

2017-1

## Benchmarking JavaScript Frameworks

Carl Lawrence Mariano  
*Technological University Dublin*

Follow this and additional works at: <https://arrow.tudublin.ie/scschcomdis>

 Part of the [Computer Engineering Commons](#)

---

### Recommended Citation

Mariano, C. L. (2017) *Benchmarking JavaScript Frameworks*. Masters dissertation, 2017. doi:10.21427/D72890

This Dissertation is brought to you for free and open access by the School of Computing at ARROW@TU Dublin. It has been accepted for inclusion in Dissertations by an authorized administrator of ARROW@TU Dublin. For more information, please contact [yvonne.desmond@tudublin.ie](mailto:yvonne.desmond@tudublin.ie), [arrow.admin@tudublin.ie](mailto:arrow.admin@tudublin.ie), [brian.widdis@tudublin.ie](mailto:brian.widdis@tudublin.ie).



This work is licensed under a [Creative Commons Attribution-Noncommercial-Share Alike 3.0 License](#)

# Benchmarking JavaScript Frameworks



**Carl Lawrence Mariano**

B.E. Hons, Computer Engineering, Dublin Institute of Technology, 2014.

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 2017**

I certify that this dissertation which I now submit for examination for the award of MSc in Computing (Advanced Software Development), 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 text 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.

*Carl Lawrence Mariano*

**Signed:** \_\_\_\_\_

**Date:**                      **03 January 2017**

## ABSTRACT

JavaScript programming language has been in existence for many years already and is one of the most widely known, if not, the most used front-end programming language in web development. However, JavaScript is still evolving and with the emergence of JavaScript Frameworks (JSF), there has been a major change in how developers develop software nowadays. Developers these days often use more than one framework in order to fulfil their job which has given rise to the problem for developers when it comes to choosing the right JavaScript framework to develop software which is partly due to the availability of countless numbers of JavaScript frameworks and libraries. Moreover, the use of JavaScript is getting more important for web development and thus, there has been major considerations done about the performance aspect of the JavaScript programming language. Thus, this work investigates current research regarding the comparison of JavaScript frameworks through the use of computer benchmarks. A benchmark reference application that simulates user events was developed which then incorporated the implementation of an application developed in each of the JavaScript frameworks chosen. In addition, software complexity metrics was introduced and experiments were conducted to measure these metrics. Overall, this research hopes to achieve a level of comparison which can further garner knowledge towards comparing JavaScript frameworks.

**Key words:** *software development, software engineering, JavaScript, performance testing, JavaScript frameworks, framework comparison, computer benchmarking, web development*

## **ACKNOWLEDGEMENTS**

I would like to express my sincere thanks to my supervisor Dr. John Gilligan for his time, patience, knowledge, advice and for constantly pushing me to get things done by keeping the standards of work high during the course of this project.

I would also like to give a special thanks to our research dissertation coordinator Dr. Luca Longo for initially helping me come up with a topic for my research, although he was busy, when I was struggling to find one.

Finally, I want to dedicate this work to my family for their never ending support especially during times of difficulty. If it wasn't for their love and support this project would have never come to fulfilment.

# TABLE OF CONTENTS

ABSTRACT .....	II
----------------	----

TABLE OF FIGURES .....	VII
------------------------	-----

1. INTRODUCTION.....	1
----------------------	---

THE ROLE OF JAVASCRIPT IN WEB DEVELOPMENT .....	1
1.1 BACKGROUND .....	2
1.2 RESEARCH PROJECT/PROBLEM.....	3
1.3 RESEARCH OBJECTIVES.....	4
1.4 RESEARCH METHODOLOGIES .....	5
1.5 SCOPE AND LIMITATIONS .....	5
1.6 DOCUMENT OUTLINE .....	7

2. LITERATURE REVIEW & RELATED WORK.....	8
--	---

2.1 BENCHMARKING.....	8
2.1.1 <i>Benchmark Definitions</i> .....	8
2.2 COMPUTER BENCHMARKING .....	9
2.2.1 <i>The Nature of Computer Benchmarks</i> .....	10
2.2.2 <i>The Importance of Benchmarking</i> .....	12
2.2.3 <i>Classifications of Benchmarks</i> .....	14
2.3 APPROACHES TO BENCHMARKING.....	17
2.3.1 <i>Processor Benchmarks</i> .....	18
2.3.2 <i>Systems Benchmarks</i> .....	19
2.3.3 <i>Programming Language Benchmarks</i> .....	19
2.4 FRAMEWORKS .....	21
2.4.1 <i>JavaScript Frameworks Comparison Research</i> .....	22
2.5 JAVASCRIPT AND JAVASCRIPT FRAMEWORKS.....	24
2.5.1 <i>AngularJS</i> .....	25
2.5.2 <i>React</i> .....	25
2.5.3 <i>BackboneJS</i> .....	26
2.5.4 <i>Execution of JavaScript</i> .....	27
2.5.5 <i>Data Bindings Explained</i> .....	27
2.5.6 <i>States Explained</i> .....	29

2.6	MODEL-VIEW-CONTROLLER (MVC).....	31
2.6.1	<i>Model</i> .....	32
2.6.2	<i>View</i> .....	32
2.6.3	<i>Controller</i> .....	33
2.7	OVERVIEW OF BENCHMARKING METRICS .....	33
2.7.1	<i>Selection of Metrics</i> .....	34
2.8	CHAPTER SUMMARY .....	37
<b>3.</b>	<b>DESIGN &amp; METHODOLOGY .....</b>	<b>39</b>
3.1	REFERENCE BENCHMARK APPLICATION DESIGNS .....	39
3.1.1	<i>Todo Benchmark Application Design</i> .....	39
3.1.2	<i>Database Benchmark Application Design</i> .....	45
3.2	EXPERIMENT DESIGNS.....	47
3.3	CHAPTER SUMMARY .....	49
<b>4.</b>	<b>IMPLEMENTATION .....</b>	<b>50</b>
4.1	SOFTWARE USED.....	50
4.2	TODO REFERENCE APPLICATION IMPLEMENTATIONS .....	52
4.2.1	<i>AngularJS Todo Application</i> .....	52
4.2.2	<i>React Todo Application</i> .....	53
4.2.3	<i>Backbone Todo Application</i> .....	57
4.3	BENCHMARKING CLOCK IMPLEMENTATION .....	60
4.4	DISCONTINUED IMPLEMENTATIONS.....	62
4.5	BENCHMARK TEST ENVIRONMENT.....	62
4.6	CHAPTER SUMMARY .....	63
<b>5.</b>	<b>RUNNING THE EXPERIMENTS &amp; EVALUATION .....</b>	<b>64</b>
5.1	EXPERIMENTATION.....	64
5.1.1	<i>Implementation of Benchmark Application</i> .....	64
5.1.2	<i>Running the Experiments</i> .....	69
5.2	EVALUATION.....	72
5.2.1	<i>Todo Application Benchmark Results</i> .....	72
5.2.2	<i>Software Complexity Measurement Results</i> .....	77
5.2.3	<i>Strengths and Limitations</i> .....	80
5.3	CHAPTER SUMMARY .....	81

<b>6. CONCLUSION &amp; FUTURE WORK.....</b>	<b>82</b>
6.1 RESEARCH OVERVIEW.....	82
6.2 EXPERIMENTATION, EVALUATION AND LIMITATIONS .....	82
6.3 CONTRIBUTIONS & IMPACT .....	84
6.4 FUTURE WORK & RECOMMENDATIONS .....	84
<b>BIBLIOGRAPHY .....</b>	<b>86</b>
<b>APPENDIX A: BENCHMARK SCRIPTS SOURCE CODE .....</b>	<b>92</b>



## TABLE OF FIGURES

FIGURE 1 DIFFERENT TYPES OF PERFORMANCE TESTS AND BENCHMARKS .....	11
FIGURE 2 ILLUSTRATION OF A COMPARISON FRAMEWORK.....	23
FIGURE 3 MODELS AND VIEWS CONCEPT IN BACKBONE.....	26
FIGURE 4 COLLECTIONS IN BACKBONE.....	27
FIGURE 5 ONE-WAY DATA BINDING .....	28
FIGURE 6 TWO-WAY DATA BINDING .....	29
FIGURE 7 MODEL-VIEW-CONTROLLER.....	31
FIGURE 8 USER INTERFACE OF TODO APPLICATION.....	41
FIGURE 9 TODO APPLICATION SHOWING COMPLETION OF AN ITEM.....	41
FIGURE 10 DEFAULT USER INTERFACE OF TODO APPLICATION.....	42
FIGURE 11 BENCHMARK APPLICATION USER INTERFACE.....	42
FIGURE 12 BENCHMARK APPLICATION SHOWING "ADDING100ITEMS" TASK BEING EXECUTED.....	43
FIGURE 13 BENCHMARK APPLICATION SHOWING "COMPLETINGALLITEMS" TASK BEING EXECUTED.....	44
FIGURE 14 SAMPLE GRAPH OF RESULTS.....	45
FIGURE 15 EXAMPLE OF A DBMONSTER APPLICATION INTERFACE.....	47
FIGURE 16 EXPERIMENT PROCESS PART A .....	48
FIGURE 17 EXPERIMENT PROCESS PART B .....	49
FIGURE 18 CHECKING VERSION OF NODEJS AND NPM .....	51
FIGURE 19 EXAMPLE SHOWING INSTALLATION OF NODE MODULE .....	51
FIGURE 20 EXAMPLE OF HOW AN INPUT IS SAVED IN ANGULARJS.....	52
FIGURE 21 EXAMPLE OF HIDING FOOTER IN ANGULARJS.....	53
FIGURE 22 EXAMPLE OF SHOWING A LIST OF TODO ITEMS IN ANGULARJS.....	53
FIGURE 23 INPUT FIELD COMPONENT IN REACT .....	54
FIGURE 24 LOGIC FOR THE INPUT FIELD COMPONENT .....	54
FIGURE 25 EXAMPLE OF HIDING/SHOWING THE FOOTER IN REACT .....	55
FIGURE 26 EXAMPLE OF A TASK ITEM IN REACT .....	55
FIGURE 27 EXAMPLE OF A CREATION OF TASK ITEM IN REACT .....	56
FIGURE 28 EXAMPLE SHOWING RENDERING OF ALL TASK ITEMS IN REACT.....	56
FIGURE 29 EXAMPLE OF THE APPLICATION VIEW IN BACKBONE .....	57

FIGURE 30 HTML VIEW OF TODO APPLICATION IN BACKBONE .....	57
FIGURE 31 EXAMPLE SHOWING INITIALIZATION OF APP .....	58
FIGURE 32 EXAMPLE SHOWING RENDER FUNCTION OF APP IN BACKBONE .....	58
FIGURE 33 EXAMPLE SHOWING FUNCTIONS TRIGGERED FOR EVENTS .....	59
FIGURE 34 EXAMPLE SHOWING GENERATION OF ATTRIBUTES AND CLEARING THE MODEL .....	59
FIGURE 35 CODE SNIPPET OF CLOCK IMPLEMENTATION .....	60
FIGURE 36 CODE SNIPPET SHOWING IMPLEMENTATION OF ASYNCHRONOUS TIMER.....	62
FIGURE 37 CLONING A GITHUB REPOSITORY.....	66
FIGURE 38 SNIPPET CODE OF ADDING A SUITE OF TEST .....	67
FIGURE 39 SNIPPET CODE FOR CREATION OF UI LAYOUT .....	68
FIGURE 40 SNIPPET CODE OF STARTTEST FUNCTION .....	68
FIGURE 41 RUNNING THE WEB SERVER.....	69
FIGURE 42 INTERFACE SHOWING THE BENCHMARK APPLICATION .....	70
FIGURE 43 IMAGE SHOWING RETRIEVAL OF FILES .....	70
FIGURE 44 EXAMPLE OF RUNNING THE COMPLEXITY REPORT TOOL.....	71
FIGURE 45 AVERAGE RESULTS GENERATED IN GOOGLE CHROME AFTER 1 RUN .....	72
FIGURE 46 AVERAGE RESULTS GENERATED IN GOOGLE CHROME AFTER 25 RUNS .....	73
FIGURE 47 AVERAGE RESULTS GENERATED IN MICROSOFT EDGE AFTER 1 RUN .....	74
FIGURE 48 AVERAGE RESULTS GENERATED IN MICROSOFT EDGE AFTER 25 RUNS .....	74
FIGURE 49 AVERAGE RESULTS GENERATED IN MOZILLA FIREFOX AFTER 1 RUN .....	75
FIGURE 50 AVERAGE RESULTS GENERATED IN MOZILLA FIREFOX AFTER 25 RUNS .....	76
FIGURE 51 FIGURE SHOWING MEAN PER-FUNCTION LOGICAL LOC .....	77
FIGURE 52 FIGURE SHOWING MEAN PER-FUNCTION CYCLOMATIC COMPLEXITY .....	78
FIGURE 53 FIGURE SHOWING MEAN PER-FUNCTION HALSTEAD EFFORT .....	79
FIGURE 54 FIGURE SHOWING MEAN PER-MODULE MAINTAINABILITY INDEX.....	80

**TABLE OF TABLES**

TABLE 1 SUMMARY OF CLASSIFICATIONS OF BENCHMARKS ..... 17

TABLE 2 OVERVIEW OF SELECTED METRICS..... 37

# 1. INTRODUCTION

This project involves the comparison of JavaScript frameworks which increasingly have become a cornerstone of web development.

This research evaluates the performance of JavaScript frameworks in order to further expand knowledge and research towards comparing JavaScript frameworks. In simple terms, a JavaScript framework is a web application framework that is written using the JavaScript language. A JavaScript framework differs to that of a JavaScript library in that, a library comes packaged with predefined functions that are ready to be used by developers straight out of the box. On the other hand, a JavaScript framework describes how an application should be built and allows for code to be reusable and more organized which in turn, reinforces the scalability and flexibility of an application.

Therefore, in this work, a number of JavaScript frameworks are evaluated by building a number of reference applications along with the use of multiple tools to perform the experiments and assess each of the framework's performance based on a number of benchmark metrics as described in this thesis. Moreover, the experiment and evaluation process was performed as fair as possible on each of the JavaScript frameworks to prevent biased results.

## *The role of JavaScript in Web development*

Organizations and enterprises in the software industry have relied hugely on the web since its initial breakthrough in order deliver products to its customers as well which allowed them to succeed. The web was invented by Sir Tim Berners-Lee in 1989 who was a British computer scientist and without the emergence of the web, major components of the web would not exist today as we see it. Therefore, it can be seen that web technologies have rapidly evolved in the web's history. With the arrival of Web 2.0, there was a massive increase in the innovations of web applications, where applications are becoming more interactive as users are now able to add customized information like posts and blogs compared to when the web were mostly read-only before. With the web being born, it too gave birth to technologies such as HTML, JavaScript and CSS which allowed for the development of rich web applications capable of adding more effects and new ways to interact with these applications.

## ***1.1 Background***

JavaScript is one of the most widely known, if not, the most used programming language in front-end web development. JavaScript is usually implemented alongside HTML and CSS3 at the client-side to create attractive, interactive and creative web designs. The evolution of JavaScript have since been exploited with the emergence of JavaScript Frameworks (JSF) due to various programming needs in order to make it easier and to better manage the development of web applications and to reduce the complexity of developing these applications (Chuan, Wang, 2009). A JavaScript framework is a given structure of how code should be written. It is a set of functions and tools that make it much easier to develop cross-browser compatible JavaScript code. In other words, it's more like a code-template for developing applications that greatly reduces the cost and time of development. A typical JavaScript framework should abstract or generalize the most complex and longest operations and ensure cross-browser support and compatibility which in turn, enables for the rapid development of software.

Developers these days often use more than one frameworks and libraries to fulfil their job especially when developing large scale and complex web applications. One of the main advantages in doing this is that they can reuse code which allows organizations to focus more of their time and attention towards designing a scalable web application by choosing the appropriate framework, one which can be embraced by all developers within a company. On the other hand, there is also a danger when it comes to choosing a less suitable framework which greatly affects the development of an application which in turn, causes a chain reaction where the quality of the application is reduced and deadlines are missed.

However, JavaScript framework comparison is not an established area and is relatively new in the field of research. The closest field towards comparing JavaScript frameworks is software architecture comparison. But, software architecture comparison too is also quite a young discipline where one of the most popular methods of comparison is the Software Architecture Analysis Method (SAAM), which originated in 1996 (Fernández-Villamor, Casillas, & Iglesias, 2008). This project was performed at Decerno which is a small IT consultancy company established in Sweden in 1984 that has been proven to build custom systems for its customers. Their focus has always been on web development and as they build custom systems, they assess

well the tools and frameworks that they're going to use and as required for the particular project. Their current focus is on building Single Page Applications utilizing the new and emerging JavaScript frameworks.

In view of this, web development framework evaluation and benchmarking in the field of software development and software engineering is the primary motivation for this project especially with the increasing demands of the web as well as the need of web developers to constantly find new ways of developing their applications more efficiently and effectively (Graziotin & Abrahamsson, 2013).

## ***1.2 Research Project/Problem***

The major challenge facing web developers is typically when choosing the right language or framework in order to fulfil their job. In the era of web development, JavaScript is the most popular client-side programming language and have seen major acceptance throughout the whole web development community. In view of this, JavaScript is garnering interest in web development especially with the emergence of new JavaScript frameworks. The most common deciding factor when it comes to choosing the right framework for the job is typically centered on the developer's familiarity, which according to (Lavanya, Ramachandran, & Mustafa, 2010) is not a right basis for choosing a framework as it tends to be subjective. However, one factor that is often overlooked is the performance of such a JavaScript framework which is an important factor especially for enterprises developing complex applications. Thus, with the increasing number of JavaScript Frameworks, web developers are often finding it difficult to select the most appropriate framework to use. For web developers, it is crucial to select a framework that best match their needs and one that provides code of high quality and performance. Therefore, web developers have been reluctant to adapt a new JSF from a framework that they've already become accustomed to as changing frameworks would mean that they would need to allocate time in learning and understanding the new framework. Moreover, they don't consider the improvements that each framework gives (Gizas, Christodoulou, & Papatheodorou, 2012). With the vast number of JavaScript frameworks available today, it is a difficult task especially for developers to get up and started with the right framework.

In view of this, few researches have been conducted in attempting to analyze the various JSF in terms of its performance that would be of great aid to developers in

selecting the most appropriate JSF in a given situation. As of today, according to [jster.net](#), thousands of JS libraries and frameworks are available, each serving different purposes. Examples include jQuery, Backbone.js, React, EmberJS, Knockout and Angular.js (Gizas, Christodoulou, & Papatheodorou, 2012; Jain, Mangal, & Mehta, 2015). Furthermore, research into comparing and evaluating JavaScript frameworks has not been sufficiently completed as all papers read suggested that future work should be done towards reaching an understanding of its performance and quality especially with new and emerging JSFs.

Therefore, this project will attempt to answer the following Research Question:

**Is it appropriate to use Computer and Software Benchmarking metrics for the comparison of JavaScript Frameworks?**

### ***1.3 Research Objectives***

The aim of this research is to garner knowledge and information around the use and performance of JavaScript frameworks and to evaluate them with the intent to applying these frameworks to various reference applications which will be hope to give an insight to researchers on the performance of JavaScript frameworks. Performance in the scope of this thesis is defined as the values returned when a benchmark metric is assessed or measured since not only the performance is being measured here but also this project utilizes various software complexity metrics as benchmarking metrics to measure the quality of software used. The results of each experiment will be performed by means of tests on each JavaScript framework and will be used to analyze the performance of each framework and then compile that knowledge to give an overall view of each of the JavaScript framework's performance.

The Project Objectives are as follows:

- To investigate the current state of the art research conducted to date on benchmarking and JavaScript framework comparison, more specifically, the performance of JavaScript frameworks.
- To develop an experiment in order to evaluate the selected JavaScript frameworks based on a number of computer benchmark metrics.
- Document and evaluate the results and findings from the experiment.

- Based on the evaluation process, give an overall view of the performance of each of the nominated JavaScript frameworks.
- Make recommendations for future research in this area.

#### ***1.4 Research Methodologies***

For this research, a secondary research will be carried out to further understand previous studies and build knowledge for a comprehensive literature review which will be followed by a quantitative research to perform an empirical study deductive reasoning approach to select appropriate benchmark metrics within the scope of this project where the results will be analyzed and presented on a number of graphs for better display of the results.

Therefore, the research is carried out in the following phases:

- Perform a literature review of benchmarking in general and JavaScript & JavaScript frameworks in order to assess the current status of JavaScript framework comparison and to inform the choice of JavaScript frameworks for the experimentation part of this research.
- Design and develop a number of reference applications software to be used for the experiment.
- Adapt the reference applications to use the selected JavaScript frameworks and apply other tools to it.
- Evaluate the performance of the frameworks by reviewing the final experimentations and the effort involved.
- Summarize the research, draw conclusions and suggest recommendations for future research.

#### ***1.5 Scope and Limitations***

This research will appraise the performance of a number of JavaScript frameworks that enable software developers to build web applications using modern web frameworks. In this case, a total of three JavaScript frameworks were selected as it may not be possible to evaluate more than three frameworks due to a shortage in time.

As the objective suggests, this research aims to evaluate the performance of JavaScript frameworks. This is specifically aimed at researchers and developers looking for more



information towards studies concerning the evaluation of JavaScript frameworks and to further enhance current studies already performed on this. Furthermore, as the scope of this project is very broad, especially in terms of benchmarking where hundreds of benchmarking metrics are available, it is impossible to include all of these metrics in this project and therefore, an attempt to gather and decide the most important metrics were performed instead. On the other hand, as mentioned in previous sections, there are thousands of JavaScript frameworks and libraries available today and thus, there are a wide range of frameworks to choose from. Therefore, a decision has been made in the selection of frameworks based on their popularity which is why only three frameworks were chosen for this project.

Consequently, experimentation in this research will include adapting and developing reference applications from previous applications developed by developers on benchmarking JavaScript frameworks which will serve as the main platform for experiments. However, most of these, if not all implementations of the applications are already out of date with new versions of each JavaScript frameworks out every so often and therefore, there is a need to update the code in order to give a more updated benchmark results. The first reference application is an implementation of a Todo application for each JavaScript framework taken from a source code repository and incorporated into one application that executes tasks within each application and produces the amount of time taken to execute those tasks. Here, the time library used to calculate the execution time is very important as it is the main factor for the benchmark as there are many clock/timer libraries available. This will further be discussed in Chapter 4 of Implementation. The second reference application will measure the runtime performance of a database application based on the implementation of the browser-perf<sup>1</sup> library adapted from the source code repository. Finally, the complexity of each Todo implementation for each JavaScript framework will also be analyzed based on a number of software complexity metrics as discussed in Chapter 3. Furthermore, Chapter 2 is a synthesis of all research reviewed in relation to Benchmarking and JavaScript frameworks. As mentioned earlier, this project will attempt to provide a performance analysis of JavaScript frameworks.

---

<sup>1</sup> Browser-perf npm, <https://www.npmjs.com/package/browser-perf>

## ***1.6 Document Outline***

This dissertation is organized as follows. Chapter 2 presents the literature review including a brief history, definitions and background on benchmarking and JavaScript frameworks. From the literature review, the design and methodology will follow up in Chapter 3 and it describes the design of the benchmark reference applications. Chapter 4 describes the tools used to build the applications and the plan and process of developing the benchmark reference applications. Chapter 5 describes the actual work carried out by means of the experiment process along with the interpretation of the results. Finally, Chapter 6 presents the conclusion and recommendations for future research.

## 2. LITERATURE REVIEW & RELATED WORK

This project is about comparing JavaScript frameworks using standard benchmarking approaches. Therefore, this chapter includes a brief introduction to benchmarking and the different types of benchmarking in relation to computer science. This is followed by a discussion of frameworks and more specifically, JavaScript frameworks. It will particularly describe the main features of the various JavaScript frameworks considered in this project. Since the key architecture of these frameworks is based around components of the Model-View-Controller (MVC) pattern, this is also discussed. Similarly, each frameworks approach to web development is considered since this is one of the aspects which distinguishes them.

### 2.1 *Benchmarking*

This section provides a general introduction to benchmarking in the field of software development and software engineering. Some common definitions of benchmarking terminologies will also be described, followed by a discussion of the importance of benchmarking as well as a description of the classifications of common types of benchmarking such as *micro* and *macro-benchmarks*.

#### 2.1.1 Benchmark Definitions

According to the International Organization for Standardization (ISO)<sup>2</sup> and the International Electrotechnical Commission (IEC)<sup>3</sup>, they define benchmark as:

*“A standard against which results can be measured or assessed.*

- (ISO/IEC 25010:2011)

Similarly, IEEE<sup>4</sup> define benchmark as:

*“A standard against which measurements or comparisons can be made.*

---

<sup>2</sup> International Organization for Standardization website, <http://www.iso.org/iso/home/about.htm>

<sup>3</sup> International Electrotechnical Commission website, <http://www.iec.ch/>

<sup>4</sup> IEEE website, <https://www.ieee.org/index.html>

*A procedure, problem, or test that can be used to compare systems or components to each other or to a standard.”*

- (IEEE 24765:2011)

SPEC<sup>5</sup> on the other hand, provides a more focused definition of benchmarking in relation to the performance of computer systems as:

*“A benchmark is a test, or set of tests, designed to compare the performance of one computer system against the performance of others.”*

- (SPEC:2013a)

In relation to computer benchmarking, (Bouckaert et al., 2010) defines computer benchmarking as:

*“The act of measuring and evaluating computational performance, networking protocols, devices and networks, under reference conditions, relative to a reference evaluation”*

Therefore, the next section describes computer benchmarking in detail.

## **2.2 Computer Benchmarking**

The purpose of benchmarking has been around for decades already and has always been the way to compare different platforms, tools, or techniques by means of performing experiments in order to find these differences (Dixit, 1993). With regards to the use of benchmarking tools, it usually refers to a program or a set of programs that are used to evaluate the performance of an application or solution under certain conditions which is relative to the performance of another application or solution. (Bouckaert et al., 2010) states that the goal of benchmarking is to enable a fair comparison between different solutions. The main advantage of benchmarking is that it helps organizations to be more open to other approaches rather than being blinded by a single approach that may seem like the best approach to solve their problems. Benchmarks is also used as a way to standardize measurements and to provide repeatable, objective and results which can be compared to other benchmarks. In the field of computer science, benchmarks are used to compare, for example, CPU performance, database management systems (DBMS), or information retrieval algorithms (Sim, Easterbrook, & Holt, 2003). In other instances, benchmarks are also used to evaluate JavaScript performances (Ratanaworabhan, Livshits, & Zorn, 2010).

