CHALMERS
UNIVERSITY OF TECHNOLOGY

A schematic for comparing web backend application frameworks

With regards to responsiveness and load scalability

Master's thesis in Software Engineering

Mathias Dosé
Hampus Lilja

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden 2015

Master's thesis 2015:June

A schematic for comparing web backend application frameworks

With regards to performance and scalability

MATHIAS DOSÉ
HAMPUS LILJA

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Division of Software Engineering
Chalmers University of Technology
Gothenburg, Sweden 2015

A schematic for comparing web backend application frameworks
With regards to responsiveness and load scalability
MATHIAS DOSÉ
HAMPUS LILJA

Supervisor: Morgan Ericsson, Department of Computer Science and Engineering
Examiner: Miroslaw Staron, Department of Computer Science and Engineering

Master's Thesis 2015:June
Department of Computer Science and Engineering
Division of Software Engineering
Chalmers University of Technology
SE-412 96 Gothenburg

Abstract

This thesis aim to engineer a schematic that is to be used for evaluating and comparing web backend application frameworks with regard to responsiveness and load scalability. A partial study was conducted in order to determine what affects these characteristics of performance in a web backend application framework. These findings were named and defined as influences, which where mapped to microbenchmarks called test types. These test types in combination with methods to visualize the results constitute the schematic. The schematic was evaluated through a running example where Express.js and .NET MVC was compared. The schematic proved successful as it was possible to draw a conclusion about these frameworks from the results in the context of use.

Keywords: web, backend, application, framework, benchmark, performance, scalability, Node.js, ASP.NET, MVC

Acknowledgements

The following thesis in Software Engineering was performed during the spring of 2015 at Chalmers University of Technology within the department of Computer Science and Engineering. We would like to thank our supervisor Morgan Ericsson for his assistance, engagement and counseling throughout this thesis, as well as our examiner Miroslaw Staron for his feedback and guidance.

Mathias Dosé, Hampus Lilja, Gothenburg, June 2015

Contents

Acronyms **xi**

List of Figures **xii**

List of Tables **xiv**

1 Introduction **1**
1.1 Background 1
1.2 Purpose of study 2

2 Foundation **3**
2.1 Performance in web applications 3
2.2 Web backend application framework 4
2.3 Benchmarks 5
2.3.1 Micro- vs Macrobenchmark 6
2.3.2 Techempower 7
2.3.3 PayPal 8

3 Research Method **9**
3.1 Design Science Research Methodology 9
3.2 Identifying influences of a WBAF 10
3.3 Constructing the Schematic 12
3.4 Evaluating the Schematic 13

4 Influences of a WBAF **15**
4.1 Request routing 17
4.2 Serialization 17
4.3 Template Engine 18
4.4 O*M 18
4.4.1 ORM 19
4.5 Cache client 20
4.6 Authentication 21
4.7 JSON Web Token 21
4.8 WebSocket 23

5 Schematic **25**
5.1 Environment and tools 25

5.2 Concurrences . . . 26
5.3 Charts . . . 26
5.3.1 Throughput Line Chart . . . 27
5.3.2 Throughput Comparison Line Chart . . . 27
5.3.3 Throughput Bar Chart . . . 28
5.3.4 Throughput Comparison Bar Chart . . . 29
5.3.5 Response time Line Chart . . . 30
5.3.6 Response time Comparison Line Chart . . . 30
5.3.7 Response time Bar Chart . . . 31
5.3.8 Response time Comparison Bar Chart . . . 32
5.4 Test types . . . 32
5.4.1 Request Routing . . . 33
5.4.2 Serialization . . . 35
5.4.3 O*M . . . 37
5.4.4 Template Engine . . . 38
5.4.5 JSON Web Token . . . 40
5.4.6 Cache client . . . 42
5.4.7 Websockets . . . 43
5.4.8 Authentication . . . 44

6 Express.js vs .NET MVC 45
6.1 Use case . . . 45
6.2 Specifications . . . 46
6.3 Test types . . . 46
6.3.1 Request Routing . . . 47
6.3.2 Serialization . . . 49
6.3.3 O*M . . . 50
6.3.4 Template engine . . . 53
6.3.5 JSON Web Token . . . 55
6.3.6 Cache . . . 56
6.3.7 Websockets . . . 58

7 Evaluation 59
7.1 Threats to Validity . . . 60

8 Conclusion 61
8.1 Future work . . . 62

Bibliography 65

Acronyms

AFBW Applicable For Building WBAF.

AGR Active GitHub Repositories.

