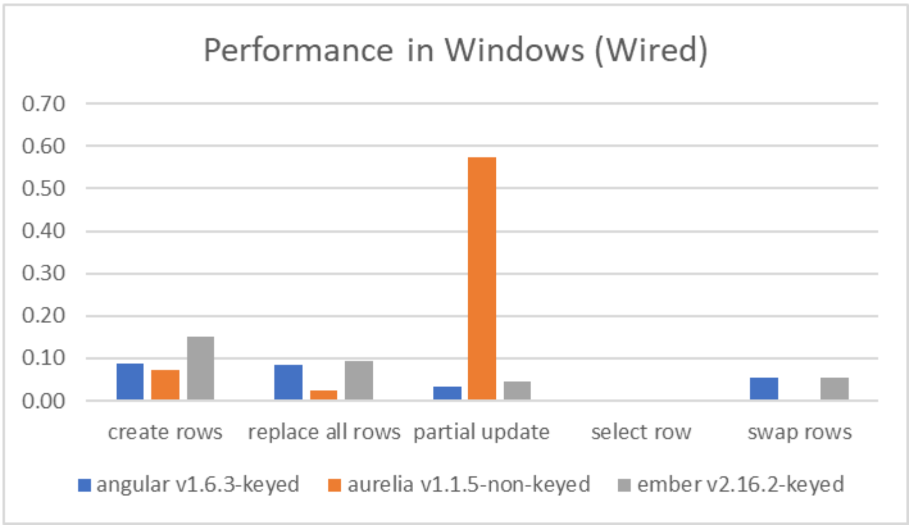


Figure 5.20 Windows performance in a wired connection (Part III)

Linux took 356.07ms on average to execute all the above operations, and Windows took 441ms to execute the same tasks. Aurelia performed better in most results except for the 'partial update' task. In this task, the framework took a significant amount of time to update the DOM nodes. Aurelia does not perform a dirt-checking like AngularJS (Section 2.2.1), but instead, it uses an observer-based mechanism that does not check anything that has not changed in the DOM[15]. Furthermore, Aurelia also does not update the DOM instantly, but it batches the changes in an aggregator, so it can happen at once. The Partial Update task demands a lot of changes in the DOM object (create ten thousand rows and update them). All these changes are batched and added up at the end of the operation, increasing the execution time of this operation. Overall, Ember still scored worst results in both operating systems except for the partial updates.

The second round of tests used a wireless connection with the same environment configuration and browser. The connection speed was on 10Mbps on average. The average number of concurrent processes in Linux was two hundred and twenty-two (222) and sixty-nine (69) in Windows.

Value	0.2	0.18	0.16	0.14	0.12	0.10	0.08	0.06	0.04	0.02
Milliseconds	1013.38	912.04	810.70	709.36	608.02	506.69	405.35	304.01	202.67	101.33

Table 5.4 Description of Y-axis values in milliseconds in Figure 5.21 and Figure 5.22

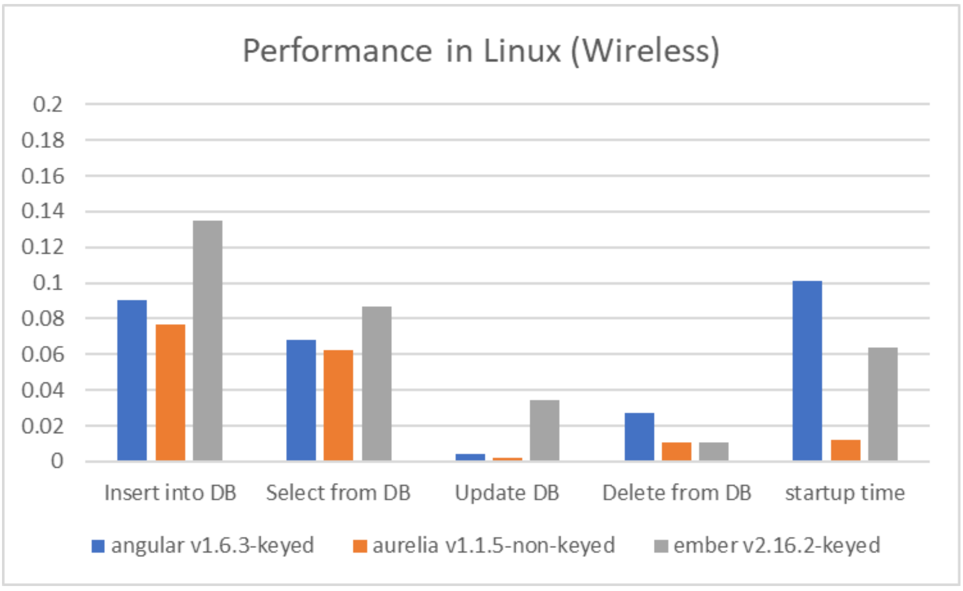


Figure 5.21 Linux performance in a wireless connection (Part I)

[15] https://github.com/aurelia/binding/issues/401

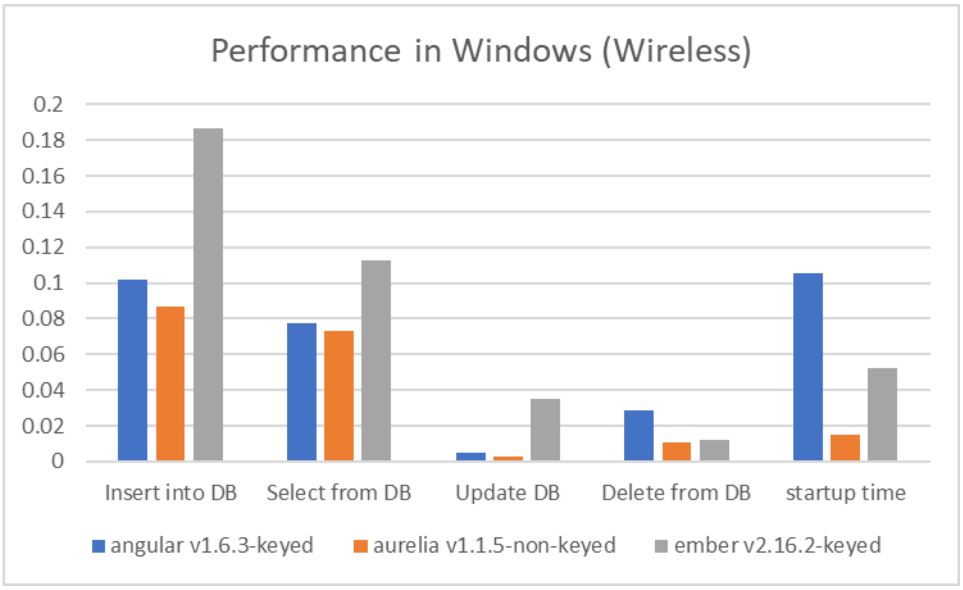


Figure 5.22 Windows performance in a wireless connection (Part I)

In the benchmarks above, Linux executed the tasks in 275.89ms on average, and Windows' average was 316.03ms. The results were similar to the execution in the wired network regarding the network speed, but, on average, they were slightly slower. Ember still has the worst performance rank overall, especially with the InsertDB operation which took 955.13ms. In comparison with the wired connection, the difference to execute the same task was 214.6ms. The disparity of the numbers raised suspicions about network instability and in the second attempt to run the same task, the operation took 713.22ms, which is 241.91ms less. The network's instability at the moment of the task's execution was the primary cause of this disparity.