---

<sup>5</sup> Standard Performance Evaluation Corporation, <https://www.spec.org/>

In addition to performance evaluations, benchmarks in computer science can also employ other measurements such as the number of false positives or negatives in detection algorithms.

### **2.2.1 The Nature of Computer Benchmarks**

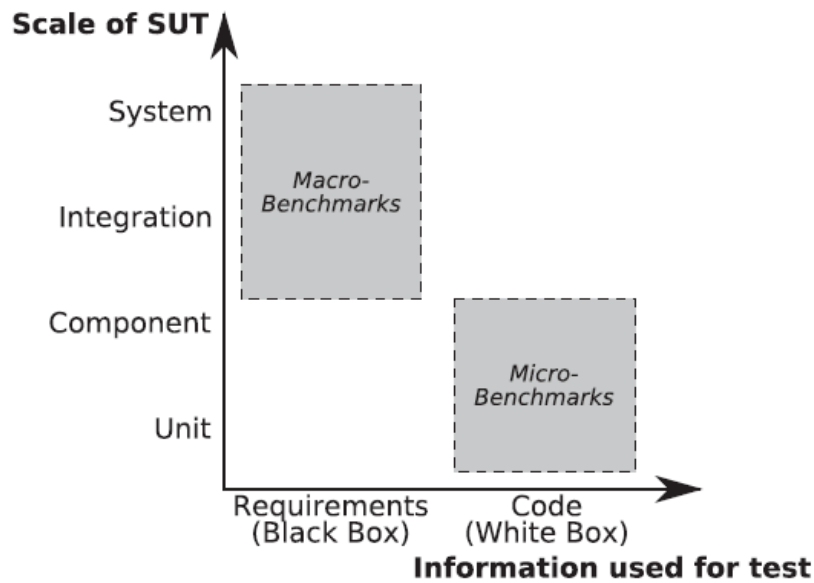
The following metrics are key benchmarking metrics:

- Time – the length of time to complete a job
- Rate - the speed at which a system can perform work

(Menasce & Almeida, 2002, Utting & Legeard, 2007).

Time and rate are the most basic measures of system performance. From the user's point of view, program or application execution time is the best indicator of system performance. Users do not want to know what happens in the background such as if the service is executed in a nearby desktop computer on a wired internet connection or if it is processed thousands of miles away on a remote server from her/his location which can be connected through various networks. Users always want fast response time. On the other hand, from a management's viewpoint, the performance of a system is defined by the rate at which a system can perform work. For example, system managers are interested in questions such as: How many transactions can the system process or execute per minute, or how many requests is the Web server able to process per second? Furthermore, both users and managers are always concerned with cost, which are reflected in questions such as: What is the system's operational cost? And what is the server purchase cost? However, despite all these viewpoints, the same or basic problem remains which is defining a good and efficient standard measure of system performance.

(Utting & Legeard, 2007) also defines the need to provide a defined usage profile (operation profile and scenario in Figure 1) especially for complex benchmark systems to produce repeatable results. The system under test (SUT) or platform that interacts with the benchmark system could be a platform made up of hardware or software, software components, or even single operation. Utting and Legeard employs three dimensions in order to classify testing which includes, the characteristics being tested, the SUT scale, and the information gathered to be used to design the test as illustrated in Figure 1. In the case of benchmarking, the characteristics is fixed at performance.



**Figure 1 Different types of performance tests and benchmarks**

Therefore, only a focus on the two axis as shown in Figure 1 is needed. The SUT scale corresponds to the size or complexity of the benchmark system whereas the information gathered to be used for the tests corresponds to the information that is used to design the benchmarks.

Benchmark results can be both informative and in some cases, may also cause confusion to users about the real capacity of systems to handle all workload being executed by actual applications. Depending on how well one evaluates the results, various interpretations of benchmark results can cause confusion to other researchers or individuals reading these results. That is why, in order for an individual or researcher to use these benchmark results, they must first understand the tasks that will be executed, the system in study, the tests, the metrics and the measurements of these metrics, and finally the results. Otherwise, there may be different interpretations of these benchmark results as opposed to following a standard way of analysing the results (Kelessidis, 2000). Therefore, (Menasce & Almeida, 2002) proposes a number of steps. The first step is to answer a number of questions which follows:

- What actual benchmark metric is being measured?
- How close does the benchmark match the user environment workload?
- What are the measurements of these benchmark metrics?

Once benchmark results are well understood, researchers or individuals can use them to all the more increase one's knowledge about the performance (values returned when a benchmark metrics are assessed) of the system or a web application as related to web

development (e.g. programming languages) under study. Benchmarking regarding programming languages in particular is further discussed in section 2.3 (Approaches to Benchmarking).

### **2.2.2 The Importance of Benchmarking**

A number of papers read have stressed the importance of benchmarks in the field of Computer Science. (Tichy, 2014) states in his article, that benchmarks are an effective and affordable way of conducting experiments in computer science. Using well-known benchmarks that are accepted by the community as representatives of significant applications, in experimental designs suggests a general acceptance of observed results. As a result, the successful evaluation of ideas with the implementation of these types of benchmarks often plays an important role in the acceptance of these formulated ideas (Adamson, Dagastine, & Sarne, 2007).

Thus, it can be said that benchmarks are a central part of scientific investigations as they are able to shape the field of computer science and drive research and product development into new directions (Adamson, Dagastine, & Sarne, 2007). Hence, the use of benchmarks is frequently accompanied by the progression of rapid technologies and especially, the employment of such performance benchmarks has contributed greatly in order to improve generations of new systems (Vieira, Madeira, Sachs, & Kounev, 2012). In summary, benchmarking is at the core of experimental research and computer science. But also, benchmarking is an important activity at the business level.

According to (Sachs, 2011), the development of benchmarks has turned into a complicated team effort involving a large group of people each with different goals and challenges compared to the development of traditional software. (Tichy, 2014) states that constructing benchmarks, in itself is hard work, and is best shared with the involvement of people within communities. Furthermore, benchmarks need to evolve from narrowly targeted tests to a broader, more generalized tests in order to prevent sticking to a specific goal. (Carzaniga & Wolf, 2002) also stressed the importance of designing benchmarks as a community activity rather than within small closed group, resulting in wider acceptance and adoption of the developed benchmark. (Sim, Easterbrook, & Holt, 2003) further urge the community to send their ideas and state that benchmarks must always be developed and with the collaboration of people within

the community, rather than by a single researcher. The quality of good benchmarks emerge from a combination of scientific discovery and popularity and acceptance in the community, which are all equally important. As mentioned in previous sections, the SPEC<sup>6</sup> Research Group is an example of such a community that is actively involved in the development of standardized benchmarks.

However, most popular benchmarks are provided by research communities or larger consortiums involving many companies, organizations or governments. Some of the most popular and widely accepted associations are the already mentioned Standard Performance Evaluation Corporation (SPEC), the Transaction Processing Performance Council (TPC)<sup>7</sup>, and the Defence Advanced Research Projects Agency (DARPA)<sup>8</sup>. The SPEC is a consortium with several groups involved, which regularly create a variety of standardized benchmarks. Their main focus are on benchmarks which compare different hardware systems or software environments. On the other hand, the TPC defines benchmarks relating to transactional processing and databases while the DARPA provides a wide variety of benchmarks including image processing or speech recognition benchmarks.

According to (Pfleeger, 1995), two of the most common empirical research and evaluation methods in software engineering include formal experiments and case studies. Experiments require a high level of control over all variables that would affect the outcome but should also provide the ability to reproduce the experiment and use the experiments for easier comparisons. On the other hand, case studies, require less control but are not often replicable and difficult to generalize. Thus, benchmarks are somewhere in the middle of formal experiments and case studies, and contain elements of both empirical methods (Sim, Easterbrook, & Holt, 2003). Similar to experiments, a benchmarks aim is for a high control of the variables that may influence the experiment and for reproducibility. On the other hand, the actual platform, tool, or technique evaluated by the benchmark can vary, thus each run of the benchmark is quite similar to a case study.

---

<sup>6</sup> Standard Performance Evaluation Corporation, <https://www.spec.org/>

<sup>7</sup> Transaction Processing Performance Council, <http://www.tpc.org/>

<sup>8</sup> Defence Advanced Research Projects Agency, <http://www.darpa.mil/>



### 2.2.3 Classifications of Benchmarks

Benchmarks can be classified into two of the most common categories in benchmarking: micro and macro benchmarks (Seltzer, Krinsky, Smith, & Zhang, 1999). *Micro-benchmarks* are designed to evaluate the performance of a very specific part of a software system, which is usually a small part. On the other hand, *Macro-benchmarks* are large and often complex benchmark systems which is designed to simulate a real system or part of a real system. The benchmark that applies to this project is micro-benchmarking as the aim of this project is to compare the performance of JavaScript frameworks based on a number of metrics which are tested by running a variety of small appropriate tasks. An example of a micro-benchmark is the comparison of the performance of various code operations. These metrics will be further discussed in Chapter 3 of the Design/Methodology section. The following sections describe these two categories of benchmarking in greater detail.

#### 2.2.3.1 Micro-Benchmarks

*Micro-benchmarks* can also be classified as *synthetic* benchmarking and are written to compare and distinguish basic concepts, such as a single operation, or small aspects of a larger system. Typical examples of these benchmarks are the comparison of different algorithms, such as sorting algorithms or performance of an operation on different hardware platforms (Waller, 2015). In relation to JavaScript, micro-benchmarks can be used to evaluate and compare various frameworks and libraries, such as the performance evaluations of JavaScript classes or selectors (Christodoulou & Gizas, 2014).

In Figure 1, Micro-Benchmarks usually correspond to the lower right corner of the scale and information axes. They are focused on a specific part of a system, usually a small part (e.g. a single unit such as an operation or a class). Additionally, this type of benchmark often use a white-box approach in their design, that is, they are designed with the actual system environment under test in mind. In theory, micro-benchmarks excel at their given task of comparing well defined, small properties. However, it is often difficult to find these small, relevant task-samples. Therefore, a lot of micro-benchmarks have only a very limited applicability in real-world scenarios.

On the other hand, the basic concepts of this benchmark make most micro-benchmarks easy to automate. They can be included in continuous integration setups to automatically record performance improvements and regressions (Bulej, Kalibera, & Tuma, 2005). In environments such continuous integration environments, code changes are tested for incompatibilities with other code changes. Additionally, code changes are also test to find new bugs. Usually, these tests are performed automatically on an integration system which notify developers of problems encountered by the system (Fowler & Foemmel, 2006).

The major disadvantage and danger of micro-benchmarks is that results may be found to simple where they often neglect other factors that may influence the results and only focus on a single aspect of the complexity of a system. This can lead to biased and false conclusions and may harm performance tunings if not performed correctly (Mogul, 1992).

#### 2.2.3.2 Macro-Benchmarks

Although macro-benchmark is not the primary benchmark technique used in this project, it is still worth to discuss in order to distinguish it from micro-benchmarking. According to (Hinnant, 1988), macro-benchmarks can also be called *natural* or complex benchmarks and are supposed to represent a relevant task-sample including other factors that may influence the results. Therefore, they often consists of a large portion of the all possible tasks. They are used to overcome the shortcomings of micro-benchmarks and macro-benchmarks usually correspond to the upper left corner of the scale and information axes in Figure 1. Macro-benchmarks typically represent large parts of systems or even complete systems. Furthermore, a black box approach is usually used to represent macro-benchmarks, that is, they are not designed with a specific system under test in mind, but rather with a more general requirements specification. In best case scenarios, macro-benchmark is the actual system under test with a realistic task-sample, for example the macro-benchmark of an online store could be a new instance of the online shop/store system, which is deployed on a similar hardware and software, and used with realistic task-samples. In relation to performance benchmarking, the workload produced by the task-samples could be higher than the

expected workload in order to discover and performance bottlenecks that may be present.

In most cases, macro-benchmarks are representations of real systems. Besides varying the hardware and software running the benchmark, the macro-benchmark itself can be an abstraction or a reduced part of the real system. For example, instead of using the real application, the benchmark may consist of a more generalized, abstract online shop, which simulates the real app. An example of such a macro-benchmark is the SPECjbb®2013<sup>9</sup> application benchmark.

In addition, when the system under test is independent from the actual benchmark system, an abstract benchmark can be common (e.g. typical task-samples). In the case of the SPECjbb®2013 application benchmark, the system under test is usually a combination of hardware which includes a specific application server, while the benchmark system could be an online shop, simulating the typical tasks that the application server might execute.

Likewise, finding a good trade-off between a realistic benchmark system with the added complexity of such a system and the deciding factor of coming up with task samples is usually difficult. Therefore, domain knowledge of experts is an invaluable asset in coming up with these tasks. Apart from the higher complexity, macro-benchmarks usually accumulate a lot of costs and therefore, is often harder to pinpoint the actual cause of problems in performance detected using these benchmarks, compared to specialized micro-benchmarks (Saavedra-Barrera, Gaines, & Carlton, 1993).

A summary of micro- and macro-benchmarks is shown in Table 1.

---

<sup>9</sup> The SPECjbb®2013 benchmark has been developed from the ground up to measure performance based on the latest Java application features, <https://www.spec.org/jbb2013/>

<b>Micro-Benchmark</b>	<b>Macro-Benchmark</b>
Also called Synthetic Benchmark	Also called Natural or Complex
Compares basic concepts, such as a single operation, or small aspects of a larger system	Compares real world tasks
Typical examples: comparison of different algorithms, such as sorting algorithms or performance of an operation on different hardware platforms	Typical examples: comparison of large and complex systems. Represents real systems.
Uses white-box approach. Programming implementation and knowledge is required	Uses black-box approach. Programming implementation and knowledge is not required
Easier to automate	Difficult to find suitable test cases
Example of benchmark: Benchmark.js (timer benchmark)	Example of benchmark: SPECjbb®2013 <sup>10</sup> application benchmark.

**Table 1 Summary of Classifications of Benchmarks**

The next section define the different approaches to benchmarking as has been used in the past.

### ***2.3 Approaches to Benchmarking***

This section provides a description of some common benchmarking approaches to different areas of computer science. These include Processor, Systems and

---

<sup>10</sup> The SPECjbb®2013 benchmark has been developed from the ground up to measure performance based on the latest Java application features, <https://www.spec.org/jbb2013/>

Programming Language benchmarking and will also include various examples of implementations in these areas.

There are many benchmark tests available at present time that are used to evaluate the performances of a wide variety of systems and components under different types of application workloads. Online repositories of research articles as well as blogs in the web is a rich source of up-to-date information regarding benchmarks. However, in order for a benchmark to be useful, it should pass the following attributes as stated by (Gray, 1993).

- **Relevance:** The benchmark should provide meaningful performance measures within a specific problem domain.
- **Understandable:** The benchmark results should be simple and easy to interpret and understand.
- **Scalable:** The benchmark tests must be applicable to a wide range of systems, in terms of cost, performance, and configuration.
- **Acceptable:** The benchmarks should present unbiased results that are recognized by users and vendors.

In view of this, as mentioned earlier, there are a number of standardized benchmarks that are available in relation to measuring system and component performance. The next few sections will describe the some of the most common types of benchmarks available today. These include System Benchmarks, Processor Benchmarks, Database Benchmarks and Programming Language Benchmarks.

### 2.3.1 Processor Benchmarks

SPEC CPU benchmark is specifically designed to provide a standard way of measuring the performance of compute-intensive workloads that are run on different system environments. SPEC CPU benchmarks are denoted as SPECxxxx, where xxxx specifies the version of the benchmark. SPEC2006<sup>11</sup> contains two suites of benchmarks called SPECint 2006<sup>12</sup> and SPECfp 2006<sup>13</sup>. The former is designed for measuring and comparing the performance of compute-intensive integer while the latter focuses on

---

<sup>11</sup> SPEC2006, <https://www.spec.org/cpu2006/>

<sup>12</sup> SPECint 2006, <https://www.spec.org/cpu2006/CINT2006/>

<sup>13</sup> SPECfp, <https://www.spec.org/cpu2006/CFP2006/>

the performance of floating point variables. Because these benchmarks are compute-intensive, they concentrate on the performance of the computer's processor, the memory architecture, and the compiler (Packirisamy, Zhai, & Yew, 2008).

### **2.3.2 Systems Benchmarks**

Systems benchmarks is another type of benchmark that measures the entire system consisting of a number of components. As mentioned in section 2.1.1, TPC defines a set of database benchmarks. TPC assesses performance of applications in relation to database transactions such as banking transactions, airline reservations (services), and inventory control (goods). TPC also measures system components such as the processor, the I/O subsystem, the network, the compilers, and the operating system. TPC maintains and run a total of six benchmarks within its Enterprise Benchmark suite and three benchmarks within its Express Benchmark suite. Examples of such benchmarks include TPC-C, TPC-DI, TPC-E and TPC-H. The purpose of TPC-C is to measure the number of transactions executed against a database. TPC-DI, which is also known as Data Integration (DI) analysis, combines and transforms data from a variety of sources and combines them into a single model representation. TPC-E benchmark measures the workload of On-Line Transaction Processing (OLTP) while TPC-H is a decision support benchmark that measures and examine large volumes of data to provide answers to critical business questions. This is measured by Composite Query-per-Hour (QphH@Size)<sup>14</sup>.

### **2.3.3 Programming Language Benchmarks**

Programming languages are constantly evolving as more languages are being developed, all serving different purposes. For example, FORTRAN and C are compiled languages and were developed mainly for compute intensive applications and system software which require both high performance. On the other hand, interpreted languages such as Ruby, Python and Perl are typically used for small tasks such as daily text parsing and web applications that do not require high performance but instead, require high productivity. Object-Oriented programming languages such as JAVA, C# and JavaScript have also been developed, with each language having slightly different implementations and use of syntaxes.

---

<sup>14</sup> TPC Benchmarks, <http://www.tpc.org/information/benchmarks.asp>

With regards to benchmarking programming languages, a number of synthetic benchmarks have been developed in an attempt to compare programming languages. Whetstone is an example of such benchmark developed in the 1970s that was developed in ALGOL which were first used in NPL Oxford University in order to create scientific programs. In addition, it was also designed to be compatible with other programming languages (Curnow & Wichmann, 1976). Curnow also implemented a FORTRAN version of the Whetstone benchmark. Furthermore, Whetstone is based on scientific programs however, the operations performed in the benchmark by floating-point is regarded as meaningless as scientific calculations.

Another example of a synthetic benchmark developed decades ago when benchmarking first started is the Dhrystone benchmark. This was also regarded as a synthetic benchmark that was based on a number of collected data from programs written in FORTRAN, Pascal, ALGOL 68, Ada and C. However, the original version of was written in C and Pascal (Weicker, 1984).

Nevertheless, these small benchmarks mentioned which include Whetstone and Dhrystone have become obsolete as modern CPUs became more advanced and got bigger cache which in turn made CPUs faster that made benchmark unreliable. Moreover, languages that use bytecode interpreter and dynamically type languages such as Java and JavaScript have emerged which utilized Virtual Machines (VM), Just-in-Time compilation (JIT) and garbage collection (GC). Therefore, these programming techniques and concepts have changed the way information is collected which have made old benchmarks such as Whetstone and Dhrystone unreliable anymore.

The Computer Language Benchmarks Game<sup>15</sup> is an online benchmark that compares and evaluates measurements of programs that are written in different programming languages. The number of languages contained in this benchmark enumerates to 27 languages with 13 benchmarks within each language. By far, this is the largest benchmark suite that is available to the public and have been implemented in a variety of programming languages such as Ada, C, Chapel, C++, C#, Java, Python and many more. This benchmark consists of two parts where the first part is an algorithmic benchmark consisting of what is called N-body physical simulation, Mandelbrot set calculation, puzzle game solver, pi digits calculate, permutations and bioinformatics algorithms. The other part are measurements regarding performance of a number of

---

<sup>15</sup> The Computer Language Benchmarks Game, <https://benchmarksgame.alioth.debian.org/>

basic operations such as threading, vector manipulation and memory usage management. However, a number of problems have been found in these benchmarks such as the benchmarks scale being small where implementations are less than 200 lines developed in Java in comparison to Dhrystone implementation of about 300 lines in Java. Also, the diversity of the applications are considered biased as these applications are all contributed by volunteers. Therefore, some of the implementations on a number of languages are incomplete and not optimized. These implementations are submitted by volunteers and each implementation is given a score where the implementation with the best score is then chosen. This means that while an implementation may be simple, there will be a trade-off between the simplicity, quality and performance of the implementation.

In previous sections, the theory and history behind benchmarking was discussed which gave an insight into how benchmarking was done before. The next section explores the definitions of frameworks along with a state of the art review of the current approaches to JavaScript framework comparison which is the main focus of this project.

## **2.4 Frameworks**

This section gives an overview of frameworks. This is followed by a state of the art review of comparing JavaScript frameworks and approaches done by various researchers to comparing these frameworks.

In literature, there have been many definitions of a software framework. (Johnson, 1997) defines a framework as a reusable design which are represented by a set of classes that are abstract. Also, he calls a framework a sort of a skeleton application in which developers are able to customize this application in whatever way they like, to suit their needs. Moreover, frameworks can also be described as a ‘semi-complete’ application. Therefore, it can be said that frameworks are purposed to allow developers to solve problems that are within the bounds or domain of the framework being used (Schmidt & Buschmann, 2003). Reinventing the wheel is definitely not recommended in any areas of software and therefore, by modularizing code, code can be reused by developers which enhances the effectiveness and capability to develop higher quality software products (J.D, Farre, Bansode, Barber, & Rea, 2011).

The next section provides a brief review of the main approaches of solving the JavaScript framework problem.



#### **2.4.1 JavaScript Frameworks Comparison Research**

A paper by Gizas, Christodoulou, and Papatheodorou attempted to evaluate the most popular JavaScript Framework (JSF) in terms of its quality and performance by taking into account some factors such as software quality and performing tests to further evaluate the selected JSFs. However, tests were only performed on JavaScript libraries and some have been discontinued. Nevertheless, the paper have recommended the use of well-known software metrics to analyze JSFs (Gizas, Christodoulou, & Papatheodorou, 2012).

Graziotin and Abrahamsson discusses in their paper that there is little research available that helps practitioners to choose the most suitable JSF to a given situation. Therefore, a research design was proposed as a way towards a comparative analysis of JSFs (Graziotin & Abrahamsson, 2013).

Ocariza Jr, Pattabiraman, and Mesbah talks about the demand for a more reliable and maintainable JavaScript-based web application and further explores the development of JavaScript MVC (Model-View-Controller) frameworks in their paper. However, they further state that there exists inconsistencies in MVC frameworks and thus, there is a need to find a way of detecting these. Therefore, a tool was proposed which automatically detects these inconsistencies in web applications (Ocariza Jr, Pattabiraman, & Mesbah, 2015).

A 2010 paper by Ratanaworabhan, Livshits, and Zorn discusses how there is surprisingly, few research papers that measures specific aspects of JavaScript considering how widely used it is. The workload of JavaScript running on different browsers were tested where a benchmark was produced as a result (Ratanaworabhan, Livshits, & Zorn, 2010). They also discuss in their paper the common behaviours that are not well emphasized in most benchmarks such as event-driven execution, instruction mix similarity, cold-code dominance, and the prevalence of short functions. The current approaches to solving the JavaScript Frameworks (JSF) varies from the papers read. Gizas, Christodoulou, and Papatheodorou suggests a way to test JSFs such as the use of quality, performance and validation test tools, each with its own metrics to evaluate JSFs. Furthermore, they also recommended using the same tools on mobile environments (Gizas, Christodoulou, & Papatheodorou, 2012).

Joorabchi, Mesbah, and Kruchten provided a way of gaining an understanding of the main challenges that developers face when developing mobile applications. The use of web-based or hybrid mobile app development frameworks were proposed to support developers with their technology selection process (Joorabchi, Mesbah, & Kruchten, 2013).

Graziotin and Abrahamsson proposed a model framework for comparing JavaScript frameworks which consists of two layers, one associated with research and the other associated with practitioners. This new framework was created as an extension of the framework developed by Gizas, Christodoulou, and Papatheodorou to further reinforce the appropriate analysis of JSF (Graziotin & Abrahamsson, 2013). Such metric that Graziotin and Abrahamsson added to the framework is the Community and Documentation metrics as shown in Figure 2.



**Figure 2 Illustration of a Comparison Framework.**

Note: Retrieved from *"Making Sense Out of a Jungle of JavaScript Frameworks"* (p. 337), by Graziotin & Abrahamsson, 2013, Springer Berlin Heidelberg.

McCabe, Weyuker and Coleman, Ash, Lowther and Oman provides a formal description of software quality metrics and how software maintainability analysis can be further used as guide to software-related decisions such as programming languages, frameworks and systems (McCabe, 1976; Weyuker, 1988; Coleman, Ash, Lowther, & Oman, 1994).

In previous sections the purpose and importance of benchmarking have been discussed which gave an insight into the history of benchmarking as well as its current state. As the primary focus of this thesis is to compare and evaluate JavaScript frameworks, thus, the next section will describe the JavaScript language as well as an exploration of the features of various JavaScript frameworks available today to determine the metrics that will be chosen for the comparison.

## ***2.5 JavaScript and JavaScript Frameworks***

This section talks about the concept of JavaScript as well as the main features of three JavaScript frameworks chosen. This is followed by some concepts regarding how JavaScript is executed in the browser, how data bindings work and the use of states.

JavaScript is an object-oriented language which was designed back in 1995 by Brendan Eich at Netscape to allow people with no programming background to extend web sites using client-side executable code (Richards, Lebresne, Burg, & Vitek, 2010). By definition, JavaScript is an interpreted programming language with the capability to use Object-Oriented (OO) principles. Syntactically, the core of JavaScript language can be seen to resemble well established programming languages such as C, C++ and Java. However, what distinguishes JavaScript from other languages is that JavaScript is a loosely typed language which means that variables, defined by the keyword `var` in JavaScript, do not need to have a specific type specified such as Integers and Strings. In addition, objects in JavaScript are similar to associative arrays (key-value pairs) where each property name (key) within the object are mapped to their corresponding arbitrary property values (value). Also, unlike the languages mentioned, it does not encourage the use of an OO principle called encapsulation or even the use of structured programming. JSON (JavaScript Object Notation) is an example of an associative array that utilizes key-value pairs. This allows JavaScript to be flexible in nature. Moreover, the OO inheritance mechanism of JavaScript is prototype-based where all JavaScript objects inherit their properties and methods from their prototype (Flanagan, 2006). Furthermore, the JavaScript language is also becoming more prominent in the world of software with the increase of JavaScript frameworks and libraries that embrace the MVC concept. Such frameworks developed include AngularJS, ReactJS, BackboneJS, EmberJS, jQuery and many more. Nevertheless, JavaScript implementations and frameworks are often compared using benchmarks (Richards, Lebresne, Burg, & Vitek, 2010). For this reason, three JavaScript frameworks were chosen for this project due to the fact that limited time is available to evaluate more frameworks. Therefore, the nominated frameworks are AngularJS, BackboneJS and React. These were selected according to its community ranging from small to large developer communities as well as their popularity which hopes to give a well-balanced

view of JavaScript frameworks. Finally, these JavaScript frameworks are further discussed in the next few sections.