CoC convention over configuration.

DIRT data intensive real-time.

DRY do not repeat yourself.

DSRM Design Science Research Methodology.

HTTP Hyper Text Transfer Protocol.

JIT just in time.

JWA JSON Web Algorithm.

JWT JSON Web Token.

MPFF Most Popular Fullstack Framework.

MVC model view controller.

ORM object relational mapper.

OS operating system.

OVB omitted-variable bias.

PM package manager.

RBC Response time Bar Chart.

RCBC Response time Comparison Bar Chart.

RCLC Response time Comparison Line Chart.

RLC Response time Line Chart.

SPA single page application.

TBC Throughput Bar Chart.

TCBC Throughput Comparison Bar Chart.

TCLC Throughput Comparison Line Chart.

TLC Throughput Line Chart.

TLFAPM Top List Found of Any Package Manager.

WBA web backend application.

WBAF web backend application framework.

WS WebSocket.

List of Figures

3.1 The DSRM process constructed by Peffers et al. 9
3.2 The process of deriving influences from programming languages . . . 12

4.1 A template engine processes a template and data to HTML 18
4.2 Illustration of how a string is encrypted and decrypted using either a secret or private/public key architecture. 22
4.3 WebSocket sequence diagram. 24

5.1 The environment prerequisite for executing the test types in the schematic. 25
5.2 Example TLC showing throughput with regard to single test type variable for different concurrency levels 27
5.3 Example TCLC illustrating a comparison of the throughput with regard to single test type variable for two WBAF's 28
5.4 Example TBC showing throughput with regard to multiple variants of a test type for different concurrency levels 29
5.5 Example TCBC illustrating a comparison of the throughput with regard to multiple variants test type for two WBAF's 29
5.6 Example RLC illustrating response time with regard to single test type variable for different concurrency levels 30
5.7 Example RCLC illustrating comparison of the response time with regard to test type variable for two WBAF's 31
5.8 Example RBC showing response time with regard to multiple variants of a test type for different concurrency levels 31
5.9 Example RCBC comparing two WBAFs by showing response time with regard to multiple variants of a test type 32
5.10 Illustration of when to add the routes in **R5**. 34
5.11 Illustration of how the two templates in the test type should be constructed. 39

6.1 Request routing .NET TLC 48
6.2 Request routing Express.js TLC 48
6.3 Request routing .NET RLC 48
6.4 Request routing Express.js RLC 48
6.5 Request routing C10 TCLC 48
6.6 Request routing C10 RCLC 48

6.7 Serialization .NET MVC TLC 49
6.8 Serialization Express.js TLC 49
6.9 Serialization .NET MVC RLC 50
6.10 Serialization Express.js RLC 50
6.11 Serialization C10 TCLC 50
6.12 Serialization C10 RCLC 50
6.13 Illustrating a case of not utilizing full CPU capacity in the O*M test type as only three threads are awake. 51
6.14 O*M .NET MVC TBC 52
6.15 O*M Express.js TBC 52
6.16 O*M .NET MVC RBC 52
6.17 O*M Express.js RBC 52
6.18 O*M C10 TCBC 53
6.19 O*M C10 RCBC 53
6.20 Template engine .NET MVC TLC 54
6.21 Template engine Express.js TLC 54
6.22 Template engine .NET MVC RLC 54
6.23 Template engine Express.js RLC 54
6.24 Template engine C10 TCLC 54
6.25 Template engine C10 RCLC 54
6.26 JSON Web Token .NET MVC TBC 55
6.27 JSON Web Token Express.js TBC 55
6.28 JSON Web Token .NET MVC RBC 56
6.29 JSON Web Token Express.js RBC 56
6.30 JSON Web Token C10 TCBC 56
6.31 JSON Web Token C10 RCBC 56
6.32 Cache .NET MVC TLC 57
6.33 Cache Express.js TLC 57
6.34 Cache .NET MVC RLC 58
6.35 Cache Express.js RLC 58
6.36 C10 TCLC 58
6.37 C10 RCLC 58

List of Tables

4.1 Result of the partial study to compile the applicable programming languages to derive from. AGR - Active GitHub Repositories, AFBW - Applicable For Building WBAF, TLFAPM - Top List Found of Any Package Manager, MPFF - Most Popular Fullstack Framework 15
4.2 Result of the inspection of the top lists of package managers. 16
4.3 Result of the inspection of the contents of fullstack frameworks. . . . 16
4.4 JWA's to use in JWT specified by IETF. 23