Value	1	0.80	0.60	0.40	0.20
Milliseconds	5066.91	4053.528	3040.146	2026.764	1013.382

Table 5.5 Description of Y-axis values in milliseconds in Figure 5.23 and Figure 5.24

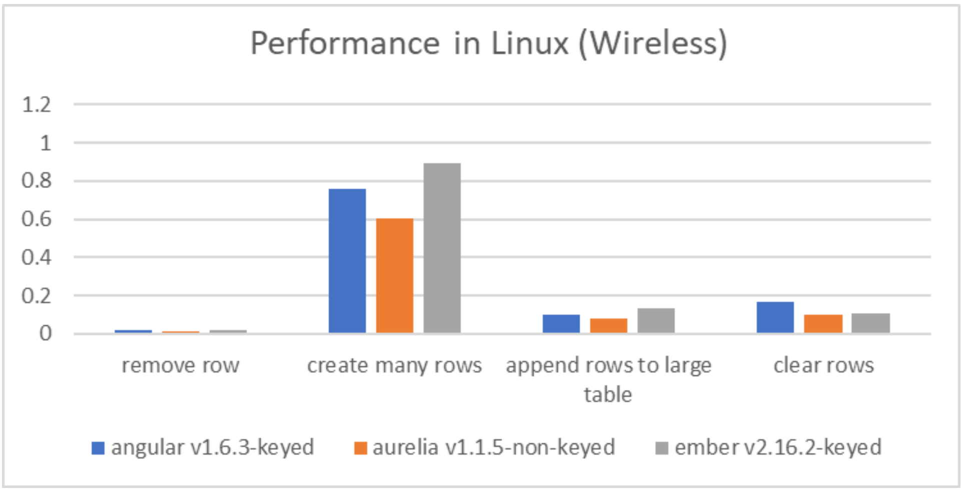


Figure 5.23 Linux performance in a wireless connection (Part II)

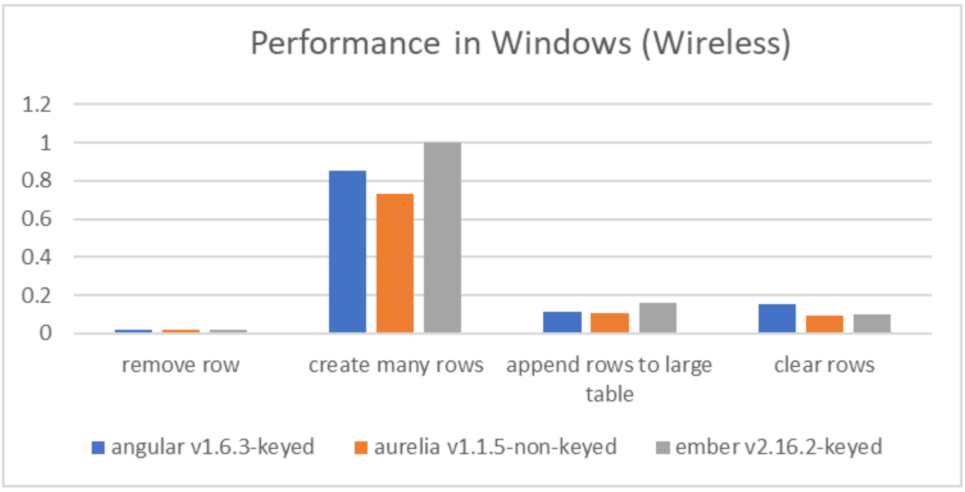


Figure 5.24 Windows performance in a wireless connection (Part II)

In the benchmarks above, Linux executed the tasks in 1272.43ms on average, and Windows's average was 1451.31ms. Although these tasks do not use a network connection to be executed, the number of concurrent processes in both operating systems was higher in the wireless connection compared to the wired connection. This factor might have influenced the results of these tests.

Value	0.60	0.50	0.40	0.30	0.20	0.10
Milliseconds	3040.15	2533.45	2026.76	1520.07	1013.38	506.69

Table 5.6 Description of Y-axis values in milliseconds in Figure 5.25 and Figure 5.26

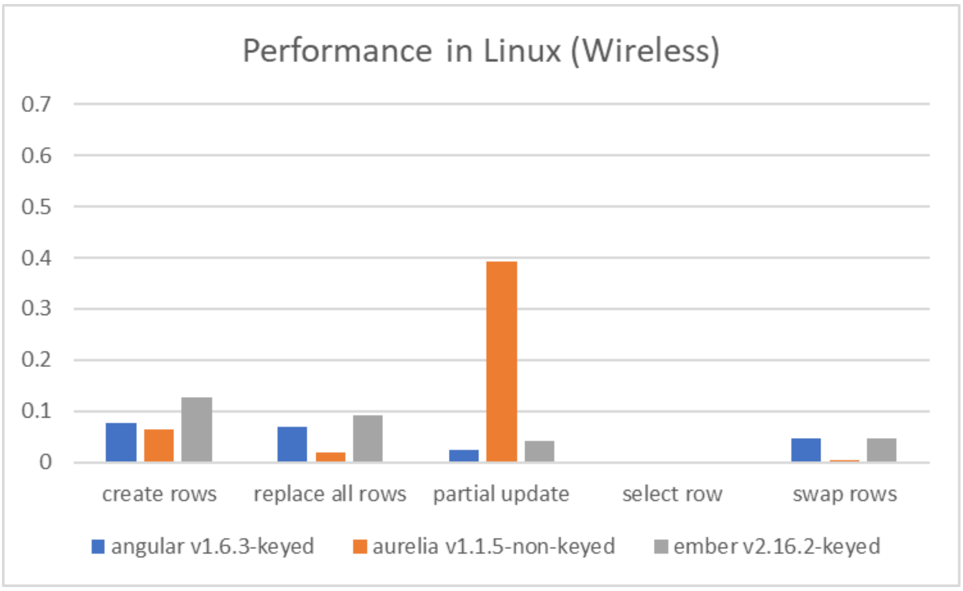


Figure 5.25 Linux performance in a wireless connection (Part III)