### 2.5.1 AngularJS

AngularJS<sup>16</sup> was developed by Google Inc. with its first release in 2010. The motivation in the creation of AngularJS is to extend the HTML vocabulary with the use of data bindings through directives. At a high level, directives can be thought of as markers on a DOM (Document Object Model) element such as attributes, element name, comment or CSS class that tell the HTML compiler (\$compile) within AngularJS to attach a specific behaviour to that DOM element (via event listeners) or transform specific DOM elements and propagate these changes to its children. This is implemented with the use of *ng-tags*, which bind the view to one or many models. In addition, the concept of dirty checking is used to check data-binding in AngularJS. This means that if a data value is bound to the view through a model, it is not immediately updated. Instead, it is updated when dirty checking within AngularJS is executed on the value. This dirty checking is performed asynchronously. In addition, AngularJS includes tests suites which makes it easier to test individual components through the use of built-in dependency injection. Finally, AngularJS is implemented using the MVC (Model-View-Controller) architecture concept where parts of an app is isolated such that the application logic is isolated from the user interface. An example of applications built with AngularJS is Netflix and YouTube for Playstation.

### 2.5.2 React

React<sup>17</sup> is a JavaScript framework developed by Facebook Inc. and was initially released in 2013. React is simply, the “V” of the MVC architecture. It makes use of a Virtual DOM which is used for efficient re-rendering of the DOM. Essentially, React makes use of components and each component have a state. React uses this state to monitor when a piece of data within the component is changed. The concept of states are explained further in section 2.6.6. Contrary to AngularJS, data bindings in React are implemented using an algorithm which Facebook calls the **diffing algorithm**.

---

<sup>16</sup> AngularJS documentation, <https://angularjs.org/>

<sup>17</sup> React, <https://facebook.github.io/react/>

According to Reacts documentation, when a component's props or state change, React decides whether an actual DOM update is necessary by comparing the newly returned element with the previously rendered one. When they are not equal, React will update the real DOM. This algorithm causes a full re-render of the application every time a state being monitored changes. Examples of applications built with React includes Instagram and Yahoo Mail.

### 2.5.3 BackboneJS

According to the online documentation, Backbone.js<sup>18</sup> is a JavaScript library that gives structure to web applications and is based on the model-view-presenter (MVP) application design paradigm. Backbone makes used of models with key-value bindings and custom events. Models can either be created, validated, destroyed or saved to the server. Whenever a UI action causes an attribute of a model to change, the model triggers a “change” event. In other words, models manages an internal table of data attributes and triggers these change events when a piece of data within it is changed (see Figure 3). **Collections** helps to handle a group of related models by providing helper functions to perform aggregations or computations against a list of models. Aside from their own events, collections also allows for listening to change events that occur to any models within the collection in one place by proxying through all of the events that occur in the models (see Figure 4). Finally, **views** are used to wait and listen for any changes from the user input which then renders the UI (User Interface) accordingly.

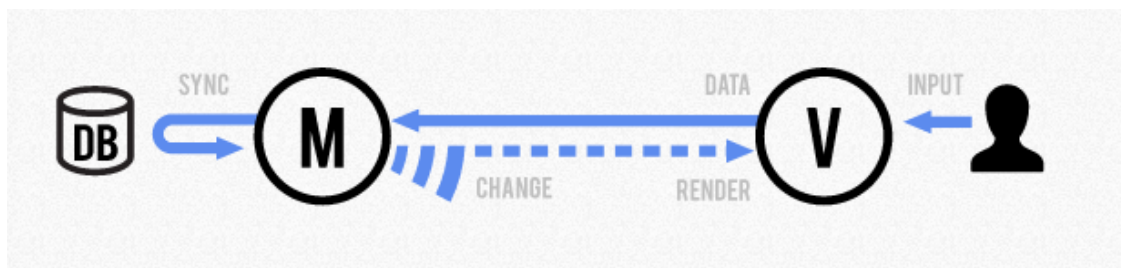
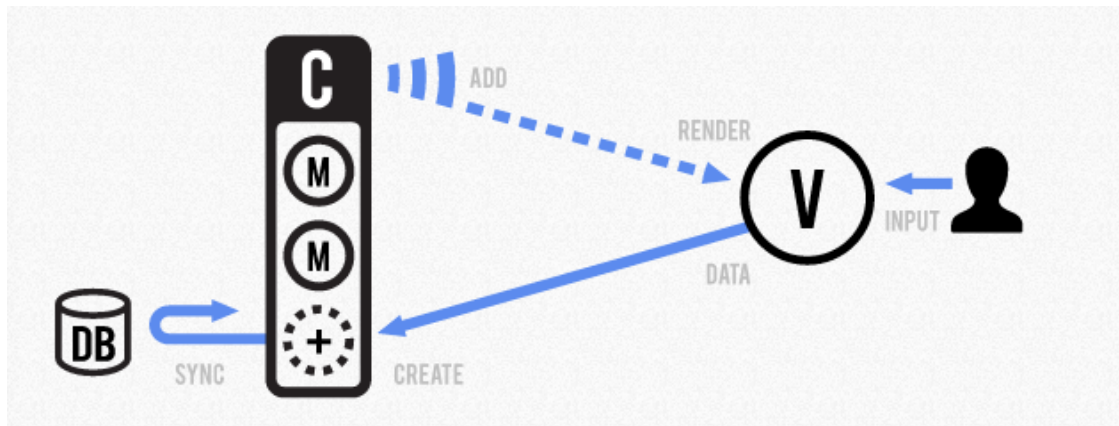


Figure 3 Models and Views concept in Backbone

<sup>18</sup> Backbone.js, <http://backbonejs.org/#Getting-started>



**Figure 4 Collections in Backbone**

#### 2.5.4 Execution of JavaScript

JavaScript is typically executed in the browser and depending on the web browser being used, each browser interprets JavaScript differently. This is due to the distinct implementations of the JavaScript engine between web browsers. Mozilla FireFox uses its own JavaScript engine called SpiderMonkey<sup>19</sup> and Google Chrome uses its own engine known as Chrome V8<sup>20</sup> written in C++. On the other hand, Apple uses JavaScriptCore<sup>21</sup> in their WebKit<sup>22</sup> browser engine and is used within Safari and their App store. There are many browser engines from some of the most well-known web browsers available, each with distinct features from one another. Therefore, it is important to test each JavaScript framework in as many browsers as possible.

#### 2.5.5 Data Bindings Explained

There are two types of data bindings which include one-way data binding and two-way data binding. The one-way binding is typically used in many traditional server-side web applications where a template and one or many data models are merged onto the server and sent back to the user's view via the web browser. Therefore, any changes that are made to the views or models are not reflected back to the user after the merge onto the web server. Therefore, in order to update the model, it is necessary for the

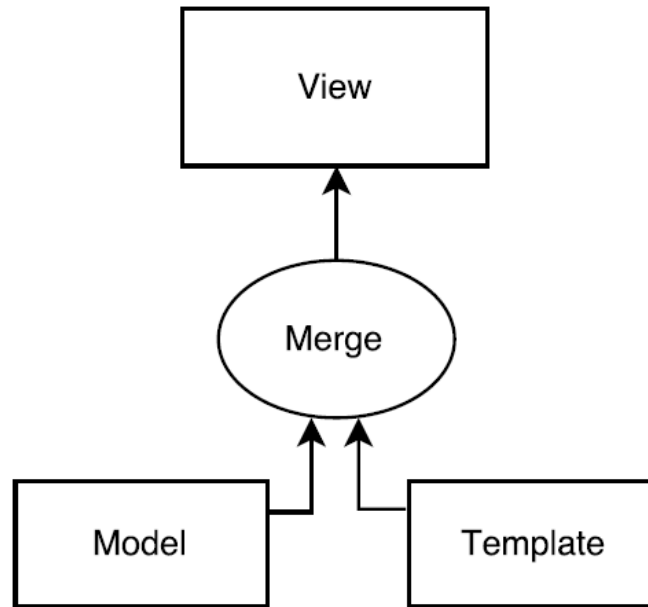
<sup>19</sup> SpiderMoney – Mozilla, <https://developer.mozilla.org/en-US/docs/Mozilla/Projects/SpiderMonkey>

<sup>20</sup> Chrome V8, <https://developers.google.com/v8/>

<sup>21</sup> JavaScript Core - Apple, <https://developer.apple.com/reference/javascriptcore>

<sup>22</sup> WebKit, <https://webkit.org/>

user to re-send the view back to the web server. These changes are then processed by the server and sent back as a new merge of the template and the models with the reflected change<sup>23</sup>.



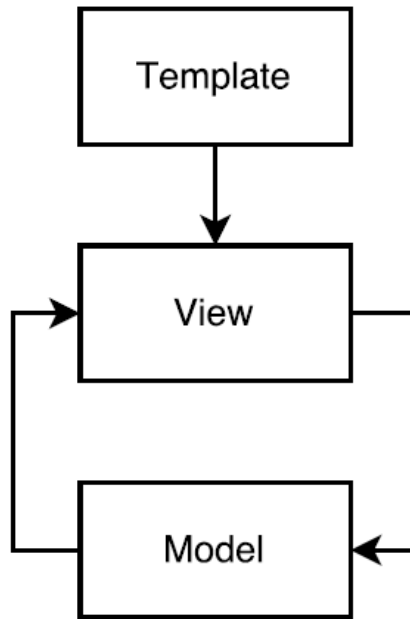
**Figure 5 One-way data binding**

On the other hand, the most common data binding concept in many JavaScript Frameworks today is the concept of two-way data binding where the view can be seen as a “single-source-of-truth” of the data models. A diagram of two-way data binding is shown in Figure 6. This means that all changes done by the user is instantly reflected onto the model and vice versa, all the changes to the model are propagated to the view<sup>6</sup>. However, since two-way data binding is bi-directional, the application might behave in a different way and finding out the cause behind this may be difficult. In a recent statement from the developer team at Facebook Inc. they said that “We found that two-way data bindings led to cascading updates, where changing one object led to another object changing, which could also trigger more updates.”<sup>24</sup> This led to some major developer problems with their Facebook Messenger application. To work their way around this, Facebook developed a new type of data binding called Flux where a

<sup>23</sup> AngularJS: Developer Guide to Data Binding, <https://docs.angularjs.org/guide/databinding>

<sup>24</sup> Flux, the Application Architecture for Building User Interfaces, <https://facebook.github.io/flux/docs/overview.html>

one-way data binding was used with four components instead of three as shown in Figure 5.



**Figure 6 Two-way data binding**

### 2.5.6 States Explained

State is the place where data comes from and have always existed in web applications. In order to increase the user interactivity, more states are required. In server-side rendering or one-way data binding, it is a cumbersome task to implement smaller changes in the components. These small changes need to retrieve the merge of the template and the model from the server before the user's view is changed. Typically, front-end applications implemented using JavaScript frameworks have more complex states than traditional server-side applications. Below lists these points<sup>25</sup>:

- Some DOM events that cause changes in state in the views by specific events such as using forms where the fields within the forms are validated and response is given back to the user.
- The state of an application can change depending on the interaction of users such as interacting with buttons that causes a new page to show up.

<sup>25</sup>

Modern web applications: an overview, <http://singlepageappbook.com/goal.html>

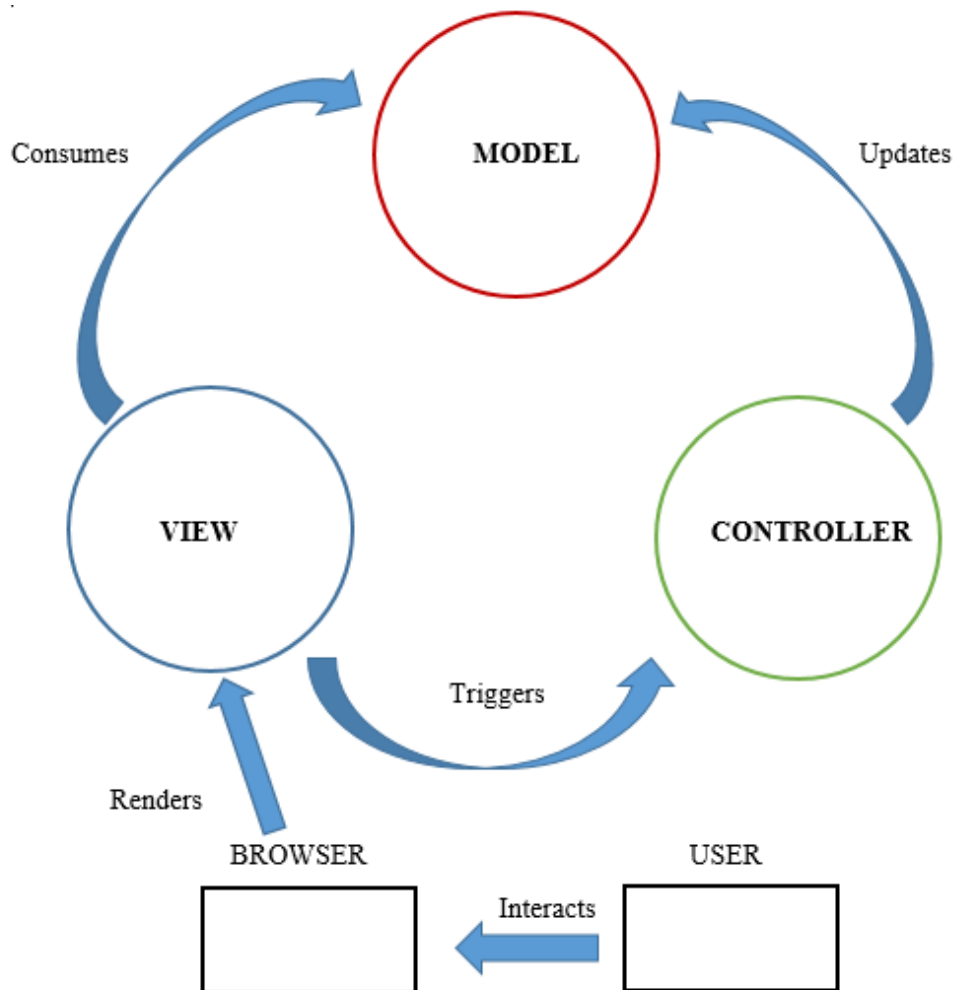


- Global state changes can also occur such as going offline in a real time application.
- Delayed data and results can happen as a result of various calls to the API where AJAX call are delayed between the application and the server.
- Data model changes as a result of a change in the data model and an update is sent to the client.

The next section will describe the MVC pattern and the components that make up the pattern as all frameworks chosen for this project is based around components of the MVC pattern.

## 2.6 Model-View-Controller (MVC)

In this section, the fundamentals of the Model-View-Controller pattern is described.



**Figure 7 Model-View-Controller**

The Model-View-Controller (See Figure 7) pattern is an architectural style or design pattern that is most commonly used by developers to separate application concerns, in that, all business logic code (Controller) is separated from the presentation (View) and access of data (Model). The MVC pattern was first introduced by Trygve Reenskaug in a programming language called Smalltalk-76, while he was visiting the Xerox Palo Alto Research Center (PARC) in the 1970s (Burbeck, 1992). Since then, the MVC pattern have vastly evolved as numerous adaptations have been seen by many which gave rise to different variations of the MVC pattern such as the model-view-presenter (MVP), model-view-viewmodel (MVVP) and model-view-adapter (MVA) patterns. Moreover, the use of the MVC pattern in web applications became so popular that

frameworks were developed based on the pattern such as Spring<sup>26</sup> framework for Java, Ruby on Rails<sup>27</sup>, Django<sup>28</sup> for Python and many of today's JavaScript frameworks such as AngularJS, ReactJS and BackboneJS. The components that make up the MVC pattern will now be discussed.

### 2.6.1 Model

Firstly, the Model is discussed. (Reenskaug, 1979) describes the Model as a representation of knowledge. In other words, it represents the permanent storage of data that is to be used in the overall design. A Model must allow access for the data to be viewed, collected and updated. The Model is technically 'blind', in that, it has no perceived knowledge of what happens to its data when used by either the View or Controller. The only purpose of the Model is to process data into its permanent storage or seek and prepare data to be used by other components and can be described as a passive component of the MVC pattern. The storage within the model may not necessarily be a database. Storage can also be in formats such as hard-coded variables and files. The Model is an integral part of the MVC pattern as without it, there would be no connection between the View and Controller.

### 2.6.2 View

The View handles the presentation of data. Data used by the View is collected via a request from the Model. The View can be seen as the starting point of interaction from the user through a web browser. Every interaction from the user triggers an action in the Controller, for example, when a button is clicked, an action is triggered in the Controller that processes the event. There is a misconception of the View as developers believe that there is an interaction between the View and Model as well as a bi-directional relationship between the Controller and View. However, this is false notion as the correct implementation of the MVC pattern disregards interaction between Models and Views as all logic is handled by the Controllers. On the other

---

<sup>26</sup> Spring Framework, <https://spring.io/>

<sup>27</sup> Ruby on Rails, <http://rubyonrails.org/>

<sup>28</sup> Django, <https://www.djangoproject.com/>

hand, there is no bi-directional relationship between the Controller and View as only the View interacts with the Controller and the Controller interacts with the View by updating the Model (Reenskaug, 1979).

### **2.6.3 Controller**

The Controller is also known as the manager and can be viewed as the brain of the MVC design pattern. The Controller handles all logic and data that user inputs/submits which in turn updates the model accordingly. It is the Controller component that end-users should be interacting with, through the View. Each function within the Controller can be viewed as a trigger function that is applied when users starts interacting with the View (Reenskaug, 1979).

The next section discusses the selection of the benchmark metrics that will be used for the experiment.

## ***2.7 Overview of Benchmarking Metrics***

For this reason, a number of benchmarking metrics are discussed in this section which are deemed to be important. However, before the actual selection of metrics are discussed, first, an overview of the purpose of benchmarking metrics is discussed. When comparing something complex, specifically the performance of JavaScript frameworks, it is vital to come up with a list of criteria that would be used to best compare these frameworks. Based on a number of papers read, (Molin, 2016) has come up with a conceptual framework describing the criteria needed to compare these frameworks. These criteria were formulated based on his own research and also by conducting various interviews to a number of professionals such as developers and fellow researchers. These include criteria such as Documentation, Popularity, Portability, Reliability, Maintainability, Reliability, Modularity, Persistence, Testability and Performance. However, since the scope of this research is focused primarily on comparing JavaScript frameworks performance by benchmarking, therefore, the focus will solely be on performance. As a refresher, performance in this context refers to values returned when benchmarking metrics are assessed.

(Molin, 2016) also states that it is naturally important to identify a number of general metrics that are useful for the comparison of JavaScript Frameworks. Therefore, his work focused more on the formulation of a set of criteria that can be referred to by other researchers wanting to create their own set of criteria. His method of defining these set of criteria included a formulation of various questions that were asked if the metrics is suitable for the actual scope of the study. These were based on papers by (Lennon, 2010), (Malmstrom, 2014) and (Salas-Zárate et al., 2015).

In the next section, the actual metrics selected are discussed which is believed to be appropriate for benchmarking JavaScript frameworks.

### 2.7.1 Selection of Metrics

This section enumerates the various benchmarking metrics chosen for comparing JavaScript frameworks. The discussion selection of metrics are split into two parts where the first discusses about the software complexity metrics whereas the latter talks about the selection of computer benchmark metrics. Therefore, a combination of metrics relating to web performance was sought out from various papers and online resources read. Firstly, the following metrics discuss metrics relating to software complexity.

**Lines of Code (LOC)** is the oldest metric for software projects. This metric was first introduced around 1960 and was first used in economics, quality studies and productivity and was quite effective for all three purposes. In the early years of programming languages, when assembly languages were still being used, the idea of lines of code was fairly simple. However, as new programming languages emerged (example, C language) came about at the time, the idea of lines of code became more complex as programming languages required a more structured flow. As a result of this sudden change, the IEEE<sup>29</sup> has standardized the use of lines of code (LOC) with the standardization of two counting methods: Physical Lines of Code (SLOC) and Logical Lines of Code (LLOC). Physical SLOC is the actual count of the number of lines in the source code excluding the comment lines whereas Logical SLOC measures the number of executable statements within the code (Park, 1992).

---

<sup>29</sup> The Institute of Electrical and Electronics Engineers, <https://www.ieee.org/index.html>

Another important metric is the **cyclomatic complexity** metric and this was developed by Thomas J. McCabe in 1976 as discussed in McCabe's paper (McCabe, 1976). This metric has been well established as it has been around for many decades. Cyclomatic complexity is the measure of the number of linearly independent paths through the source code of a program. (McCabe, 1976) proposed an upper limit to this metric as a value higher than the value 10 would indicate less manageable modules. Therefore, the lower the cyclomatic complexity, the better. Furthermore, (McCabe, 1976) suggested that if the cyclomatic complexity exceeds the value 10, then programmers should split a software module into smaller parts. There are tools available to calculate the cyclomatic complexity metric such as the JScomplexity<sup>30</sup> which is a software complexity analysis tool for JavaScript.

**Halstead complexity** measures is another software metric that is considered in regards to benchmarking (Halstead, 1977). This metric was first introduced by Maurice Howard Halstead in 1977 and are based on a number of values which include: the number of distinct operator ( $n1$ ), the number of distinct operands ( $n2$ ), the total number of operators ( $N1$ ) and the total number of operands ( $N2$ ). From these values, several measures can be calculated using the following formulas:

$$1) \text{ Volume (V)} = (N1 + N2) \times \log_2(n1 + n2)$$

$$2) \text{ Difficulty (D)} = \frac{n1}{2} \times \frac{N2}{n2}$$

$$3) \text{ Effort (E)} = D \times V$$

$$4) \text{ Time (T)} = \frac{E}{18} \text{ seconds}$$

$$5) \text{ Bugs (B)} = \frac{\frac{E}{18}}{3000}$$

Fortunately, the Halstead complexity measures can also be calculated using the tool provided by JSComplexity.

**Maintainability Index** is another metric and it was designed by Paul Oman and Jack Hagemester in 1991 which measures how maintainable or easy it is to support and

---

<sup>30</sup> JSComplexity, <https://github.com/slyg/jscomplexity>, <https://github.com/escomplex/complexity-report>

change the source code. The maintainability index is calculated as a factored formula consisting of Source Lines of Code (SLOC), Cyclomatic Complexity and Halstead Volume (Oman & Hagemeister, 1992). Its values range from  $-\infty$  to +171 on a logarithmic scale. A measured value of 65 is considered difficult to maintain as proposed by Oman and Hagemeister and therefore, higher maintainability index value is considered good and easier to maintain. The formula for the maintainability index is as follows:

$$MI = \frac{171 - 5.2 \times \ln(\text{Lines of Code}) - 0.23 \times (\text{Cyclomatic Complexity}) - 16.2 \times \ln(\text{Halstead Volume})}{171} \times 100$$

(Calero, Piattini, & Genero, 2001) proposes metrics regarding access to databases such time it takes to perform **CRUD** (Create, Read, Update and Delete) operations as many applications make extensive use of databases and therefore, measuring performance in accessing databases is important. (Palmer, 2002) talks about **render-time** as a metric in his paper. It is the time elapsed from the request to when the user sees the actual website content appear on the page. This metric is important as no user likes staring at a blank page while waiting for the web page to render in the background. (Christodoulou & Gizas, 2014) states that 37% of consumers will shop elsewhere if a mobile site or app fails to load in 3 seconds. Finally, since most of the operations that each JavaScript framework take place on the client side. It would be interesting to see how the results turns out.

Therefore, Google<sup>31</sup> introduces page-level metrics which consists of top-level measurements that are captured and displayed on the webpagetest.org tool. One important metric within this list of page-level metrics is the **speed index** which measures the average time at which parts of a web page becomes visible to the user's view. It is expressed in milliseconds and is dependent on the size of the view port. It is especially useful for comparing user's experience of pages against each other (before and after optimizing) and should therefore be used in combination with other metrics such as **load time** and **render time** to better understand a website's performance.

---

<sup>31</sup> Google Developers, <https://developers.google.com/web/>

(Stepniak & Nowak, 2016) analyzed their web system by incorporating metrics from the Google Timeline Event Reference<sup>32</sup>. Research by Google into event metrics were deemed important in the measurement of web performance. Particularly, the interest is on **render and frame measurements** which have been found to be useful for developers to measure in their applications. Table 2 summarizes the metrics selected for the comparison of JavaScript frameworks. The next section explores the definitions of frameworks along with a state of the art review of the current approaches to JavaScript framework comparison which is the main focus of this project.

Metric	Source
Lines of Code	(Park, 1992)
Cyclomatic Complexity	(McCabe, 1976)
Halstead Complexity	(Halstead, 1977)
Maintainability Index	(Oman & Hagemeister, 1992)
Database metrics	(Calero, Piattini, & Genero, 2001)
Page-load/render-time	(Christodoulou & Gizas, 2014), ("Metrics - WebPagetest Documentation," 2008)
Speed index	("Metrics - WebPagetest Documentation," 2008)
Render and frame measurements	("Timeline Event Reference   Web," n.d.)

**Table 2 Overview of selected metrics**

## 2.8 Chapter Summary

This chapter gave an overview and purpose of benchmarking. Different types of benchmarking were also described which shows the different variations of

<sup>32</sup> the Google Timeline Event Reference, [https://developers.google.com/web/tools/chrome-devtools/evaluate-performance/performance-reference#common\\_timeline\\_event\\_properties](https://developers.google.com/web/tools/chrome-devtools/evaluate-performance/performance-reference#common_timeline_event_properties)



benchmarking available today. Moreover, the state of play for JavaScript framework comparison was discussed which talked about the various approaches implemented by researchers in an attempt to compare different JavaScript frameworks as well as the discussion of the main features of AngularJS, React and BackboneJS, all of which are increasing in popularity and garnering interest in the development of web applications. Furthermore, this chapter also gave an overview of the fundamentals of the Model-View-Controller design pattern which had since become widely accepted by the developer community since its initial development. The evolution of the MVC pattern had seen a wide spread of modified versions of the MVC design pattern with the emergence of various frameworks as enumerated in section 2.6. Finally, the selection of metrics were discussed in section 2.7 which were deemed to be the most important metrics for benchmarking.