5.1 The values of concurrences to use in the test types 26
5.2 The values of variables to use in request routing benchmark 34
5.3 The values of variables to use in request routing benchmark 36
5.4 The variants in the O*M benchmark 38
5.5 The values of variables to use in template engine benchmark 39
5.6 By BlueKrypt, calculated key sizes in bits safe for use in 2015 by using methods from different publications. 40
5.7 The variants in the JSON Web Token test type 42
5.8 The number of data entries that shall be fetched from the external cache in Cache benchmark 43
5.9 The length of each message to be sent by the benchmarking tool. . . 44

1

Introduction

1.1 Background

The web is still a young phenomena that is evolving at a rapid rate, the complex data intensive real-time (DIRT) [32] applications and the responsive mobile applications that is seen today have come a long way from the static web sites of the 90's. Companies that wish to stay competitive with their products and consulting services are required to keep an eye open for new technologies, to mitigate the risk of what they are providing becoming outdated.

Selecting a technology stack when developing a new web application is a complex task, where one major decision is to choose a web backend application framework (WBAF). There are different ways to deal with this decision, ranging from purchasing reports from technology research companies to establishing an evaluation committee that would by consensus reach a decision. If there would be no significance on time to market and developer cost would be zero, the ultimate way to evaluate a WBAF would be to build the whole application in the WBAF's of interest and simply measure and compare the results in a production environment. Naturally this is an unrealistic approach because of the mentioned prerequisites.

Groupon allocated three months to meetings and research in their decision of a WBAF when rebuilding their backend [84]. They elicited a set of requirements that the application should meet, and evaluated a couple of WBAF's by building simple prototypes and comparing their performance. The creation of these prototypes enabled the developers not only to evaluate the performance criteria, but also the developer motivation to work in said WBAF and its ecosystem maturity. Naturally they didn't affirm the choice solely on these artificial experiments, rather it enabled them to make a more thought through and contemplated choice based on the additional knowledge acquired.

The WBAF evaluation criteria are contextual, meaning what's an important requirement for one project might be insignificant for another. Evaluation criteria can be developer productivity, scalability or ecosystem maturity etc. A reason to the complexity of choosing a WBAF is that the importance of the evaluation criteria varies to this extent. Another reason to why it is so difficult is the absence of open objective opinions of when to use which framework. There is a lot of evangelists for every WBAF that proclaim that their WBAF is the solution to the problem.

One evaluation criteria being scalability, is defined by [1] as the ability of a system to handle growth in traffic (load scalability), and the ability to expand in a chosen dimension without architectural modifications (structural scalability). Responsiveness is the ability of a system to meet its objectives for response time or throughput.

Throughput can be measured by how many requests per second an application can manage for a given hardware setup. More efficient solutions will require less resources, especially in multibillion dollar businesses where every percentage gain in hosting costs can have huge impact on the revenue [101].

In a web backend application context, response time refers to how much time it takes for the server to respond to a specific request, i.e the request latency [38]. With the web transitioning from the concept of returning static HTML pages to todays AJAX heavy single page applications (SPA), not only has the throughput demand of the applications seen an huge increase, but also the significance of the applications to provide good response time [4]. Responsiveness being a key ingredient in SPA's stress the importance of a low request latency, especially on the mobile platform where the connectivity usually are worse. Amazon did A/B tests on this matter and results showed that an increased latency of 100ms decreased the sales with 1%. Google noticed during experiments that an increase of 0.5s in requesting a search resulted in 20% decrease in traffic and revenue [26].

1.2 Purpose of study

The aforementioned complexity of conducting a deliberate choice of WBAF in combination with the claimed significance of the evaluation criteria load scalability and responsiveness, leads to the demand of the product of this thesis with the main goal and research questions as followed.

The goal of this thesis is to analyze web backend application frameworks to evaluate what affects its performance with respect to responsiveness and load scalability.

RQ1 What affects the load scalability and responsiveness of a web backend application framework?

RQ2 How to compare load scalability and responsiveness of web backend application framework's?

This thesis will produce a schematic with a set of test types that will be used to conduct a comparison between different WBAFs.

2

Foundation

2.1 Performance in web applications

Smith and Williams defines performance as *the degree to which a software system or component meets its objectives for timeliness* [16]. There are many dimensions of performance, and two important ones are responsiveness and scalability. Responsiveness is the ability of a system to meet its objectives for response time or throughput. Response time being the time required to respond to an event, and throughput is the number of events processed in a specified duration [43]. Scalability is defined by Bondi as the ability of a system to continue to fulfill its requirement of responsiveness as the demand for the software function increases (load scalability), and the ability to expand in a chosen dimension without architectural modifications (structural scalability) [1].