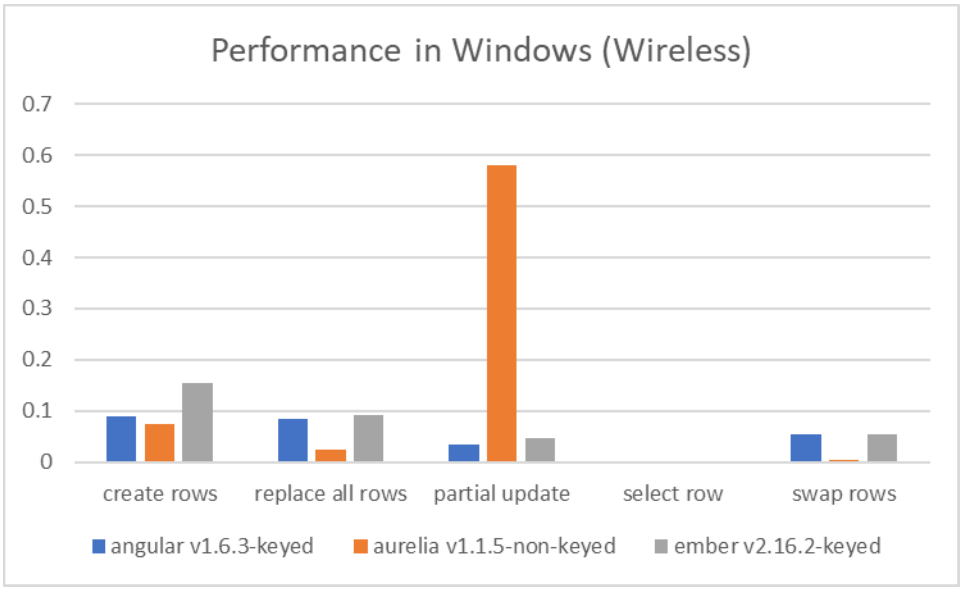


Figure 5.26 Windows performance in a wireless connection (Part III)

In the benchmarks above, Linux executed the tasks in 351.14ms on average, and Windows's average was 445.57ms. In comparison with the wired connection, a slight difference occurred with the wireless connection due to the number of processes running concurrently. When the benchmarks were running in the wired connection, the wireless network card was disabled, and consequently, the processes attached to this hardware were also inactivated. The number of processes influenced the results of these benchmarks.

5.3.2 Complexity measurement results

This section will present the results of the complexity measurement tool (complexity-report) over the js-framework-benchmark application. The complexity report was run on the compiled applications. The compilation process is showed in section 5.2.1.

AngularJS compiles the application in one file (dist\main.js), Aurelia compiles its version of the application in two files (scripts\app-bundle.js and scripts\vendor-bundle.js), and Ember also compiles its version of the application in two files (dist\assets\ember-temp…js and dist\assets\vendor-…js).

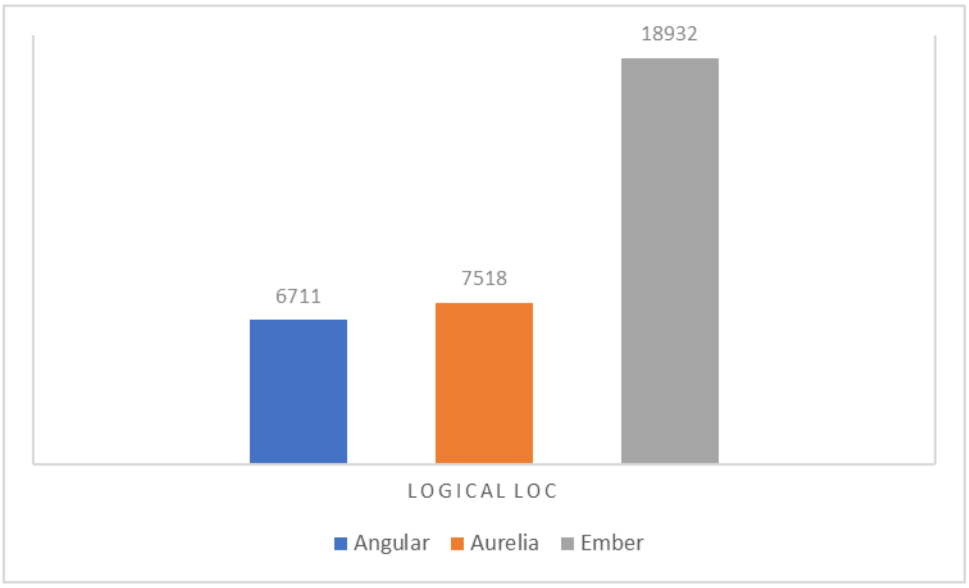


Figure 5.27 Logical Lines of Code comparison

Figure 5.27 shows the total logical lines of code in each framework. Physical Lines of Code (SLOC) and Logica Lines of Code (LLOC) are discussed in details in section 2.4. Angular has the lowest number of logical lines of codes because it loads the browser's libraries instead of loading its modules. Ember has the most significant number of LLOC because it needs to specify its library in the compiled files. The higher number of LLOC in Ember justifies its lowest performance in the overall benchmark execution because it takes more time to access and execute the functions in comparison to its competitors AngularJS and Aurelia. The compiled file shows only six SLOC for AngularJS and two SLOC for Aurelia. Ember has four thousand, one hundred and thirty-one physical lines of code in its compiled file. However, SLOC, regarding JavaScript execution time, does not impact on the framework's performance.

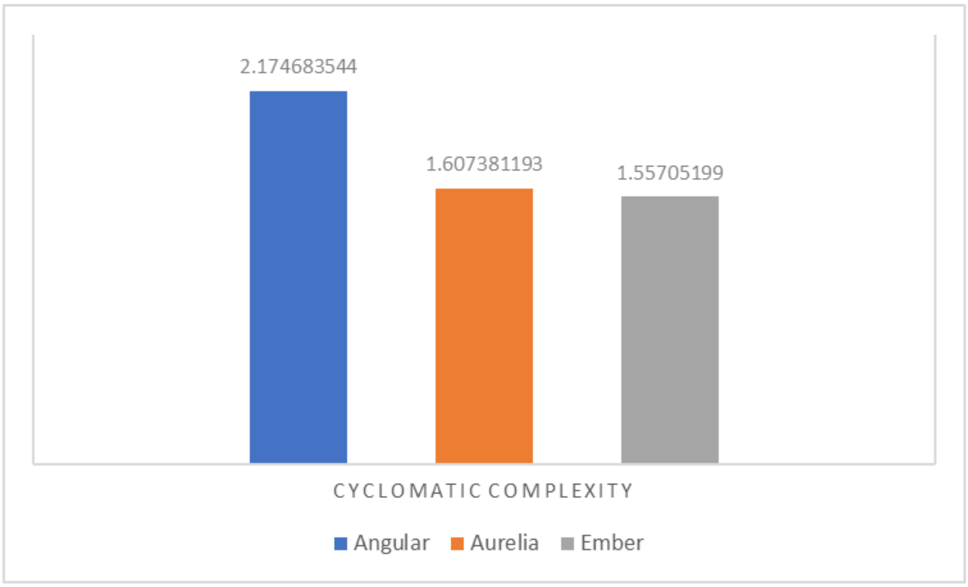


Figure 5.28 Cyclomatic complexity comparison

Figure 5.28 shows the cyclomatic complexity of each code. In this comparison, the lower the result, the less code complexity. Cyclomatic complexity counts the number of different paths a method can take. The complexity tool uses McCabe's definitions as mentioned in section 2.4. Although AngularJS has a small number of LOC, it is the most complicated code to understand due to the number of paths in the algorithm.

