



Measuring Maintainability and latency of Node.js frameworks.

Sabry Kadi

Faculty of Computing
Blekinge Institute of Technology
SE-371 79 Karlskrona Sweden

This thesis is submitted to the Faculty of Computing at Blekinge Institute of Technology in partial fulfilment of the requirements for the bachelor's degree in software engineering. The thesis is equivalent to 10 weeks of full-time studies

Contact information:

Author:

Sabry Kadi

Sabry_kadi@hotmail.com

University advisor:

Emil Folino

Department of Computer Science

Faculty of Computing
Blekinge Institute of Technology
SE-371 79 Karlskrona, Sweden

Internet: www.bth.se
Phone: +46 455 38 50 00
Fax: +46 455 38 50 57

ABSTRACT

Context: Node.js is an established web framework built using JavaScript. As a result, there are a wide variety of frameworks that have emerged that specialise in different quality attributes and functionalities. Some of which are heavily geared to performance and benchmarking while other might focus on security, availability, robustness etc.

Objectives: The project aims to explore different Node.js server-side frameworks and determine their maintainability using metrics such as Halstead metrics, Maintainability index, source line of code as well as Logical source lines of code. This thesis also explores if there is a correlation between the quality attributes maintainability and performance.

Realisation: In order to explore the different quality attributes, the thesis relied upon experiments and a literature review. The hierarchical method in this thesis was first to examine their performance, later examine their overall maintainability. Examined is also the impact of comments and how they can affect the results of the maintainability index

Results: The results indicate all the selected frameworks have a low-to borderline medium cyclomatic complexity, also a high degree of maintainability using two different 3 metric maintainability index formulas. The latency tests indicate the different framework produce similar performance results.

Conclusion:

Concluded in this thesis is, there seem to be no relationship between both lines of code , logical lines of code and cyclomatic complexity. There also seem to be no correlation between Halstead volume and the overall maintainability index for both the 3 metric formulas used. The performance results indicate that the latency test is isolated from maintainability, where cyclomatic complexity, maintainability index, Halstead volume etc do not affect the latency of the framework.

However, there seem to be a relationship between Halstead Effort and Cyclomatic Complexity using one of the 3 metric formulas i.e., as the cyclomatic complexity decreases the overall maintainability (using Halsted's effort instead of Halstead's volume) increases.

Keywords: Maintainability index, Latency, Node JS,
Maintainability, Halstead, Cyclomatic Complexity

TABLE OF CONTENTS

<i>Abstract</i> -----	3
<i>Table of contents</i> -----	4
1. <i>Introduction</i> -----	6
2. <i>Background</i> -----	7
2.1 <i>Inclusion criteria</i> -----	7
2.2 <i>Purpose</i> -----	8
2.3 <i>Scope And Limitations</i> -----	8
2.4 <i>What is maintainability?</i> -----	9
2.5 <i>Measuring maintainability</i> -----	10
2.6 <i>Maintainability index metrics</i> -----	11
2.7 <i>Modifiability</i> -----	14
2.8 <i>Correlation between Maintainability and Modifiability</i> -----	14
2.9 <i>Performance</i> -----	15
3. <i>Research Questions</i> -----	16
3.1 <i>Research Question 1</i> -----	16
3.2 <i>Research Question 2</i> -----	16
3.3 <i>Research Question 3</i> -----	17
3.4 <i>Research Question 4</i> -----	17
4. <i>Maintainability index formulas</i> -----	18
5. <i>Research Method</i> -----	20
5.1 <i>Selected Frameworks</i> -----	20
5.2 <i>Method for measuring maintainability index (RQ2)</i> -----	20
5.3 <i>Performance measuring method (RQ1)</i> -----	23
5.4 <i>Test details</i> -----	24
5.5 <i>Autocannon Configuration</i> -----	26
6. <i>Literature Review Method(RQ4)</i> -----	27
6.1 <i>Input for RQ4</i> -----	27
6.2 <i>Validation template</i> -----	27
6.3 <i>Scientific research and papers</i> -----	28
6.4 <i>literature review constraints</i> -----	29
7. <i>Results of literature review</i> -----	30
8. <i>Performance Test results</i> -----	35
9. <i>Results of literature review RQ4</i> -----	39

10.	<i>Maintainability index Results</i>	40
11.	<i>Analysis of the performance test</i>	41
12.	<i>Analysis of the Maintainability index results</i>	44
13.	<i>Conclusion</i>	47
14.	<i>Validity Threats</i>	48
15.	<i>Future Work</i>	49
	<i>References</i>	50
	<i>Appendix</i>	58
	<i>Code Files</i>	59

1. INTRODUCTION

Web Frameworks provide a foundation in which the system can be used to be developed under [1]. The Web framework can include classes, predefined code segments for specific actions, functions, and different processing helper modules that simplifies the development process. The different modules both help maintain the source code but also offer benefits such as faster development times and some Webb framework even provides pretested code [1][2][3]. This in the end, streamlines the production almost in all areas of development.

Webb Frameworks can generally be divided into two groups, client-side / front-end frameworks and server-side/backend frameworks [1]. The backend Webb-frameworks can be used to implement CRUD, Sockets, Security operations as well as End point creations among various other features, front-end frameworks on the other hand can help rendering processes, design/layout, parsing, data processing [4].

This thesis will focus on examining a set of server-side Webb frameworks and measure their Maintainability and performance.

2. BACKGROUND

2.1 INCLUSION CRITERIA

The pool of existing Node.js frameworks are extensive, Different developing entities aim to improve different areas in both simplifying the development processes and at time tune the framework towards special quality attributes.

Including every single framework is a difficult task, therefore the selected frameworks will be selected based on the inclusion criteria and a maximum of 8 frameworks will be studied and explored further.

The inclusion criteria are a composition of the framework in which the framework might be relevant to the study.

1. Must be a Node.JS backend framework
2. Must be open source under MIT or similar license
3. Must support routing.
4. Must be relevant in terms of updates, the framework must have been updated within the time span of maximum of 2 years (September 2019)
5. Must be natively written in JavaScript
6. Popularity

2.1.1 INCLUSION CRITERIA MOTIVATION

1. This thesis aims to examine frameworks/libraries that are built upon the Node.js environment. This is to limit deviations or impacts caused by different environments.
2. The web frameworks must be altered in this thesis such as removing comments and files etc, therefore for this possibility frameworks need to be open source.
3. Routing is defined as “...Routing refers to determining how an application responds to a client request to a particular endpoint, which is a URI (or path) and a specific HTTP request method (GET, POST, and so on).”¹ The frameworks selected will make use of traditional HTTP request methods, therefore the frameworks must be able to process HTTP methods.
4. Stale or outdated frameworks are not relevant to this study, this is due to maintainability index measures the simplicity of applying maintenance activities on a system. Essentially, if a system does not need to be updated, it is perhaps counterproductive to measure its maintainability.

¹ <https://expressjs.com/en/starter/basic-routing.html>

5. The reason is to limit differences and impacts made by different programming languages. In essence, different programming language can vary in implementation therefore this can impact complexities and lines of code i.e. factors that developers may not be able to control.
6. The Node.js frameworks must have an established user-base, this reason is selected arbitrarily. However, its linked to the notion of the more userbase the more frequent the source code will be up to date.

2.2 PURPOSE

There aren't any industry guidelines for selecting frameworks; rather the general selection process is narrowed down to a set of considerations, such as quality attributes like scalability and learnability or perhaps the general effort it will take to understand the framework [1]. In my understanding there have been very few and limited server-side evaluation of frameworks and how well different frameworks operate on the server side.

Another aspect of evaluating frameworks is to examine how the system/systems are susceptible to change. A great deal of the overall cost of development over the system life cycle is postproduction. One of the factors that drive the postproduction cost is how maintainable the system inherently is [5][6][7]

This thesis aims to explore the relationship between performance in terms of latency and the overall maintainability of the system. This thesis also includes various metrics that could be used to determine the system maintainability and the effects/impacts different metrics have on the outcome.

2.3 SCOPE AND LIMITATIONS

Because of the time and the complexity, this thesis will be limited to a total of eight Node.js server-side frameworks. There will be no emphasise on budling a client-side page and connecting the system to databases. The implementation of The REST API endpoints must adhere to the documentation provided by the developing entity.

The maintainability index will be measured using various measuring tools. Files such as readme's, test scripts, package-JSON's, node modules etc will not be included in the evaluation of the framework. The selected tools all support the ability to not calculate certain files and folders from the calculations.

As eluded by in the inclusion criteria, The framework must have had updates relatively recent. However, updates to documentation or tests etc does not count as an update to the framework. Rather, an update is defined in this instance, as an update to the source files that adheres directly to the framework functionalities.

2.4 WHAT IS MAINTAINABILITY?

Maintainability is quality attribute defined as “...degree of effectiveness and efficiency with which a product or system can be modified to improve it, correct it or adapt it to changes in environment, and in requirements” [12]

according to both *ISO/IEC/IEEE 14764* and *ISO 25010* standards. Fundamentally, maintainability is the ease a software can be maintained, in addition to how easy it is for the developers to alter, remove or enhance the system

Maintainability measurements and metrics allows the organizations forecast and asses the quality of the system. Explicitly, if a system is built to have a high degree of maintainability it can habitually translate into low maintenance efforts. Effort can be both in physical time for the developing team to enhance, remove or alter features or in terms of financial cost.[22] Both of which aims to decrease the postproduction costs.

According to *ISO/IEC/IEEE 14764*, there are several types of activities for maintenance Examples as such are:

- **Corrective maintenance:** Maintenance to detect and/or correct bugs, errors or faults in the system. [25]
- **Adaptive maintenance:** Maintenance to adapt to environmental factors such as new hardware, operating systems or even rules and regulations.[25]
- **Perfective maintenance:** Maintenance for software evolution, i.e., introduction of new features and enhancements but also removal or altering of existing features. [25]
- **Preventive maintenance:** Assessment and maintenance of the software to enhance the life span of the system, i.e., updating documentation, optimizing the source code etc. this is to detect and mitigate faults in hopes to prevent them becoming operational errors. [25]

The subsidiary benefits of high degree of maintainability are also the system’s ability to become adaptable, giving the organization the possibility to alter the system according to environmental factors (such as the market, rules and regulations, requirements etc.), but also to introduce new features to the system itself. Hence, having a high degree of maintainability poses advantages that in many cases do in fact translate to reduced efforts in the long run. [22]

2.5 MEASURING MAINTAINABILITY

2.5.1 DESIGN TIME

There are several techniques for determining maintainability. Maintainability is a quality attribute that is often measured during the design stages of the development [13] A case in point being to identify low cohesive areas which can be a sign of bad design or perhaps even complicated areas of the system that need to be “untangled”. [16].

Decreasing “module dependencies” i.e., decreasing module coupling [17] even measuring consistency of the documentation (for instances does the code conform to the documentation or how much deviation is found between the documentation and the code) are all methodologies of *preventative* measures that inherently aim to prevent design “error” and/or decisions that later might possibly require refactoring, or if left untreated yield a lower degree of maintainability. [13]

2.5.2 POST DEVELOPMENT

There are studies indicating there are conclusions to be drawn between quantitatively mathematical measurements i.e., code metrics and maintainability of the code. [13][19][20][21].

However, some studies have criticized code metrics for being questionable due various factors, one of which is the requirement of having a code base built [13] to even attempt to gauge the maintainability of the system during or post development. Unlike design time, prediction techniques can be employed at various stages of development, whereas statistical evaluations require input in form of a code base. Using SLOC as the **only** tool for gauging maintainability might not translate to a factor for measuring maintenance, this notion is further described in section 2.6.

2.5.3 MAINTAINABILITY INDEX

The maintainability index is a metric that measures the maintainability of a system, The metric incorporates Cyclomatic complexity, Lines of code, and Halstead metrics in order to assess and produce a value indicating the system overall maintainability. Throughout the years the formula has been expanded and several variations of the formula have emerged. [31]

The study by the developers of the formula and Hewlett and Packard in 1994 concluded the following “... *Our results indicate that automated maintainability analysis can be conducted at the component level, the subsystem level, and the whole system level to evaluate and compare software*” using maintainability index. *This is also confirmed by research and studies such as “Development and Application of an Automated Source Code Maintainability Index”* [30]

Thus, indicating that the use of MI is optimal in two instances mainly for Periodic software assessments (asses the system over time) and software change integration (assess software and determine if the system requires refracturing) [30]

2.6 MAINTAINABILITY INDEX METRICS

2.6.1 LINES OF CODE(LOC/SLOC) AND ITS LIMITATIONS

The metric has been under discussion for several years.[18], in essence Lines of code metric is used to measure the source code in physical lines and as a result illustrate the size of the software.