Therefore, this research seeks to go one step further and evaluate a number of JavaScript frameworks and measure their performance by considering some important metrics provided by numerous researchers as the basis for the comparison.

The next chapter covers the design of the reference applications used in the experiment and the design of the experiment itself as well as the metrics decided upon for the comparison and analysis of JavaScript frameworks.

### 3. DESIGN & METHODOLOGY

This chapter gives an overview and design of the study and reference applications used in the experiment followed by the description of the methodology implemented to fulfil the experiments.

#### 3.1 *Reference Benchmark Application Designs*

This section discusses the design of the reference applications used for the experiments.

##### 3.1.1 **Todo Benchmark Application Design**

The first reference application aims to compare the dominant JavaScript frameworks in terms of its performance. As mentioned before, performance are the values returned for when metrics are assessed. A decision has been made to choose a reference application that is developed using all three JavaScript frameworks. However, in order to do this, it is imperative that one should be careful when implementing the applications as it may lead to biased programming that is typically relative to an expertise of a developer on a particular framework.

Fortunately, there is an open source project developed by founders and lead developers Addy Osmani and Sindre Sorhus called TodoMVC<sup>33</sup> where a generic Todo application has been implemented in almost every JavaScript framework that exists today. For this reason, only one Todo application was considered as the basis of the comparison of performance due to the fact that this application is an open-source project which is constantly being produced, maintained and updated by expert developers in each JavaScript framework community. Therefore, it is safe to assume that the implementations are the best possible implementations for each specific framework. However, other Todo applications may be considered in this case, however, Todo applications developed by individual developers may yield more different results

---

<sup>33</sup> TodoMVC project, <http://todomvc.com/>

compared to a more dedicated project like TodoMVC where development to the application are shared amongst other developers where specific changes are then made that that is optimal for the application.

Figure 8 shows the user interface of the Todo application. Its main functionality is to be able to create, read and update items while in turn, can be marked as completed and deleted from the list. The application contains three main components which include an input field, item list and a footer, each implemented slightly different by each JavaScript framework.

**Input field:** This input field is where items are entered and added to the list. At the bottom of the item list, the item is created as an active item and can be marked as completed or uncompleted via the button left to the item.

**Task list:** This component is where all items are contained in and shown to the user. The individual items may be marked as completed or active depending on the user as shown in Figure 9. Also, items can be edited by double clicking on the item or removed from the list via an (X) button that shows up when the item is hovered by the mouse pointer.

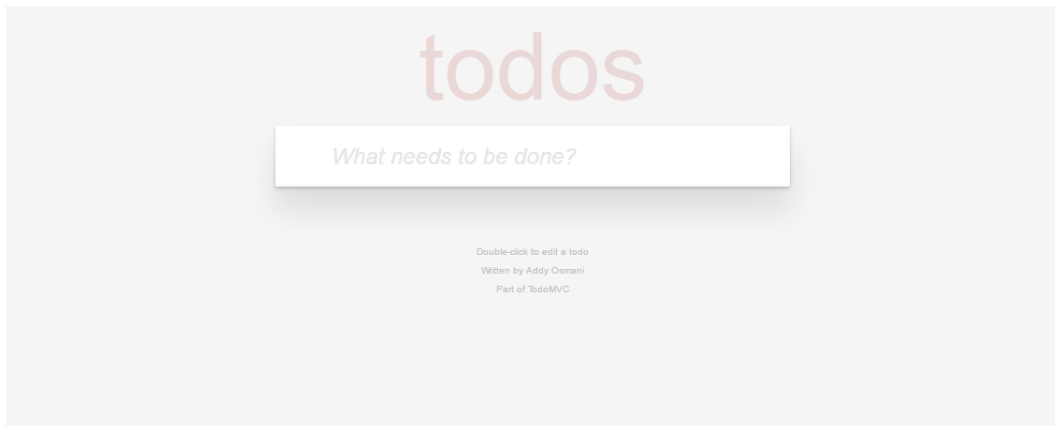
**Footer:** This component is the row at the bottom of the list. This footer component contains four buttons and one counter. The counter shows the number of active items left in the list. The buttons “All”, “Active” and “Completed” are where items are sorted and shows the different views of the items. By default, the “Clear completed” button is not shown in the initial view of the Todo application and is only shown when items are marked as completed. The “Clear completed” button removes all completed items from the list. In addition, the footer is hidden when there are no items in the list as shown in Figure 10.



**Figure 8 User Interface of Todo Application**

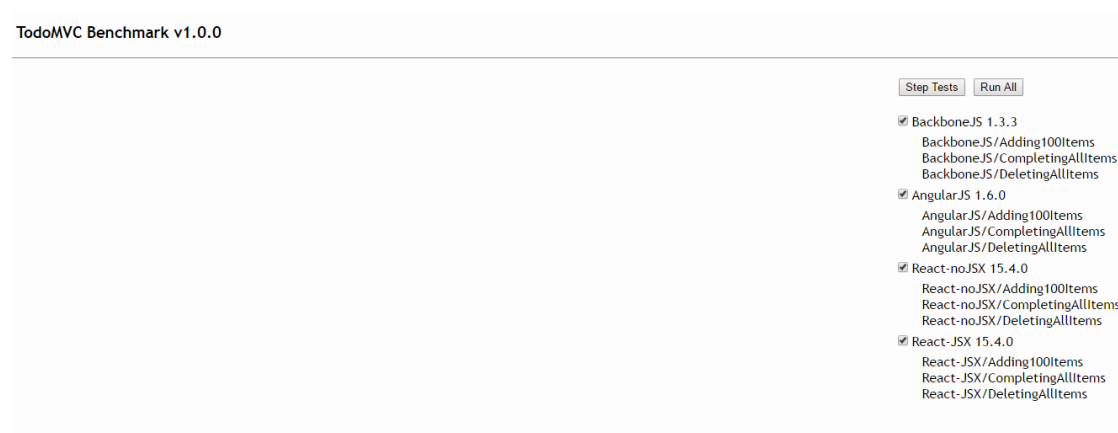


**Figure 9 Todo Application showing completion of an item**



**Figure 10 Default User Interface of Todo Application**

Figure 11 shows the user interface for the benchmark application. On the right hand side of the application shows the JavaScript frameworks that were selected for the benchmark tests. It lists Backbone JS, AngularJS and React. As can be seen there are two versions of the Todo application implementation for React, one that was implemented using the standard React implementation, and another implemented without the use of JSX which is a special feature in React that allows the use of HTML like syntax that extends JavaScript without any defined semantics. However, JSX is not necessary in React and requires a transpiler in order to convert JSX to JavaScript as JSX is not intended to be implemented by browsers or engines. However, JSX makes developing in React much simpler. Moreover, the use of JSX in this case is only intended for comparison purposes which is further described in Chapter 4.



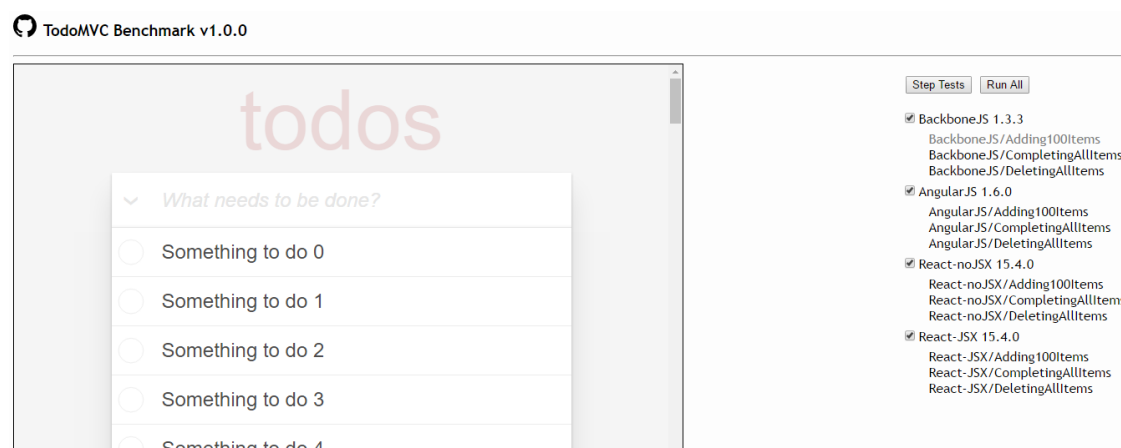
**Figure 11 Benchmark Application User Interface**

### 3.2.1.1 Tests for Todo Application

In order to inspect each application implementations, a set of benchmark tests were performed. There are two parts to this test, one that runs the benchmark application and one that runs a tool called complexity-report<sup>34</sup> which is a command line tool that performs analysis and generates report on the code used. This tool will collect the software complexity metrics as described in section 3.1.1. The implementation will further be discussed in Chapter 4.

The benchmark application is run when the “Run All” button is clicked. There are three tasks to be executed by each tests within the instances of the Todo application which includes adding, completing and deleting items from the list. Each tests are executed in sequence and in each of the 4 instances of the application and within three browsers (Chrome, Firefox and Microsoft Edge). Furthermore, each tests were performed 20 times in order to validate the results and to achieve a more balanced result.

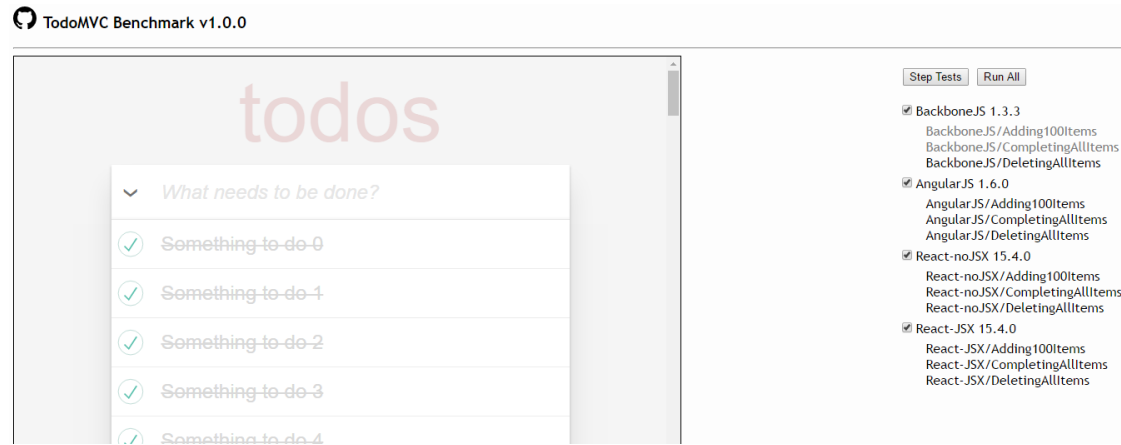
The first task to be executed is “Adding100items”. As the name suggests, starting from a clean slate (starting a new instance of a browser and closing all background applications), this submits one hundred new items to the Todo application and adds to its list as shown in Figure 12. Furthermore, each Todo item is added as “Something to do + (incremental number)” to the list and updates the application’s internal data and DOM.



**Figure 12 Benchmark Application showing "Adding100items" task being executed**

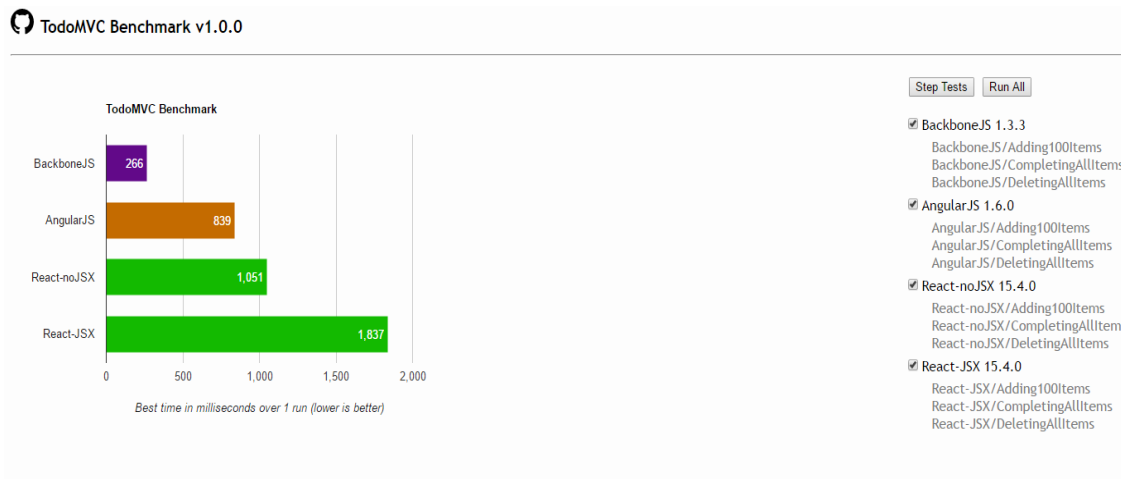
<sup>34</sup> Complexity-report, <https://www.npmjs.com/package/complexity-report>

The next task executed after adding 100 items to the list is the “CompletingAllItems” task. This marks all items in the list as completed and moves on to the next task as shown in Figure 13.



**Figure 13 Benchmark Application showing "CompletingAllItems" task being executed**

Finally, the last task simulates the user activity of clicking on the “Clear Completed” button located at the footer of the Todo application. This results in the application deleting each item within the list one at a time until there are no more items left in the list. The full cycle of deleting the items starts with the benchmark application simulating the clicking of the “Clear Completed” button. This button triggers the instance of the Todo application that is currently running in order to identify each item and remove the items from its internal memory representation (including the in-browser database) and from the DOM (Document Object Model). After the benchmark application has finished executing the test, it moves on to the next set of tests until there are no more tests left to be executed. Finally, the benchmark application also provides a visual representation of the time taken to execute the set of tests for each instance of the Todo application at the end as shown in Figure 14. This sorts the time in ascending order, starting from the fastest to the slowest time. The results are shown in Chapter 4.



**Figure 14 Sample Graph of Results**

### 3.1.2 Database Benchmark Application Design

This reference application is a standard rendering benchmark app that was adapted from DBMonster Core<sup>35</sup> which is a tool used to test an application under performance heavy database load. The DBMonster application was originally developed in JAVA, however, multiple implementations in JavaScript have since been developed and have been used to compare each implementation.

The idea of DBMonster is quite simple. DBMonster is a table-oriented database, meaning that it generates data for tables one by one. In addition, DBMonster can generate data for the following data types:

- Strings – for SQL char, varchar and text.
- Integers – SQL int4 and int8.
- Numbers – SQL arbitrary numeric type and arbitrary precision.
- Booleans – SQL boolean.
- Timestamps – SQL datetime and timestamp.

DBMonster involves rendering a two-dimensional array of fake database monitoring data to demonstrate a framework's 'repaint performance'. DBMonster was originally developed to test EmberJS performance. However, this benchmark aims to test it for different JavaScript frameworks.

<sup>35</sup> DBMonster Core, <http://dbmonster.sourceforge.net/>



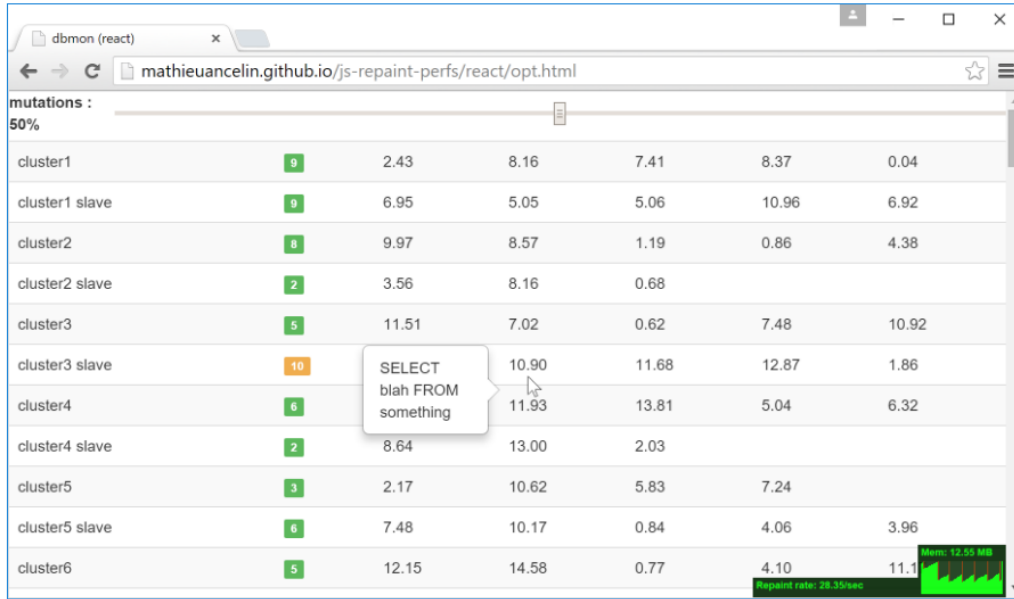
Figure 17 shows the interface for the DBMonster application. It is worth to note the most important features to look at in this application as described below.

- 1) **Smooth scrolling:** users should be able to scroll the page up and down.
- 2) **Popup tracking:** when hovering the mouse over the grid, a popup is triggered that follows and updates.
- 3) **Repaint rate:** At the bottom of the application there is an indicator that shows the repaint rate (measured in frames per second) and memory usage (measured in MB). Repaint rate measures the number of times in a second that a new set of data is being updated or rendered by the DBMonster application. Therefore, the higher the number, the faster the render time and hence, the better.
- 4) **Memory:** To the right of the Repaint rate monitor is the measurement for the memory usage of DBMonster application. This is measured in MB/s (Megabytes per second).
- 5) **Mutations slider:** at the top of each DBMonster implementation is a slider. This controls the amount of the data being processed. As the mutation slider increases, the higher the DOM updates and vice-versa. Therefore, when the mutation slider is at 1% (very low), there should be an increase in the repaint rate as there are less DOM updates being done. On the other, if the repaint rate doesn't change as the mutation rate is decreased, it means the JavaScript framework isn't efficient at tracking changes or identifying when to update the DOM.

In addition, Mathieu Ancelin<sup>36</sup> has put together a website that has aggregated the DBMonster implementations of popular JavaScript frameworks which will be adapted to run the benchmark tests.

---

<sup>36</sup>DBMonster implementations, <http://mathieuancelin.github.io/js-repaint-perfs/>



**Figure 15 Example of a DBMonster Application Interface**

### 3.2.2.1 Tests for DBMonster Application

In order to run this benchmark application, a tool called browser-perf<sup>37</sup> will be adapted to automate the tests which includes a scroll/smoothness tests from telemetry<sup>38</sup> that is already integrated in browser-perf which will be used to collect metrics from the DBMonster application. The tool contains hundreds of metrics based on the render and frame measurements as described in Section 3.1.1. However, given the time remaining at the time of writing this thesis, it may be difficult to analyse all metrics. This will be further discussed in chapters 4 and 5.

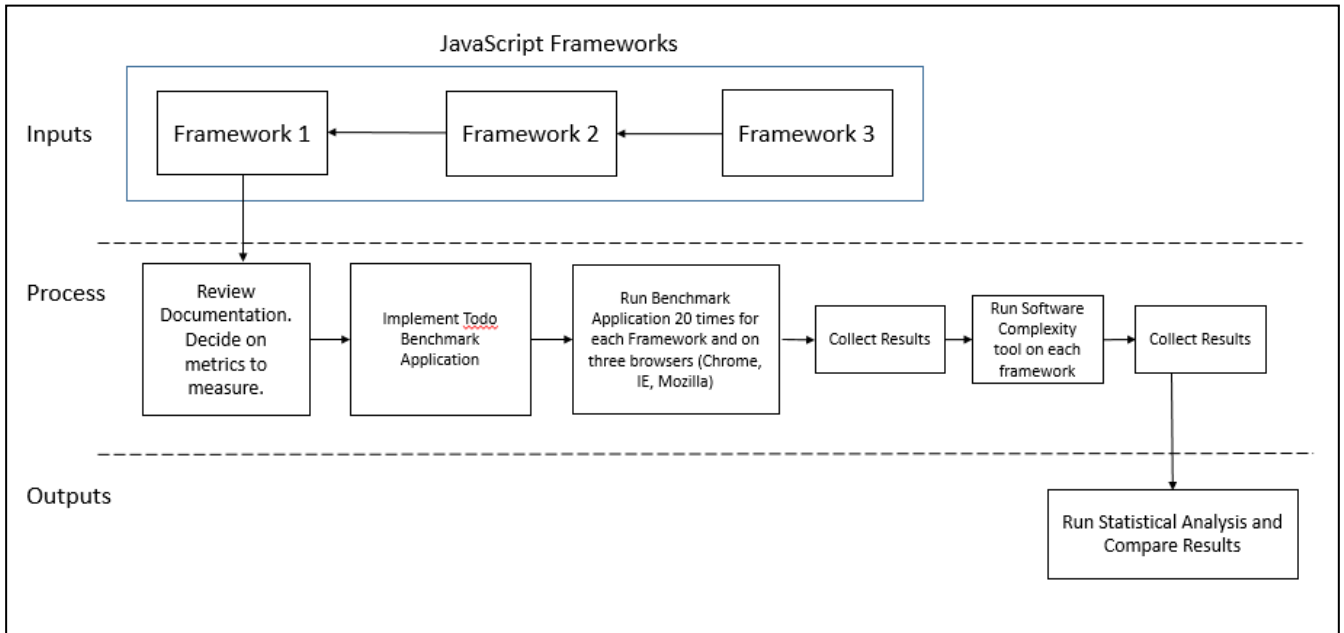
## 3.2 Experiment Designs

The aim of the experiment is to compare the performance of JavaScript Frameworks based on computer and software benchmarking metrics. In the context of this research, performance means the values returned when metrics are assessed. The experiments are split into two parts. One that runs experiments for the Todo application and another that is run for the DBMonster application. For the first part of the experiments, the process is shown in Figure 16. Each JavaScript framework was run at random for 20

<sup>37</sup> Browser-perf, <https://github.com/axemclion/browser-perf>

<sup>38</sup> Catapult Telemetry, <https://catapult.gsrc.io/telemetry>

times each on the benchmark application on three browsers (Chrome, Edge, and Mozilla) to validate the results and to give a balanced view of the results. A basic statistical analysis approach was used in order to compare and contrast the results gathered. After this is done, the experiment that gathers the software complexity measurements of the code used was conducted. This is also analysed based on a statistical analysis approach.

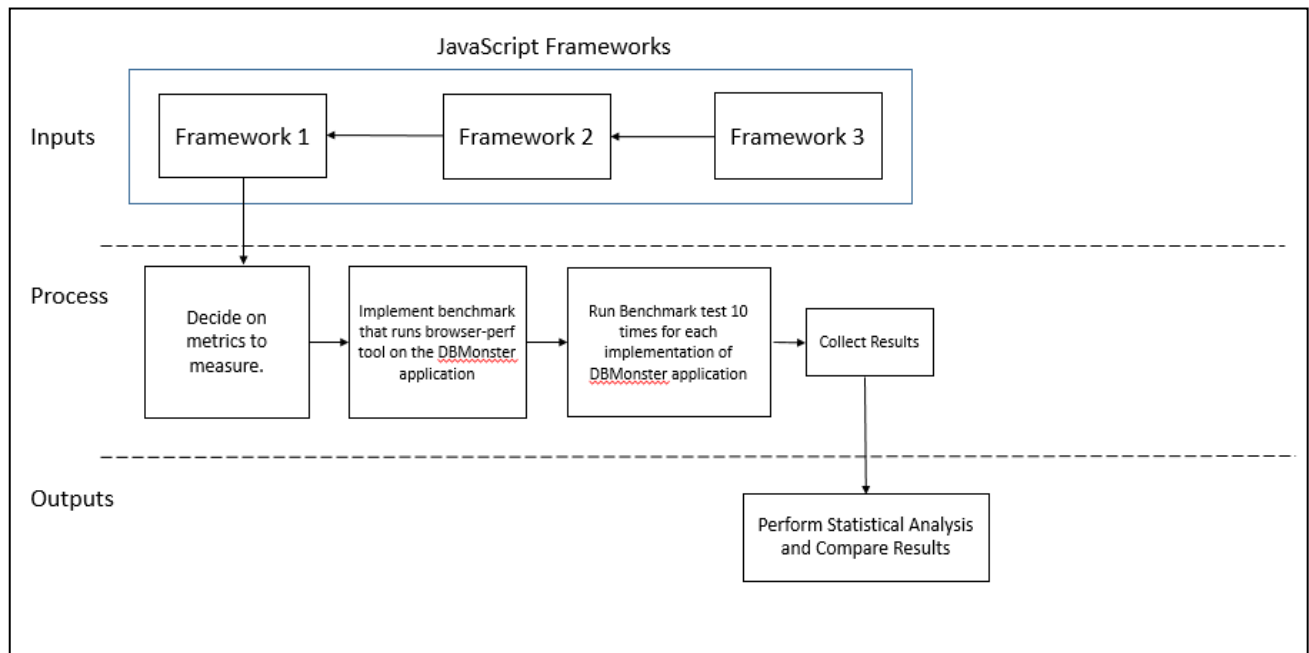


**Figure 16 Experiment Process Part A**

The next set of experiments for the DBMonster benchmark application will follow a process as illustrated in Figure 17. A thorough research into finding the best implementations of the DBMonster application that utilized JavaScript Frameworks was performed beforehand which includes recommendations from reading various posts and comments on github<sup>39</sup> source code repository while taking into account the number of contributions from developers on the nominated JavaScript frameworks. The experiment will run by choosing first, the best implementations of the DBMonster application in JavaScript for each nominated frameworks. Then, a benchmark script will be created that will utilize the API from the browser-perf tool which will automate the test for each chosen implementation of DBMonster application and will output the

<sup>39</sup> Github, <https://github.com/>

results of the metrics to a file. Finally, the file containing the results of the benchmark will be analysed.



**Figure 17 Experiment Process Part B**

### 3.3 Chapter Summary

This chapter presented a high level overview of the designs of the benchmark reference applications along with an overview of the experiment design. In the next chapter, the implementation of the applications are described in detail.