There is no ideal number for the complexity of a function, but a cyclomatic complexity above twenty indicates that the code should be rewritten.

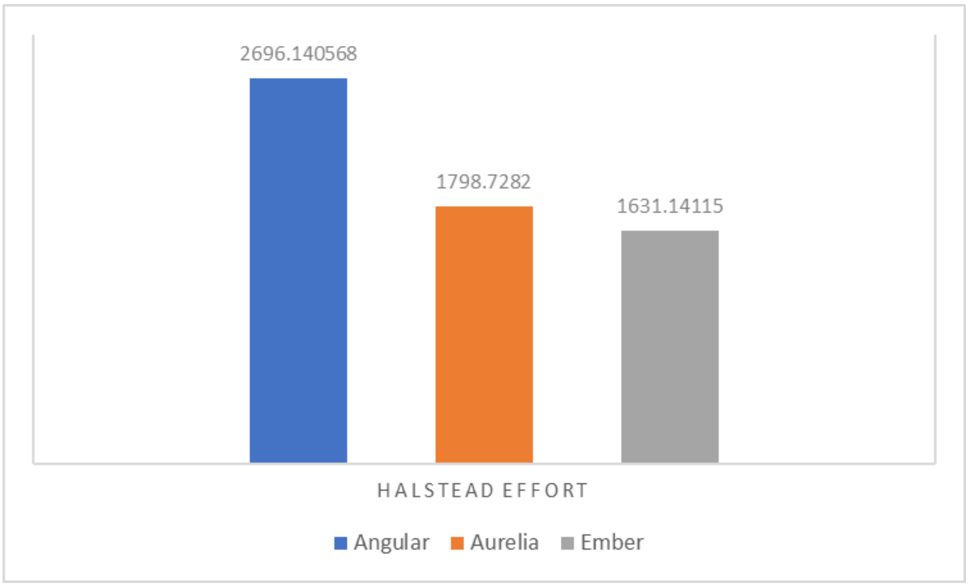


Figure 5.29 Halstead effort comparison

Halstead effort calculates the number of distinct operators, the number of distinct operands, the total number of operators and the total number of operands in each function. These metrics are used to assess the complexity of the system, and they are discussed in section 2.4. This metric is the base for the maintainability index, and Figure 5.29 shows a mean per-function of this metric.

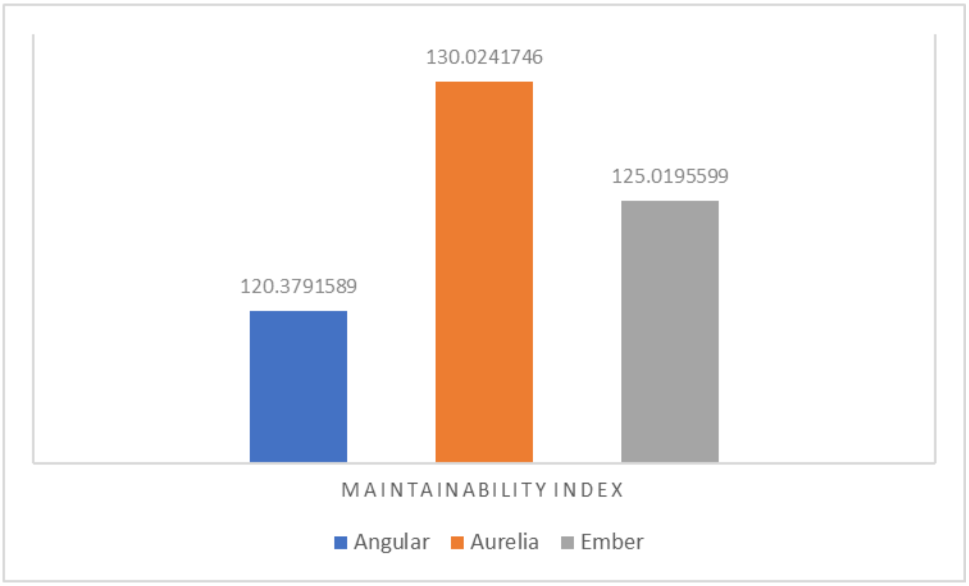


Figure 5.30 Maintainability index comparison

The maintainability index is measured on a logarithmic scale, calculated from the logical lines of code, cyclomatic complexity and Halstead effort. Figure 5.30 shows the maintainability index of each framework. As discussed in section 2.4, the higher the maintainability index, the easier to maintain the code. Therefore, Aurelia is the most straightforward framework to maintain due to the results of this comparison.

AngularJS has fewer LOC in comparison to the other frameworks, but it is more complicated, and it might be hard for a developer to understand its code.

Figure 5.31 shows a mean per-function parameter counter comparison. A parameter counter is the number of parameters obtained from the signature of each function. A higher number of parameters indicates excessive work in a function and a complicated interface. This situation must be avoided.

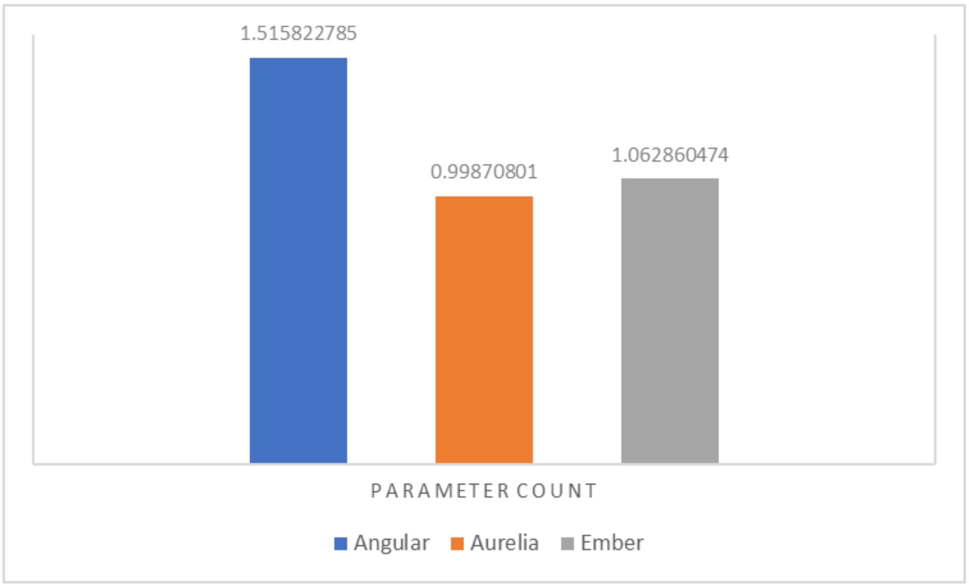


Figure 5.31 Parameter counter comparison

5.3.3 Comparison conclusion and suggestions

Aurelia had the best performance in almost every benchmark. Aurelia is the newest framework, released in 2016, and it is built with the most recent and sophisticated modules for JavaScript applications. Aurelia presented the lowest code complexity metric, and the implementation of the database operations in this framework was quite straightforward. Ember has an elaborate design structure but once understood; this framework is a powerful tool able to make the code well organized. However, this whole structure makes it the worst framework regarding performance, but the second one in complexity and maintainability. The new version of AngularJS is called Angular 2 (or just Angular), and it promises to bring improvements related to the data binding and startup time. AngularJS was used in this research as a reference to how the frameworks have been evolving throughout time, and the disparity between Aurelia and AngularJS has shown those improvements.