In the context of a WBA, throughput refers to number of requests being processed during a specified duration, and response time refers to how long time it take for the WBA to respond to the incoming requests.

Load scalability is, in the context of a WBA, the ability to continue to serve requests with the required response time when the amount of concurrent requests are increased. Furthermore, structural scalability is, in the context of a WBA, the ability to deploy additional processes of a specific WBA behind a load balancer to ensure that the system is load scalable [92]. In practise this usually means that the new WBA process is hosted on a new node, as the reason for deploying it is that the current nodes have reached a resource bottleneck. WBAs that are more load scalable will thus require less nodes, which results in a lower hosting cost for the company [101].

Performance is a pervasive quality of software systems; everything affects it, from the software itself to all underlying layers, such as operating system, middleware, hardware, communication networks, etc [49]. These elements that affect the performance of an entity are called Influences in this thesis. These Influences are characterized by a transitive relation, thus the web backend application framework (WBAF) will affect the web application performance.

2.2 Web backend application framework

There are some ambiguities to what a web application actually is. Some refers the term to the application that is rendered and then executed in the browser, i.e the HTML and Javascript, while other definitions include or limits the definition to the part which is run on the server [17, 105]. AngularJs [7] - a client side Javascript framework, and Express.js [22] - a server side Javascript framework, is both defined on their respective homepage as frameworks for building web applications. The definition that is going to be used in this thesis is; a web application comprise all the elements that are required for the web based solution to meet its functional requirements.

With the term web application occupied with the definition in this chapter, in combination with the demand for a common term that refers to the backend part of the web application, the term web backend application (WBA) has been coined. The WBA comprise the operating system (OS), the web server, the platform, the WBAF and the implementation.

The ambiguity when defining web application naturally follows when trying to define web application framework, thus usage of that term has been avoided. Aligned with the newly coined term web backend application, a framework for building such applications is consequently termed web backend application framework (WBAF).

In a sense all web sites works in the same way. A client, which is often represented by a browser, makes a request to a WBA. The request is routed to a machine that hosts a WBA. The WBA interprets the request and sends a response to the client. This means that there are a lot of functionality that has to act in the same way for each and every WBA. To make web development less tedious and repeatable, libraries that makes it easier to communicate were built for different platforms [41].

The term web server has two meanings, it can resemble the physical computer or the software that is used to host anything web related. However, in this thesis it's going to be used as reference to the latter. Web servers can host web applications or static files and provide a set of features, e.g load balancing and serving error pages. Examples are IIS [57], Apache [8] and Nginx [65]. Some platforms can act as its own web server, NodeJs is an example platform of this phenomena.

Libraries are also used to abstract other types of common or complex functionalities in web development. Some examples of those functions are database-access, caching, authentication etc. When multiple libraries that concerns web backend development are combined and labeled it is called a WBAF. A WBAF is not required to have a specific set of functionalities to be called a WBAF, what's included is up to the author. It can range from just including the minimum abbreviations such as communicating with the client to communicating with the database and all of the logic in between.

In the web context, a platform refers to the underlying system or API on which web applications or web sites can run [104, 24, 105]. Platforms are commonly mistaken for the programming language, i.e that C# [54] would be a platform, which isn't

the case. Platforms doesn't have a one-to-one relation with the language itself. An example is both the .NET [55] and Mono [60] platforms use C# as programming language while still being two different platforms.

The abstraction of commonly needed functionality does not only help developers with complex and tedious tasks, it also helps to enforce some of the software engineering principle's such as do not repeat yourself (DRY), which means that code repetition should be kept at a minimum. Every WBAF enforces the DRY principle to some degree by keeping shared functions in libraries, thus decreasing the need of manually written code at different places in the application [27]. Other good practises that can be enforced by the WBAF are for example the paradigms such as convention over configuration (CoC) and patterns like model view controller (MVC) [19].

2.3 Benchmarks

It is a cumbersome task to measure performance in computer systems. One reason is because of the difficulty to quantify the results of experiments performed on such a complex and dynamic system that have such a large number of ever changing parts as a computer system has. Some people will even go so far as to say that Computer Science is not real science because of it's inability to draw deterministic results from experiments [39]. Even so, scientists, engineers and organizations have tried to find ways of measuring and benchmarking computer components and software. One approach is to isolate specific parts in the system, thus making the environment more controlled. Most of the performance benchmarks published by researchers and organizations are done on hardware components such as CPU and GPU and not in the area of software performance [106]. One reason might be that it's easier to isolate a specific hardware component in a computer system then it is to isolate a software that runs in the system making use of several hardware components.