## 4. IMPLEMENTATION

This chapter will catalogue and describe the different software used in building the reference applications which were used to run the experiments. In addition, other aspects of the implementation are also described here such as the problems encountered, special features used for measuring the time as well as implications for the application development.

### 4.1 *Software Used*

As the aim of this thesis is to compare JavaScript frameworks, thus, the language chosen for the development of the applications is the JavaScript language. In addition, three JavaScript frameworks were selected to be compared. These include AngularJS, React and BackboneJS. The choice of source code editor was Visual Studio Code<sup>40</sup> which is a source code editor developed by Microsoft and is compatible with any operating system including Linux and macOS. Visual Studio Code is relatively new and was initially released in 2015 which allows developers to develop in almost all programming languages available today. This was chosen as the main source code editor due to its ease of use and support for debugging, syntax highlighting, code completion and source code version control.

Furthermore, NodeJS<sup>41</sup> was used as the runtime environment for all benchmark applications used in this project. NodeJS is simply an environment on which JavaScript code can be executed. NodeJS contains a built in http-server that is used to serve html pages. However, NodeJS is not just a webserver and what makes it powerful is that it allows developers to share code through the use of npm (node package manager). NPM<sup>42</sup> allows code to be bundled into reusable code known as packages or modules and distributed in the registry of modules (node package manager) which in turn, can be used by other developers in their applications. Every version of NodeJS comes with npm preinstalled in which developers can check for the

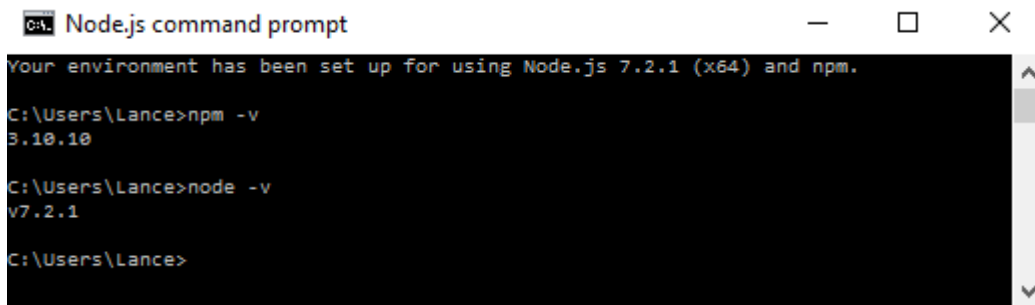
---

<sup>40</sup>Microsoft Visual Studio Code, <https://code.visualstudio.com/>

<sup>41</sup> NodeJS website, <https://nodejs.org/en/>

<sup>42</sup> NPM, <https://docs.npmjs.com/getting-started/what-is-npm>

version of their NodeJS environment as well as the npm version they are using by typing the code as shown in Figure 18.



```
C:\> Node.js command prompt
Your environment has been set up for using Node.js 7.2.1 (x64) and npm.

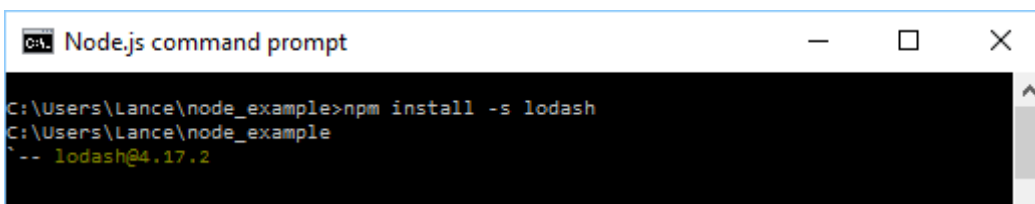
C:\Users\Lance>npm -v
3.10.10

C:\Users\Lance>node -v
v7.2.1

C:\Users\Lance>
```

**Figure 18 Checking version of NodeJS and npm**

Node modules can be installed in two ways either locally or globally. Developers use which kind of installation they prefer based on how they want to use the package. For example, if their application depends on their own module using something like Node.js’ “**require**”, then they would want to install the module locally using the **npm install -s <package\_name>** command. In contrast, if developers want to use a module as a command line tool, then install it globally using the **npm install -g <package\_name>** command. When a module is installed, a **node\_modules** folder will be created within the directory that they are working where modules are stored. An example of installing a module is shown in Figure 19. The command shown locates and installs the lodash module from the node package manager directory. In addition, a **node\_modules** folder is created within the directory being worked on where the module is stored. The directory layout of the project is described in section 5.1.1.



```
C:\Users\Lance> Node.js command prompt
C:\Users\Lance> npm install -s lodash
C:\Users\Lance>
-- lodash@4.17.2
```

**Figure 19 Example showing installation of node module**

The main source code repository used in this project is GitHub<sup>43</sup>. GitHub is a version control manager that is used by many open source projects to manage and distribute projects. In particular, all source code for both reference applications were adapted

<sup>43</sup> GitHub repository, <https://github.com/>

from the GitHub repository. However, code, especially for the Todo application, were outdated at the time of writing this thesis as newer versions of JavaScript frameworks were released. Therefore, some minor changes were done on the code which should make a slight difference in performance.

In the next section, the implementation of the Todo application for each JavaScript frameworks chosen is described.

## 4.2 *Todo Reference Application Implementations*

In this section, the implementation of the Todo application for each nominated JavaScript frameworks are described. In particular, the implementation is described by comparing the source code used to implement its core features and components as mentioned in Chapter 3. Therefore, three JavaScript frameworks were chosen in total.

### 4.2.1 AngularJS Todo Application

As mentioned earlier, the source code for the Todo Application was adapted from the TodoMVC project which is located in the GitHub repository. One of the main goals of AngularJS is to provide an extension to HTML. This is done with what is known as directives. Directives can be thought of as markers on a DOM (Document Object Model) element such as attributes, element name, comment or CSS class that tell the HTML compiler (\$compile) within AngularJS to attach a specific behaviour to that DOM element (via event listeners) or transform specific DOM elements and propagate these changes to its children. This is implemented with the use of *ng-tags*, which bind the view to one or many models with the help of controllers.

```
<body>
  <section id="todoapp" ng-controller="TodoCtrl as TC">
    <header id="header">
      <h1>todos</h1>
      <form id="todo-form" ng-submit="TC.addToDo()">
        <input id="new-todo" placeholder="What needs to be done?" ng-model="TC.newTodo.title" autofocus>
      </form>
    </header>
```

Figure 20 Example of how an input is saved in AngularJS

As seen in Figure 20, a data binding between the input field and the model exists. The **ng-model** directive binds the input field to **TC.newTodo.title**. The controller in this case is defined by the code **TodoCtrl as TC**. By binding the title value to **newTodo**, the controller is able to create a new task item (object) by accessing the input field. In

order to submit this value, the **ng-submit** directive is used. The function **TC.addTodo()** handles the creation of items.

```
<footer id="footer" ng-show="TC.todos.length" ng-cloak>
```

**Figure 21 Example of hiding footer in AngularJS**

The footer is shown by the **ng-show** directive, as seen in Figure 21. The **ng-show** directive is used to show or hide the footer whenever the expression inside the tag is validated to either true or false. **TC.todos.length** is evaluated to true if the length of the task list is greater than zero. The footer is hidden if the value of the length is zero. The **ng-cloak** directive is used to avoid flickering while loading the application. This causes the footer to stay hidden until the expression inside the **ng-show** is fully validated.

```
<li ng-repeat="todo in TC.todos | filter:TC.statusFilter track by $index"
    ng-class="{completed: todo.completed, editing: todo === TC.editedTodo}">
  <div class="view">
    <input class="toggle" type="checkbox" ng-model="todo.completed">
    <label ng-dblclick="TC.editTodo(todo)">{{todo.title}}</label>
    <button class="destroy" ng-click="TC.removeTodo($index)"></button>
  </div>
```

**Figure 22 Example of showing a list of todo items in AngularJS**

As seen in Figure 22, the **ng-repeat** directive is used to iterate over a list of todo items. A filter is applied so the items are sorted according to their status. The **ng-class** directive extends the HTML class-tag, which can alter the visual style of the HTML. The **ng-class** in this example alters how the item looks like if its status is either set to completed or edited, and adds the HTML-tag accordingly. However, this depends on the state of the task.

#### 4.2.2 React Todo Application

The Todo application implemented in React is split into four components which include the input field, the task list, the task items and the footer. If one component uses another, both are rendered. In this case, the main application is the input field, which uses the list and the footer, and the list using the task items.



```

<header className="header">
  <h1>todos</h1>
  <input
    className="new-todo"
    placeholder="What needs to be done?"
    value={this.state.newTodo}
    onKeyDown={this.handleNewTodoKeyDown}
    onChange={this.handleChange}
    autoFocus={true}
  />
</header>

```

Figure 23 Input field component in React

```

handleNewTodoKeyDown(event) {
  if (event.keyCode !== ENTER_KEY) {
    return;
  }

  event.preventDefault();

  var val = this.state.newTodo.trim();

  if (val) {
    this.props.model.addToDo(val);
    this.setState({newTodo: ''});
  }
}

```

Figure 24 Logic for the input field component

In Figures 23 and 24, the main code for the input field is shown as a separate component with its own internal logic. In Figure 25, the visual elements, represented by HTML, of the component are shown. JavaScript doesn't allow code to be mixed up with HTML, however, React's feature known as JSX allows this to happen. It is possible to write React code without using JSX, however, JSX allows developers to write cleaner code and type less code as rendering can be done by combining HTML-like syntax code with JavaScript.

The logic shown in Figure 24 shows the **handleNewTodoKeyDown** function. This is triggered when a user presses a key in the **onKeyDown** of Figure 23. This function checks whether the Enter key is pressed or not. If the Enter key is pressed, the value

from the input field is saved and a trim function is called on the input. The **addTodo** function handles the new item entered where it is then passed on to the model and the **setState** function resets the state of the input field to an empty string.

```
if (activeTodoCount || completedCount) {  
  footer =  
    <TodoFooter  
      count={activeTodoCount}  
      completedCount={completedCount}  
      nowShowing={this.state.nowShowing}  
      onClearCompleted={this.clearCompleted}  
    />;  
}
```

**Figure 25 Example of hiding/showing the footer in React**

In Figure 25, the **TodoFooter** component and its associated logic is shown. If **activeTodoCount** (the amount of active tasks) or **completedCount** (amount of completed tasks) is larger than zero, the footer is shown and the values from the model is bound to it. The **TodoFooter** component then uses these values and re-renders the application.

```
var todoItems = shownTodos.map(function (todo) {  
  return (  
    <TodoItem  
      key={todo.id}  
      todo={todo}  
      onToggle={this.toggle.bind(this, todo)}  
      onDestroy={this.destroy.bind(this, todo)}  
      onEdit={this.edit.bind(this, todo)}  
      editing={this.state.editing === todo.id}  
      onSave={this.save.bind(this, todo)}  
      onCancel={this.cancel}  
    />  
  );  
}, this);
```

**Figure 26 Example of a task item in React**

```

    if (todos.length) {
      main = (
        <section className="main">
          <input
            className="toggle-all"
            type="checkbox"
            onChange={this.toggleAll}
            checked={activeTodoCount === 0}
          />
          <ul className="todo-list">
            {todoItems}
          </ul>
        </section>
      );
    }

```

Figure 27 Example of a creation of task item in React

```

    return (
      <div>
        <header className="header">
          <h1>todos</h1>
          <input
            className="new-todo"
            placeholder="What needs to be done?"
            value={this.state.newTodo}
            onKeyDown={this.handleNewTodoKeyDown}
            onChange={this.handleChange}
            autoFocus={true}
          />
        </header>
        {main}
        {footer}
      </div>
    );

```

Figure 28 Example showing rendering of all task items in React

In Figure 26, React creates all items as separate components `<TodoItem />` component. These are created and saved with their properties in a variable called **todoItems**. This list is bound to the view `{todoItems}` where it is rendered as shown in Figure 27. On the other hand, the main render function renders all components created as shown in Figure 28.

### 4.2.3 Backbone Todo Application

In Backbone, the overall application view is presented in Figure 29 and is the top-level piece of the user interface. The corresponding HTML is also shown in Figure 30.

```
// Our overall **AppView** is the top-level piece of UI.
app.AppView = Backbone.View.extend({

  // Instead of generating a new element, bind to the existing skeleton of
  // the App already present in the HTML.
  el: '.todoapp',

  // Our template for the line of statistics at the bottom of the app.
  statsTemplate: _.template($('#stats-template').html()),

  // Delegated events for creating new items, and clearing completed ones.
  events: {
    'keypress .new-todo': 'createOnEnter',
    'click .clear-completed': 'clearCompleted',
    'click .toggle-all': 'toggleAllComplete'
  },
},
```

Figure 29 Example of the application view in Backbone

```
<body>
  <section class="todoapp">
    <header class="header">
      <h1>todos</h1>
      <input class="new-todo" placeholder="What needs to be done?" autofocus>
    </header>
    <section class="main">
      <input class="toggle-all" id="toggle-all" type="checkbox">
      <label for="toggle-all">Mark all as complete</label>
      <ul class="todo-list"></ul>
    </section>
    <footer class="footer"></footer>
  </section>
```

Figure 30 HTML view of Todo application in Backbone

```

initialize: function () {
  this.allCheckbox = this.$('.toggle-all')[0];
  this.$input = this.$('.new-todo');
  this.$footer = this.$('.footer');
  this.$main = this.$('.main');
  this.$list = this.$('.todo-list');

  this.listenTo(app.todos, 'add', this.addOne);
  this.listenTo(app.todos, 'reset', this.addAll);
  this.listenTo(app.todos, 'change:completed', this.filterOne);
  this.listenTo(app.todos, 'filter', this.filterAll);
  this.listenTo(app.todos, 'all', _.debounce(this.render, 0));

  // Suppresses 'add' events with {reset: true} and prevents the app view
  // from being re-rendered for every model. Only renders when the 'reset'
  // event is triggered at the end of the fetch.
  app.todos.fetch({reset: true});
},

```

Figure 31 Example showing initialization of App

```

// Re-rendering the App just means refreshing the statistics -- the rest
// of the app doesn't change.
render: function () {
  var completed = app.todos.completed().length;
  var remaining = app.todos.remaining().length;

  if (app.todos.length) {
    this.$main.show();
    this.$footer.show();

    this.$footer.html(this.statsTemplate({
      completed: completed,
      remaining: remaining
    }));

    this.$('.filters li a')
      .removeClass('selected')
      .filter('[href="#/" + (app.TODOFilter || '') + "']')
      .addClass('selected');
  } else {
    this.$main.hide();
    this.$footer.hide();
  }

  this.allCheckbox.checked = !remaining;
},

```

Figure 32 Example showing render function of App in Backbone

```

// Add a single todo item to the list by creating a view for it, and
// appending its element to the `<ul>`.
addOne: function (todo) {
  var view = new app.TodoView({ model: todo });
  this.$list.append(view.render().el);
},

// Add all items in the **Todos** collection at once.
addAll: function () {
  this.$list.html('');
  app.todos.each(this.addOne, this);
},

filterOne: function (todo) {
  todo.trigger('visible');
},

filterAll: function () {
  app.todos.each(this.filterOne, this);
},

```

Figure 33 Example showing functions triggered for events

```

// Generate the attributes for a new Todo item.
newAttributes: function () {
  return {
    title: this.$input.val().trim(),
    order: app.todos.nextOrder(),
    completed: false
  };
},

// If you hit return in the main input field, create new **Todo** model,
// persisting it to *localStorage*.
createOnEnter: function (e) {
  if (e.which === ENTER_KEY && this.$input.val().trim()) {
    app.todos.create(this.newAttributes());
    this.$input.val('');
  }
},

// Clear all completed todo items, destroying their models.
clearCompleted: function () {
  _.invoke(app.todos.completed(), 'destroy');
  return false;
},

```

Figure 34 Example showing generation of attributes and clearing the model

In Figure 29, as with all implementations, the whole application view is bound to the element with class **todoapp** present in the HTML view (see Figure 30). Events are delegated for events such as creating new items and clearing finished or completed ones. When the first view is rendered, all relevant events are bound on the Todos collection when items are either added or changed. Figure 31 shows the names of the

functions called when an event is triggered. The actual logic of the functions are illustrated in Figures 32-34.

In the next section, the implementation of the benchmarking clock is described in detail.

### 4.3 Benchmarking Clock Implementation

This section will describe the implementation of the clock used in order to calculate the execution time of all tests within the Todo benchmark application.

```
BenchmarkRunner.prototype._runTest = function(suite, testFunction, prepareReturnValue, callback)
{
    var now = window.performance && window.performance.now ?
    function ()
    { return window.performance.now(); } : Date.now;

    var contentWindow = this._frame.contentWindow;
    var contentDocument = this._frame.contentDocument;

    var startTime = now();
    testFunction(prepareReturnValue, contentWindow, contentDocument);
    var endTime = now();
    var syncTime = endTime - startTime;

    var startTime = now();
    setTimeout(function () {
        setTimeout(function () {
            var endTime = now();
            callback(syncTime, endTime - startTime);
        }, 0)
    }, 0);
}
```

Figure 35 Code snippet of Clock implementation

Figure 35 is the code snippet in implementing the clock used for the benchmark. This snippet of code is part of the benchmark script created in order to run the tests and measure the time taken to execute the tests. The main clock used in running the benchmarks is the **Window.performance**<sup>44</sup> API. This API allows developers to have access to certain functions for measuring the performance of web pages and web applications, including other APIs such as the **Navigation Timing**<sup>45</sup> API and other high resolution time data. The clock is monitored by calling the **Performance.now()** method within the **Window.performance** API that returns a

---

<sup>44</sup> Window.performance API, <https://developer.mozilla.org/en-US/docs/Web/API/Window/performance>

<sup>45</sup> Navigation Timing API, [https://developer.mozilla.org/en-US/docs/Web/API/Navigation\\_timing\\_API](https://developer.mozilla.org/en-US/docs/Web/API/Navigation_timing_API)

**DOMHighResTimeStamp**<sup>46</sup> that is used to store a time value as a double type. In this case, the time is measured in milliseconds and returns time elapsed since what is known as the time origin. In this case, the time origin is the actual time in a browser window when the user views a web page or document.

The `Date.now()` is another type of clock and can be used as an alternative to the `Window.performance` API. `Date.now()` is a method that is typically used in UNIX systems as it depends on a system clock. The value it returns is the time in milliseconds since 1 January 1970. The major difference between `performance.now()` and `Date.now()` is that the former is a high-end resolution timer that is typically more accurate in measuring web pages that require more precise measurements such as media (audio, video and gaming). On the other hand, `Date.now()` is more dependent on system clocks which typically runs on UNIX systems as it is formerly based on the Unix epoch. Also, `Date.now()` is a relatively old clock compared to `performance.now()` as the latter is only available in newer browsers.

As shown in Figure 36, the time is calculated by first storing the value of the current time in a variable. In this case, the current time is stored in a **startTime** variable. Next, the function that executes the events that triggers the addition, completion and deletion of items from the Todo application is executed. This function then executes the tests in sequence. When this is done, `performance.now()` is called again to store the current time after the execution of **testFunction**. Now, in order to calculate the total time taken to execute the testFunction, the difference between the `endTime` and `startTime` is calculated which returns the total time taken to execute all tests. The purpose of calculating the time before and after the execution of the testFunction is due to the fact that testFunction is the one that gets executed when the Todo benchmark tool is run. Therefore, in order to calculate the total time taken to execute the benchmark test, timestamps are stored before and after the execution of the test function where the difference between the two timestamps produces the total time taken to execute the test.

---

<sup>46</sup>DOMHighResTimeStamp, <https://developer.mozilla.org/en-US/docs/Web/API/DOMHighResTimeStamp>



```
var startTime = now();
setTimeout(function () {
    setTimeout(function () {
        var endTime = now();
        callback(syncTime, endTime - startTime);
    }, 0)
}, 0);
```

Figure 36 Code snippet showing Implementation of Asynchronous Timer

#### 4.4 *Discontinued Implementations*

Due to limited time, where the latter part of this project was spent abroad in the Philippines and with limited resources, a project management decision was made to discontinue the implementation of the DBMonster application and other experiments regarding the measurement of the speed index and page load as it would not be feasible to run the experiments in the Philippines due to a lack of internet connectivity in the area resided in. Instead a focus was made on running the experiments on the Todo application. However, this may not give a well-balanced view of the overall results. This is further discussed in Chapter 5. The next section describes the test environment the experiments were executed on as well as the rationale behind the test environment.

#### 4.5 *Benchmark Test Environment*

The test environment used where benchmarks were performed on was Windows. The test environment was treated so that it reaches as close to an ideal platform for running benchmarks. Therefore, before running the benchmarks, an attempt to end as much background processes and applications as possible was performed beforehand each time the benchmark application tool is run. Also, all disk logging was disabled as recommended by Intel<sup>47</sup> as it is found that read/write operations of hard disks can have an effect in the execution time of programs.

The set up used is as follows:

**Processor:** i7 4770k 4<sup>th</sup> Generation Ivy Bridge with 8GB memory (2013)

**Hard drive:** Samsung 840 Pro 120GB SSD with 1TB WD mechanical hard drive

**Operating system:** Windows 10 Pro

**Network: Ethernet LAN connection:** 240Mbps Download/20Mbps Upload

---

<sup>47</sup> Intel, (<http://www.intel.com/content/dam/doc/white-paper/intel-it-optimizing-pc-performance-paper.pdf>)

**Browser versions:** Google Chrome version 55.0.2883.87, Mozilla Firefox version 50.1.0, Microsoft Edge version 38.14393.0.0

## ***4.6 Chapter Summary***

This chapter described the implementation of the Todo reference applications and compared the code used for implementing the Todo applications that was built using the selected JavaScript frameworks as well as a description of the clock used to monitor the time in the benchmarks. The next chapter covers the execution of experiments as well as the evaluation of the results.

## 5. RUNNING THE EXPERIMENTS & EVALUATION

This chapter describes the experiments conducted to compare and evaluate JavaScript frameworks as well as the results obtained from the experiments. The experiments follows a process as described in Chapter 3 of the design and methodology section. In addition, as outlined in Section 3.1.1, this chapter also discusses the evaluation of the results collected.

### 5.1 *Experimentation*

This section describes the experiments conducted which includes the process followed to fulfil the experiments. Although the experiments were not completed as planned, the major focus on comparing execution time was still performed. As mentioned in Chapter 3, the Todo application was implemented in three JavaScript frameworks using the TodoMVC project. These frameworks include AngularJS, React and BackboneJS. Typically, before starting a new JavaScript framework, small examples or tutorials were implemented to be familiar with the frameworks. Thus, an initial review of the documentation of each JavaScript framework was conducted first, followed by an implementation of the sample applications from each tutorial.

#### 5.1.1 Implementation of Benchmark Application

The experiment was conducted by implementing a benchmark application which integrates all three JavaScript Frameworks so that one benchmark application runs all tests on all frameworks. This was done by adding a benchmark script that runs all tests which calculates the execution time of all tasks ran. The project structure is enumerated below.

- Todomvc-master (root directory)
  - node\_modules (sub-folder)
  - resources (sub-folder)
    - benchmark-runner.js
    - manager.js
    - tests
  - todomvc (sub-folder)

- angularjs (sub-folder)
  - js
    - controllers
      - todoCtrl.js
    - directives
      - todoFocus.js
    - services
      - todoStorage.js
    - app.js
  - node\_modules
  - package.json
  - index.html
- backbone (sub-folder)
  - js
    - collections
      - todos.js
    - models
      - todo.js
    - views
      - app-view.js
      - todo-view.js
    - routers
      - router.js
    - app.js
  - node\_modules
  - index.html
  - package.json
- react (sub-folder)
  - js
    - app.js
    - footer.js
    - todoItem.js
    - todoModel.js

- utils.js
- node\_modules
- package.json
- index.html
- react-es2015 (sub-folder)
  - js
    - app.js
    - index.js
    - todoFooter.js
    - todoItem.js
    - todoModel.js
    - utils.js
  - node\_modules
  - bundle.js
  - package.json
  - webpack.config
  - index.html
- index.html
- package.json

```

C:\WINDOWS\System32\WindowsPowerShell\v1.0\Powershell.exe
Windows PowerShell
Copyright (C) 2016 Microsoft Corporation. All rights reserved.

C:\Users\Lance\Documents\GitHub> git clone https://github.com/tastejs/todomvc.git
Cloning into 'todomvc'...
remote: Counting objects: 29091, done.
remote: Compressing objects: 100% (9/9), done.
Receiving objects: 21% (6232/29091), 11.57 MiB | 105.00 KiB/s
  
```

**Figure 37 Cloning a GitHub repository**

The TodoMVC project was cloned from the GitHub repository by running the command as shown in Figure 37. The main page (index.html) read the benchmark scripts from the resources folder.