AngularJS introduced new concepts to the field of JavaScript frameworks, and they are still relevant to this day. Ember is a robust framework created to develop ambitious applications, and it organises the code better compared to AngularJS and Aurelia. Aurelia is the fastest framework in this comparison. However, the framework has just been released, and it still lacks a big community and support in comparison with its competitors.

5.3.4 Strengths and limitations

The JavaScript framework benchmark is an open source on-going project that has been growing since its first release. The constant updates and the number of contributors to this project make it extremely relevant and reliable as the community of developers are enthusiastic and dedicated to this project and always bringing improvements and bug fixes to this application. The benchmark application also contains a good number of frameworks implemented which can aid developers in choosing between the great variety of open source frameworks available.

Although the benchmark application has a complete set of metrics and JavaScript frameworks, it only performs the tests in one browser, Google Chrome. Future implementations of these applications intend to include more browsers such as Mozilla Firefox and Microsoft Edge. The application is also expanding the number of metrics, and after the running of this experiment a new CPU and memory metrics were implemented, but they are not included in this project.

The research also focused on the environment where the experiment should be conducted to find variables that could influence the performance of JavaScript frameworks. The comparison was conducted on two different operating system on the same machine, and it was verified that concurrent processes have a minimal impact on the performance of JSFs. The results showed that concurrent processes do not have a significant impact on the framework's performance.

However, network instability could cause some delays in the execution time of JSFs with web servers. As it was shown in one of the benchmarks (Figure 5.22), network instability caused a delay in the database operation 'InsertDB' where Ember took more time compared to the second round of tests.

The metrics related to database operations contributed to the simulation of a real web application where CRUD operations are executed using JavaScript through JSFs. The previous work did not consider using the database in a JavaScript framework comparison.

The research used three JSFs due to time constraints. The results of this comparison would benefit from the use of a greater variety of JSFs

5.4 Chapter summary

This chapter describes the implementation and execution of the JavaScript framework comparison experiment. The environment preparation and the necessary steps to run the experiment were covered at the beginning of this chapter. The benchmark framework needs to be installed in the machine while the RESTful API needs to be deployed in a cloud web server, in this case, Google Cloud Platform was the platform chosen to run this experiment. This chapter also covered the complexity measurement tool to evaluate the complexity metrics of each framework. These metrics were discussed in section 2.4 of this research.

The results provided by each tool (JavaScript framework benchmark and complexity tool) were evaluated and discussed, highlighting the main differences between each JSF and their strengths and weaknesses.

6 CONCLUSION

6.1 Introduction

JavaScript is a front-end programming language, and with the aid of HTML5 and CSS, it has been experiencing an increasing number of developers dedicated to this language. JavaScript frameworks help programmers to develop SPAs much faster with a set of functionalities that save the developer`s time when building those applications. The popularity of JavaScript created a vast number of JSFs available for developers today, and Section 1.2 described the problems that these professionals are facing when choosing the framework that best suits their projects. The aim of this research was to determine the factors that could influence the adoption of JavaScript frameworks based on software metrics and environment configuration. Three JSFs were selected for this experiment, and the results were discussed in Chapter 5.

6.2 Research overview

This research investigated the effects of the environment on JavaScript applications using JavaScript frameworks. The main goal of this research was to use benchmark metrics, presented in Section 2.4, to evaluate JSFs in different environments to measure the effects of the environment on those frameworks. Three JSFs were selected to run the experiment in a benchmark framework using two different operating systems (Linux and Windows) on the same machine. A dual-boot technique was chosen over the virtual machine because the two systems can coexist on the same machine without interfering with each other’s performance.

Two different network connections were used to run the experiment. DIT (Kevin Street) offers fast Wired and Wireless network connections and maintains a stable connectivity using fiber backbone connections. The network specifications are described in Section 3.5.

Google Cloud Platform (GCP) was the cloud-based service chosen to host an implementation of a RESTful API to execute CRUD operations in a database. Both, the RESTful application and the database are hosted in the GCP. GCP has a massive infrastructure with data centres scattered around the globe and a good availability of its services. It can host Node.js applications and Relational Database System Management Systems such as MySQL.

6.3 Experimentation, Evaluation and Limitations

The benchmark application ran all the speed metrics successfully and showed clear differences between the JavaScript frameworks tested. The project only includes three JSFs due to its time constraints, and the benchmark application offers more JSFs to be implemented and tested.

AngularJS was one of the frameworks chosen because the researcher already had knowledge and experience with this tool. Even though there are similarities between AngularJS, Ember and Aurelia, the researcher was not familiar with these frameworks which complicated the implementation of security metrics in this project.

The experiment used a cloud-based platform to simulate transactions between a user computer and a web server. The results showed that environment configurations have little impact on the performance of JSFs as the results did not change from one environment to another. In other words, environment configurations do not affect the performance of JSFs in comparison with other JSFs.

The comparison of a JSF in the same environment but with a different configuration showed that the environment slightly decreases the performance of this JSF. For instance, AngularJS in Windows with an average of sixty-seven processes running concurrently performs better than AngularJS running in the same OS with an average of a hundred and twenty-two processes running concurrently.

Although the environment configurations of the selected OS's do not significantly impact the performance of JSFs, the research lacks more systems running JavaScript as the literature review showed that JavaScript is not only limited to execution in Desktop browsers but also on different platforms such as mobiles and video games.

The results also showed that stable internet connections do not affect the performance of JSFs. However, the connection speed and the network type (wired and wireless) slightly affect the execution time of JavaScript as the data needs to be transferred using these channels. In this experiment, the network has different configurations for each network as Section 3.5 described.

The experiment only included Google Chrome as the browser for running the benchmarks. The benchmark application has only implemented this browser, and future implementations will contain different browsers such as Microsoft Edge and Mozilla Firefox.

6.4 Results Summary

This section summarises the results and findings of this research

- Aurelia is the newest and fastest framework compared to Ember and AngularJS.
- Ember has the most organised structure and it can handle ambitious JavaScript applications.
- AngularJS has been restructured and updated to Angular2 (or just Angular) and it eliminated the dirt checking of the data binding processes. Evaluation on the new version should be conducted to asses the impact of those changes.
- Environmental configurations have little impact on the performance of JavaScript applications using JSFs. The main factor the affects execution time of JavaScript application is the JSF itself.