The simple concept of more LOC equates to more effort is not explicitly true in all cases [24] for instance, writing several variable declarations in one line might yield a minute LOC value, this does not equate to easily maintained code compared to several lines of variable declarations. [24]. Although its questionability, the lines of code metric have been used successfully in mathematical formulas used to detect errors. [23][28]

By measuring the size of the software, the metric gives an insight of what type of effort is required to maintain the software. The notion being, the more line of code equates to more effort required to alter the source code. [23],

Further constrains of Lines of code:

- I. A code base must be built, subsequently the more code, the more accurate the LOC will reflect the maintenance required (if LOC is the only input of maintainability measurement) [23]
- II. Reducing lines of code, can be a subjective function of how skilled the developer inherently is i.e., knowledgeable developers can develop the same code with less lines of code. Writing compact modules may decrease the LOC but may not inevitably affect the overall maintainability. [23]
- III. LOC is not a uniform solution, meaning different languages may require different number of lines of code for different procedures in comparison to other languages, coupled with no industry standard figure for what constitutes a “good” LOC total makes the value of LOC metric somewhat unsubstantiated. [23][52].

2.6.2 COMMENTS ON THE CODE

Maintainability index do come with drawback and that is the use of SLOC, because SLOC considers **all** lines of code including comments as eluded by in the previous section. Ultimately, it's a human that will alter the code, thus having descriptive comments to the person conducting the maintenance will benefit the maintenance process (comments need to be maintained with the code base, having outdated comment is problematic and again having comment in the code, doesn't necessary transmute into more easily maintained code) [30][53].

The developers of the maintainability index (P Oman and J Hagemester, 1992) further explained” *...large comment blocks, especially in small modules, unduly inflated the resulting maintainability indices*”

A scientific article by Kurt D Welker also further emphasised the claim “... *Comments, just like source code, will degrade over time as maintenance activities are performed unless specific actions are taken to keep them from becoming inaccurate*” [31], thus indicating the comments are a point of discussion and their use might affect the overall maintainability of the code.

Depending on the developing entity and the method of development. Some developers might be required to add comments and perhaps even adhere to organizational guidelines of how to construct those comments, alternatively some developers might not require commenting on source code files. Both of which have an impact to the maintainability aspect of the system.

2.6.3 HALSTEAD'S METRICS

In 1977 Maurice Halstead, developed a series of metrics in order to measure program complexity through the source code. More specifically, his aim was through operands and operators determine the source code complexity [34].

2.6.4 HALSTEAD VOLUME

The Halstead volume aims to measure the “size” of an algorithm measured in bits [35][36]

Halstead Volume : $V = N \log_2 (n) \Rightarrow V = (N1 + N2) \log_2 (n1 + n2)$ [37]

- $n2$ = number of unique or distinct operands.
- $n1$ = number of unique or distinct operators.
- $N1$ = total number of occurrences of operators.
- $N2$ = total number of occurrences of operands.

2.6.5 HALSTEAD EFFORT

Halstead's effort metric aims to gauge and quantify the general effort it would take to implement the algorithm into source code.

Halstead hypothesis was to examine if there is a correlation between a software and the mental effort required to implement it, one of the driving factors for the study was Stroud's number/range to assess the mental effort required to build a system [37]

$$\text{Halstead Effort} : E = V * D \Rightarrow E = V * \left(\frac{n1}{2}\right) * \left(\frac{N2}{n1}\right) \quad [37]$$

The Halstead effort metric is controversial in its use, due to some assumptions made by Halstead that was deemed to be unsubstantiated. [37][38] An example of which is the assumption made by Halstead of how many elementary decisions the mind can make and the use of Stroud's Range. [37][38]

2.6.6 CYCLOMATIC COMPLEXITY (CC)

Developed by Thomas. J. McCabe in 1976, code complexity is defined as the difficulty of understanding the code, the difficulty in fixing and updating the code [40] [39]. The cyclomatic complexity main purpose is to measure the complexity using flow graphs i.e., decision points.[30][39]

The metric defines and considers all decision choices a function can make. For instance, a clear static variable that is assigned a value will produce a lower/better mark while conditional-statements and loops will submit a higher/worse score. The general score is later an indication of the overall complexity of the function/class or module [39][41][42].

$$\text{Cyclomatic Complexity: } CC = E - N + (2P) \quad [39]$$

- E = number of edges in the Flow Graph
- N = number of nodes in the Flow Graph
- P = number of nodes with return/exit commands

2.7 MODIFIABILITY

Alterations made to software's occur for different reasons. Some of which can be an improvement of a feature/function, an expansion of the software system, or at times even eliminate existing features.

Modifiability in its core, is all about change and thus we study the constraints of making said change. Thereby, modifiability is a QA² that focuses on how the system can handle a change. [13][8]

Although highly dependent on the lifecycle of the system, some types of software are designed to be short lived therefore applying high degree of modifiability might increase the pre-launch cost for something that, in the end will not require regular adjustments [11].

There are several different enquiries that pertain to modifiability, questions such as what can change and at which rate? Essentially there are some emphases also on the general cost of a change made. [8]

The cost of such change can be both financial and effort, but also the rate in which several other components will require changes as a result [17].

In broad terms modifiability can be summed to *“Degree to which a product or system can be effectively and efficiently modified without introducing defects or degrading existing product quality.”* [12]

2.8 CORRELATION BETWEEN MAINTAINABILITY AND MODIFIABILITY

Maintainability encompasses several other QA² such as modularity, Reusability, analysability, modifiability and testability [12].

Some scholars, studies and sources suggest the QA² maintainability is a sub-category of modifiability [14][15]. The latest ISO 25010 standard, however, concludes modifiability being a subcategory of maintainability [12][13]

² QA: Quality attribute

2.9 PERFORMANCE

2.9.1 OVERVIEW

Performance is a quality attribute that produces quantitative results such as response times, resource usage and throughput among others various statistical units, these elements are called *performance requirements* [45], In essence performance testing is the process of examining key indicators that when under some *type* of pressure produce quantitative results based on the pressure they are put under [46][45]

The results produced are indicators that explain both the limit of the software and indirectly argue about the systems *performance requirements*. Thus, when testing a system some principles might become more prioritized than others.[45]

Performance testing produce metrics that can argue about other respective quality attributes such as security, scalability, availability (fault tolerance, recovery.... etc) etc. [47]

2.9.2 PERFORMANCE TESTING

There are several types of subgroups in terms of performance testing [48], Typically, one of the interesting factors of performance testing is, through a user perspective, measure throughput using response time under a specific load.[47]

The reason to test is to examine how the system handles different workloads, this can be a sudden increase in the number of users, concurrent users, measure maximum workload for specific actions, system bottlenecks and perhaps even determine weak point of the system etc. [49]

3. RESEARCH QUESTIONS

OVERVIEW

The performance and maintainability quality attributes have a relationship to each other, at times a beneficial and other times conflicting.

The performance attribute negatively affects maintainability [29]. And since performance and maintainability are conflicting a trade-off between the two-quality attribute can at times be inevitable [3][27].

At the other end, performance issues are “easier” to solve in maintainable code than the less maintainable code, furthermore, the chances of performance issues are less prevalent in maintainable code [63].

Maintainability is defined as an “optimisation enabler” to other quality attributes such as performance. Therefore, there is a “secondary” effect of designing for maintainability, which is the benefits of optimisation for performance aspects [83].

This thesis aims to examine if a specific performance aspect affects maintainability or metrics included in the maintainability index.

3.1 RESEARCH QUESTION 1

how extensively do the different frameworks differ in terms of performance?

There are in essence identical design solutions, but the implementation can be implemented in various ways [9] and thus, this question examines if a single implementation offers advantages in comparison to others in terms of performance and Maintainability.

Answered through an experiment.

3.2 RESEARCH QUESTION 2

What degree of maintainability is induced by the framework and is there a correlation between performance and maintainability?

The use of maintainability index, cyclomatic complexity and Logical SLOC we aim to measure the overall maintainability (with respect to the metrics). These factors are explained and motivated in the upcoming section “Maintainability”

Answered Through an experiment

3.3 RESEARCH QUESTION 3

Is there a correlation between Halstead's metrics (Effort, Volume, Difficulty) and their impact on the maintainability index? In essence, do the metrics for maintainability index measurement correlate to each other?

The main question is, do the metrics in maintainability index have a correlation to each other? In this thesis we aim to explore the relationship between cyclomatic complexity and the maintainability index (Halstead Volume and effort.) and its parameters such as source line of code, Logical SLOC, Halstead Volume and effort.

Answered through experiment

3.4 RESEARCH QUESTION 4

What are the main differences between the frameworks selected?

The frameworks all aim to solve design problems. however, the implementation may vary. Some might perhaps be geared to modularity, features etc while other might be geared towards performance. This thesis aims to explore what the frameworks are geared towards and if the framework is geared towards performance. In order to evaluate the claim, the only input to this exploration, is the Framework developers own claims.

Answered through input from RQ1 and a literature/Documentation study.

4. MAINTAINABILITY INDEX FORMULAS

Through the years different variations of the original maintainability index formula have emerged. Some have been to counter-act problems with comments other have emerged to further simplify the results etc.

Single metric formula:

$$MI = 125 - 10 \times \log(\mathit{aveE}) \quad [33]$$

The single metric formula encompasses the average Halstead effort metric. This is the least accurate formula and does not consider variable such as the size of the system nor the complexity of the modules that is measured. It relies heavily on the effort variable. [31]

The 3-metric formulas/original formula:

$$i. \quad MI = 171 - 5.2 \times \ln(\mathit{aveV}) - 0.23 \times \mathit{aveV}(g') - 16.2 \times \ln(\mathit{aveLOC}) \quad [31]$$

$$ii. \quad MI = 171 - 3.4 \times \ln(\mathit{aveE}) - 0.23 \times \mathit{aveV}(g') - 16.2 \times \ln(\mathit{aveLOC}) \quad [31]$$

The 3 metric formulas make use of several different variables and are deemed sufficient if no assessment of the comments in the source code is made. [31]. The formulas are largely the same except for their main use of the Halstead metrics, where "I, uses the average Halstead volume which is less controversial than ii which implements the effort metric.

The 4 metric formula / Maintainability index SEI [44]:

$$MI = 171 - 5.2 \times \ln(\mathit{aveV}) - 0.23(\mathit{aveV}(g')) - 16.2 \times \ln(\mathit{aveLoc}) + (50 \times \sin(\sqrt{2.46 \times \mathit{perCM}})) \quad [44]$$

In order to counteract the problem with comments (determining if the comments of the source code are relevant), a new extension of the original formula was developed in 1994 by the same founders (P Oman and J Hagemeister) of the original formula [30]. The newly revised formula was promoted by Software engineering Institute (SEI) in 1997 and was recommended to be used for measuring maintainability of a program, including the use of Cyclomatic complexity, Halstead complexity measures etc. [44]

The 5 metric formula:

$$MI = 138 - 2.76 \times \ln(\mathit{aveE}) - 0.33 \times \mathit{aveVG2} - 12.2 \times \ln(\mathit{aveLOC}) + 0.88 \times \mathit{aveCMT} + 104 \times \mathit{EDOC} \quad [33]$$

A further improvement of the 4 metric formula is the 5 metric system. In which the extended cyclomatic is used, the main difference being that the extended cyclomatic complexity emphasises Boolean operators i.e., comparisons [42][65].

Furthermore, the formula also takes physical lines of comments and thus limiting the subjective input from the programmer, and lastly the EDOC is two-part subjective ranking/scoring system [33]:

1. an evaluation of documentation is conducted and is scored between 0 and 15 points
2. 0 to 5 points is given based upon the ease of building the system itself.

Abbreviation	Description
<i>aveE</i>	Average Halstead effort per module.
<i>aveV</i>	Average Halstead volume per module.
<i>aveV(g')</i>	Average extended cyclomatic complexity per module
<i>aveLOC</i>	Average Lines of Code per module
<i>perCM</i>	Percentage of comments in the source code
<i>aveVG2</i>	Average <i>extended</i> cyclomatic complexity
<i>aveCMT</i>	Average number of lines of comments per module
<i>EDOC</i>	A scoring system

5. RESEARCH METHOD

5.1 SELECTED FRAMEWORKS

Based upon the inclusion criteria, Table 1 illustrates the frameworks that will be explored further and therefore be evaluated.

Table 1 Frameworks/libraries in the study

FRAMEWORK	Updated (Visited August 2021)
Hapi.js	July 2021
Express.js	August 2021
Resify.js	July 2021
Fastify.js	Jan 2021
Sails.js	dec 2020
Koa.js	June 2021
Total.js	June 2021
Moleculer.js	July 2021

5.1.1 SPECIAL CONSIDERATIONS

The frameworks were searched on Google and GitHub and examined based on the inclusion criteria's,

There are several other frameworks that could very well be in Table 1. Such as Vapr.js, meteor.js and Derby.js however due to the time restrictions, the selected frameworks had to be limited in their number. Ultimately the final verdict was based upon how well each framework fulfils the inclusion criteria from their popularity became the deciding factor. There were other frameworks that were considered such as Nest.js, Adonis.js and Feather.js. These frameworks were unfortunately not included due to their use of typescript rather than JavaScript.