- resources
  - benchmark-runner.js

- manager.js
- tests.js

```

1 var numberOfItemsToAdd = 100;
2 var Suites = [];
3 Suites.push({
4   name: 'BackboneJS',
5   url: 'todomvc/backbone/index.html',
6   version: '1.3.3',
7   prepare: function (runner, contentWindow, contentDocument) {
8     //The contentWindow property returns the Window object generated by an iframe element (through the window object,
9     //you can access the document object and then any one of the document's elements).
10    contentWindow.Backbone.sync = function () {}
11    return runner.waitForElement('.new-todo').then(function (element) {
12      element.focus();
13      return element;
14    });
15  },
16  tests: [
17    new BenchmarkTestStep('Adding' + numberOfItemsToAdd + 'Items', function (newTodo, contentWindow, contentDocument) {
18      var appView = contentWindow.appView;
19      for (var i = 0; i < numberOfItemsToAdd; i++) {
20        var inputEvent = document.createEvent('Event');
21        inputEvent.initEvent('input', true, true);
22        newTodo.value = 'Something to do ' + i;
23        newTodo.dispatchEvent(inputEvent);
24
25        var keypressEvent = document.createEvent('Event');
26        keypressEvent.initEvent('keypress', true, true);
27        keypressEvent.which = 13; // VK_ENTER
28        newTodo.dispatchEvent(keypressEvent);
29      }
30    }),
31    new BenchmarkTestStep('CompletingAllItems', function (newTodo, contentWindow, contentDocument) {
32      var checkboxes = contentDocument.querySelectorAll('.toggle');
33      for (var i = 0; i < checkboxes.length; i++)
34        checkboxes[i].click();
35    }),
36    new BenchmarkTestStep('DeletingAllItems', function (newTodo, contentWindow, contentDocument) {
37      var deleteButtons = contentDocument.querySelectorAll('.destroy');
38      for (var i = 0; i < deleteButtons.length; i++)
39        deleteButtons[i].click();
40    })
41  ]
42 });

```

**Figure 38 Snippet Code of Adding a Suite of Test**

As shown in Figure 38, the tests.js file contains all test subjects. The **Suites** array is used to add the test suites. In addition, the **BenchmarkTestStep** function creates a new instance of the object using the **new** keyword for each step to be executed including the addition, completion and deletion of the todo items.

```

11 function createUIForSuites(suites, onstep, onrun) {
12     var control = document.createElement('nav');
13     var ol = document.createElement('ol');
14     var checkboxes = [];
15
16     /* var button = document.createElement('button');
17     button.textContent = 'Step Tests';
18     button.onclick = onstep;
19     control.appendChild(button); */
20
21     var button = runButton = document.createElement('button');
22     button.textContent = 'Run All';
23     button.onclick = onrun;
24     control.appendChild(button);
25
26     for (var suiteIndex = 0; suiteIndex < suites.length; suiteIndex++) {
27         var suite = suites[suiteIndex];
28         var li = document.createElement('li');
29         var checkbox = document.createElement('input');
30         checkbox.id = suite.name;
31         checkbox.type = 'checkbox';
32         checkbox.checked = true;
33         checkbox.onchange = (function (suite, checkbox) { return function () { suite.disabled = !checkbox.checked; runs = []; } })(suite, checkbox);
34         checkbox.onchange();
35         checkboxes.push(checkbox);
36
37         li.appendChild(checkbox);
38         var label = document.createElement('label');
39         label.appendChild(document.createTextNode(formatTestName(suite.name) + ' ' + suite.version));
40         li.appendChild(label);
41         label.htmlFor = checkbox.id;
42
43         var testList = document.createElement('ol');
44         for (var testIndex = 0; testIndex < suite.tests.length; testIndex++) {
45             var testItem = document.createElement('li');
46             var test = suite.tests[testIndex];
47             var anchor = document.createElement('a');
48             anchor.id = suite.name + '-' + test.name;
49             test.anchor = anchor;
50             anchor.appendChild(document.createTextNode(formatTestName(suite.name, test.name)));

```

Figure 39 Snippet Code for Creation of UI Layout

```

63
64 function startTest() {
65
66     var match = window.location.search.match(/[?&]r=(\d+)/),
67         timesToRun = match ? +(match[1]) : 1
68
69     var runner = new BenchmarkRunner(Suites, {
70         willRunTest: function (suite, test) {
71             if (!navigator.userAgent.match("MSIE 9.0")) test.anchor.classList.add('running');
72         },
73         didRunTest: function (suite, test) {
74             var classList = test.anchor.classList;
75             if (!navigator.userAgent.match("MSIE 9.0")) classList.remove('running');
76             if (!navigator.userAgent.match("MSIE 9.0")) classList.add('ran');
77         },
78         didRunSuites: function (measuredValues) {
79             var results = '';
80             var total = 0; // FIXME: Compute the total properly.
81             for (var suiteName in measuredValues) {
82                 var suiteResults = measuredValues[suiteName];
83                 for (var testName in suiteResults.tests) {
84                     var testResults = suiteResults.tests[testName];
85                     for (var subtestName in testResults) {
86                         results += suiteName + ' : ' + testName + ' : ' + subtestName
87                             + ' : ' + testResults[subtestName] + ' ms\n';
88                     }
89                 }
90                 results += suiteName + ' : ' + suiteResults.total + ' ms\n';
91                 total += suiteResults.total;
92             }
93             results += 'Run ' + (runs.length + 1) + '/' + timesToRun + ' - Total : ' + total + ' ms\n';
94
95             if (!results)
96                 return;
97
98             console.log(results)
99
100             runs.push(measuredValues)
101             timesRan++
102             if (timesRan >= timesToRun) {

```

Figure 40 Snippet Code of startTest Function

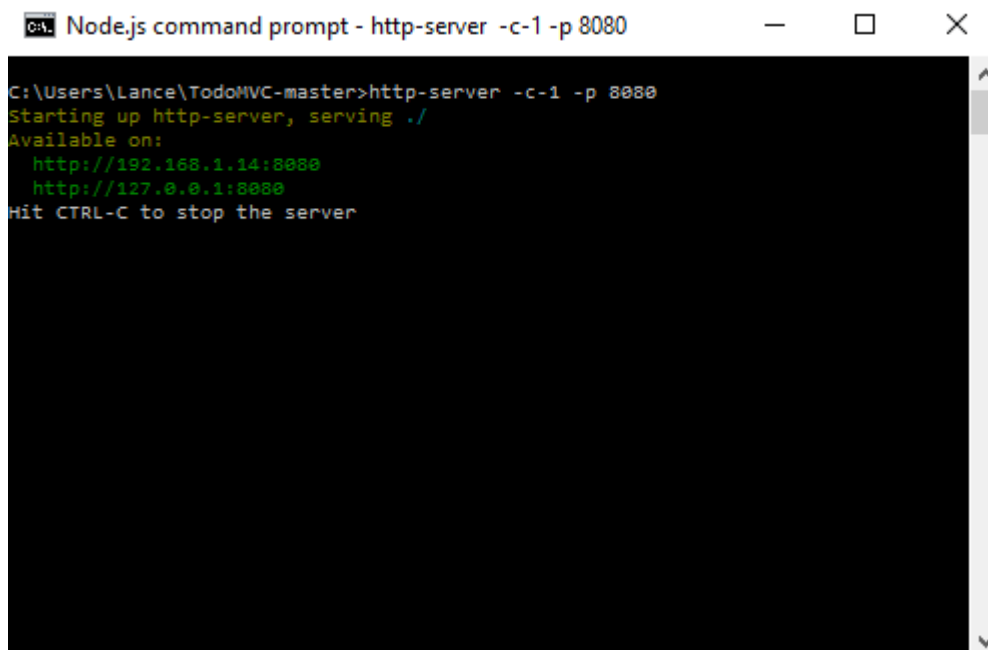
The file `manager.js` performs all test preparations such as preparing the UI layout and the display of graph of the results as shown in Figure 39. A **startTest** function is present in the `manager.js` file that executes the functions needed from the `benchmark-runner.js` file to run the benchmarks. This is shown in Figure 40.

The execution of tests takes place in the `benchmark-runner.js` file. Inside this file is a **BenchmarkRunner** function which calls a number of functions, which are extended from the parent class (**BenchmarkRunner**) using the **prototype** keyword that executes a number of step functions. Such function includes a **\_runTestAndRecordResults** function that executes the **\_runTest** function containing the clock and other benchmark steps that calculates the executed time as described in Section 4.3.

### 5.1.2 Running the Experiments

This section describes the execution of experiments. These are divided into two further smaller sections where one describes the experiment regarding the benchmark application and the other describes the experiment regarding the running of the software complexity tool on the Todo implementations in different JavaScript frameworks.

#### 5.1.2.1 Executing the Todo Benchmark Application



```
Node.js command prompt - http-server -c-1 -p 8080
C:\Users\Lance\TodoMVC-master>http-server -c-1 -p 8080
Starting up http-server, serving ./
Available on:
  http://192.168.1.14:8080
  http://127.0.0.1:8080
Hit CTRL-C to stop the server
```

Figure 41 Running the Web Server



Firstly, the experiment is started by firing up a web server within the directory the Todo application is located. This is done by executing the command as shown in Figure 41.

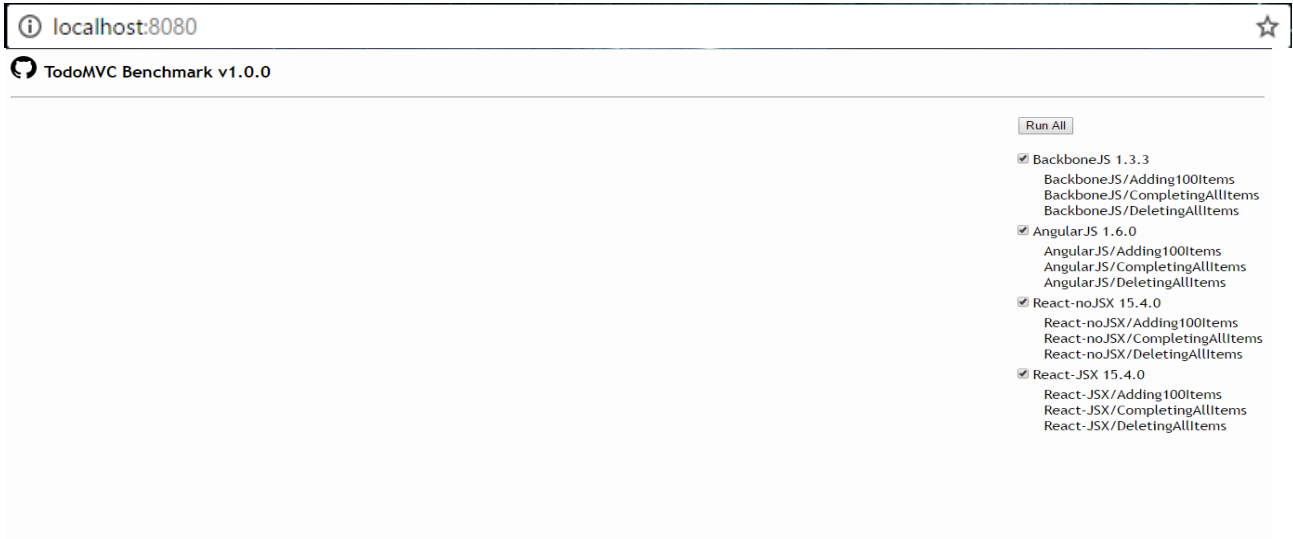


Figure 42 Interface Showing the Benchmark Application

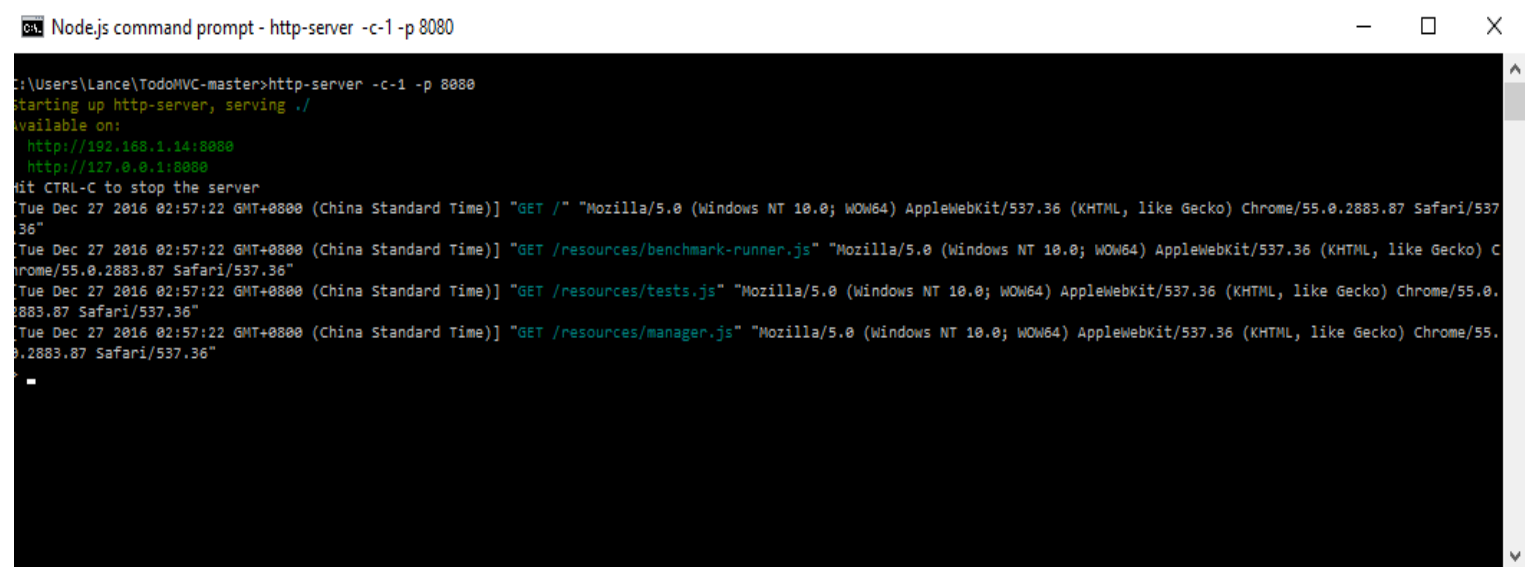


Figure 43 Image Showing Retrieval of Files

Next, the benchmark application is accessed by typing the URL of the localhost in the browser being used. This is done by typing the localhost address in the URL bar as shown in Figure 42. When this is accessed, all files listed in index.html are retrieved

along with the benchmark resources as shown in Figure 43. The benchmark application is now ready to be executed.

The user interaction (add item) is emulated by sending a keydown, keyup (or keypress) event with the number keyCode = 13 to the input of the Todo application. Next, each item in the list are set to complete by selecting all items and sending a click event to each checkbox input. When the benchmark application is run, each task within the benchmark application is executed in sequence. The frameworks are then shuffled after all tasks are executed and the graph is printed out on the browser. The chart is implemented by loading the Google Charts<sup>48</sup> API. The results are further discussed in Section 5.2.

#### 5.1.2.2 Executing the Software Complexity tool

The tool complexity-report<sup>49</sup> was used to retrieve the software complexity metrics. Complexity-report is a node.js based command-line tool that performs software complexity analysis. The tool produces a number of metrics such as Lines of Code, Cyclomatic Complexity, Halstead Complexity, and Maintainability Index as was described in Section 3.1.1. In order to run the tool, it requires to be installed first on node.js by running the command:

npm install complexity-report

The tool is executed by running the command:

cr [options] <path>

```
C:\Users\Lance\TodoMVC-master\todomvc>cr

Usage: index [options] <path>

Options:
  -h, --help                output usage information
  -c, --config <path>      specify path to configuration JSON file
  -o, --output <path>      specify an output file for the report
  -f, --format <format>    specify the output format of the report
  -e, --ignoreerrors        ignore parser errors
  -a, --allfiles            include hidden files in the report
  -p, --filepattern <pattern> specify the files to process using a regular expression to match against file names
  -P, --dirpattern <pattern> specify the directories to process using a regular expression to match against directory names
  -x, --excludepattern <pattern> specify the the directories to exclude using a regular expression to match against directory names
  -M, --maxfiles <number>  specify the maximum number of files to have open at any point
  -F, --maxfod <first-order density> specify the per-project first-order density threshold
  -O, --maxcost <change cost> specify the per-project change cost threshold
  -S, --maxsize <core size> specify the per-project core size threshold
  -M, --minmi <maintainability index> specify the per-module maintainability index threshold
  -C, --maxcyc <cyclomatic complexity> specify the per-function cyclomatic complexity threshold
  -Y, --maxcyden <cyclomatic density> specify the per-function cyclomatic complexity density threshold
  -D, --maxhd <halstead difficulty> specify the per-function Halstead difficulty threshold
  -V, --maxhv <halstead volume> specify the per-function Halstead volume threshold
  -E, --maxhe <halstead effort> specify the per-function Halstead effort threshold
  -s, --silent              don't write any output to the console
  -l, --logicalor           disregard operator || as source of cyclomatic complexity
  -w, --switchcase         disregard switch statements as source of cyclomatic complexity
  -i, --forin               treat for...in statements as source of cyclomatic complexity
  -t, --trycatch           treat catch clauses as source of cyclomatic complexity
  -n, --nemi               use the Microsoft-variant maintainability index (scale of 0 to 100)
  -Q, --nocoresize         don't calculate core size or visibility matrix

C:\Users\Lance\TodoMVC-master\todomvc>cr -o angular.json ./angular.js
```

**Figure 44 Example of Running the Complexity Report Tool**

<sup>48</sup> Google Charts, <https://developers.google.com/chart/>

<sup>49</sup> Complexity-report, <https://www.npmjs.com/package/complexity-report>

There exists a number of options by adding the option `-help` at the end of the command. In this case, the output file option is used to write out all complexity metric results and later viewed for the evaluation process as shown in Figure 44.

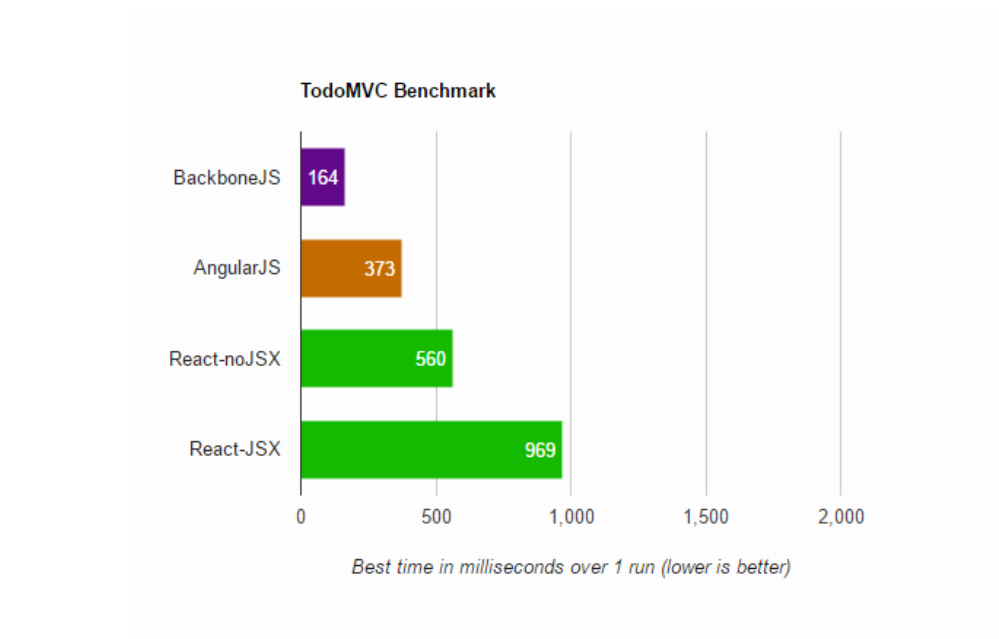
**5.2 Evaluation**

This section is divided into two sections where the first evaluates the results from the Todo Benchmark application and the other evaluates the results from the Complexity-report tool.

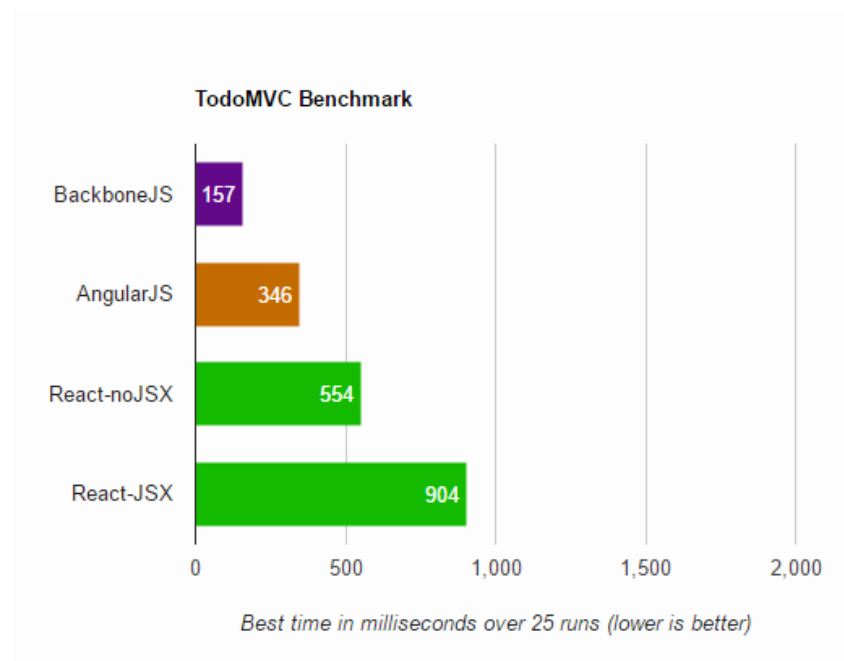
**5.2.1 Todo Application Benchmark Results**

Each benchmark were run on three browsers (Chrome, Edge and Mozilla) and for 25 times to ensure validity of results. Thus, this section is segregated into sub-sections according to each web browser used.

*Google Chrome*



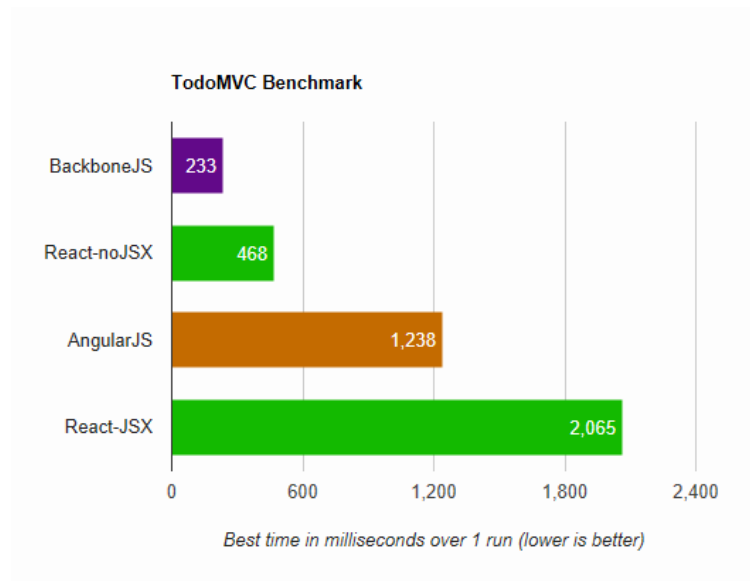
**Figure 45 Average Results generated in Google Chrome after 1 Run**



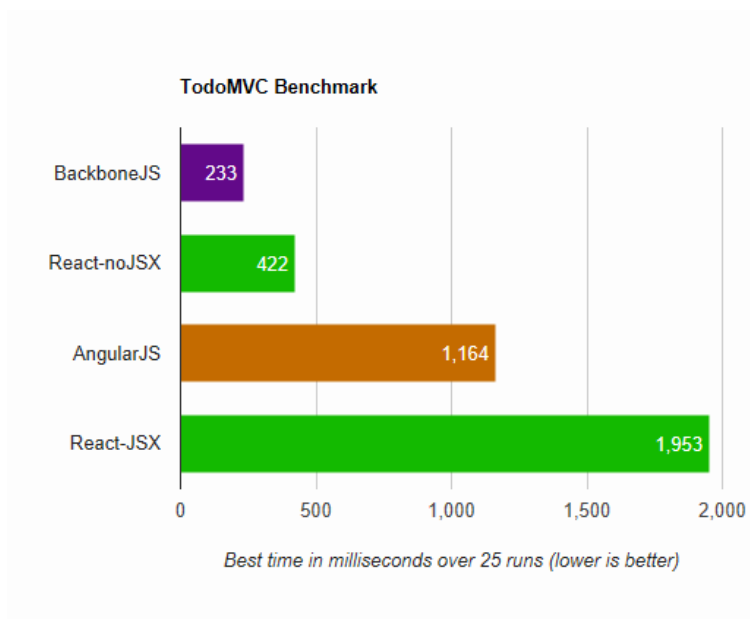
**Figure 46 Average Results generated in Google Chrome after 25 Runs**

The benchmark was first run on Google Chrome and Figures 45 and 46 illustrate the results gained from the benchmarks which are averaged and then sorted from fastest to slowest execution time. As can be seen from the results, BackboneJS achieved the fastest results averaging about 157ms of execution time. The slowest time achieved by a JavaScript framework is about 904 ms which was achieved from the use of JSX feature in React. There is a slight change of time from Run 1 to Run 25 as shown in the results which may be due to the fact that the JavaScript engine may still be warming up within the browser. The reason for this is because JavaScript needs to be compiled down into native code which is specific to the platform the code is running on. In this case, Google Chrome is the platform. Therefore, the compiled code needs to be interpreted by Chrome's JavaScript engine first before executing within the browser. However, since the performance of JavaScript engines is beyond the scope of this project, it is not discussed in greater detail.

## Microsoft Edge



**Figure 47 Average Results generated in Microsoft Edge after 1 Run**

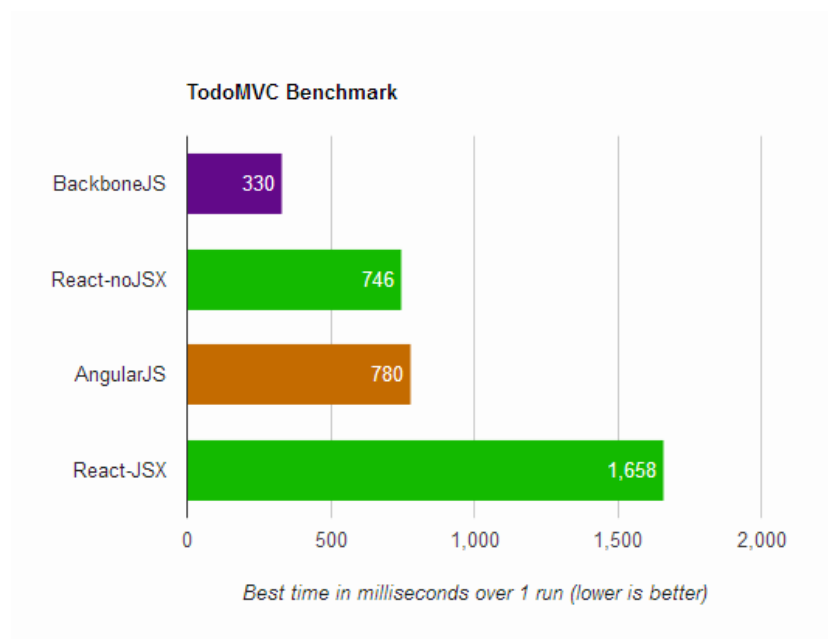


**Figure 48 Average Results generated in Microsoft Edge after 25 Runs**

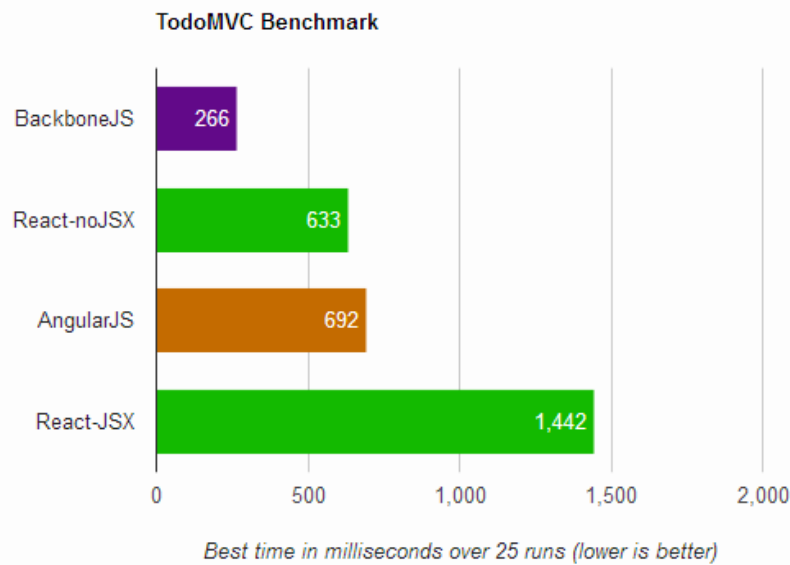
Figures 47 and 48 illustrates the results obtained from Microsoft Edge. Yet again the results are run for 25 times and averaged. As can be seen from both figures, there is a difference in results as compared to when the benchmark is run on Google Chrome. The biggest difference is with React-noJSX and AngularJS where before, AngularJS executed quite well on Google Chrome and the use of the standard implementation of the Todo application without the use of JSX seemed to gain faster on Microsoft Edge. However, in this case, AngularJS achieved a slower execution time of at least three

times slower than the previous results in Chrome whereas React's implementation of the Todo application without the use of JSX seemed to have performed faster AngularJS. On the other hand, BackboneJS still outperforms all JavaScript frameworks selected while React-JSX still underperforms all other JavaScript frameworks with execution time doubling compared to Google Chrome.