The first and most difficult hazard that should be recognized when benchmarking in Computer Science is the statistical hazard omitted-variable bias (OVB). Especially when trying to isolate a specific part in a system the risk of inflicting OVB increases. OVB appears when variables that have impact on the end result is, deliberately or unintentionally, not taken into the equation. For example, comparing the performance of a precompiled language against a just in time (JIT) compiled language will always benefit the precompiled language on short test runs. If the JIT's warm up time is not taken into consideration the test will not spend the same amount of time testing the actual code of the JIT language that it does testing the precompiled code thus making the JIT results skewed. There can be even more shifty variables that are harder to detect that inflicts unpredicted results. Some of these variables could be the compiler settings that was used when building the software, the heap size of the garbage collector, the operating system caching etc. A solution to tackle some of these hazards are to re-run the experiment in the same environment to get relevant result, and either include more data and variables in the equation or be as transparent as possible about the specifications and settings used when the exper-

iment was conducted so that the results could be analysed with their environment [15].

Second, there is a hazard that software have been tuned just to score higher in a set of specific benchmarks. One recent event that got much media attention was when it became known how Samsung and other phone producers had programmed their phone software to notice when a benchmark was performed on the system and then releasing more CPU and memory power from the system to the benchmark [85].

Third, when tests that are the basis for a benchmark gets known by companies that wants to compete on the ranking list they would want to improve their software in that specific area. This pursuit of higher rankings by the means of only focusing on increasing performance for the benchmark specific operations might decrease the performance in other areas that doesn't get the same amount of attention.

Fourth, benchmarks that are conducted by organizations that have something to gain on the scores should be scrutinized for biased test environments, test types, test permutations etc.

Finally, as for everything in Computer Science, if benchmarks are not frequently updated they soon get outdated. Outdated benchmarks can become trivial i.e. no longer test anything of importance or in worst case claim to test one thing while in practise they test a completely different thing due to speedup in some areas [15, 39].

In this thesis the focus is not primarily on benchmarking but instead on the actual comparison of WBAFs. This means that the result of testing a single WBAF doesn't say anything until there is another result from a different WBAF that has been tested on the same environment. The hazards described above does still apply to a comparison but in this case the environment can be tailored to the exact same environment as the one that will host the application. This mitigates the risk of getting different result when the final product is deployed.

2.3.1 Micro- vs Macrobenchmark

To avoid confusion and misunderstandings it is important to specify which type of benchmark that is used when benchmarking an application or software. In worst case scenario a microbenchmark recognized as a macrobenchmark can make an organization choose WBAF on the wrong premises.

Measuring the performance of an isolated operation in a web application is a typical use case of a microbenchmark. To be precise and to minimize the OVB hazard in a test, the code that executes the operation in question should be as simple as possible. By writing a simple and precise code it ensures that the exact steps for performing the operation of the benchmark is tested. This makes it easier to specify which part of the WBAF that have been tested. However, a real world application

often have more factors that affect the performance such as authentication, larger logic blocks, more complex data relations etc.

Macrobenchmark To measure the performance of a WBAF in a real world implementation a benchmark of the type macro needs to be conducted. Unlike microbenchmarks where the workload is usually set to a high generic value without a real scenario in mind, an analysis or a realistic idea of the workload that the actual application should be able to handle is important to have and to make use of in the benchmarks. As stated above a macrobenchmark should be performed on hardware and software that has similar setup as the environment that will run the application in production. Finally, a good macrobenchmark should run the tests for a longer period of time to get a more realistic time scenario than the microbenchmarks [25, 15].

2.3.2 Techempower

Techempower is a company that consults in the field of web development and their philosophy is that performance when developing for the web matters a great deal [94]. They also have, if not the only, one of the biggest ongoing WBAF performance comparison and benchmarking on the web. The benchmarks are focused on a couple of normally occurring actions performed by web applications such as JSON serialization, database queries and templating. The tests are run a couple of times per year and the results are published on their web page by ranking the WBAFs from best to worst throughput. They are transparent about which environment that is used and the code that is executed. The code is mostly contributed by the community surrounding the benchmark suite and everyone is welcome to contribute to the code base which is located on GitHub.

The code that is exercised in each test is just the bare minimum to be able to perform the operation specified for each test. This in combination with generic data and non-motivated workloads during the tests characterizes the Techempowers benchmarks of the microbenchmark type.