5.2 METHOD FOR MEASURING MAINTAINABILITY INDEX (RQ2)

Before alluding to the methods. The 4-5 metric MI - formulas requires assessment of the source code and a conclusion must be drawn whether most of the comments are correct and valid, since this requires subjective input the 4 and 5 metric systems will not be relevant. The original 3-metric MI-formula is sufficient, if no assessment of the comments can be made. [31]

5.2.1 3-METRIC SYSTEM

- a) $MI = 171 - 5.2 \times \ln(\mathit{aveV}) - 0.23 \times \mathit{aveV}(g') - 16.2 \times \ln(\mathit{aveLOC})$
- b) $MI = 171 - 3.4 \times \ln(\mathit{aveE}) - 0.23 \times \mathit{aveV}(g') - 16.2 \times \ln(\mathit{aveLOC})$

The original(3-metric) formulas will be used, the reasoning behind this choice is due to research indicating the 3 metric system is sufficient if there is not a human assessment that deem most comments valid [31].

Typically, there are 2 different variations of the formula were:

- a) Uses Halstead's volume(V) to calculate the MI,
- b) Uses Effort(E), where effort is defined as $E = D * V$,

D is defined as: $D = \left(\frac{n1}{2}\right) * \left(\frac{N2}{n2}\right)$ [37]

- o $n2$ = number of unique or distinct operands.
- o $N2$ = total number of occurrences of operands.

5.2.2 MICROSOFT VERSION OF THE MAINTAINABILITY INDEX

Despite the scale being strictly defined, there are constraints and problematic elements of the scale.

For instance, the original 3-metric formula as it is, can produce negative values and thus it can be difficult to gauge the differences between a maintainability index with the score of zero versus a negative value. [51]

Original maintainability index:

$$MI = 171 - 5.2 \times \ln(\mathit{aveE}) - 0.23 \times \mathit{aveV}(g') - 16.2 \times \ln(\mathit{aveLOC})$$

Microsoft Maintability index :

$$MI = \text{MAX}(0, (171 - 5.2 \times \ln(\mathit{aveV}) - 0.23 \times \mathit{aveV}(g') - 16.2 \times \ln(\mathit{aveLOC})) \times 100/171)$$

In order to counter-act this problem Microsoft created a formula that solves this problem while still preserving the integrity of the original formula. This is done by rounding negative values to zero and translate the score to a range between 0-100. [51]

5.2.3 SCALES AND THRESHOLDS

For the 3-metric system, the scale of the maintainability index is described as the following [30]:

Index	Description
>85	high degree of maintainability. Indicating the source code is highly maintainable.
65-85	moderately maintainable code
<65	Difficulties arises if maintenance is required.

Original Maintainability index thresholds

The Microsoft maintainability scale is defined as following [51]:

Index	Description
0-9	Red (Reason for concern)
10-19	Yellow (Fair)
20-100	Green (Good)

Microsoft Maintainability index thresholds

Cyclomatic complexity thresholds per module is defined as following [79]:

Index/complexity	Risk level
1-10	Low risk
11-20	Medium risk
21-50	High Risk
>50	Very high Risk

Cyclomatic complexity thresholds

5.2.4 METHOD OF MEASUREMENTS

The units (Halstead Effort, Cyclomatic Complexity and LOC) will be measured by an arbitrarily selected software, and the values will be verified with another arbitrarily selected software.

If the values differentiate another similar software will be selected until the values are redundant across several other software.

5.2.5 SOFTWARE CANDIDATES

Inclusion criteria:

To be a valid nominee, the measuring software must:

- Be able to measure the units SLOC (Source Lines Of Code), HV (Halstead volume and/or Halstead Effort) and CC (cyclomatic complexity)
- Be able to measure JavaScript code.
- Be open sourced.

Selected Software's:

- I. **escomplex**³: Measures CC, LOC, Halstead volume and Effort, MI
- II. **complexity-report**⁴: CC, LOC, Halstead volume and Effort, MI
- III. **plato**⁵: LOC, Halstead volume and Effort, MI (Microsoft Scale)

5.3 PERFORMANCE MEASURING METHOD (RQ1)

This thesis will focus on the response times of the different server-side frameworks. The area to be tested and measured is the response time/latencies of the different frameworks ability to resolve requests in the framework of a REST API.

5.3.1 SOFTWARE CANDIDATES AND TECHNICAL DETAILS

The requests will be measured using HTTP/1.1 performance testing tool. The pool of such software's is large and extensive both in their functionalities and their measurements. For instance, there are tools that can simulate real virtual users and environments with various metrics and payloads, all the way to tools that are simple, and perhaps not as extensive in their metrics and capabilities in comparison to extensive tools.

The optimal testing tool for this thesis are tools of minimalistic variety, since the assessments is unilaterally the response times. Therefore, there aren't any need for extensive virtual base support or sophisticated environment, for this reason HTTP testing tools that are minimalistic and do not include any overhead, load times, GUI's or any factors that can greatly affect the hardware will take precedence.

³ <https://www.npmjs.com/package/escomplex>

⁴ <https://www.npmjs.com/package/complexity-report>

⁵ <https://www.npmjs.com/package/plato>

5.4 TEST DETAILS

5.4.1 MACHINE

Machine: macOS Catalina Version 10.15.5 | 16gb 1600 MHz DDR3

Processor: 2,2 GHz Quad-Core Intel Core i7

Node: v15.8.0

Date of performance test: 10-July-2021 and 02-September-2021

5.4.2 SOFTWARE CANDIDATES

- Autocannon ⁶ : Primary testing tool
- Postman ⁷ : Reserve.
- JMeter ⁸ : Secondary testing tool

5.4.3 TEST PARAMETERS

The number of virtual users and number of concurrent users are all parameters that can gravely affect the results. Typically, when load testing, factors such as Centralized/decentralized architecture, requirements, budget etc. [64], including number of users to simulate are studied.

In this case the number of virtual users is a function of how much the virtual users, concurrent users etc. impacts the hardware. Therefore, selecting a virtual users base and number of concurrent users etc., that are beyond the capacity for the hardware to handle might alter the results.

A general rule of thumb is a single core can handle around 500-750 Virtual users for simple HTTP applications per agent-computer [80], this number decreases depending on how intensive the server logic is. Factors such as number of cores, number of CPU's, CPU speed, memory size and ram available etc are all factors that can affect the results.

In summary, despite the figures and general guidelines, there are numerous factors that can alter and skew the results. [80][81][82]

In order to minimise the impacts, a general upper baseline was established based upon the hardware. This is done by evaluating the error rates, standard deviation and processor utilisations in an increased virtual user base.

⁶ <https://www.npmjs.com/package/autocannon>

⁷ <https://www.postman.com/>

⁸ <https://jmeter.apache.org/>

Running 2000 virtual users across 4 workers yields i.e. machine incapable of processing the workload.

Framework	Avg-RT(ms)	errors	timeouts	Avg-requests/s	stdev
Express	548.69	~8000	~6000	839.23	617.75

Optimal Parameters based upon processor utilisation, error rate and standard deviation (requests/s , bytes/s and latency)

Virtual users	750
Time(s)	40(default)
Workers	4
Pipelines	1

5.4.4 LIMITATION OF THE LATENCY TEST

In essence, assuming a “POST” request is issued, how promptly the framework can resolve the end points is measured.

Emphasis will not be placed on:

- I. The size of the payload i.e., the size of the data to be posted.
- II. Successful post to a database or other storage medium (physical or virtual).
- III. Client-side/Front-end processing i.e., rendering processes etc.

The performance will measure the average response times for four different HTTP methods for Rest API service:

- I. GET Requests
- II. POST Requests
- III. PUT Requests
- IV. DELETE Requests

5.5 AUTOCANNON CONFIGURATION

Each test is ran two times, first one being a “warmup” in order to examine the hardware. The next to extract the data.

commands for running the test, “-c” number of virtual users, “-d” duration in seconds, “-w” number of workers, “-m” HTTP request method

End point	Command
GET	<code>autocannon -c 750 -d 40 -w 4 -m GET localhost:3000/get</code>
POST	<code>autocannon -c 750 -d 40 -w 4 -m POST -H 'content-type=application/json' -b '{"hello": "world"}' http://localhost:3000/post</code>
PUT	<code>autocannon -c 750 -d 40 -w 4 -m PUT -H 'content-type=application/json' -b '{"hello": "world"}' http://localhost:3000/put</code>
DELETE	<code>autocannon -c 750 -d 40 -w 4 -m DELETE -H 'content-type=application/json' -b '{"hello": "world"}' http://localhost:3000/delete</code>

6 LITERATURE REVIEW METHOD(RQ4)

In order to select a source that same source must be validated, and claims made must be examined and checked.

6.1 INPUT FOR RQ4

The main input to the RQ4 are sources and claims made by the framework developing entity themselves. There will be no inclusion of claims made by other sources. This is due to framework developing entity cannot be held accountable for the claims made by external unaffiliated sources.

However, the *description* of the framework (in this thesis) can use external sources for instance strength and weaknesses of the frameworks etc.

These elements will follow the Validation template located in the next section (6.2).

External resources

External resources are defined in this thesis as articles or sources that are not made by the developing entity. Thus, if scholars /authors/ reviewers etc that are not affiliated to the developing entity or the framework organisation, make a claim. The framework organisation cannot be held accountable for that statement.

6.2 VALIDATION TEMPLATE

OVERVIEW

The validation template is evaluation framework designed for a systematic approach to evaluate specifically internet resources and articles [66]. By adequately evaluating the internet sources, the problem of validity is somewhat mitigated thus the risk of misinformation is minimised and controlled.

Identify the Purpose and the intended audience:

- E.g., news articles, provide information, advertising etc
- Main audience? E.g., scholars or people with specialised knowledge

Authority and credibility of the author:

- Qualification of the author.
- Publisher i.e., university, organizational etc publications.
- Organisational, article or blogs websites.

Accuracy and validity

- Relevance of the citations, sources used. Are the claims backed by relevant studies supplying evidence of the claim?
- Are the claims redundant across other sources or is the source claim unique?
- If the claims are original, how were they gathered? validity of the study etc

Timelines and urgency

- Is the information relevant in terms of time? Or is it outdated?
- If outdated, do the claims still hold overtime? For example, Historical evidence.

Objectivity and bias

- Occurrence of opinions and subjective notions on the text.
- What Agendas are being applied.

These principles are applied to the selection of internet resources gathered. Claims that are subjective but are still validated in for instance other websites or sources, will be categorised as valid despite being subjective. Redundant claims are valued higher than unique claims.

6.3 SCIENTIFIC RESEARCH AND PAPERS

Scientific papers will take presidency of internet/webpage sources. In order to find relevant research paper, google scholar was used to initially find resources pertaining to the subject in question.

For instance, google scholar was used to find relevant research papers that pertain to the subject such as lines of code, later if the research paper is perhaps lacking in relevant information, the snowball method is used to locate more and/or related research.

Example of keywords used on google scholar to find relevant studies:

“Logical line of code”
“Maintainability software”
“Maintainability index”
“Maintainability metrics”
“Maintainability measurements”
“Software performance “
“Halstead metrics”
....
..
.

6.4 LITERATURE REVIEW CONSTRAINTS

Some resources on the subject rely on sources that are removed, outdated (not due to physical date but rather due to technological advancements made from the time in which the study was made), and/or even other industries that does not pertain to computer science.

These sources can still be valid in the “grand scheme” depending on how heavily they rely on the “invalid” reference.

An examination of the paper is done, and the spectrum of the paper is identified thus if the paper draws conclusions or analysis based upon the invalid reference, then the paper is not used as reference, however if the paper has redundant resources (that is valid) for a exert, statement or similar the problem in this case is mitigated.

At times hindrances arise where referenced sources are kept internal to the publishing entity, such papers/studies will not be used since they can't be accessed.

7. RESULTS OF LITERATURE REVIEW

7.1 EXPRESS.JS GITHUB STARS: 53,945 (JULY 2021)

FRAMEWORK OVERVIEW

Express.js is a backend development JavaScript framework that operates as a mediator/extension for Node.js [67]. The Express.js framework is used to develop **REST API** end points and is mainly used to develop the server [67][71].

Express.js is defined as a simplistic framework, with the uppermost focus on the framework being as minimalistic [67], and flexible [67][68][70]. According to [67], the reasoning is to limit the developer's exposure to complexity of the project.

The framework is designed to be fast and offer extensive http utilities [68]. Express.js is also reasoned to be flexible because of its high-level abstraction [69][70].

The Express.js framework is a very popular framework and thus the community of the open-source framework has extensive support and a large community, this advantage makes the framework lucrative for developers opting to add modules and packages that work well with the Express.js framework.[67].

Another aspect of the framework is it comes with engines that are designed to handle the webpage data such as Jade and EJS, these are the main template engines that are used in Express.js.[70]

QUALITY ATTRIBUTE CLAIMS MADE:

- Robustness⁹
- Performance¹⁰

⁹ <https://github.com/expressjs/express>

¹⁰ <https://expressjs.com/>

7.2 HAPI.JS

GITHub STARS: 13,411(JULY 2021)

FRAMEWORK OVERVIEW

Hapi.js is an open-source server-side framework developed by Walmart labs. Like various other Node.js server-side frameworks, the framework allows the developer to create and manage REST APIs among other numerous features. [74]

The framework is more in geared toward quality rather than achieving the best optimal performance, focusing on readability, predictability and overall developer satisfaction.