### ***Mozilla Firefox***



**Figure 49 Average Results generated in Mozilla Firefox after 1 Run**



**Figure 50 Average Results generated in Mozilla Firefox after 25 Runs**

Finally, the Todo benchmark application was run on Mozilla Firefox browser. The results generated, as illustrated in Figures 49 and 50 show that there is somewhat a strong similarity of results between React-noJSX implementation and AngularJS implementation. The benchmark was run for a total of 25 runs to obtain the same level of results each time. Yet again, BackboneJS performed the fastest out of all implementations of the Todo application whereas the implementation of the Todo application in React which utilized JSX performed the least which totals to all browsers where React-JSX performed the least. On the other hand, Backbone still outperforms all JavaScript frameworks.

As can be seen from the results presented above, there are major differences in the performances of JavaScript frameworks used in this project especially when tested on different browsers. This may be due to the fact that the tools used are optimized for specific browsers. As an example, NodeJS is built on Google Chrome's V8 engine which may well be one reason why Google Chrome's results were seen as faster than other browsers tested. Moreover, the implementation of the Todo application using BackboneJS and AngularJS used less lines of code than React which may be a factor when it comes to the performance of an application built using JavaScript frameworks. On the other hand, React's feature called JSX is relatively new and thus it may not have been fully optimized yet for use in browsers as it requires the process of

**transpiling** code to take place where code written in newer JavaScript standards such as ES6 to be transformed to an older standard (ES5) which the web browser can interpret and process. As of now, ES6 or JavaScript’s ECMAScript 6 standard is not supported by all browsers. Finally, one of the reasons BackboneJS out performed all other JavaScript frameworks used is because of the fact that it is a lightweight framework that has a net size of around 6.5kb and around 43.5kb with the required dependencies.

5.2.2 Software Complexity Measurement Results

This section illustrates by the use of bar graphs, the results obtained from running the software complexity tool (complexity-report) on the Todo implementations. It is worth to note that each run ignores all node modules within the node\_modules folder in the project and analyses solely all source code used within each **js** folder. Also, all software complexity measures are calculated per-function (method) in the source code within the tool.

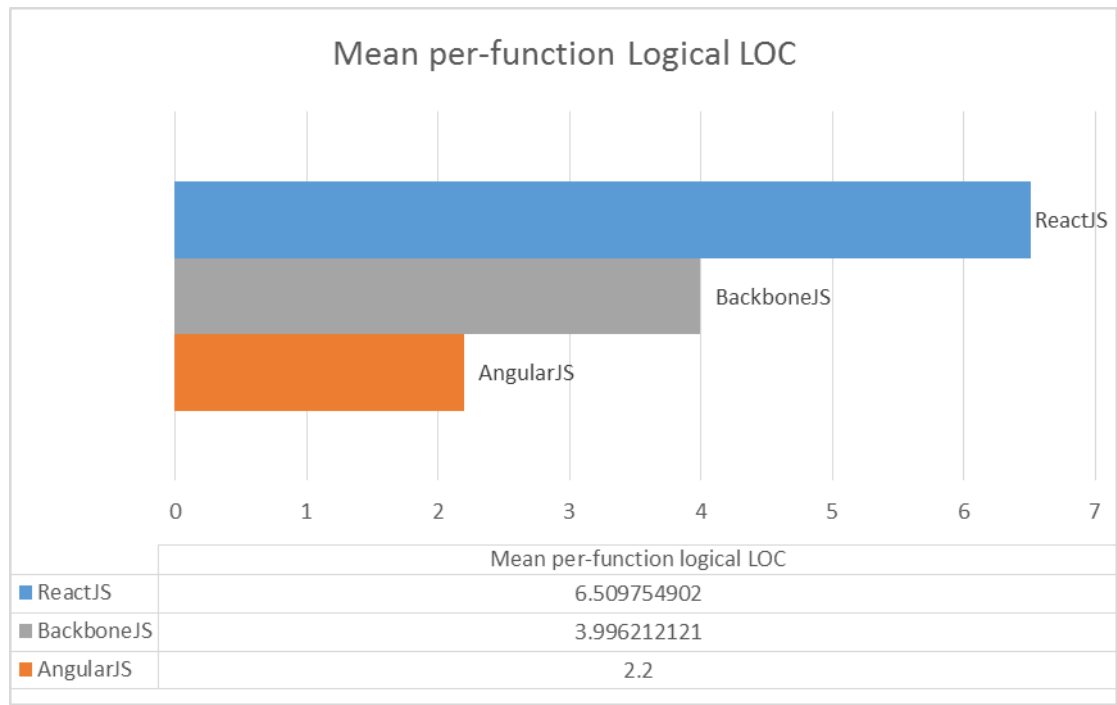


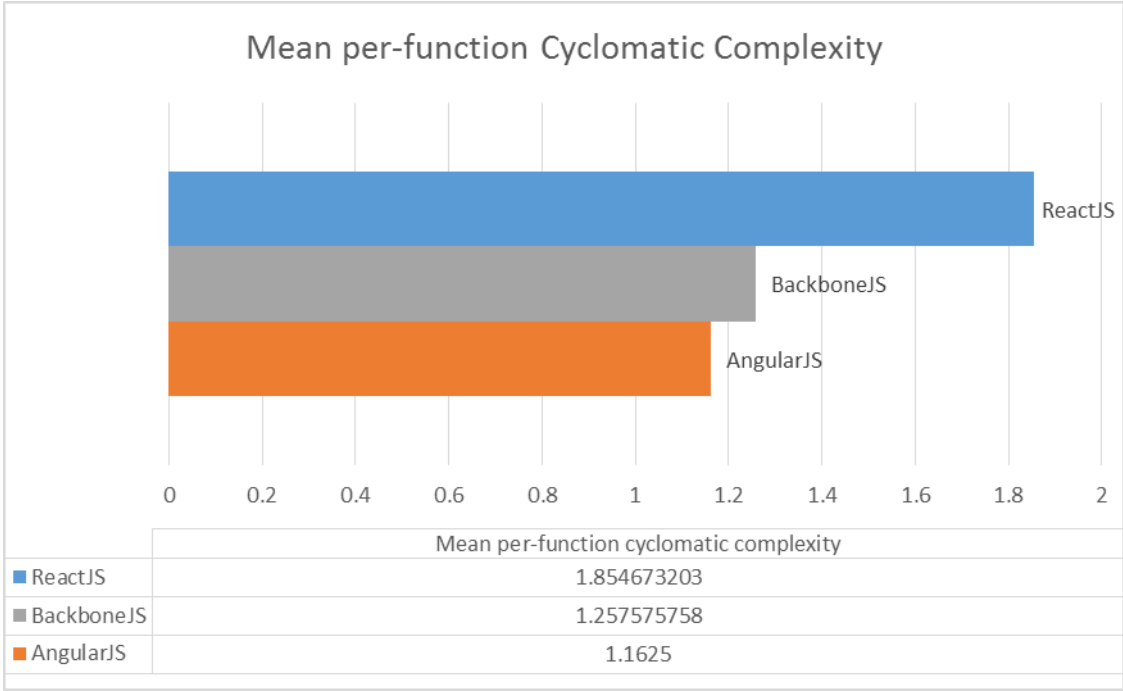
Figure 51 Figure showing Mean per-function Logical LOC

The mean per-function Logical LOC (Lines of Code) ranges from 2.2 – 6.5 logical lines per function. The highest value comes from ReactJS where the mean value of



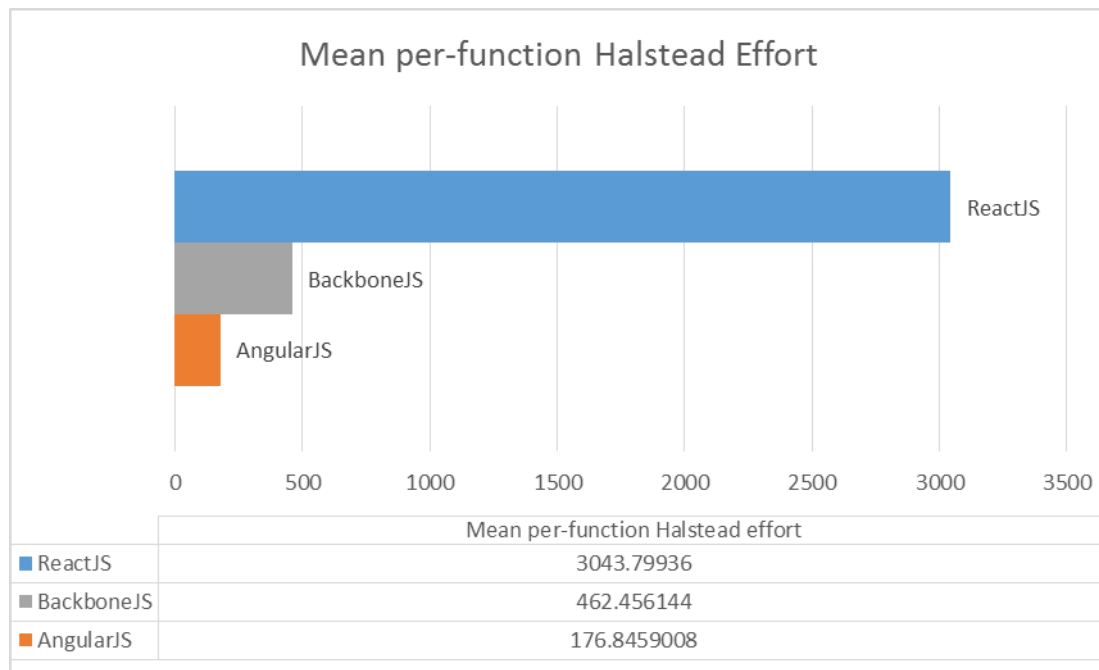
logical LOC per-function is found to be around 6.51 (corrected to two decimal places) which shows that React’s implementation of the Todo application required the most amount of code to write.

On the other hand, AngularJS looks attractive when it comes to the amount of code a developer has to write in order to develop an application. Second place comes from BackboneJS which is quite reasonable when it comes to the amount of code required to develop an application.



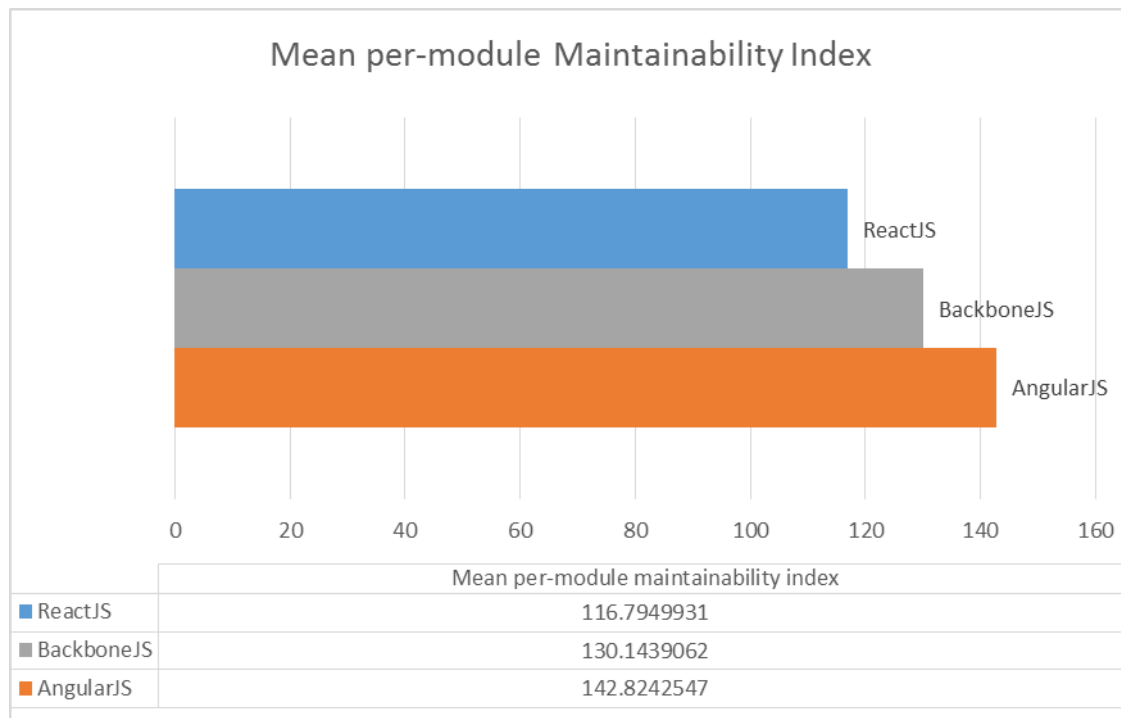
**Figure 52 Figure showing Mean per-function Cyclomatic Complexity**

The mean per-function Cyclomatic Complexity ranges from 1.16-1.85 where AngularJS yet again generating the least value in relation to its Cyclomatic Complexity and React generating the greatest value. As mentioned in Section 3.1.1, the lower the Cyclomatic Complexity is, the better.



**Figure 53 Figure showing Mean per-function Halstead Effort**

Only, the mean-per function Halstead Effort is shown as this is the only Halstead value that was generated by the software complexity tool. The difference between the lower and higher values are quite dramatic for the Halstead Effort where ReactJS implementation of the Todo application generated a whopping value of 3043.8 and AngularJS, producing the least value of 176.85 with Backbone producing a value of 462.46 which shows the major differences between all values.



**Figure 54 Figure showing Mean per-module Maintainability Index**

Finally, the mean per-module (file) Maintainability Index was generated from all three implementations of the Todo application. As was mentioned in Section 3.1.1, the values range from infinity up to 171 where higher values are considered to have better maintainability. Also, the threshold for the Maintainability Index is identified as 65 below which an application is considered difficult to maintain. From Figure 54, it can be seen that AngularJS generated the highest Maintainability Index whereas ReactJS generated the lowest value of the Maintainability Index. However, each implementation doesn't fall below the threshold, therefore, each implementation are considered easy to maintain.

### 5.2.3 Strengths and Limitations

#### Strengths

- i. The findings of this experiment have been based on a Todo application of the TodoMVC project which were contributed by experts. Therefore, each implementation of the Todo application developed in various JavaScript frameworks are the best possible implementations.

- ii. The focus of this study have been on benchmarking and the use and application of metrics as a way to compare JavaScript frameworks. A focus which is somehow lacking in a number of previous works in this field.
- iii. Results from updated JavaScript frameworks and the use of various features showed the differences in performance.

### Limitations

- i. This research is limited by the short number of JavaScript frameworks tested. The quality and the variety of the findings would be greatly increased if more frameworks were used.
- ii. The test environment used may not have been the best environment to perform the experiments. Also, it would be interesting to see performance results on other devices.
- iii. The calculation of the time using the benchmark clock presented may not have been the most accurate way of calculating the time.
- iv. The software complexity tool generated limited results as only the Halstead Effort metric was generated by the tool.
- v. The experiment regarding the DBMonster application was discontinued due to the fact that resources weren't available at the time of running the latter part of the experiments which would have produced additional viable results.

### **5.3 Chapter Summary**

This chapter describes the experiments conducted in order to compare and evaluate three JavaScript frameworks, the first of which describes the experiments conducted in order to generate the benchmark results for the Todo application where the benchmarks were run on a total of three web browsers (Chrome, Edge and Firefox). This was followed by the use of an analysis tool on each Todo implementation selected that generated the measures for the software complexity metrics. Finally, the strengths and findings of the results were enumerated.

The final chapter summarizes the whole dissertation including concluding remarks as well a discussion on other areas that may serve as future work for other researchers wanting to explore in the same direction as this project.

## **6. CONCLUSION & FUTURE WORK**

There are many JavaScript frameworks available today. As was discussed in Chapter 1, one major concern facing web developers is typically when choosing the right language or framework in order to fulfil their needs. Therefore, the problem of choosing the right framework all comes down to how well each JavaScript framework is assessed in order to give developers an insight into how well each framework performs, which this research aims to address in which metrics were selected and assessed based on the implementation of a benchmark application which ran tests on all three JavaScript frameworks selected.

### ***6.1 Research Overview***

This research carried out a comparison of three JavaScript frameworks which were carried out based on various benchmark metrics presented in Section 2.4.1. By evaluating each JavaScript framework according to these benchmark metrics, an initial comparison was carried through the use of experiments in order to measure these metrics which hoped to achieve a level of comparison and evaluation suitable for developers and researchers to look at.

Therefore, the aim of this work was to garner knowledge around the research of comparing JavaScript frameworks, the results of which allowed a quantitative assessment of the measurements of benchmark metrics by conducting various experiments. However, problems arose while working on this project due to a number of factors as discussed in the next section.

### ***6.2 Experimentation, Evaluation and Limitations***

The experimentation phase of this work was not completed to the initial level anticipated at the beginning of this project. Expectations for this project was quite high, however, it did not reach the high level of expectation due to a number of factors which reduced the amount of time given to the project which in turn, delayed the completion of milestones set beforehand.

As the majority of this research is based on writing code in JavaScript, a thorough review of documentations and practice of tutorials were conducted in order to gain a full understanding of the programming language. On top of this, there was no initial knowledge of each JavaScript framework chosen as each framework implements software slightly differently from one another. Therefore, the effort in understanding and implementing the benchmark reference applications was underestimated. This was partially due to the steep learning curve of AngularJS and ReactJS. Also, the choice of tools in order to conduct the experiments and develop the reference applications was a difficult task especially considering the vast amount of tools available. In addition, building the benchmark reference application was one way to evaluate these frameworks but it did not fully evaluate each JavaScript framework nor did it exercise the full capabilities of the frameworks. Moreover, the number of JavaScript frameworks evaluated were limited to three, however, more frameworks would be ideal to give a more balanced view of the comparison of JavaScript frameworks.

Finally, the latter part of this project was spent overseas in the Philippines due to a tragic loss in the family which required the whole family to travel to the Philippines to be reunited and spend time with other family members which greatly reduced the amount of time put in to this project. While there, it was very difficult to gain access to the internet and even if an access to the internet is granted, speeds were not feasible to conduct the experiments as connections were not constant as internet connections are being interrupted and disconnected regularly. This was due to the fact that the area where the experiments were to be conducted lacked a robust internet connection as internet connections in the Philippines are on a major upgrade which have started in 2016 which hopes to finish all upgrades and developments in the coming months or years according to the Philippine government. Therefore, a decision was made to discard the implementation and conduction of experiments regarding the DB Monster application and the measurement of the render time and speed index of each Todo implementation which would have been beneficial to the results of this project.

### ***6.3 Contributions & Impact***

The JavaScript frameworks selected are some of the most well-known client side frameworks being used by developers in the software industry these days. That is why this research aims to compare the most well-known JavaScript frameworks according to the size of its community so as to make known to other developers how each JavaScript framework perform. Moreover, the findings presented in this thesis have contributed to current research as up to date findings were presented such the performance of newer versions of JavaScript frameworks used. Furthermore, there has been numerous attempts to compare JavaScript frameworks, however, previous comparisons made between these frameworks rarely account the metrics being measured and are somewhat biased when it comes to the development of reference applications in order to test the frameworks. Also, similar research are out of date as major releases of each framework are constantly being distributed which vendors claim, to have made big improvements. Moreover, there exists only a few research that adopted software complexity metrics in their evaluation process which are deemed to be important benchmark metrics as outlined in Chapter 3.

### ***6.4 Future Work & Recommendations***

The following are recommendations and suggestions which can add to this research to gain further knowledge to the comparison of JavaScript frameworks and to encourage other developers and researchers to conduct their own research and experiments to further reinforce the evaluation methods of comparison of JavaScript frameworks. These recommendations include:

- Applying the same approach used in this research with the addition of more JavaScript frameworks would better inform developers and researchers of the performance of a wide range of JavaScript frameworks.
- Benchmarking is an effective way of evaluating JavaScript frameworks if done correctly. However, improvements can be made to the process followed in this project by further applying more benchmarking techniques available.
- A number of experiments were unable to be conducted in this research. Application of the experiments missed in this research would add viable results.

- Perhaps implement the benchmark clock differently and use Benchmark.js instead which is another alternative to the clock implemented in this research.
- Consider more benchmark metrics to further expand the comparison of JavaScript frameworks.



## BIBLIOGRAPHY

- Adamson, A., Dagastine, D., & Sarne, S. (2007). SPECjbb2005 – A year in the life of a benchmark. In: Standard Performance Evaluation Corporation (SPEC) Benchmark Workshop. *SPEC*, 1–4.
- Bulej, L., Kalibera, T., & Tuma, P. (2005). Repeated results analysis for middleware regression benchmarking. *Performance Modeling and Evaluation of High-Performance Parallel and Distributed Systems* (pp. 345–358).
- Burbeck, S. (1992). Applications programming in smalltalk-80 (tm): How to use model-view-controller (mvc). *Smalltalk-80 v2*, 5. Retrieved from [https://www.researchgate.net/profile/Steve\\_Burbeck/publication/238719652\\_Applications\\_programming\\_in\\_smalltalk-80\\_how\\_to\\_use\\_model-view-controller\\_\(mvc\)/links/5575a00508ae7536375024c7.pdf](https://www.researchgate.net/profile/Steve_Burbeck/publication/238719652_Applications_programming_in_smalltalk-80_how_to_use_model-view-controller_(mvc)/links/5575a00508ae7536375024c7.pdf)
- Calero, C., Piattini, M., & Genero, M. (2001). A case study with relational database metrics. *Proceedings ACS/IEEE International Conference on Computer Systems and Applications*, 485–487.
- Carzaniga, A., & Wolf, A. L. (2002). A Benchmark Suite for Distributed Publish/Subscribe Systems. Technical report CUCS- 927-02. *Department of Computer Science, University of Colorado*.
- Christodoulou, S. P., & Gizas, A. B. (2014). PERFORMANCE EVALUATION FRAMEWORK OF ALL CLASSES OF SELECTORS FOR JAVASCRIPT LIBRARIES. *IADIS International Journal on WWW/Internet*, 12(2). Retrieved from [http://search.ebscohost.com/login.aspx?direct=true&profile=ehost&scope=site&authType=crawler&jrnl=16457641&AN=111559093&h=q0IIPeWZ2MROxoMBKHVjiug%](http://search.ebscohost.com/login.aspx?direct=true&profile=ehost&scope=site&authType=crawler&jrnl=16457641&AN=111559093&h=q0IIPeWZ2MROxoMBKHVjiug%2F)

2Fi%2Bv0%2FBiiBi7vhEzFliBHCoyB7SwggvPIkKmwePW1Fqyy2LWRORJWr2yF  
UskUsQ%3D%3D&crl=c

Curnow, H. J., & Wichmann, B. A. (1976). A synthetic benchmark. *The Computer Journal*, 19(1).

Dixit, K. M. (1993). *Overview of the SPEC Benchmarks*. Retrieved from <http://research-srv.microsoft.com/en-us/um/people/gray/BenchmarkHandbook/chapter9.pdf>

Fenton, N. (1994). Software measurement - A necessary scientific basis. *IEEE Trans. Software Engineering*, (20(3)), 199–206.

Fernández-Villamor, J. I., Casillas, L. D., & Iglesias, C. (2008). A Comparison Model for Agile Web Frameworks. *Proceedings of the 2008 Euro American Conference on Telematics and Information Systems, EATIS '08*, 14:1–14:8.

Flanagan, D. (2006). *JavaScript: the definitive guide*. O'Reilly Media, Inc.

Fowler, M., & Foemmel, M. (2006). Continuous integration. *Thought-Works*) <Http://www.Thoughtworks.com/Continuous Integration. Pdf>, 122.

Gray, J. (1993). *The Benchmark Handbook for Database and Transaction Processing Systems* (2nd ed.). Morgan Kaufmann.

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* (Vol. 7983, pp. 334–337). Berlin, Heidelberg: Springer Berlin Heidelberg. Retrieved from [http://link.springer.com/10.1007/978-3-642-39259-7\\_28](http://link.springer.com/10.1007/978-3-642-39259-7_28)

Halstead, M. H. (1977). *Elements of Software Science*. New York: Elsevier North-Holland.

Hinnant, D. F. (1988). Accurate Unix benchmarking: Art, science, or black magic? *IEEE Micro* 8.5, 64–75.

J.D, M., Farre, C., Bansode, P., Barber, S., & Rea, D. (2011). *Reusing Open Source Code*. Microsoft Press.