Critics of Techempower argues that it's only interesting to compare WBAFs that run on the same platform because of the big impact the platform has on the WBAFs performance. They also think that the tests are way to small and the time spent testing the actual logic is next to none compared to the time spent in the kernel of the web servers OS and the internal network communication. Instead they propose to extract the IO operations and test them in a separate test by varying the size of the header and entity values [50]. Another critic thinks that the measure request per second or response, in Techempowers tests, doesn't say anything about performance. He claims that when measuring performance the metric that should be in focus should instead be the duration of each request. The critic conclude that measuring requests per second only gives the throughput of the application and that is only interesting if one is building an application that needs to serve millions of concurrent users such as Facebook or Google [47].

When analysing the results of the different tests displayed on Techempowers website one framework stands out from the rest. Not that it outperforms the other frameworks but while this framework score among the top result on several tests there is little information of this framework on the web. The framework in question is Gemini, and when looked into it, it soon becomes clear that this is a WBAF developed by Techempower them self [10]. The code is not open sourced or publicly available which makes it impossible for outsiders to analyze it and to see what makes this WBAF so fast. Techempower never claim to be unbiased but they don't show any clear indication of the opposite either. Evidently Techempower is not completely unbiased and the fact that it's impossible to scrutinize their Gemini code in order to look for benchmark hazards such as software tuning they leave some unanswered questions.

With this said, Techempower can and should still be considered as a valid source of inspiration when it comes to benchmarking WBAFs. Thanks to the clear explanations and descriptions on how the benchmarks are constructed and executed they contribute with great value to how performance in WBAFs could be measured. Performance differences are important when choosing a framework, and Techempower does it's part to provide help with the choice of the best performing WBAF.

2.3.3 PayPal

In 2013 PayPal decided to look for alternatives for their WBAF. The reason was that it was hard to find fullstack developers, e.g. developers that could work with both the client- and the server-side logic. PayPal's original server platform was built with Java and the client side platform made use of JavaScript. The WBAF that made them interested for an alternative was Node.js which uses JavaScript on the server side as well. While the main reason wasn't the performance factor it would show that it was a considerable difference between the alternative WBAF's. In a blog post by Harrell, J. [40] he describes the test set up and results from the performed benchmark tests.

The benchmark is performed on the hardware that is used to host paypal.com and the tested routes executed the same logic that would be executed in production. Because of the real world environment these benchmarks can be categorized as macrobenchmarks.

While they display and described the results in a clear and concrete way they do not discuss why there are such a difference in performance. Also, by not releasing the source code that is executed in the tests it's impossible for outsiders to draw own conclusions on what it is that is effecting the performance and to look for benchmarking hazards. Some of the benchmarking hazards that these tests are prone to encounter are that the software for one of the WBAF might have been tuned and implemented with awareness of the benchmark in mind. Secondly, under the development of PayPal's new architecture a Node.js framework were constructed and while this framework is open source they still have a lot to gain if the community grows around their framework. This might make the results biased.

3

Research Method

3.1 Design Science Research Methodology

In Design Science Research Methodology (DSRM) there are 6 activities that should be performed and the iterative cycle could be done through activity 2-6. The activity that starts the research doesn't have to be activity 1. Depending on the context of the research the entry point to DSRM could vary from activity 1 through 4. A problem centered research should enter the process on activity 1 while researches that are more centered around the objective should enter on activity 2. If the design and development is the center of the research the entry point could be activity 3 and finally, if the research is initiated in a client context the 4th activity could be a suitable starting point. See figure 3.1 for the whole methodology process.