However, the main selling point of Hapi.js is the security and testability facets, when it pertains to security concerns, the developers of the framework wanted to prevent any threats that are linked to open-source dependencies that are usually handled by the package handler NPM¹¹ since they are deemed at times to be unsafe. The developer elected to create a framework that has internal modules rather external dependencies, the internal modules are at times referred as plugins and is used to extend Hapi.js capabilities. By default, this makes Hapi.js an extensive and secure framework according to the developers [75].

QUALITY ATTRIBUTE CLAIMS MADE:

- Security¹²
- Code Readability¹²
- Predictability¹²

7.3 RESTIFY.JS

GITHub STARS: 10,194(JULY 2021)

FRAMEWORK OVERVIEW

The Restify.js framework is a framework that specialised in building only semantically correct REST API.[76]

The Framework is restricted to only build REST API and due to its small size one of the main advantages of the framework is its design for performance. The framework is quite new and have received several updated and the GitHub repository is managed quite often. Restify.js have a lot of similarities to the framework express [75]

Restify.js framework have also been shown to perform better in term of response times compared to express. [77][78]

QUALITY ATTRIBUTE CLAIMS MADE:

- Debuggability¹³
- Performance¹³

¹¹ NPM: Node package manager

¹² <https://hapi.dev/>

¹³ <http://restify.com/>

7.4 FASTIFY.JS GITHUB STARS: 19,986(JULY 2021)

FRAMEWORK OVERVIEW

The Fastify.js framework was developed to be a fast framework, that offers the general http utilities to manage and request http request and responses and build complete REST API's.[10]

One of the great benefits of Fastify.js is it was designed, and its components selected specifically to offer the best performance [10]. An example of such a performance component is the use of *PINO* which is a high performing logger and the json parser that is included in the framework.[10][73]

QUALITY ATTRIBUTE CLAIMS MADE:

- Performance ¹⁴

7.5 SAILS.JS GITHUB STARS: 21,990(JULY 2021)

FRAMEWORK OVERVIEW

The sails.js framework is a vast framework supporting several different features. The framework is built using MVC architecture and has the capability of autogenerating extensive portions of the required code at start-up [72], For instance if the developer wanted to create a REST API the entire API can be generated by a single command. The framework is built upon the framework express.js and is further constructed with ensembles of different modules and components with a high degree of maintainability and simplicity [52].

However, one of the main disadvantages of this framework is the performance, one of which is whenever a change to a controller, routes or functions, the framework requires a restart of the application using “Sails.js lift” command [55]

Eluded by in the previous paragraph, a lot of the code is auto generated i.e several auto generated files and code structures. At the early stages of deployment explanations of the auto generated items was not included in the documentation [55].

Another common census is the suboptimal internal ORM¹⁵ called Waterline.

The ORM have been argued to be very ineffective due to

- Queries are changed, meaning the ORM¹⁵ is case insensitive and will change all uppercase letters to lower.
- No support for transactions

However, the developers of the framework have been able to resolve some of these issues with the Waterfall ORM¹⁵ in recent times.

¹⁴ <https://www.fastify.io/>

¹⁵ ORM: Object Rational Mapping

QUALITY ATTRIBUTE CLAIMS MADE:

- Security ¹⁶
- Lightweight ¹⁶
- Scalability [72]
- Maintainability ¹⁶

7.6 MOLECULER.JS GITHUB STARS: 4,510(JULY 2021)

FRAMEWORK OVERVIEW

Moleculer.js is a microservice Node.js framework. The framework is geared towards performance, efficiency, progressive and scalability. The vast framework supports a range of different functionalities such as load balancing and fault tolerance. [57]

Moleculer.js with the microservice structure was developed in the hopes of issues relating to distributed systems, it's also claimed to be scalable efficient and progressive [58]

The framework according to the framework developers, very developer friendly. During the latest release the developers added logs, metrics measurements as well as tracing tools to identify bottlenecks and interactions between modules.[59]

QUALITY ATTRIBUTE CLAIMS MADE:

- Availability-Fault tolerance ¹⁷
- Performance ¹⁷

7.7 TOTAL.JS GITHUB STARS: 4,131(JULY 2021)

FRAMEWORK OVERVIEW

Total.js framework is a full stack Node.js framework that is mainly used to create and manage restful services and web application. The Total.js framework is an open-source project written in only JavaScript. The framework has placed extra emphasis on the documentation such that the documentation is available offline.[54]

The framework supports various functionalities, which ranges everything from web sockets to embedded NoSQL database that is built into the framework itself.

The framework also aims to function without 3rd party libraries or modules (it is able to function with external 3rd modules/packages) , this mitigates the problems of needing 3rd party compiling modules, when developing hybrid or web application. The framework developing entity claims the framework offers “great performance and stability” [50][54]

¹⁶ <https://sailsjs.com/features>

¹⁷ <https://moleculer.services/>

QUALITY ATTRIBUTE CLAIMS MADE:

- Stability¹⁸
- Scalability¹⁸
- Optimisation (low CPU and memory usage)¹⁸
- Extendibility¹⁸

7.8 KOA.JS

GITHUB STARS: 31,371(JULY 2021)

FRAMEWORK OVERVIEW

The main selling point of Koa.js is its claim of being lightweight,[32] The framework is developed by the team that developed the Express.js framework with the incorporation of ES6.[43]

One of the Koa.js main selling point is developer's ability to build their own middleware since the framework does not come “shipped” with a middleware [43]

Koa.js uses “*context objects*” abbreviated as “*ctx*”, the object is used to encapsulate request and response objects [32].

QUALITY ATTRIBUTE CLAIMS MADE:

- Robust¹⁹
- Interoperability¹⁹

¹⁸ <https://www.totaljs.com/platform/>

¹⁹ <https://koajs.com>

8. PERFORMANCE TEST RESULTS

Table 1 Express.js total mb read:

Endpoint	Avg - RT ²⁰ (ms)	Avg - requests/s	Avg - bytes/s	Number of total requests (k = 1000)	Latency Standard deviation(ms)
GET	126	5944.5	1.55 MB	239k	25.5
POST	134.1	5571.8	1.4 MB	224k	19.2
PUT	137.5	5434.2	1.24 MB	218k	22.9
DELTE	134.4	5552	1.27 MB	223k	19.7

The express framework produces on the lower end of the performance test. The average request the framework can handle within 40 seconds are ~ 225 thousand requests. The total average latency for the end points rounds to 132 ms with a standard average deviation of approximately 22 ms. No errors or timeouts was recorded.

Table 2 Moleculer.js

Endpoint	Avg - RT ²⁰ (ms)	Avg - requests/s	Avg - bytes/s	Number of total requests (k = 1000)	Latency Standard deviation(ms)
GET	52.1	14213.6	3.31 MB	569k	12.4
POST	77.2	9680.6	2.16 MB	388k	17.5
PUT	74.7	9970.1	2.22 MB	400k	14.5
DELTE	73	10181.4	2.27 MB	408k	11

The Moleculer.js framework produces on the lower end of the response time test. The average request the framework can handle within 40 seconds are ~ 435 thousand requests. The total average latency for the end points rounds to 68 ms with a standard average deviation of approximately 13 ms. No errors or timeouts was recorded.

²⁰ Avg-RT: Average – Response Time.

Table 3 Hapi.js.

Endpoint	Avg - RT(ms)	Avg - requests/s	Avg - bytes/s	Number of total requests (k = 1000)	Latency Standard deviation(ms)
GET	71	10706	2 MB	429k	22.1
POST	107.4	6943.6	1.3 MB	278k	25.8
PUT	106.2	7007.2	1.3 MB	281k	18.9
DELTE	102.9	7241.9	1.1 MB	290k	16.7

The hapi.js framework produces on the lower end of the performance test. The average request the framework can handle within 40 seconds are ~ 313 thousand requests. The total average latency for the end points rounds to 95 ms with a standard average deviation of approximately 20 ms. During the post request, 6 errors occurred and 0 timeouts.

Table 4 Sails.js

Endpoint	Avg - RT (ms)	Avg - requests/s	Avg - bytes/s	Number of total requests (k = 1000)	Latency Standard deviation(ms)
GET	655.3	1175.3	472 KB	48k	233.7
POST	708	1052.3	413 KB	43k	159.6
PUT	695.6	1074.8	422 KB	44k	134.6
DELTE	749.6	1062.7	417 KB	44k	167.7

The sails.js framework produces on the lower end of the performance test. The average request the framework can handle within 40 seconds are ~ 44 thousand requests. The total average latency for the end points rounds to 701 ms with a standard average deviation of approximately 170 ms. 12 errors and 0 timeouts occurred during POST method, 2 errors and 0 timeouts for PUT method.

Table 5 Restify.js

Endpoint	Avg - RT (ms)	Avg - requests/s	AVG bytes/sec	Number of total requests (k = 1000)	Latency Standard deviation(ms)
GET	49.5	15374.5	3.03 MB	616k	15.3
POST	62.9	12375	2.33 MB	473k	13.8
PUT	70	10615.6	2 MB	425k	22.0
DELTE	64.1	11596.6	2.18 MB	465k	17.3

The Restify.js framework produces on the lower end of the performance test. The average request the framework can handle within 40 seconds are ~ 489 thousand requests. The total average latency for the end points rounds to 61 ms with a standard average deviation of approximately 16 ms. No errors or timeouts were recorded

Table 6 Total.js

Endpoint	Avg - RT (ms)	Avg - requests/s	AVG bytes/sec	Number of total requests (k = 1000)	Latency Standard deviation(ms)
GET	50.9	14642.9	4.8 MB	586k	16
POST	93.4	7980.5	2.5 MB	320k	45.6
PUT	79	9429.8	3 MB	378k	44
DELTE	64.5	11548.2	3.7 MB	463k	13.2

The total.js framework produces on the lower end of the performance test. The average request the framework can handle within 40 seconds are ~ 425 thousand requests. The total average latency for the end points rounds to 70 ms with a standard average deviation of approximately 25 ms. 2 errors were recorded during post request, no timeouts.

Table 7 Koa.js

Endpoint	Avg - RT (ms)	Avg - requests/s	AVG bytes/sec	Number of total requests (k = 1000)	Latency Standard deviation(ms)
GET	47	15792.9	3.1 MB	632k	15.4
POST	60.8	12214.9	2.2 MB	489k	15
PUT	67.3	11029.4	2 MB	442k	18.3
DELTE	59.7	12426.8	2.3 MB	498k	12.4

The koa.js framework produces on the lower end of the performance test. The average request the framework can handle within 40 seconds are ~ 510 thousand requests. The total average latency for the end points rounds to 58 ms with a standard average deviation of approximately 15 ms. no errors were recorded during post request, no timeouts.

Table 8 Fastify.js

Endpoint	Avg - RT (ms)	Avg - requests/s	AVG bytes/sec	Number of total requests (k = 1000)	Latency Standard deviation(ms)
GET	40.4	18487.7	3.6 MB	740k	9.8
POST	75.5	9926.6	1.8 MB	398k	13.3
PUT	77.7	9567.7	1.8 MB	383k	14
DELTE	82.7	8990.1	1.6 MB	360k	16.7

The Fastify.js framework produces on the lower end of the performance test. The average request the framework can handle within 40 seconds are ~ 448 thousand requests. The total average latency for the end points rounds to 66 ms with a standard average deviation of approximately 13 ms. no errors were recorded during post request, no timeouts.

9. RESULTS OF LITERATURE REVIEW RQ4

The result of the literature works in conjunction with the performance results. This is to identify and map the developing entities claim of high performing to the performance results gathered.

Table 9.1 composition of the performance results

Framework	Avg - RT (ms)	Avg- request (k = 1000)	Avg-stdev(ms)	Performance claim
<i>Express.js</i>	132	225	22	Yes
<i>Moleculer.js</i>	68	435	13	Yes
<i>Hapi.js</i>	95	313	20	no
<i>Sails.js</i>	701	44	170	no
<i>Restify.js</i>	61	489	16	Yes
<i>Total.js</i>	70	425	25	Yes
<i>Koa.js</i>	58	510	15	no
<i>Fastify.js</i>	66	448	13	Yes

Table 9.1 illustrates the different systems under test averages. The standard deviation further illustrates how tightly the response times are wound to the mean. i.e., the smaller standard deviation the more consistent the response times are. The performance claims are inputs from the literature review, which examined if the framework documentation make the statement of the framework is geared toward the performance quality attribute.

Figure 9.2 performance(latency) vs claim, red is whether the framework make the claim of high performance. Sorted on lowest response times first.