6.5 Contributions and Impact

The main contribution of this work is its emphasis on the impact of the environment where the JavaScript is being executed. Previous research in this field did not include some variables that could alter the performance results of the execution time of JavaScript frameworks. This project investigated concurrent processes running in the machine where the JavaScript is being executed to assess the effect of these processes on the JavaScript applications. The results showed that concurrent processes have little impact on the execution of JavaScript frameworks and the deterministic factors involving the performance of JSFs are within the implementation of these frameworks. In other words, the framework implementation is the cause of a lower execution time of the JavaScript application.

The research also addressed the issue of cloud-based services. The findings demonstrated that database operations do not alter the execution time of JavaScript applications as they use asynchronous methods to execute an HTTP request. However, the response time of these servers relies on the network’s connectivity and stability which could increase the execution time of an operation (e.g. ‘InsertDB’).

6.6 Future Work and Recommendations

There are a number of ways in which this work can be pushed forward to expand the knowledge of JavaScript framework comparison. Some of the recommendations were suggested by previous work and still have not been achieved.

1. Implementing the database operations in different frameworks and running the benchmarks in those JSFs would give developers a better understanding of the actual marketplace and help them to decide on the best choice for their project.
2. Performance evaluation has different approaches, and it would be interesting to apply the different methods discussed in Section 2.3.2
3. Including more browsers in this experiment such as Microsoft Edge and Mozilla Firefox
4. Running the experiment on different platforms such as mobiles to identify possible performance issues connected to an entirely different environment.

BIBLIOGRAPHY

A. Rajaram, Jiang Hu, & R. Mahapatra. (2006). Reducing clock skew variability via crosslinks. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, *25*(6), 1176–1182. https://doi.org/10.1109/TCAD.2005.855928

Alves, T. L., Ypma, C., & Visser, J. (2010). Deriving metric thresholds from benchmark data. In *2010 IEEE International Conference on Software Maintenance* (pp. 1–10). IEEE. https://doi.org/10.1109/ICSM.2010.5609747

Ayala, C., Hauge, Ø., Conradi, R., Franch, X., Li, J., & Velle, K. S. (2009). Challenges of the Open Source Component Marketplace in the Industry. In C. Boldyreff, K. Crowston, B. Lundell, & A. I. Wasserman (Eds.), *Open Source Ecosystems: Diverse Communities Interacting: 5th IFIP WG 2.13 International Conference on Open Source Systems, OSS 2009, Skövde, Sweden, June 3-6, 2009. Proceedings* (pp. 213–224). Berlin, Heidelberg: Springer Berlin Heidelberg. https://doi.org/10.1007/978-3-642-02032-2_19

B. M. Subraya, & S. V. Subrahmanya. (2000). Object driven performance testing of Web applications. In *Proceedings First Asia-Pacific Conference on Quality Software* (pp. 17–26). https://doi.org/10.1109/APAQ.2000.883774

Barkmann, H., Lincke, R., & Löwe, W. (2009). Quantitative Evaluation of Software Quality Metrics in Open-Source Projects. In *2009 International Conference on Advanced Information Networking and Applications Workshops* (pp. 1067–1072). https://doi.org/10.1109/WAINA.2009.190

Chang, J. M., Ho, P. C., & Chang, T. C. (2014). Securing BYOD. *IT Professional*, *16*(5), 9–11. https://doi.org/10.1109/MITP.2014.76

Cristian, F. (1996). Synchronous and Asynchronous. *Commun. ACM*, *39*(4), 88–97. https://doi.org/10.1145/227210.227231

Denaro, G., Polini, A., & Emmerich, W. (2004). Early Performance Testing of Distributed Software Applications. In *Proceedings of the 4th International Workshop on Software and Performance* (pp. 94–103). New York, NY, USA: ACM. https://doi.org/10.1145/974044.974059

Dietrich, S. W., Brown, M., Cortes-Rello, E., & Wunderlin, S. (1992). A Practitioner's Introduction to Database Performance Benchmarks and Measurements. *The Computer Journal*, *35*(4), 322–331. https://doi.org/10.1093/comjnl/35.4.322

Draheim, D., Grundy, J., Hosking, J., Lutteroth, C., & Weber, G. (2006). Realistic load testing of Web applications. In *Conference on Software Maintenance and Reengineering (CSMR'06)* (p. 11 pp.-70). https://doi.org/10.1109/CSMR.2006.43

Filipe, R., Boychenko, S., & Araujo, F. (2015). On Client-Side Bottleneck Identification in HTTP Servers. In *10th International Conference on Internet and Web Applications and Services. IARIA* (pp. 22–27).

Fourment, M., & Gillings, M. R. (2008). A comparison of common programming languages used in bioinformatics. *BMC Bioinformatics*, *9*(1), 82. https://doi.org/10.1186/1471-2105-9-82

Friedman, D. P., & Wise, D. S. (1978). Aspects of Applicative Programming for Parallel Processing. *IEEE Trans. Comput.*, *27*(4), 289–296. https://doi.org/10.1109/TC.1978.1675100

Geiger, N., George, T., Hahn, M., Jubeh, R., & Zündorf, A. (2010). Using Actions Charts for Reactive Web Application Modeling. In F. Daniel & F. M. Facca (Eds.), *Current Trends in Web Engineering: 10th International Conference on Web Engineering ICWE 2010 Workshops, Vienna, Austria, July 2010, Revised Selected Papers* (pp. 49–60). Berlin, Heidelberg: Springer Berlin Heidelberg. https://doi.org/10.1007/978-3-642-16985-4_5

Gizas, A., Christodoulou, S., & Papatheodorou, T. (2012a). Comparative evaluation of javascript frameworks. In *Proceedings of the 21st International Conference on World Wide Web* (pp. 513–514). ACM. Retrieved from http://dl.acm.org/citation.cfm?id=2188103

Gizas, A., Christodoulou, S., & Papatheodorou, T. (2012b). Comparative Evaluation of Javascript Frameworks. In *Proceedings of the 21st International Conference on World Wide Web* (pp. 513–514). New York, NY, USA: ACM. https://doi.org/10.1145/2187980.2188103

Graziotin, D., & Abrahamsson, P. (2013). Making Sense Out of a Jungle of JavaScript Frameworks. In J. Heidrich, M. Oivo, A. Jedlitschka, & M. T. Baldassarre (Eds.), *Product-Focused Software Process Improvement: 14th International Conference, PROFES 2013, Paphos, Cyprus, June 12-14, 2013. Proceedings* (pp. 334–337). Berlin, Heidelberg: Springer Berlin Heidelberg. https://doi.org/10.1007/978-3-642-39259-7_28

Horký, V., Libič, P., Steinhauser, A., & Tůma, P. (2015). DOs and DON'Ts of Conducting Performance Measurements in Java. In *Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering* (pp. 337–340). New York, NY, USA: ACM. https://doi.org/10.1145/2668930.2688820

J. L. Henning. (2000). SPEC CPU2000: measuring CPU performance in the New Millennium. *Computer*, *33*(7), 28–35. https://doi.org/10.1109/2.869367

J. W. Haskins, & K. Skadron. (2001). Minimal subset evaluation: rapid warm-up for simulated hardware state. In *Proceedings 2001 IEEE International Conference on Computer Design: VLSI in Computers and Processors. ICCD 2001* (pp. 32–39). https://doi.org/10.1109/ICCD.2001.955000

Jain, N., Mangal, P., & Mehta, D. (2015). AngularJS: A modern MVC framework in JavaScript. *Journal of Global Research in Computer Science*, *5*(12), 17–23.