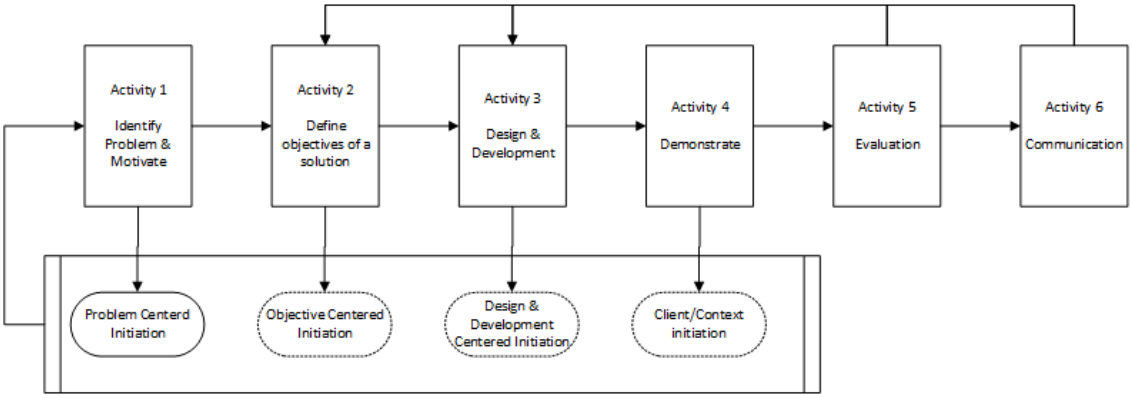

Figure 3.1: The DSRM process constructed by Peffers et al.

Activity 1 is the identifying and motivating activity. This is where the research is justified by defining and explaining the problem and the value that a solution to the problem would have. Activity 2 is where the means of achieving a solution to the problem are defined. The knowledge about the problem from activity 1 combined with knowledge about current solutions should be used to define possible and feasible solutions. In activity 3 an artifact is designed and constructed. A design research artifact can be anything that includes research contributions to solve the research problem. The demonstrating activity that is activity 4 is where the artifact constructed in the previous activity is demonstrated on its ability to solve parts of the research problem. Some of the demonstrating techniques that

could be conducted are case study, experiments, proofs etc. It is also important to demonstrate knowledge on how the artifact solves the problem. Activity 5 is where the evaluation of the solution takes place. The result from activity 4 is compared to the objective defined in activity 2 to observe how well the implemented solution performs in regards to the desired solution. It's important that there is knowledge on how to interpret the metrics and how to analyze the results. After activity 5 have been performed it is possible to iterate back to either activity 2 to find another solution or to activity 3 to improve the artifacts effectiveness to solve the problem. The final activity, activity 6, is where the research shall be communicated to the relevant audience through different publication channels. It's important that the research paper is written in a clear and professional way to get the audience to focus on the important aspects of the research. After this activity it is also possible for external researcher to either continue the research or iterate back to complement or extend the current research [42].

Hevner et al. [5] have established seven guidelines that works as a basis for Design researchers that strives for effective design-science research. These guidelines are:

3.2 Identifying influences of a WBAF

Influences are elements that affect the performance of an entity. They are characterized by an transitive relation, meaning that if A influences B and B influences C, then A will also influence C. This term was coined early in this project with the purpose of enabling a common way of referring to these elements.

In order to engineer the test types for the schematic, the influences of a WBAF had to be identified. However, there is no definition or limit on how much functionality should be included in a WBAF, that's a decision for each creator of each WBAF. Consequently, there was no way of defining all the influences of a WBAF. Instead test types for the most used influences was developed, therefor these had to be identified. In order to identify these popular influences, the nature of the concept micro framework and fullstack framework was utilized in a small study. This study resembles activity 2 in the DSRM process.

A fullstack framework comes packed with all necessary features for development of a WBA. However, a fullstack WBAF doesn't necessarily limit the developer to use the provided features. An example is the ASP.NET MVC fullstack WBAF that comes with an ORM component called Entity Framework [58], which can be replaced with another ORM component of choice, e.g NHibernate [66].

A micro framework applies the mindset of a minimal and extensible core, where features are applied as optional packages rather than out of the box. However, request routing must be included. Examples are the Express.js framework for Node.js, or Flask [23] for Python, they don't come with features like an ORM component or JSON web token out of the box, however these features are available as packages. These packages usually serve a single purpose, and its category can be directly

mapped to an influence of a WBAF. Installing and managing these packages are most commonly done in a so called package manager (PM). Homepages or third party sites of these PM's usually supply a top list of the most downloaded packages, which was inspected in order to identify the most used influences. This was done for the top lists of the most popular package manager of the most popular programming languages that are Applicable For Building WBAF.

In order to evaluate which programming language was the most popular, its Active GitHub Repositories (AGR) was examined. If the language then fulfilled the criteria of being Applicable For Building WBAF (AFBW) and Top List Found of Any Package Manager (TLFAPM), it was then chosen as one of the six programming languages in the study. Later in the study the Most Popular Fullstack Framework (MPFF) of each language would be needed, thus it was also included in this partial study. Which fullstack framework was MPFF for each language had already been calculated by HotFrameworks.com [30] and was thus used in this study. HotFrameworks.com combines GitHub and StackOverflow score [89] to determine popularity. The GitHub score is based on the number of stars the framework's repository has on GitHub, and the StackOverflow score is based on the number posts tagged with the framework in question [31], both scores are on a log scale.