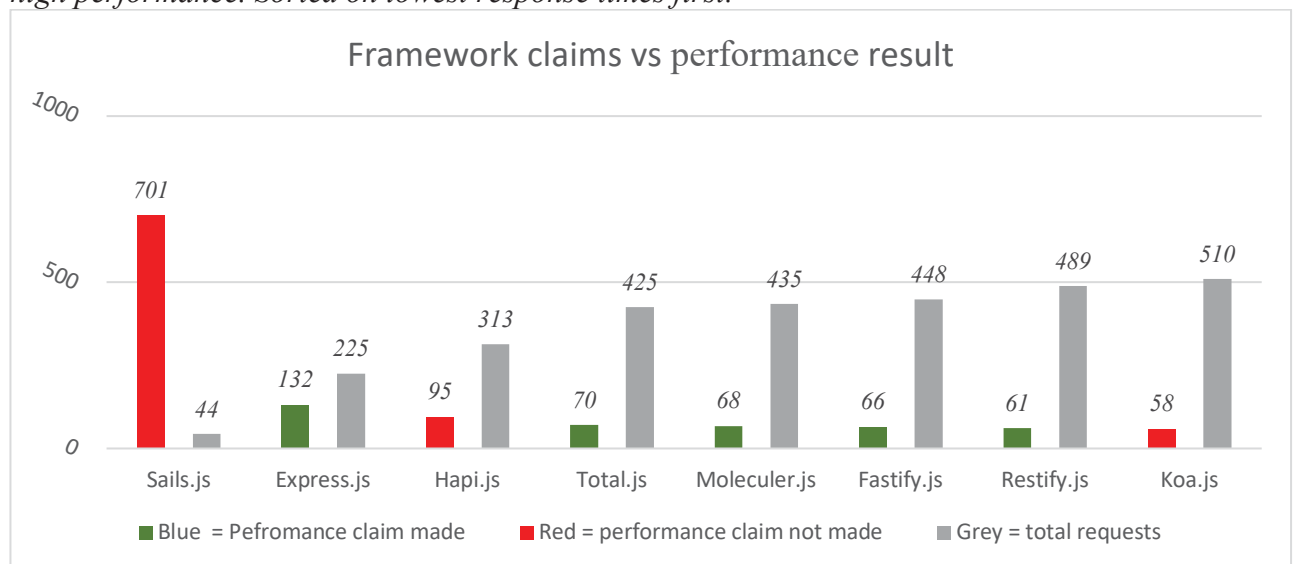


Figure 9.2 further showcases the differences of the frameworks in terms of response times, performance claim and number of requests resolved during the 40s test duration. The graph further confirms the obvious which is the number of total requests grow as the response times decrease.

10. MAINTAINABILITY INDEX RESULTS

Table 14 Average/mean per Function/Module

Framework	(Mean) Maintainability index	(Mean) cyclomatic complexity	(Mean) Logical SLOC
Express.js	111.9	2.6	7.1
Fastify.js	84.6	6.5	21.3
Hapi.js	103.2	4.5	10.8
Koa.js	99.6	5.2	12
Moleculer.js	110.1	1.5	8.1
Restify.js	110.9	2.6	7.7
Sails.js	105.4	3	9.1
Total.js	104.4	3.4	9.2

The maintainability index is calculated using the original 3-metric formula. Each module in the framework (if it pertains to the framework functionalities) is calculated and an average is extracted. The ranges are mapped to their corresponding scales i.e., maintainability index to its respective scale and the same for cyclomatic complexity values.

Table 15 MS code complexity per system/framework, 3 metric system

Framework	Microsoft Scale	Total Logical SLOC	Total Lines of code
<i>Express.js</i>	67.3	2685	4074
<i>Fastify.js</i>	63.7	4029	5534
<i>Hapi.js</i>	61.8	3947	5406
<i>Koa.js</i>	69.3	1479	2841
<i>Moleculer.js</i>	76.7	14753	41910
<i>Restify.js</i>	67.4	5478	9485
<i>Sails.js</i>	66.4	9257	19482
<i>Total.js</i>	60.7	54039	70473

The Microsoft maintainability index is calculated using the original 3-metric formula but with two major differences:

- the formula incorporates Halstead volume rather than the effort variable that is used on the original maintainability 3-metric formula.
- Not counting the previous point, the formula is similar but its rebased to a value between the ranges 0-100.

Also, included on table 15, is the size of the system in total lines of code versus logical lines of code. Thus, the impacts/differences in comments, coding styles etc are visible.

11. ANALYSIS OF THE PERFORMANCE TEST

From the literature review, A great deal of framework make the claim of being fast or high performing. Some of the frameworks make explicit performance claim with factual data or benchmarks some even compare their framework to other frameworks. For instance, A figure such ‘n’ ms for a routine is hard to quantify its worth in terms of performance, this can be due to the non-existing subsequent data to compare the figure to, to clarify, essentially depending on what is being tested the term “fast” can be subjective.

Table 16 Composition of the different latencies in (ms)

Framework	GET	POST	PUT	DELETE
<i>Express.js</i>	126	134.1	137.5	134.4
<i>Fastify.js</i>	40.4	75.5	77.7	82.7
<i>Hapi.js</i>	71	107.4	106.2	102.9
<i>Koa.js</i>	47	60.8	67.3	59.7
<i>Moleculer.js</i>	52.1	77.2	74.7	73
<i>Restify.js</i>	49.5	62.9	70	64.1
<i>Sails.js</i>	655.3	708	695.6	749.6
<i>Total.js</i>	50.93	93.43	79.08	64.5

Looking strictly at the response time during the requests, all the frameworks produce their best time during the GET request. The difference is due to how the different HTTP methods are processed. Where in a GET request data is sent on the request header and the method being idempotent, the post request however, sends data using the message body along with a request header.

In summary, the different HTTP request methods are processed differently and its one of the main reasons for framework having shorter response times on the GET HTTP methods compared to POST method for instance.

Examining the Sails.js framework it produces significant differences in comparison to the other frameworks. The sails.js framework is built upon the Express.js framework [60], however the framework come shipped with extensively more features and modules. According to the developing entity, the framework offers high level features such as auto generating REST API’s.

The sails framework structure is also different in comparison to for instance express.js, fastify.js, koa.js etc. For instance, when a GET request is issued in a Sails.js server it gets resolved by a “Controller “and an “Action”. In essence when a request is received by the sails.js server its parsed, and routed to a controller, which later issues an action for that specific command. The sails.js also contains error handlers [72]. This perhaps adds computational overhead greater than frameworks such as Fastify.js and Express.js produce

In Express.js for instance, the server listens to the port. Given a GET request, the server responds with what’s given in that function. In comparison to the sails.js server the express server is efficient but perhaps more error prone, however due to the simplicity of the server each request (assuming its correct) gets resolved at a faster rate.[62]

Framework	Avg-stdev(ms)	Avg-latency(ms)
<i>Express.js</i>	22	132
<i>Moleculer.js</i>	13	68
<i>Hapi.js</i>	20	95
<i>Sails.js</i>	170	701
<i>Restify.js</i>	16	61
<i>Total.js</i>	25	70
<i>Koa.js</i>	15	58
<i>Fastify.js</i>	13	66

Table 17 avg latency in ms

Looking at table 17, The koa.js framework is the best performing frameworks in terms of having the lowest response times, this notion is further displayed on table 16 where the framework displays consistent response times throughout the different end points this is also further proved by the framework's fairly low standard deviation.

While Fastify.js framework is the best performing on the GET endpoint it does not uphold consistency throughout the different HTTP request methods.

The standard deviation of the different framework shows how consistent the response times are, if a framework on table 17 contains a lower degree of response time but a high deviation. This indicates the value is less precise due to the inconsistent response times. For instance, comparing moleculer.js and fastify.js, the moleculer.js although minimal, is more consistent in its processing times than the fastify.js framework.

Table 18 composition of various metrics sorted on lowest response time first.

Framework	Avg-RT(ms)	(Tot)SLOC	(Tot)LSLOC	CC	MS-MI	MI
<i>Koa.js</i>	58	2841	1479	5.2	69.3	99.6
<i>Restify.js</i>	61	9485	5478	2.6	67.4	110.9
<i>Fastify.js</i>	66	5534	4029	6.5	63.7	111.9
<i>Moleculer.js</i>	68	41910	14753	1.5	76.7	110.1

Illustrated in table 18, selected are the four best performing frameworks in terms of response times. Both the koa.js and the fastify.js framework produces the highest mean cyclomatic complexity out collectively all the frameworks. Whereas the restify.js and the moleculer.js frameworks receive the lowest mean cyclomatic complexity scores.

The size of the different frameworks also varies. Looking at the Logical lines of code, koa.js is the smallest of all the frameworks, fastify.js and restify.js are within the mid-range and the moleculer.js framework is on the higher range in terms of size calculated by logical source lines of code.

Looking at the framework's overall maintainability index, Despite the koa.js being the smallest framework it does not receive the highest maintainability score or the lowest complexity, rather koa.js receives the lowest maintainability index score of all the frameworks. Furthermore, large frameworks such as moleculer.js receive a higher maintainability score and lower complexity despite their size.

On the Microsoft maintainability index scale, koa.js receives a score above the mean and median.

Restify.js and moleculer.js produce low response times despite being mid to high range on their logical source lines of code. Both the frameworks produce maintainability indexes on the upper most range, this is also true on the measurements of the Microsoft scale.

12. ANALYSIS OF THE MAINTAINABILITY INDEX RESULTS

The maintainability index at table 14, indicate most frameworks have a high degree of maintainability, whereas only one of the frameworks (Fastify.js) is on the border between high and medium maintainability.

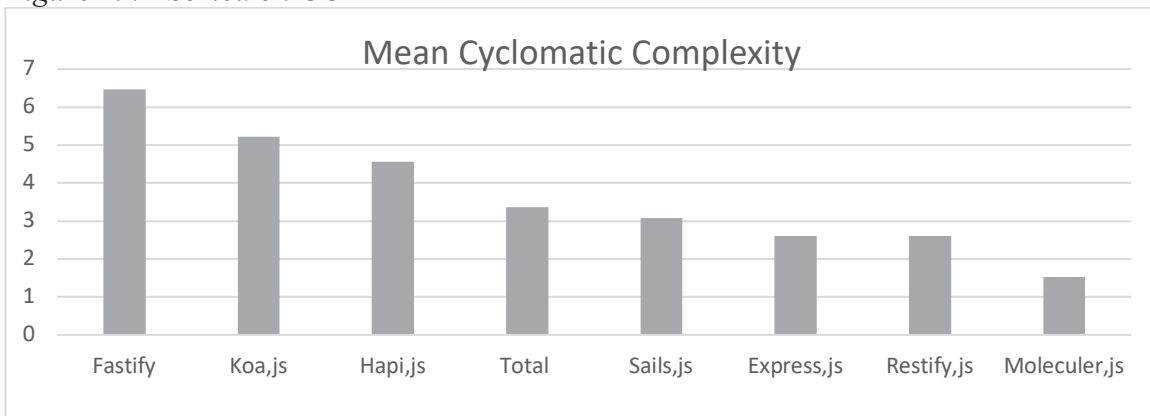
The reasoning behind this is, by examining the Logical SLOC and the cyclomatic complexity in table 19.A, we can see an indication as to why the framework receives a lower score. The fastify.js framework receives the highest cyclomatic complexity despite being on the low to medium in size in terms of logical source lines of code.

The research on the correlation between Cyclomatic complexity and lines of code is conflicting, at some instances the correlation between the metrics is clear so much so, that the two metrics grow linearly and cyclomatic complexity is deemed as redundant. [26], Other research suggest that there are no linear correlation and there are in fact differences between the two different metrics [61]. The research on the relation to logical source lines of code and cyclomatic complexity is limited.

Table 19.A. Sorted on CC

Framework	(Mean) cyclomatic complexity	Total logical SLOC	Total Lines of code
<i>Fastify.js</i>	6,5	4029	5534
<i>Koa.js</i>	5,2	1479	2841
<i>Hapi.js</i>	4,5	3947	5406
<i>Total.js</i>	3,4	54039	70473
<i>Sails.js</i>	3	9257	19482
<i>Express.js</i>	2,6	2685	4074
<i>Restify.js</i>	2,6	5478	9485
<i>Moleculer.js</i>	1,5	14753	41910

Figure 19.B Sorted on CC



From the data gathered there is no evident correlation between the per module cyclomatic complexity and both the logical SLOC and the SLOC, meaning that the 2 metrics are different in this instance.

This notion is enforced due to some frameworks are substantially larger (LSLOC) but still display a low degree of cyclomatic complexity, and on the other end, some frameworks such as `express.js` also display a low degree of cyclomatic complexity despite being smaller in size (LSLOC).

SLOC and the Logical SLOC (Table 19.A) transcendentally have a strong relationship, for instance the logical SLOC will almost at all instances be lower than the total Lines of Code. Furthermore, by comparing the SLOC and the Logical SLOC the different methodologies/coding styles such as comment on the code start to emerge. Logical SLOC only counts lines that contain actual source code, whereas SLOC simply counts every line independent of what it contains.

Table 20, Composition of total number of comments and SLOC

Framework	Comment lines	Total logical SLOC	Total Lines of code
<i>Express.js</i>	1587	2685	4074
<i>Fastify.js</i>	312	4460	5534
<i>Hapi.js</i>	253	3947	5406
<i>Koa.js</i>	1235	1411	2841
<i>Moleculer.js</i>	11050	25692	41910
<i>Restify.js</i>	3998	5478	9485
<i>Sails.js</i>	6664	9257	19482
Total.js	3627	54039	70473

Illustrated in table 20 is the relationship between lines of comments in the code and the total Lines of Code.