- Johnson, R. E. (1997). Components, frameworks, patterns. In *ACM SIGSOFT Software Engineering Notes* (Vol. 22, pp. 10–17). ACM. Retrieved from <http://dl.acm.org/citation.cfm?id=258378>
- Kelessidis, V. (2000). BENCHMARKING. *Thessaloniki Technology Park*.
- Kim, J. S., & Hsu, Y. (2000). Memory system behavior of Java programs: methodology and analysis. In *Proceedings of the International Conference on Measurement and Modeling of Computer Systems*, 264–274.
- Lennon, J. (2010, February 2). Compare JavaScript frameworks [CT316]. Retrieved November 25, 2016, from <http://www.ibm.com/developerworks/library/wa-jsframeworks/index.html>
- Malmstrom, T. J. (2014). Structuring modern web applications: A study of how to structure web clients to achieve modular, maintainable and longlived applications. Retrieved from <http://www.diva-portal.org/smash/record.jsf?pid=diva2:753082>
- Menasce, D. A., & Almeida, V. A. (2002). Benchmarks and Performance Tests. In *Capacity Planning for Web Services: Metrics, Models, and Methods* (pp. 261–303). Prentice Hall PTR.
- Mikowski, M. S., & Powell, J. C. (2013). Single Page Web Applications. *B and W*. Retrieved from <http://deals.manningpublications.com/spa.pdf>
- Mogul, J. C. (1992). SPECmarks are leading us astray. In: Proceedings of the Third Workshop on Workstation Operating Systems (WWOS '92). *IEEE Computer Society*, 160–161.
- Molin, E. (2016). Comparison of Single-Page Application Frameworks. Retrieved from [http://www.nada.kth.se/~ann/exjobb/eric\\_molin.pdf](http://www.nada.kth.se/~ann/exjobb/eric_molin.pdf)
- Ocariza Jr, F. S., Pattabiraman, K., & Mesbah, A. (2015). Detecting inconsistencies in JavaScript MVC applications. In *Proceedings of the 37th International Conference on*

- Software Engineering-Volume 1* (pp. 325–335). IEEE Press. Retrieved from <http://dl.acm.org/citation.cfm?id=2818796>
- Oman, P., & Hagemester, J. (1992). Metrics for assessing a software system's maintainability. *Proceedings Conference on Software Maintenance*, 337–344.
- Packirisamy, V., Zhai, A., & Yew, P. (2008). Exploring Speculative Parallelism in SPEC2006. *Department of Computer Science and Engineering University of Minnesota*.
- Palmer, J. W. (2002). Web Site Usability, Design, and Performance Metrics. *Information Systems Research*, 151–167.
- Park, R. E. (1992). *Software size measurement: A framework for counting source statements*. DTIC Document. Retrieved from <http://oai.dtic.mil/oai/oai?verb=getRecord&metadataPrefix=html&identifier=ADA258304>
- Pfleeger, S. L. (1995). Experimental design and analysis in software engineering. *Annals of Software Engineering*, 219–253.
- Ratanaworabhan, P., Livshits, B., & Zorn, B. G. (2010). JSMeter: Comparing the Behavior of JavaScript Benchmarks with Real Web Applications. *WebApps*, 10, 3–3.
- Reenskaug, T. M. H. (1979). The original MVC reports. Retrieved from <https://www.duo.uio.no/handle/10852/9621>
- Rentrop, J. (2006). *Software Metrics as Benchmarks for Source Code Quality of Software Systems*. University of Amsterdam.
- Richards, G., Lebresne, S., Burg, B., & Vitek, J. (2010). An analysis of the dynamic behavior of JavaScript programs. In *ACM Sigplan Notices* (Vol. 45, pp. 1–12). ACM. Retrieved from <http://dl.acm.org/citation.cfm?id=180659>

- Rosenberg, J. (1997). Some Misconceptions About Lines of Code. *Fourth International Software Metrics Symposium*, 137.
- Saavedra-Barrera, R. H., Gaines, R. S., & Carlton, M. J. (1993). Characterizing the performance space of shared memory computers using Micro-Benchmarks. Technical report USC-CS-93-547. *Technical Report USC-CS-93-547. Department of Computer Science, University of Southern California, CA, USA.*
- Sachs, K. (2011). Performance Modeling and Benchmarking of Event- Based Systems. *PhD Thesis. TU Darmstadt, Germany.*
- Salas-Zárate, M. P., Hernández, G. A., García, R. V., Mazahua, L. R., González, A. R., & Cuadrado, J. L. (2015). Analyzing best practices on web development frameworks: The lift approach. *Science of Computer Programming*, (102), 1 – 19.
- Samoladas, I., Stamelos, I., Angelis, L., & Oikonomou, A. (2004). Open source software development should strive for even greater code maintainability. *Commun. ACM*, 47(10), 83–87.
- Schmidt, D. C., & Buschmann, F. (2003). Patterns, frameworks, and middleware: their synergistic relationships. In *Software Engineering, 2003. Proceedings. 25th International Conference on* (pp. 694–704). IEEE. Retrieved from [http://ieeexplore.ieee.org/xpls/abs\\_all.jsp?arnumber=1201256](http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1201256)
- Seltzer, M., Krinsky, D., Smith, K., & Zhang, X. (1999). The case for application-specific benchmarking. In: *Proceedings of the Seventh Workshop on Hot Topics in Operating Systems (HotOS-VII)*. *IEEE Computer Society*, 102–107.
- Sim, S. E., Easterbrook, S., & Holt, R. C. (2003). Using benchmarking to advance research: A challenge to software engineering. In *Proceedings of the 25th International Conference on Software Engineering* (pp. 74–83). IEEE Computer Society. Retrieved from <http://dl.acm.org/citation.cfm?id=776826>

- Stępnia, W., & Nowak, Z. (2016). Performance Analysis of SPA Web Systems. *Springer International Publishing*, 521, 235–247.
- Tichy, W. F. (2014). Ubiquity symposium: The science in computer science - Where's the science in software engineering? *Ubiquity*.
- Timeline Event Reference | Web. (n.d.). Retrieved December 15, 2016, from <https://developers.google.com/web/tools/chrome-devtools/evaluate-performance/performance-reference>
- Trostler, M. E. (2013). *Testable JavaScript* (1st ed). Sebastopol, CA: O'Reilly Media.
- Unicode Consortium (Ed.). (1996). *The Unicode standard* (Version 2.0). Reading, Mass: Addison-Wesley Developers Press.
- Utting, M., & Legeard, B. (2007). Practical Model-Based Testing – A Tools Approach. (p. 33, 34).
- Vieira, M., Madeira, H., Sachs, K., & Kounev, S. (2012). Resilience Benchmarking. In: Resilience Assessment and Evaluation of Computing Systems. *Springer*, 283–301.
- Waller, J. (2015). *Performance Benchmarking of Application Monitoring Frameworks*. BoD–Books on Demand. Retrieved from [http://books.google.com/books?hl=en&lr=&id=qsErBgAAQBAJ&oi=fnd&pg=PA1&dq=%22necessary+to+know+the+performance+impact+of+each%22+%22analysis+of%22+%22feasibility+and+practicality+of+the+approach+and+validate+the%22+%22futile.+Repeatability+of+scienti%EF%AC%81c+experiments+is+essential+to%22+%22ots=hGfcHQVoFg&sig=rwKE\\_7W7zcoHsi94fAckvovUAI4](http://books.google.com/books?hl=en&lr=&id=qsErBgAAQBAJ&oi=fnd&pg=PA1&dq=%22necessary+to+know+the+performance+impact+of+each%22+%22analysis+of%22+%22feasibility+and+practicality+of+the+approach+and+validate+the%22+%22futile.+Repeatability+of+scienti%EF%AC%81c+experiments+is+essential+to%22+%22ots=hGfcHQVoFg&sig=rwKE_7W7zcoHsi94fAckvovUAI4)
- Weicker, R. P. (1984). Dhrystone: a synthetic systems programming benchmark. *Communications of the ACM*, 27(10), 1013–1030.

## APPENDIX A: BENCHMARK SCRIPTS SOURCE CODE

### resources/tests.js

```
var numberOfItemsToAdd = 100;
var Suites = [];
Suites.push({
  name: 'BackboneJS',
  url: 'todomvc/backbone/index.html',
  version: '1.3.3',
  prepare: function (runner, contentWindow, contentDocument) {
    //The contentWindow property returns the Window object
    //generated by an iframe element (through the window object,
    //you can access the document object and then any one of the
    //document's elements).
    contentWindow.Backbone.sync = function () {}
    return runner.waitForElement('.new-todo').then(function
(element) {
      element.focus();
      return element;
    });
  },
  tests: [
    new BenchmarkTestStep('Adding' + numberOfItemsToAdd +
'Items', function (newTodo, contentWindow, contentDocument) {
      var appView = contentWindow.appView;
      for (var i = 0; i < numberOfItemsToAdd; i++) {
        var inputEvent = document.createEvent('Event');
        inputEvent.initEvent('input', true, true);
        newTodo.value = 'Something to do ' + i;
        newTodo.dispatchEvent(inputEvent);

        var keypressEvent = document.createEvent('Event');
        keypressEvent.initEvent('keypress', true, true);
        keypressEvent.which = 13; // VK_ENTER
        newTodo.dispatchEvent(keypressEvent);
      }
    }),
    new BenchmarkTestStep('CompletingAllItems', function
(newTodo, contentWindow, contentDocument) {
      var checkboxes =
contentDocument.querySelectorAll('.toggle');
      for (var i = 0; i < checkboxes.length; i++)
        checkboxes[i].click();
    }),
    new BenchmarkTestStep('DeletingAllItems', function (newTodo,
contentWindow, contentDocument) {
      var deleteButtons =
contentDocument.querySelectorAll('.destroy');
      for (var i = 0; i < deleteButtons.length; i++)
        deleteButtons[i].click();
    })
  ]
});

Suites.push({
  name: 'AngularJS',
  url: 'todomvc/angularjs/index.html',
```

```

    version: '1.6.0',
    prepare: function (runner, contentWindow, contentDocument) {
        return runner.waitForElement('#new-todo').then(function
(element) {
            element.focus();
            return element;
        });
    },
    tests: [
        new BenchmarkTestStep('Adding' + numberOfItemsToAdd +
'Items', function (newTodo, contentWindow, contentDocument) {
            for (var i = 0; i < numberOfItemsToAdd; i++) {
                var inputEvent = document.createEvent('Event');
                inputEvent.initEvent('input', true, true);
                newTodo.value = 'Something to do ' + i;
                newTodo.dispatchEvent(inputEvent);

                var submitEvent = document.createEvent('Event');
                submitEvent.initEvent('submit', true, true);
                newTodo.form.dispatchEvent(submitEvent);
            }
        }),
        new BenchmarkTestStep('CompletingAllItems', function
(newTodo, contentWindow, contentDocument) {
            var checkboxes =
contentDocument.querySelectorAll('.toggle');
            for (var i = 0; i < checkboxes.length; i++)
                checkboxes[i].click();
        }),
        new BenchmarkTestStep('DeletingAllItems', function (newTodo,
contentWindow, contentDocument) {
            var deleteButtons =
contentDocument.querySelectorAll('.destroy');
            for (var i = 0; i < deleteButtons.length; i++)
                deleteButtons[i].click();
        })
    ]
});
Suites.push({
    name: 'React-noJSX',
    url: 'todomvc/react/index.html',
    version: '15.4.0',
    prepare: function (runner, contentWindow, contentDocument) {
        contentWindow.Utils.store = function () {}
        return runner.waitForElement('.new-todo').then(function
(element) {
            element.focus();
            return element;
        });
    },
    tests: [
        new BenchmarkTestStep('Adding' + numberOfItemsToAdd +
'Items', function (newTodo, contentWindow, contentDocument) {
            for (var i = 0; i < numberOfItemsToAdd; i++) {
                var inputEvent = document.createEvent('Event');
                inputEvent.initEvent('input', true, true);
                newTodo.value = 'Something to do ' + i;
                newTodo.dispatchEvent(inputEvent);

                var keydownEvent = document.createEvent('Event');
                keydownEvent.initEvent('keydown', true, true);

```



```

        keydownEvent.keyCode = 13; // VK_ENTER
        newTodo.dispatchEvent(keydownEvent);
    }
    }),
    new BenchmarkTestStep('CompletingAllItems', function
(newTodo, contentWindow, contentDocument) {
    var checkboxes =
contentDocument.querySelectorAll('.toggle');
    for (var i = 0; i < checkboxes.length; i++)
        checkboxes[i].click();
    }),
    new BenchmarkTestStep('DeletingAllItems', function (newTodo,
contentWindow, contentDocument) {
    var deleteButtons =
contentDocument.querySelectorAll('.destroy');
    for (var i = 0; i < deleteButtons.length; i++)
        deleteButtons[i].click();
    })
    ]
});

Suites.push({
    name: 'React-JSX',
    url: 'todomvc/react-es2015/index.html',
    version: '15.4.0',
    prepare: function (runner, contentWindow, contentDocument) {
        // contentWindow.Utils.store = function () {}
        return runner.waitForElement('.new-todo').then(function
(element) {
            element.focus();
            return element;
        });
    },
    tests: [
        new BenchmarkTestStep('Adding' + numberOfItemsToAdd +
'Items', function (newTodo, contentWindow, contentDocument) {
            for (var i = 0; i < numberOfItemsToAdd; i++) {
                var inputEvent = document.createEvent('Event');
                inputEvent.initEvent('input', true, true);
                newTodo.value = 'Something to do ' + i;
                newTodo.dispatchEvent(inputEvent);

                var keydownEvent = document.createEvent('Event');
                keydownEvent.initEvent('keydown', true, true);
                keydownEvent.keyCode = 13; // VK_ENTER
                newTodo.dispatchEvent(keydownEvent);
            }
        }),
        new BenchmarkTestStep('CompletingAllItems', function
(newTodo, contentWindow, contentDocument) {
            var checkboxes =
contentDocument.querySelectorAll('.toggle');
            for (var i = 0; i < checkboxes.length; i++)
                checkboxes[i].click();
        }),
        new BenchmarkTestStep('DeletingAllItems', function (newTodo,
contentWindow, contentDocument) {
            var deleteButtons =
contentDocument.querySelectorAll('.destroy');
            for (var i = 0; i < deleteButtons.length; i++)
                deleteButtons[i].click();
        })
    ]
});

```

```

    })
  ]
});

```

## resources/manager.js

```

var runs = [],
    res = document.getElementById('results'),
    timesRan = 0,
    runButton

function formatTestName(suiteName, testName) {
    return suiteName + (testName ? '/' + testName : '');
}

function createUIForSuites(suites, onstep, onrun) {
    var control = document.createElement('div');
    var ol = document.createElement('ol');
    var checkboxes = [];

    /* var button = document.createElement('button');
    button.textContent = 'Step Tests';
    button.onclick = onstep;
    control.appendChild(button); */

    var button = runButton = document.createElement('button');
    button.textContent = 'Run All';
    button.onclick = onrun;
    control.appendChild(button);

    for (var suiteIndex = 0; suiteIndex < suites.length;
suiteIndex++) {
        var suite = suites[suiteIndex];
        var li = document.createElement('li');
        var checkbox = document.createElement('input');
        checkbox.id = suite.name;
        checkbox.type = 'checkbox';
        checkbox.checked = true;
        checkbox.onchange = (function (suite, checkbox) { return
function () { suite.disabled = !checkbox.checked; runs = []; }
})(suite, checkbox);
        checkbox.onchange();
        checkboxes.push(checkbox);

        li.appendChild(checkbox);
        var label = document.createElement('label');

        label.appendChild(document.createTextNode(formatTestName(suite.name)
+ ' ' + suite.version));
        li.appendChild(label);
        label.htmlFor = checkbox.id;

        var testList = document.createElement('ol');
        for (var testIndex = 0; testIndex < suite.tests.length;
testIndex++) {
            var testItem = document.createElement('li');
            var test = suite.tests[testIndex];

```

```

        var anchor = document.createElement('a');
        anchor.id = suite.name + '-' + test.name;
        test.anchor = anchor;

    anchor.appendChild(document.createTextNode(formatTestName(suite.name,
    test.name)));
        testItem.appendChild(anchor);
        testList.appendChild(testItem);
    }
    li.appendChild(testList);

    ol.appendChild(li);
}

control.appendChild(ol);

return control;
}

function startTest() {

    var match = window.location.search.match(/\[?&]r=(\d+)/),
        timesToRun = match ? +(match[1]) : 1

    var runner = new BenchmarkRunner(Suites, {
        willRunTest: function (suite, test) {
            if (!navigator.userAgent.match("MSIE 9.0"))
test.anchor.classList.add('running');
        },
        didRunTest: function (suite, test) {
            var classList = test.anchor.classList;
            if (!navigator.userAgent.match("MSIE 9.0"))
classList.remove('running');
            if (!navigator.userAgent.match("MSIE 9.0"))
classList.add('ran');
        },
        didRunSuites: function (measuredValues) {
            var results = '';
            var total = 0;
            for (var suiteName in measuredValues) {
                var suiteResults = measuredValues[suiteName];
                for (var testName in suiteResults.tests) {
                    var testResults = suiteResults.tests[testName];
                    for (var subtestName in testResults) {
                        results += suiteName + ' : ' + testName + ' :
' + subtestName
                        + ': ' + testResults[subtestName] + '
ms\n';
                    }
                }
                results += suiteName + ' : ' + suiteResults.total + '
ms\n';
                total += suiteResults.total;
            }
            results += 'Run ' + (runs.length + 1) + '/' + timesToRun +
' - Total : ' + total + ' ms\n';

            if (!results)
                return;

            console.log(results)
        }
    });
}

```

```

        runs.push(measuredValues)
        timesRan++
        if (timesRan >= timesToRun) {
            timesRan = 0
            reportFastest()
            shuffle(Suites);
        } else {
            setTimeout(function () {
                runButton.click()
            }, 0)
        }
    }
}); //end runner

var currentState = null;
function callNextStep(state) {
    runner.step(state).then(function (newState) {
        currentState = newState;
        if (newState)
            callNextStep(newState);
    });
}

// Don't call step while step is already executing.
document.body.appendChild(createUIForSuites(Suites,
    function () { runner.step(currentState).then(function (state)
{ currentState = state; }); },
    function () {
        var analysis = document.getElementById("analysis");
        analysis.style.display = 'none';
        localStorage.clear();

        callNextStep(currentState);
    }));
function reportFastest () {
    var results = {}
    runs.forEach(function (runData) {
        for (var key in runData) {
            results[key] = Math.min(results[key] || Infinity,
runData[key].total)
        }
    });
    drawChart(results);
}
} //end startTest

google.load("visualization", "1", {packages:["corechart"]});
function drawChart(results) {
    var rawData = [];
    for (var key in results) {
        var color = colorify(key);
        rawData.push([ key, Math.round(results[key]), color ]);
    }
    rawData.sort(function(a, b){ return a[1] - b[1] })
    rawData.unshift([ "Project" , "Time", { role: "style" } ])
    var data = google.visualization.arrayToDataTable(rawData);

    var view = new google.visualization.DataView(data);
    view.setColumns([0, 1,
        { calc: "stringify",

```

```

        sourceColumn: 1,
        type: "string",
        role: "annotation" },
    2]);

var runWord = "run" + (runs.length > 1 ? "s" : "");
var title = "Best time in milliseconds over " + runs.length +
    " " + runWord + " (lower is better)";

var options = {
    title: "TodoMVC Benchmark",
    width: 600,
    height: 400,
    legend: { position: "none" },
    backgroundColor: 'transparent',
    hAxis: {title: title},
    min: 0,
    max: 1500
};
var analysis = document.getElementById("analysis");
analysis.style.display = 'block';
var barchart = document.getElementById("barchart_values");
var chart = new google.visualization.BarChart(barchart);
chart.draw(view, options);
}

function shuffle ( ary ) {
    var i = ary.length;
    if ( i == 0 ) return false;
    while ( --i ) {
        var j = Math.floor( Math.random() * ( i + 1 ) );
        var tempi = ary[i];
        var tempj = ary[j];
        ary[i] = tempj;
        ary[j] = tempi;
    }
}

function colorify(n){
    var c = 'rgb(' + ( Math.max(0, (n.toLowerCase().charCodeAt(3 %
n.length) - 97) / 26 * 255 | 0) ) +
        ", " + ( Math.max(0, (n.toLowerCase().charCodeAt(4 %
n.length) - 97) / 26 * 255 | 0) ) +
        ", " + ( Math.max(0, (n.toLowerCase().charCodeAt(5 %
n.length) - 97) / 26 * 255 | 0) ) + " )"
    return c
}

window.addEventListener('load', startTest);

```

## resources/benchmark-runner.js

```
function SimplePromise() {
    this._chainedPromise = null;
    this._callback = null;
}

SimplePromise.prototype.then = function (callback) {
    if (this._callback)
        throw "SimplePromise doesn't support multiple calls to then";
    this._callback = callback;
    this._chainedPromise = new SimplePromise;

    if (this._resolved)
        this.resolve(this._resolvedValue);

    return this._chainedPromise;
}

SimplePromise.prototype.resolve = function (value) {
    if (!this._callback) {
        this._resolved = true;
        this._resolvedValue = value;
        return;
    }

    var result = this._callback(value);
    if (result instanceof SimplePromise) {
        var chainedPromise = this._chainedPromise;
        result.then(function (result) {
            chainedPromise.resolve(result);
        });
    } else {
        this._chainedPromise.resolve(result);
    }
}

function BenchmarkTestStep(testName, testFunction) {
    this.name = testName;
    this.run = testFunction;
}

function BenchmarkRunner(suites, client) {
    this._suites = suites;
    this._prepareReturnValue = null;
    this._measuredValues = {};
    this._client = client;
}

BenchmarkRunner.prototype.waitForElement = function (selector) {
    var promise = new SimplePromise;
    var contentDocument = this._frame.contentDocument;

    function resolveIfReady() {
        var element = contentDocument.querySelector(selector);
        if (element)
            return promise.resolve(element);
        setTimeout(resolveIfReady, 50);
    }

    resolveIfReady();
}
```

```

    return promise;
}

BenchmarkRunner.prototype._removeFrame = function () {
    if (this._frame) {
        this._frame.parentNode.removeChild(this._frame);
        this._frame = null;
    }
}

BenchmarkRunner.prototype._appendFrame = function (src) {
    var frame = document.createElement('iframe');
    frame.style.width = '800px';
    frame.style.height = '600px';
    document.body.appendChild(frame);
    this._frame = frame;
    return frame;
}

BenchmarkRunner.prototype._waitAndWarmUp = function () {
    var startTime = Date.now();

    function Fibonacci(n) {
        if (Date.now() - startTime > 100)
            return;
        if (n <= 0)
            return 0;
        else if (n == 1)
            return 1;
        return Fibonacci(n - 2) + Fibonacci(n - 1);
    }

    var promise = new SimplePromise;
    setTimeout(function () {
        Fibonacci(100);
        promise.resolve();
    }, 200);
    return promise;
}

// This function ought be as simple as possible. Don't even use
// SimplePromise.
BenchmarkRunner.prototype._runTest = function(suite, testFunction,
prepareReturnValue, callback)
{
    var now = window.performance && window.performance.now ?
    function ()
    { return window.performance.now(); } : Date.now;

    var contentWindow = this._frame.contentWindow;
    var contentDocument = this._frame.contentDocument;

    var startTime = now();
    testFunction(prepareReturnValue, contentWindow, contentDocument);
    var endTime = now();
    var syncTime = endTime - startTime;

    var startTime = now();
    setTimeout(function () {
        setTimeout(function () {

```

```

        var endTime = now();
        callback(syncTime, endTime - startTime);
    }, 0)
    }, 0);
}

function BenchmarkState(suites) {
    this._suites = suites;
    this._suiteIndex = -1;
    this._testIndex = 0;
    this.next();
}

BenchmarkState.prototype.currentSuite = function() {
    return this._suites[this._suiteIndex];
}

BenchmarkState.prototype.currentTest = function () {
    var suite = this.currentSuite();
    return suite ? suite.tests[this._testIndex] : null;
}

BenchmarkState.prototype.next = function () {
    this._testIndex++;

    var suite = this._suites[this._suiteIndex];
    if (suite && this._testIndex < suite.tests.length)
        return this;

    this._testIndex = 0;
    do {
        this._suiteIndex++;
    } while (this._suiteIndex < this._suites.length &&
this._suites[this._suiteIndex].disabled);

    return this;
}

BenchmarkState.prototype.isFirstTest = function () {
    return !this._testIndex;
}

//Prepares frameworks from suite array
BenchmarkState.prototype.prepareCurrentSuite = function (runner,
frame) {
    var suite = this.currentSuite();
    var promise = new SimplePromise;
    //Iframe onload event for todo app
    //A window has onload event which fires when it is loaded
    completely
    frame.onload = function () {
        //From Suite Array (tests.js)
        suite.prepare(runner,
frame.contentWindow,
frame.contentDocument).then(function (result) {
promise.resolve(result); });
    }

    //get url from Suite array
    frame.src = suite.url;
    return promise;
}

```



```

BenchmarkRunner.prototype.step = function (state) {
    if (!state)
        state = new BenchmarkState(this._suites);

    var suite = state.currentSuite();
    if (!suite) {
        this._finalize();
        var promise = new SimplePromise;
        promise.resolve();
        return promise;
    }

    if (state.isFirstTest()) {
        this._measuredValuesForCurrentSuite = {};
        var self = this;
        return state.prepareCurrentSuite(this,
this._appendFrame()).then(function (prepareReturnValue) {
            self._prepareReturnValue = prepareReturnValue;
            return self._runTestAndRecordResults(state);
        });
    }

    return this._runTestAndRecordResults(state);
}

BenchmarkRunner.prototype._runTestAndRecordResults = function (state)
{
    var promise = new SimplePromise;
    var suite = state.currentSuite();
    var test = state.currentTest();

    if (this._client && this._client.willRunTest)
        this._client.willRunTest(suite, test);

    var self = this;
    setTimeout(function () {
        self._runTest(suite, test.run, self._prepareReturnValue,
function (syncTime, asyncTime) {
            var suiteResults = self._measuredValues[suite.name] ||
{tests:{}, total: 0};
            self._measuredValues[suite.name] = suiteResults;
            suiteResults.tests[test.name] = {'Sync': syncTime,
'Async': asyncTime};
            suiteResults.total += syncTime + asyncTime;

            if (self._client && self._client.willRunTest)
                self._client.didRunTest(suite, test);

            state.next();
            if (state.currentSuite() != suite)
                self._removeFrame();
            promise.resolve(state);
        });
    }, 0);
    return promise;
}

BenchmarkRunner.prototype._finalize = function () {
    this._removeFrame();
}

```

```
    if (this._client && this._client.didRunSuites)
        this._client.didRunSuites(this._measuredValues);

    // FIXME: This should be done when we start running tests.
    this._measuredValues = {};
}
```