Kambona, K., Boix, E. G., & De Meuter, W. (2013). An Evaluation of Reactive Programming and Promises for Structuring Collaborative Web Applications. In *Proceedings of the 7th Workshop on Dynamic Languages and Applications* (p. 3:1–3:9). New York, NY, USA: ACM. https://doi.org/10.1145/2489798.2489802

Klein, J., & Spector, L. (2007). Unwitting Distributed Genetic Programming via Asynchronous JavaScript and XML. In *Proceedings of the 9th Annual Conference on Genetic and Evolutionary Computation* (pp. 1628–1635). New York, NY, USA: ACM. https://doi.org/10.1145/1276958.1277282

Krishnamurthy, D., Rolia, J. A., & Majumdar, S. (2006). A synthetic workload generation technique for stress testing session-based systems. *IEEE Transactions on Software Engineering*, *32*(11).

Kureshi, I., Holmes, V., & Liang, S. (2010, September). *Hybrid HPC – Establishing a Bi-Stable Dual Boot Cluster for Linux with OSCAR middleware and Windows HPC 2008 R2*. Retrieved from http://eprints.hud.ac.uk/id/eprint/9897/

Mariano, C. L. (2017). *Benchmarking JavaScript Frameworks* (Masters dissertation). Dublin Institute of Technology, Dublin, Ireland. https://doi.org/10.21427/D72890

May, N., Kossmann, D., Kaufmann, M., & Fischer, P. M. (2013). *Benchmarking Databases with History Support* (pp. 1–44). ETH Zurich. https://doi.org/10.3929/ethz-a-009994978

Menasce, D. A., & Almeida, V. (2001). *Capacity Planning for Web Services: Metrics, Models, and Methods* (1st ed.). Upper Saddle River, NJ, USA: Prentice Hall PTR.

Nambiar, R., & Poess, M. (2013). Keeping the TPC Relevant! *Proc. VLDB Endow.*, *6*(11), 1186–1187. https://doi.org/10.14778/2536222.2536252

O. Hauge, T. Osterlie, C. F. Sorensen, & M. Gerea. (2009). An empirical study on selection of Open Source Software - Preliminary results. In *2009 ICSE Workshop on Emerging Trends in Free/Libre/Open Source Software Research and Development* (pp. 42–47). https://doi.org/10.1109/FLOSS.2009.5071359

Okur, S., Hartveld, D. L., Dig, D., & Deursen, A. van. (2014). A Study and Toolkit for Asynchronous Programming in C#. In *Proceedings of the 36th International Conference on Software Engineering* (pp. 1117–1127). New York, NY, USA: ACM. https://doi.org/10.1145/2568225.2568309

P. Miguel, J., Mauricio, D., & Rodríguez, G. (2014). A Review of Software Quality Models for the Evaluation of Software Products. *International Journal of Software Engineering & Applications*, *5*(6), 31–53. https://doi.org/10.5121/ijsea.2014.5603

P. Saxena, D. Akhawe, S. Hanna, F. Mao, S. McCamant, & D. Song. (2010). A Symbolic Execution Framework for JavaScript. In *2010 IEEE Symposium on Security and Privacy* (pp. 513–528). https://doi.org/10.1109/SP.2010.38

Pano, A., Graziotin, D., & Abrahamsson, P. (2016). What leads developers towards the choice of a JavaScript framework? *CoRR*, *abs/1605.04303*. Retrieved from http://arxiv.org/abs/1605.04303

Park, R. E. (1992). *Software size measurement: A framework for counting source statements* (No. No. CMU/SEI/92-TR-20) (pp. 1–13). CARNEGIE-MELLON UNIV PITTSBURGH PA SOFTWARE ENGINEERING INST. Retrieved from http://oai.dtic.mil/oai/oai?verb=getRecord&metadataPrefix=html&identifier=ADA258304

Ramos, M. E., & Valente, M. T. (2014). Análise de Métricas Estáticas para Sistemas JavaScript. In *Anais do II Workshop on Software Visualization, Evolution and Maintenance* (pp. 30–37).

Ratanaworabhan, P., Livshits, B., & Zorn, B. G. (2010). JSMeter: Comparing the Behavior of JavaScript Benchmarks with Real Web Applications. In *Proceedings of the 2010 USENIX Conference on Web Application Development* (pp. 3–3). Berkeley, CA, USA: USENIX Association. Retrieved from http://dl.acm.org/citation.cfm?id=1863166.1863169

Richards, G., Gal, A., Eich, B., & Vitek, J. (2011). Automated Construction of JavaScript Benchmarks. In *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications* (pp. 677–694). New York, NY, USA: ACM. https://doi.org/10.1145/2048066.2048119

S. Oney, & B. Myers. (2009). FireCrystal: Understanding interactive behaviors in dynamic web pages. In *2009 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)* (pp. 105–108). https://doi.org/10.1109/VLHCC.2009.5295287

S. Tilkov, & S. Vinoski. (2010). Node.js: Using JavaScript to Build High-Performance Network Programs. *IEEE Internet Computing*, *14*(6), 80–83. https://doi.org/10.1109/MIC.2010.145

Schreier, S. (2011). Modeling RESTful Applications. In *Proceedings of the Second International Workshop on RESTful Design* (pp. 15–21). New York, NY, USA: ACM. https://doi.org/10.1145/1967428.1967434

Silberschatz, A., Galvin, P. B., & Gagne, G. (2014). *Operating system concepts essentials*. John Wiley & Sons, Inc.

Souders, S. (2008). High-performance Web Sites. *Commun. ACM*, *51*(12), 36–41. https://doi.org/10.1145/1409360.1409374

Stol, K.-J., & Ali Babar, M. (2010). Challenges in Using Open Source Software in Product Development: A Review of the Literature. In *Proceedings of the 3rd International Workshop on Emerging Trends in Free/Libre/Open Source Software Research and Development* (pp. 17–22). New York, NY, USA: ACM. https://doi.org/10.1145/1833272.1833276

Suh, Y.-K., Snodgrass, R. T., Kececioglu, J. D., Downey, P. J., Maier, R. S., & Yi, C. (2017). EMP: execution time measurement protocol for compute-bound programs. *Software: Practice and Experience*, *47*(4), 559–597. https://doi.org/10.1002/spe.2476

T. J. McCabe. (1976). A Complexity Measure. *IEEE Transactions on Software Engineering*, *SE-2*(4), 308–320. https://doi.org/10.1109/TSE.1976.233837

Tanenbaum, A. S. (2009). *Modern operating system*. Pearson Education, Inc.