Different frameworks can perhaps use different coding styles, for instance the `Sails.js` framework have extensive comments through the code files, with both commented code and various descriptions that go into detail (*see appendix 1A*)

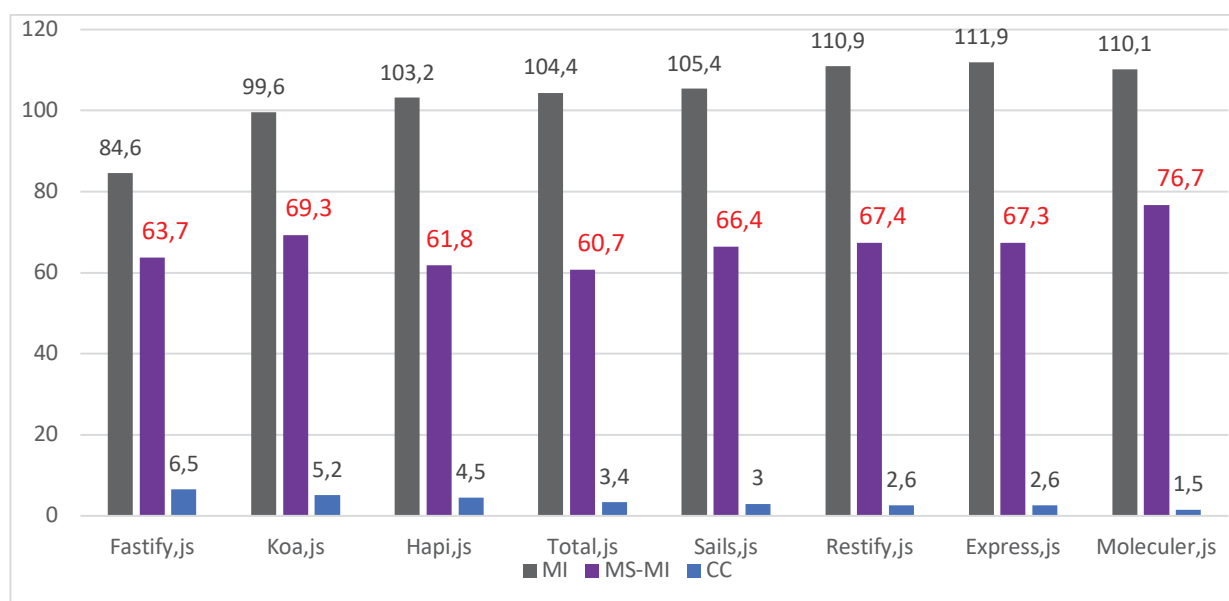
Some of the frameworks have specialised “`readme`” in the source folders containing descriptions rather than having them in the source code.

All the frameworks follow a distinct comment style, explaining parameters / returns / dependencies of the functions/modules (*see appendix 1b*). This comment style seems to be a tool to determine what is expected from the function and identifying the parameters and dependencies of the function. Although the specialised comments (*appendix 1B*) can be deemed helpful, like the code they need to be maintained if the source code undergoes maintenance [31] On the opposite side of the spectrum a well-versed developer who has knowledge of the system might not deem the specialised comments as helpful

Table 21, original Maintainability index vs Microsoft maintainability index

Framework	(Mean) Maintainability index	MS Maintainability index
<i>Express.js</i>	111.9	67.3
<i>Fastify.js</i>	84.6	63.7
<i>Hapi.js</i>	103.2	61.8
<i>Koa.js</i>	99.6	69.3
<i>Moleculer.js</i>	110.1	76.7
<i>Restify.js</i>	110.9	67.4
<i>Sails.js</i>	105.4	66.4
<i>Total.js</i>	104.4	60.7

Table 21 illustrates the different rankings where MS-maintainability index uses Halstead's volume, and the Mean maintainability index using the Halstead's effort which is the function of *program difficulty* times the *program volume*. The results indicate the two variables are inherently different and depending on which variable is used different results are produced.



maintainability indexes, sorted by cyclomatic complexity in descending order

Analysing the results from the maintainability indexes we can observe the effects of the program Effort vs Volume. There is a correlation between the original maintainability index results and the cyclomatic complexity (larger = worse). As the cyclomatic complexity decreases the maintainability of the system shows an increased trajectory. The cyclomatic complexity in this case, show a correlation to the maintainability index, specifically favouring the *program effort* Halstead metric rather than the volume as the input in the original 3-metric formula.

However, this correlation does not transcend into the Microsoft's maintainability index scale, more specifically, the correlation cannot be observed between Halstead volume and cyclomatic complexity.

13. CONCLUSION

Concluded in this thesis is, the maintainability index will reflect differently depending on which Halstead metric is passed into the 3-metric formula. The cyclomatic complexity and the Halstead effort do in fact show a relation whereas no distinct relationship is observed between Halstead's volume and Cyclomatic complexity.

There also seem to be no distinct relationship between response times for the HTTP request methods and MS-maintainability index, cyclomatic complexity and/or Source Lines of Code/Logical Source Lines of Code.

Koa.js which is deemed to be the best performing framework and the smallest in Logical source lines of code produce a high maintainability index according to the original maintainability scale and slightly above average according to the Microsoft maintainability index scale. Moleculer.js produces a performance result in the midrange despite being 10x larger in size (Logical SLOC) compared to koa.js, it also manages to produce the highest Microsoft maintainability and second highest maintainability according to the original maintainability index scale, the moleculer.js also produces the lowest complexity of all the frameworks.

Finally, concluded in this thesis the response times differ, all the frameworks produce better performance results for GET – requests. Therefore, despite being built upon the same environment (node JS) the results ultimately show differences.

Overall, the Frameworks themselves show a high degree of maintainability in both the scales, with a degree of complexity that does not warrant any distress according to both the original maintainability scale, Microsoft scale and the cyclomatic complexity scale.

The result from this study conforms to similar research conclusions, i.e., further indicates that the different metrics such as the Halstead's metrics, cyclomatic complexity and the maintainability index's need to be studied and examined further.

14. VALIDITY THREATS

14.1 LIMITED VIRTUAL USERS

The test parameters and the test cases are limited to the machine that have been used. This essentially means that the result is indicative to the limited virtual users and number of worker threads etc. Thus, broad conclusions cannot be made because the test result from a small virtual user base might not be translated to similar result in a larger virtual user base with more capable hardware. The conclusions thereof are concluded based upon the specific machine, environment, test software and the test parameters.

14.2 MAINTENANCE THREATS

This research relies upon specific metrics and formulas to gauge the overall maintainability. Therefore, the maintainability conclusion is based upon these metrics, which at time have been a point of discussion and their validity in practical use, questioned.

Different frameworks offer different functionalities, some of the frameworks follow MVC patterns, other come shipped without middleware and some even require modules to support routing. These factors most likely will impact the results of the Lines of code.

A vast framework in this case might be penalised in terms of the Logical lines of code metric for offering a great verity of functionalities.

There is also the risk of affecting the mean complexity and the Halstead metrics by introducing more features to the system if they exceed the average cyclomatic complexity and the averages Halstead metrics.

In order to address this notion, careful evaluation and an analysis of logical lines of code, response times, maintainability indexes etc was done to determine if the notion is sound.

14.3 RELIABILITY

The latency test and the results from the maintainability calculations are made using open-sourced software's. The validity (in terms of accuracy) of the software has in a limited scope been double checked using similar software's. However, the results are still constricted by the software accuracy.

14.4 EXPERIMENT ENVIRONMENT

In order to limit the impact of background processes and threads, all background tasks and threads was closed in advance. Despite this precaution, background processes or other external processes might have begun during the test which might have impacted the results.

14.5 EXTERNAL FACTORS THAT AFFECT MAINTAINABILITY

The maintainability index measures a small subset of maintenance. Time to test, localisation of the item to be maintained, how well versed the developer is etc are not examined by the maintainability index. Thus, in reality, all of these factors can greatly affect the overall maintainability of the system. Therefore, the conclusions, analysis and the results are strictly bound to the results from the study and should not be translated to real-world/practical scenarios.

14.6 USER ERROR

Before attempting to gauge the different frameworks maintainability, some files from the framework had to be skipped. Example of such files are test files, readmes and files that don't necessarily pertain to the framework's functionality. Most of the framework had their framework functionalities specifically on folders called "lib" however some frameworks have different functionalities scattered through a wider variety of folders, those frameworks required careful evaluation of which files/modules pertain to framework functionality, this does not guarantee the selection of modules relevant to be error-free.

15. FUTURE WORK

Studies and the results from this thesis indicate they are more work to be done. The maintainability index is ever evolving concept and can be both specific and misleading depending on the measured software and even language used, therefore establishing industry standard can be difficult since the metric includes some subjective notions and assumption made by the originators of the different formulas. Further studies such as rather than measuring entire frameworks, a study focusing on specific areas of the frameworks might yield a more levelled playing field, this is instead of measuring entire frameworks unaffected by their functionalities, size etc. Also, a comparison between the maintainability index of different architectural patterns such as MVC or MVVM verses traditional patterns could present interesting results.

The reason to gauge maintainability is an asset for a lot of organisations and there are various tools and organisations that work with measuring software maintainability. Extension of this thesis could be to examine the maintainability index using other tools and metrics, larger virtual user base and better hardware for the performance test. Further studies could also be at organisational level/specific for instance implement and further improve the measurement tool to be more compatible with their code base.

REFERENCES

- [1] "Server-side web frameworks ", Developer.mozilla.org, 2021. [Online]. Available: https://developer.mozilla.org/en-US/docs/Learn/Server-side/First_steps/Web_frameworks. [Accessed: 29- Mar-2021].
- [2] "Framework Definition", Techterms.com, 2013. [Online]. Available: <https://techterms.com/definition/framework>. [Accessed: 12- May- 2021].
- [3] R. Kazman, M. Klein, M. Barbacci, T. Longstaff, H. Lipson and J. Carriere, "The architecture tradeoff analysis method," in Proc. Fourth IEEE International Conference on Engineering of Complex Computer Systems (Cat. No.98EX193), 1998, pp. 68-78. [Online] Available: <https://ieeexplore.ieee.org/document/706657> [Accessed: 29- August- 2021].
- [4] "server-side Frameworks", Mozilla.org, Mar 2021, [Online]. Available: https://developer.mozilla.org/en-US/docs/Learn/Tools_and_testing/Client-side_JavaScript_frameworks [Accessed: 16-May-2021]
- [5] Software maintenance cost", Galaroth.org, Jan 2018, [Online]. Available: <https://galorath.com/software-maintenance-costs/> [Accessed: 16-May-2021]
- [6] M. Matinlassi, E. Niemelä , "The impact of maintainability on component-based software systems," In Proc. 29th Euromicro Conference, 2003, pp. 25-32, [Online] Available: <https://ieeexplore.ieee.org/document/1231563> [Accessed: 10- July-2021]
- [7] P. Marounek,. "Simplified approach to effort estimation in software maintenance." Journal of Systems Integration, vol. 3, pp 51-63, 2012. [Online] Available: <http://si-journal.org/index.php/JSI/article/viewFile/123/99> [Accessed: 09-September- 2021]
- [8] P. Bengtsson, N. Lassing, J. Bosch, and H.V. Vliet, "Analyzing Software Architectures for Modifiability." Hogskolan Karlskrona/Ronneby, 2000. [Online] Available: <https://www.diva-portal.org/smash/get/diva2:838104/FULLTEXT01.pdf> [Accessed: 22- August- 2021]
- [9] L. Bass, P. Clements and R. Kazman. *Software Architecture in Practice*, 3rd ed, MA: Addison-Wesley Professional, 2012 , pp 60-62
- [10]" Fastify, Fast and low overhead web framework for Node.js", Fastify.io, 2018. [Online]. Available: <https://www.fastify.io/>. [Accessed: 16- June- 2021].
- [11] J. McGovern, S. Tyagi, M. Stevens and S. Mathew, *Java Web Services Architecture*, 1st ed, MA: Morgan Kaufmann , 2003, pp.117 – 124 .
- [12] *System and software quality models*, ISO/IEC 25010, 2011

- [13] P. Bengtsson, "Architecture-Level Modifiability", PhD dissertation, Blekinge Institute of Technology, Ronneby, 2002. [Online]. Available: <http://www.diva-portal.org/smash/get/diva2:837619/FULLTEXT01.pdf> [Accessed: 04-June- 2021].
- [14] M. Barbacci, "Software quality attributes: modifiability and usability." Software Engineering Institute (SEI), Carnegie Mellon University, PA, 2004. [Online]. Available: <http://www.ieee.org.ar/downloads/Barbacci-05-notas1.pdf> [Accessed: 02-June-2021].
- [15] F. Bachmann, L. Bass and R. Nord, "Modifiability Tactics," Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pennsylvania, Tech. Report CMU/SEI-2007-TR-002, 2007, [Online] Available: <https://apps.dtic.mil/sti/pdfs/ADA472581.pdf> [Accessed 11-September- 2021].
- [16] A. Oy, "Project Metrics Help - Cohesion metrics", Aivosto.com, 2017. [Online]. Available: <https://www.aivosto.com/project/help/pm-oo-cohesion.html>. [Accessed: 21- June-2021].
- [17] E. Yourdon, L.L. Constantine, *Structured Design: Fundamentals of a Discipline of Computer Program and System Design*, 2nd ed, NJ: Prentice Hall, 1979, pp 95-96
- [18] J. Rosenberg, "Some misconceptions about lines of code," , In Proc. Proceedings Fourth International Software Metrics Symposium,1997, pp. 137-142, . [Online]. Available: <https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=637174>. [Accessed: 21- July-2021].
- [19] N. Chapin, J. E. Hale, K. Khan, J. F. Ramil and W.G. Tan, "Types of software evolution and software maintenance." , Journal of software maintenance and evolution: Research and Practice, vol. 13, no. 1, pp. 3-30, 2001. [Online]. Available: <http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.511.1671&rep=rep1&type=pdf>. [Accessed: 11- June- 2021].
- [20] P. Oliveira, M. T. Valente, F. P. Lima, "Extracting relative thresholds for source code metrics," In Proc. 2014 Software Evolution Week - IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE), 2014, pp. 254-263, [Online]. Available: <https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=6747177> [Accessed: 21- June- 2021].
- [21] S. Maheswari, K Chitra - "AN EMPIRICAL DESIGN AND CODE METRICS FOR PREDICTION OF SOFTWARE DEFECTS", International Journal of Pure and Applied Mathematics, vol. 119 no. 16, pp 671-677, 2018 , [Online]. Available: <https://acadpubl.eu/hub/2018-119-16/1/68.pdf6747177> [Accessed: 21- June- 2021].
- [22] M. Riaz, E. Mendes, E. Tempero, "A systematic review of software maintainability prediction and metrics," 3rd International Symposium on Empirical Software Engineering and Measurement, pp. 367-377, 2009, [Online]. Available: <https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=5314233> [Accessed: 22-July-2021].

- [23]. K. Bhatt, V. Tarey , P. Patel – “Analysis Of Source Lines Of Code(SLOC) Metric” ,International Journal of Emerging Technology and Advanced Engineering, vol. 2 no. 5, pp. 150-154, 2012 [Online]. Available: <http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.413.4352&rep=rep1&type=pdf> [Accessed: 21- July- 2021].
- [24] B. Parareda, M Pizka. "Measuring productivity using the infamous lines of code metric." In proc. Proceedings of SPACE 2007 Workshop, Nagoya, Japan. 2007. [Online]. Available: <https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.110.4666&rep=rep1&type=pdf#page=13> [Accessed: 23- July- 2021].
- [25] Software Engineering - Software Life Cycle Processes - Maintenance, ISO/IEC 14764, 2006
- [26] Y. Tashtoush , M. Al-Maolegi, and B. Arkok, "The correlation among software complexity metrics with case study.” International Journal of Advanced Computer Research, vol. 4 no. 2, pp. 414-419, 2014 ,[Online]. Available: <https://arxiv.org/pdf/1408.4523.pdf> [Accessed: 22-July - 2021].
- [27] D. Häggander, P. Bengtsson, J. Bosch and L. Lundberg, ”Maintainability myth causes performance problems in parallel applications”. In Proc. of the IASTED 3rd International Conference on Software Engineering and Applications , pp. 288-294, 1999, [Online]. Available: <http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.42.540&rep=rep1&type=pdf> [Accessed: 12- September- 2021].
- [28] H. Zhang, "An investigation of the relationships between lines of code and defects," In Proc. IEEE International Conference on Software Maintenance, 2009, pp. 274-283, [Online]. Available: <https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=5306304> [Accessed: 21- June- 2021].
- [29] K. Henningson, “Tradeoffs and conflicts between quality attributes”, M.S. thesis, Blekinge Institute of Technology, Ronneby,2001 [Online]. Available: <https://www.diva-portal.org/smash/get/diva2:832582/FULLTEXT01.pdf> . [Accessed: 17- September - 2021].
- [30] D. Coleman, D. Ash, B. Lowther and P. Oman, "Using metrics to evaluate software system maintainability," Computer, vol. 27, no. 8, pp. 44-49,1994, [Online]. Available: <https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=303623> [Accessed: 06- June- 2021].
- [31] K. D. Welker "The software maintainability index revisited." *CROSSTALK The Journal of Defense Software Engineering*, vol. 14, pp.18-21.2001, [Online]. Available: <http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.444.9945&rep=rep1&type=pdf> [Accessed: 22-September - 2021].
- [32] “next generation web framework for node.js”, koajs.com, 2021. [Online]. Available: <https://koajs.com>. [Accessed: 05- June- 2021].

- [33] H. R. Bhatti, "Automatic Measurement of Source Code Complexity", M. S. thesis, Lulea University of Technology, Lulea, 2011, [Online]. Available: <https://arxiv.org/pdf/1408.4523.pdf> [Accessed: 11- June- 2021].
- [34] T. Hariprasad, G. Vidhyagaran, K. Seenu and C. Thirumalai, "Software complexity analysis using halstead metrics," In Proc. International Conference on Trends in Electronics and Informatics (ICEI), pp. 1109-1113, 2017, [Online]. Available: <https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=8300883> [Accessed: 11- June- 2021].
- [35] "Measurement of Halstead Metrics with Testwell CMT++ and CMT Java (Complexity Measures Tool)", Verifysoft.com, 2021. [Online]. Available: https://www.verifysoft.com/en_halstead_metrics.html. [Accessed: 07- Jul- 2021].
- [36] H. Zuse, "Resolving the mysteries of the Halstead measures. ", In Proc. Praxis der Software-Messung: Tagungsband des DASMA Software Metrik Kongresses MetriKon'05, pp. 107-122 ,2005. [Online]. Available: <http://www.horst-zuse.homepage.t-online.de/z-halstead-final-05-1.pdf> [Accessed: 11- July- 2021].
- [37] M. H. Halstead , "Toward a theoretical basis for estimating programming effort." In Proc ACM '75: Proceedings of the 1975 annual conference, pp. 222-224. 1975. [Online]. Available: <https://docs.lib.purdue.edu/cgi/viewcontent.cgi?article=1091&context=cstech> [Accessed: 24- July- 2021].
- [38] V. Y. Shen, S. D. Conte and H. E. Dunsmore, "Software Science Revisited: A Critical Analysis of the Theory and Its Empirical Support," in Proc. IEEE Transactions on Software Engineering, vol. SE-9, no. 2, pp. 155-165,1983. [Online]. Available: <https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=1703032> [Accessed: 24- July- 2021].
- [39] Graylin JAY, Joanne E. HALE, Randy K. SMITH, David HALE, Nicholas A. KRAFT, Charles WARD - "Cyclomatic Complexity and Lines of Code: Empirical Evidence of a Stable Linear Relationship" – 2009. [Online]. Available: <https://www.scirp.org/html/779.html> [Accessed: 24- July- 2021].
- [40] W.A Harrison, K.Magel, R.M Kluczny and A.D DeKock, "Applying software complexity metrics to program maintenance," in Computer(IEEE Computer Society), vol. 15, no. 9, pp. 65-79. 1982,. [Online]. Available: https://scholarsmine.mst.edu/cgi/viewcontent.cgi?article=1127&context=bio_inftec_facwork [Accessed: 24- July- 2021].
- [41] C. Arapidis , *Sonar code quality testing essentials*. 1st ed. Birmingham(UK): Packt Publishing, 2012 , pp.197 – 200.
- [42] J. Britton, "What Is Cyclomatic Complexity?", Perforce Software, 2016. [Online].

Available: <https://www.perforce.com/blog/qac/what-cyclomatic-complexity>. [Accessed: 14-Jun- 2021].

[43] J.M. Rishard, "Introduction to Koa.js", medium.com, 2021. [Online]. Available: <https://medium.com/geekculture/introduction-to-koa-js-c332931c6b24>. [Accessed: 04- Aug- 2021].

[44] M. Bray, K. Brune, D.A Fisher, J. Foreman, and M Gerken, "C4 Software Technology Reference Guide-A Prototype", Carnegie-Mellon University, Pittsburgh, Pa: Software Engineering Inst, 1997,[Online]. Available: <https://apps.dtic.mil/sti/pdfs/ADA320732.pdf> [Accessed: 21- July- 2021].

[45] A. B. Bondi. 2012. "Best practices for writing and managing performance requirements: a tutorial". In Proc. ACM/SPEC International Conference on Performance Engineering (ICPE '12) vol. 3, pp. 1-8, 2012, [Online]. Available: <https://pdfs.semanticscholar.org/daaa/a43707a937a4fea3ac8237fed65cdd23b027.pdf> [Accessed: 10- July- 2021].

[46] F. I. Vokolos, and E. J. Weyuker. "Performance testing of software systems." In Proc. International Workshop on Software and Performance, vol. 1, pp. 80-87. 1998. [Online]. Available: <https://cdn-stg.scdn8.secure.raxcdn.com/static/media/ed4856b1-edb3-45ef-bc9d-3376fb1039b1.pdf> [Accessed: 22-July - 2021].

[47] H. Sarojadevi , "Performance testing: methodologies and tools." Journal of Information Engineering and Applications vol. 1 no. 5 pp. 5-13 ,2011, [Online]. Available: <https://core.ac.uk/reader/234676950> [Accessed: 21- July- 2021].

[48] J.D. Meier, C. Farre, P. Bansode, S. Barber, and D. Rea , "Performance Testing Guidance for Web Applications", Docs.microsoft.com, 2007. [Online]. Available: [https://docs.microsoft.com/en-us/previous-versions/msp-n-p/bb924375\(v=pandp.10\)](https://docs.microsoft.com/en-us/previous-versions/msp-n-p/bb924375(v=pandp.10)). [Accessed: 26-August- 2021].

[49] G. Denaro, A. Polini, and A. Emmerich, , " Early performance testing of distributed software applications". In Proc. International Workshop on Software and Performance, vol. 4, pp. 94-103. 2004, [Online]. Available: https://discovery.ucl.ac.uk/id/eprint/871/1/3.6_paper.pdf [Accessed: 26- August- 2021].

[50] P. Sirka, "Total.js 2.0: web framework for node.js", medium.com, 2016. [Online]. Available: <https://petersirka.medium.com/total-js-2-0-web-framework-for-node-js-e3e7afbe5e88>. [Accessed: 20- June- 2021].

[51] M. Jones and G. Hogenson, "Code metrics - Maintainability index range and meaning", docs.microsoft.com, 2021. [Online]. Available: [https://docs.microsoft.com/en-us/previous-versions/msp-n-p/bb924375\(v=pandp.10\)](https://docs.microsoft.com/en-us/previous-versions/msp-n-p/bb924375(v=pandp.10)). [Accessed: 02- June- 2021].

[52] V. Nguyen, S. D. Rubin, T. Tan, and B. Boehm. "A SLOC counting standard." In proc: International Forum on COCOMO and Systems/Software Cost Modelling, vol. 2007, pp. 1-16, 2007, [Online]. Available:

<http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.366.196&rep=rep1&type=pdf>.
[Accessed: 14- September- 2021].

[53] A. Liso , “Software maintainability metrics model: An improvement in the coleman-oman model.”, CROSSTALK The Journal of Defense Software Engineering '01, pp.15-17. 2001, [Online]. Available:
<http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.124.3934&rep=rep1&type=pdf>
[Accessed: 14 - September- 2021].

[54] P. Sirka, "Top 10 reasons to use Total.js Platform", medium.com, 2021. [Online]. Available: <https://petersirka.medium.com/top-10-reasons-to-use-total-js-platform-7d54e8bc2670>. [Accessed: 14- June- 2021].

[55] K. Burke, "Don't Use Sails (or Waterline)", kevin.burke.dev, 2015. [Online]. Available: <https://kevin.burke.dev/kevin/dont-use-sails-or-waterline/>. [Accessed: 12- August- 2021].

[56] "Features", Sailsjs.com. Date not published. [Online]. Available: <https://sailsjs.com/features>. [Accessed: 02- August- 2021].

[57] M. Norbert, "Moleculer, a progressive microservices framework for Node.js", medium.com, 2017. [Online]. Available: <https://medium.com/moleculer/moleculer-a-modern-microservices-framework-for-nodejs-bc4065e6b7ba>. [Accessed: 2- June- 2021].

[58] A. Mazayev, "Moleculer v0.14 — Making Microservices Accessible for Everyone", medium.com, 2020. [Online]. Available: <https://medium.com/moleculer/moleculer-v0-14-making-microservices-accessible-for-everyone-eadeefa5156c>. [Accessed: 02- June- 2021].

[59] A. Mazayev , “An Introduction to Moleculer JS” , Sep.8 , 2019, [Video file] [Online] Available: <https://www.youtube.com/watch?v=t4YR6MWrugw>
[Accessed: 05- July- 2021].