Teitel, R. F. (1981). Volume Testing of Statistical/Database Software. In W. F. Eddy (Ed.), *Computer Science and Statistics: Proceedings of the 13th Symposium on the Interface* (pp. 113–115). New York, NY: Springer US. https://doi.org/10.1007/978-1-4613-9464-8_16

Vokolos, F. I., & Weyuker, E. J. (1998). Performance testing of software systems (pp. 80–87). ACM Press. https://doi.org/10.1145/287318.287337

APPENDIX A: TABLE RESULTS FROM THE BENCHMARK APPLICATION

	angular v1.6.3-keyed	aurelia v1.1.5-non-keyed	ember v2.16.2-keyed
create rows	404.89	337.02	667.27
Processes	218	217	214
replace all rows	371.2	118.56	449.41
Processes	218	217	213
partial update	133.55	2064.63	225.37
Processes	218	213	214
select row	9.28	14.64	17.24
Processes	218	213	214
swap rows	244.54	29.6	253.82
Processes	218	212	214
remove row	92.99	85.49	112.23
Processes	218	212	214
create many rows	3816.75	3255.13	4439.38
Processes	218	212	214
append rows to large table	528.42	398.58	669.96
Processes	216	212	214
clear rows	876.6	492.98	476.88
Processes	216	212	213
Insert into DB	454.4	368.37	710.8
Processes	216	212	213
Select from DB	386.34	322	506.73
Processes	216	212	213
Update DB	33.47	22.46	152.81
Processes	216	212	213
Delete from DB	147.97	65.41	66.68
Processes	216	212	213
startup time	479.8	71.9	339.8
Processes	209	205	206
slowdown geometric mean	**1.78**	**1.22**	**2.11**

Table Appendix A - Wired connection in Linux (10 runs)

	angular v1.6.3-keyed	aurelia v1.1.5-non-keyed	ember v2.16.2-keyed
create rows	451.26	379.14	770.3
Processes	67	67	69
replace all rows	432.99	130.17	484.12
Processes	67	67	68
partial update	182.36	2913.02	242.17
Processes	67	67	67
select row	11.08	14.88	11.63
Processes	67	67	67
swap rows	278.03	32.37	281.43
Processes	67	67	67
remove row	115.22	106.88	118.79
Processes	67	67	67
create many rows	4309.51	3672.35	5066.91
Processes	67	67	66
append rows to large table	595.84	562.17	813.78
Processes	67	67	68
clear rows	793.41	505.99	504.14
Processes	68	67	67
Insert into DB	515.99	439.12	740.53
Processes	67	67	67
Select from DB	435.64	332.25	550.15
Processes	67	67	67
Update DB	35.07	25.07	173.07
Processes	67	67	67
Delete from DB	154.43	64.49	73.5
Processes	67	67	67
startup time	638.4	91.5	277.8
Processes	62	62	62
slowdown geometric mean	**1.75**	**1.22**	**1.97**

Table Appendix A - Wired connection in Windows (10 runs)

	angular v1.6.3-keyed	aurelia v1.1.5-non-keyed	ember v2.16.2-keyed
create rows	403.13	341.21	654
Processes	229	224	222
replace all rows	365.02	112.34	474.21
Processes	228	224	222
partial update	135.33	1997.58	223.2
Processes	228	224	223
select row	15.35	10.87	17.8
Processes	228	224	223
swap rows	243.28	27.88	245.98
Processes	225	224	223
remove row	94.95	90.4	108.94
Processes	220	224	224
create many rows	3834.47	3075.4	4532.7
Processes	220	224	224
append rows to large table	514.56	409.41	686.19
Processes	220	224	224
clear rows	862.35	517.78	542.07
Processes	220	224	223
Insert into DB	468.01	399.31	693.72
Processes	220	224	223
Select from DB	354.69	326.57	449.64
Processes	220	224	223
Update DB	32.7	22.31	184.58
Processes	220	224	223
Delete from DB	149.14	65.44	66.62
Processes	220	224	222
startup time	521.3	71.8	332.6
Processes	213	218	215

Table Appendix A - Wireless connection in Linux (10 runs)

type	angular v1.6.3-keyed	aurelia v1.1.5-non-keyed	ember v2.16.2-keyed
create rows	456.89	382.93	792.03
Processes	68	67	68
replace all rows	436.01	133.04	476.01
Processes	68	67	68
partial update	183.84	2941.75	243.85
Processes	68	67	68
select row	11.56	13.04	13.71
Processes	68	67	68
swap rows	280.3	35.79	282.92
Processes	68	67	68
remove row	113.29	103.71	122.45
Processes	68	67	68
create many rows	4305.13	3701.43	5065.15
Processes	68	67	67
append rows to large table	588.39	565.17	819.85
Processes	67	67	67
clear rows	791.03	495.29	507.23
Processes	67	67	67
Insert into DB	524.06	447.67	955.13
Processes	75	67	67
Select from DB	403.37	381.76	581.73
Processes	67	67	67
Update DB	36.35	26.57	188.52
Processes	67	67	67
Delete from DB	156.82	64.21	72.94
Processes	75	67	67
startup time	542.4	85.3	273.7
Processes	62	62	62
slowdown geometric mean	**1.7**	**1.22**	**1.99**

Table Appendix A - Wireless connection in Windows (10 runs)

AngularJS	**Results**
Mean per-function logical LOC	4.246203
Mean per-function parameter count	1.515823
Mean per-function cyclomatic complexity	2.174684
Mean per-function Halstead effort	2696.141
Mean per-module maintainability index	120.3792
dist\main.js	
Physical LOC	6
Logical LOC	6711
Mean parameter count	1.515823
Cyclomatic complexity	1857
Cyclomatic complexity density	27.67%
Maintainability index	120.3792

Table Appendix A - AngularJS complexity measurement results

Aurelia	**Results**
Mean per-function logical LOC	2.877216
Mean per-function parameter count	0.998708
Mean per-function cyclomatic complexity	1.607381
Mean per-function Halstead effort	1798.728
Mean per-module maintainability index	130.0242
scripts\app-bundle.js	
Physical LOC	1
Logical LOC	135
Mean parameter count	0.666667
Cyclomatic complexity	14
Cyclomatic complexity density	10.37%
Maintainability index	138.254
scripts\vendor-bundle.js	
Physical LOC	1
Logical LOC	7383
Mean parameter count	1.330749
Cyclomatic complexity	1988
Cyclomatic complexity density	26.93%
Maintainability index	121.7944

Table Appendix A - Aurelia complexity measurement results

Ember	
Mean per-function logical LOC	3.633435
Mean per-function parameter count	1.06286
Mean per-function cyclomatic complexity	1.557052
Mean per-function Halstead effort	1631.141
Mean per-module maintainability index	125.0196
dist\assets\ember-temp-….js	
Physical LOC	33
Logical LOC	211
Mean parameter count	0.741379
Cyclomatic complexity	22
Cyclomatic complexity density	10.43%
Maintainability index	126.8302
dist\assets\vendor-….js	
Physical LOC	4098
Logical LOC	18721
Mean parameter count	1.384342
Cyclomatic complexity	3893
Cyclomatic complexity density	20.79%
Maintainability index	123.209

Table Appendix A - Ember complexity measurement results