[60] “Frameworks built on Express”, <https://expressjs.com/>, Mar 2021, [Online]. Available: <https://expressjs.com/en/resources/frameworks.html> [Accessed: 16- September-2021]

[61] D. Landman, A. Serebrenik and J. Vinju, "Empirical Analysis of the Relationship between CC and SLOC in a Large Corpus of Java Methods," 2014 IEEE International Conference on Software Maintenance and Evolution, 2014, pp. 221-230, [Online]. Available: <https://www.win.tue.nl/~aserebre/Landman2015-ccsloc-jsep2015-preprint.pdf>. [Accessed: 12- September- 2021].

[62] L. Vivah, "THE BEGINNER'S GUIDE: Understanding Node.js & Express.js fundamentals", *Medium*, 2017. [Online]. Available: <https://medium.com/@LindaVivah/the-beginners-guide-understanding-node-js-express-js-fundamentals-e15493462be1>. [Accessed: 16- Sep- 2021].

[63] F. Abdullah, "Evaluating Impact of Design Patterns on Software Maintainability and Performance", M.S thesis, UNIVERSITY OF OSLO, Oslo, 2017. ,[Online]. Available:

https://www.duo.uio.no/bitstream/handle/10852/60024/1/FinalThesis_Farooq.pdf. [Accessed: 12- July- 2021].

[64] Z. M. Jiang, A. E. Hassan, G. Hamann and P. Flora, "Automatic identification of load testing problems," 2008 IEEE International Conference on Software Maintenance, 2008, pp. 307-316 ,[Online]. Available: <https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=4658079>. [Accessed: 12- Jul- 2021].

[65] A. Oy, "Project Metrics Help - Complexity metrics", Aivosto.com, 2021. [Online]. Available: <https://www.aivosto.com/project/help/pm-complexity.html>. [Accessed: 16- Sep- 2021].

[66] J. Mandalios, "RADAR: An approach for helping students evaluate Internet sources", Journal of information science vol.39 no.4 , pp.470-478.2013, [Online]. Available: https://ils.unc.edu/courses/2015_spring/inls151_002/Readings/Mandalios_RADAR_2013.pdf [Accessed: 12- Jul- 2021].

[67] E. Brown, *Web development with Node and Express*, 1st ed. MA: O'Reilly Media, 2014, pp. 1-4.

[68] "Fast, unopinionated, minimalist web framework for Node.js", expressjs.com, 2010. [Online]. Available: <https://expressjs.com/>. [Accessed: 28- August- 2021].

[69] "Express/Node introduction", developer.mozilla.org, 2021. [Online]. Available: https://developer.mozilla.org/en-US/docs/Learn/Server-side/Express_Nodejs/Introduction. [Accessed: 23- August- 2021].

[70] C. Arapidis, *Sonar code quality testing essentials*, 1st ed. Birmingham: Packt Publishing, 2012, pp. 197 – 200.

[71] A. Vlăduțu, "Mastering Web Application Development with Express", 1st ed. Birmingham(UK): Packt Publishing, 2014, pp. 26-29.

[72] M. McNeil and I. Nathan, "*Sails.js in action*", 1st ed. NY: Manning Publications Co., 2017.

[73] M. Collina, "Introducing Fastify, a Speedy Node.js Web Framework", thenewstack.io, 2017. [Online]. Available: <https://thenewstack.io/introducing-fastify-speedy-node-js-web-framework/>. [Accessed: 03- August- 2021].

[74] "The Simple, Secure Framework Developers Trust", hapi.js.dev. [Online]. Available: <https://hapi.js.dev/#quality>. [Accessed: 06- July- 2021].

[75] A. Khandelwal, "Introduction to Hapi.js (Part 1)", medium.com, 2019. [Online]. Available: <https://medium.com/the-resonant-web/an-introduction-to-hapi-js-part-1-5f2edcb5eced>. [Accessed: 18- July- 2021].

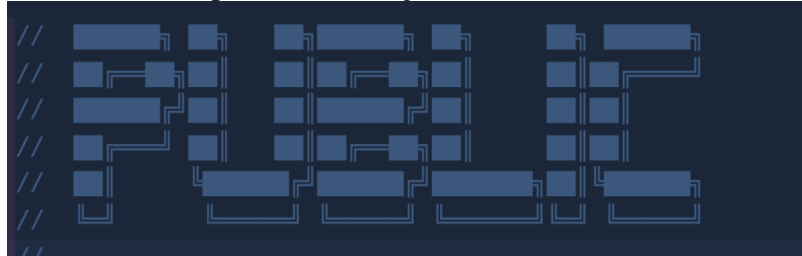
- [76] "API Guide | restify | About restify", Man.hubwiz.com. [Online]. Available: <http://man.hubwiz.com/docset/Restify.docset/Contents/Resources/Documents/index.html>. [Accessed: 05- July- 2021].
- [77] "Benchmarks", Fastify.io. [Online]. Available: <https://www.fastify.io/benchmarks/>. [Accessed: 08- July- 2021].
- [78] "Node.js performance vs Hapi, Express, Restify, Koa & More", Raygun Blog, 2017. [Online]. Available: <https://raygun.com/blog/nodejs-vs-hapi-express-restify-koa/>. [Accessed: 12- July- 2021].
- [79] P. Malhotra.,K. Shah, J. Rathod, and M. Mehta, "Python Based Software for Calculating Cyclomatic Complexity." International Journal of Innovative Science, Engineering and Technology vol.2 no.3 , pp.546-549 2015. [Online]. Available: https://ijiset.com/vol2/v2s3/IJISSET_V2_I3_86.pdf [Accessed: 12- July- 2021].
- [80] P. Varhol, "How Many Virtual Users Per Machine?", *Telerik Blogs*, 2013. [Online]. Available: <https://www.telerik.com/blogs/how-many-virtual-users-per-machine>. [Accessed: 10- September- 2021].
- [81] "System Requirements for Large-Scale Load Tests | ReadyAPI Documentation", Support.smartbear.com, 2021. [Online]. Available: <https://support.smartbear.com/readyapi/docs/performance/intro/system-requirements.html>. [Accessed: 16- September- 2021].
- [82] "Performance and Scalability Matrix", Microfocus.com. [Online]. Available: <https://www.microfocus.com/documentation/silk-performer/195/en/silkperformer-195-webhelp-en/SILKPERF-F26CE9A9-PERFSCALABILITYMATRIX-REF.html>. [Accessed: 16- September- 2021].
- [83] R. Van Der Leek, J. Visser, G. Wijnholds, S. Rigal and P. V. Eck, "What is maintainability?", *O'Reilly Media*, 2016. [Online]. Available: <https://www.oreilly.com/content/what-is-maintainability/>. [Accessed: 05- September- 2021].

APPENDIX

1A

Sails.js:

File: lib/hooks/pubsub/index.js



```
* Usage:
*
*   ``
*   getUrlFor('DuckController.quack');
*   // => '/ducks/:id/quack'
*
*   getUrlFor({ target: 'DuckController.quack' });
*   // => '/ducks/:id/quack'
*   ``
*/
```

1B

Example Comment styles

```
/**
 * Initialize a new routing Layer with given `method`, `path`, and `middleware`.
 *
 * @param {String|RegExp} path Path string or regular expression.
 * @param {Array} methods Array of HTTP verbs.
 * @param {Array} middleware Layer callback/middleware or series of.
 * @param {Object=} opts
 * @param {String=} opts.name route name
 * @param {String=} opts.sensitive case sensitive (default: false)
 * @param {String=} opts.strict require the trailing slash (default: false)
 * @returns {Layer}
 * @private
```

CODE FILES

Express.js

```
'use strict'

const express = require('express')
const app = express()
app.get('/get', (req, res) => {
  res.send({ message: 'Hello world' })
})
app.post('/post', (req, res) => {
  res.send("ok")
});
app.put('/put', (req, res) => {
  res.send("ok")
});
app.delete('/delete', (req, res) => {
  res.send("ok")
});

app.listen(3000)
```

Fastify.js

```
'use strict'

const app = require('fastify')()
app.get('/get', (req, res) => {
  res.send({ message: 'Hello world' })
})
app.post('/post', (req, res) => {
  res.send({ message: 'ok' })
  res.send();
});
app.put('/put', (req, res) => {
  res.send({ message: 'ok' })
  res.send();
});
app.delete('/delete', (req, res) => {
  res.send({ message: 'ok' })
  res.send();
});

app.listen(3000)
```

Hapi.js – P1

```
'use strict'

require('make-promises-safe')
const Hapi = require('@hapi/hapi')
async function start () {
  const server = Hapi.server({ port: 3000, debug: false })
  server.route({
    method: 'GET',
    path: '/get',
    config: {
      cache: false,
      response: {
        ranges: false
      },
      state: { parse: false }
    },
    handler: function (request, h) {
      return { message: 'Hello World' }
    }
  })
  server.route({
    method: 'POST',
    path: '/post',
    config: {
      cache: false,
      response: {
        ranges: false
      },
      state: { parse: false }
    },
    handler: function (request, h) {
      return { message: 'ok' }
    }
  })
  server.route({
    method: 'PUT',
    path: '/put',
    config: {
      cache: false,
      response: {
        ranges: false
      },
      state: { parse: false }
    },
    handler: function (request, h) {
      return { message: 'ok' }
    }
  })
}
```

```
})
```

Hapi.js – P2

```
server.route({
  method: 'DELETE',
  path: '/delete',
  config: {
    cache: false,
    response: {
      ranges: false
    },
    state: { parse: false }
  },
  handler: function (request, h) {
    return { message: 'ok' }
  }
})
await server.start()
}
start()
```

Koa.js

```
'use strict'

const Koa = require('koa')
const router = require('koa-router')() // middleware
const app = new Koa()

router.get('/get', async function (ctx) {
  return ctx.body = ({
    message: 'Hello World!'
  })
}))

router.post('/post', async function (ctx) {
  return ctx.body = ({
    message: 'ok'
  })
})

router.put('/put', async function (ctx) {
  return ctx.body = ({
    message: 'ok'
  })
})

router.delete('/delete', async function (ctx) {
  return ctx.body = ({
    message: 'ok'
  })
})

app.use(router.routes())
app.listen(3000)
```

Moleculer.js

```
"use strict";

const ApiGateway = require("moleculer-web");
module.exports = {
  name: "api",
  mixins: [ApiGateway],
  // http://moleculer.services/docs/moleculer-web.html
  settings: {
    port: process.env.PORT || 3000,
    routes: [{
      aliases: {
        "GET /get"(req, res) {
          res.setHeader('Content-Type', 'application/json');
          res.end(JSON.stringify({ message: "Hello World!" }));
        },
        "POST /post"(req, res) {
          res.setHeader('Content-Type', 'application/json');
          res.end(JSON.stringify({ message: "ok" }));
        },
        "PUT /put"(req, res) {
          res.setHeader('Content-Type', 'application/json');
          res.end(JSON.stringify({ message: "ok" }));
        },
        "DELETE /delete"(req, res) {
          res.setHeader('Content-Type', 'application/json');
          res.end(JSON.stringify({ message: "ok" }));
        },
      }
    ]
  }
};
```

Restify.js

```
'use strict'

const restify = require('restify')

const app = restify.createServer()

app.get('/get', (req, res) => {
  res.send({ message: 'Hello world' })
})

app.post('/post', (req, res) => {
  res.send({ message: 'ok' })
});

app.put('/put', (req, res) => {
  res.send({ message: 'ok' })
  res.end(); // end the response
});

app.del('/delete', (req, res) => {
  res.send({ message: 'ok' })
  res.end(); // end the response
});

app.listen(3000)
```

Sails.js:

1) Route-mappings:

```
module.exports.routes = {

  'GET /get': { controller: 'Vcontroller', action: 'get' },
  'POST /post': { controller: 'Vcontroller', action: 'post' },
  'PUT /put': { controller: 'Vcontroller', action: 'put' },
  'DELETE /delete': { controller: 'Vcontroller', action: 'delete' },
};
```

2) Vcontroller

```
module.exports = {

  get: function(req,res){
    res.send({message: 'Hello world' });
  },
  post: function (req,res){
    res.send({ message: 'ok' })
    res.end(); // end the response
  },
  put: function (req,res){
    res.send({ message: 'ok' })
    res.end(); // end the response
  },
  delete: function (req,res){
    res.send({ message: 'ok' })
    res.end(); // end the response
  }
};
```

```
}  
  
};
```

Total.js

```
'use strict'  
  
const total = require('total.js')  
  
total.http('release', {  
  ip: '127.0.0.1',  
  port: 3000  
})  
  
const ok = "ok"  
  
total.route('GET /get', function response () {  
  this.json({ message: 'Hello world' })  
})  
  
total.route('POST /post', function response () {  
  this.json(message: 'ok' )  
})  
  
total.route('PUT /put', function response () {  
  this.json(message: 'ok' )  
})  
  
total.route('DELETE /delete', function response () {  
  this.json(message: 'ok' )  
})  
})
```