The inspection of each top list of the six package managers was done according a predefined process defined in the process diagram in figure 3.2. Some packages was only used for development, e.g testing or building, thus these was skipped. Out of the packages that are to be used during run time, a best guess approach was used to determine if the category of these could be translated into an influence of a WBAF, this approach sufficed as a first filter. Furthermore, there were a lot of duplicate packages in the same category, e.g "fancyName-O*M-mysql" and "otherFancyName-O*M-mongodb" both belongs to the O*M category, these were distinctly added with regards to its category. Finally, the inspection prolonged until at least 10 possible influences were found and 150 packages had been inspected. If the influence then was found in more than half (three) of the result sets in this partial study, then it was considered a popular influence in a majority of the package managers, and would thus progress to the next part of the study.

The nature of fullstack frameworks were used as a second filter to verify that the best guessed influences found in the previous part of the study actually were influences of a WBAF. What characterizes a fullstack framework is that it contains all the necessary, in the creator's point of view, features to build WBA's. Consequently, the best guessed influences in the previous part of the study was matched against the content of the Most Popular Fullstack Framework (MPFF) of the most popular programming languages in the first part of this study. When evaluating if an influence is part of the fullstack framework, both its core assembly and its dedicated packages were examined. The requirements for a package to be dedicated to a fullstack framework was that it had to have a dependency on the core, and it had to be created or verified by the creator of the core. If the influence then was part of more than or equal of half (three) of the fullstack frameworks it was considered one of the more popular WBAF influences, and was thus included as a test type in the schematic.

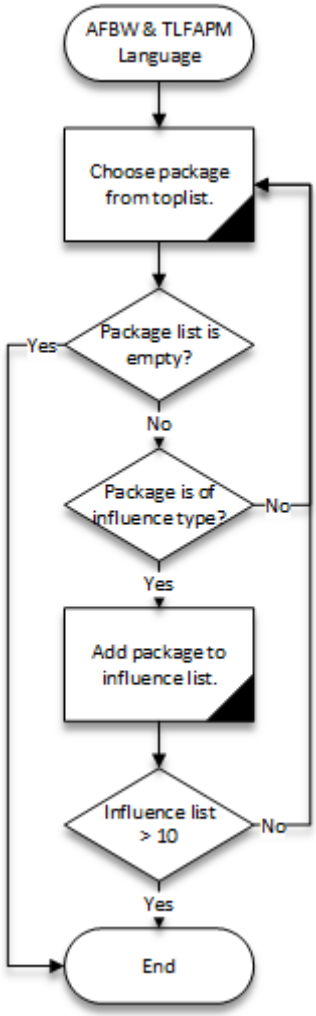

Figure 3.2: The process of deriving influences from programming languages

3.3 Constructing the Schematic

The artifact of this thesis, i.e. the schematic, consists of a number of test types, where each test type represent one of the derived influences. The structure and the content of the schematic and each test type were initially unknown, and was gradually reformed in an iterative manner through activity 3 to 5 in the DSRM process.

In order to engineer the test types, a deeper understanding of each influence was required. Therefor research about each of these influences were conducted and compiled. The aim of this in depth research was to find out how the influence affected the responsiveness and load scalability of a web backend application. The research and the development of the test types were done iteratively. The implementation of each test type were developed in one of the example WBAFs, i.e. .NET MVC or Express.js. When the test type was considered complete, its understandability was evaluated by implementing the test type in the other WBAF that hadn't been used

in the first place.

With the in depth analysis of the influences it could be determined if the test type would be engineered with an adjustable variable that would determine the result, or with multiple variants of same test type. If the influence had several features or possible usages, it would be engineered as a variant test type. Complexity in visualization delimited the engineering of combining variable and multiple variants within the same test type. During development local benchmarks were performed to see which variables could be adjusted to make an impact on the responsiveness and load scalability.

When a couple of test types had been engineered they were analyzed and common denominators were identified. This was done in order to give the test types a more unified look thus lowering the threshold for future users to understand the process of implementing these tests. The analyse also showed that the results from each test could be streamlined into different charts, making the performance impact more visible. These charts could also be given a unified look where line charts were used to display results from test types were a variable impacted the result and bar charts for test types that had multiple variants.

3.4 Evaluating the Schematic

When the test types had been constructed they were evaluated by running the tests in accordance to the specifications given by the schematic. The implementation that was executed for the test was the code developed during the construction of the test type. If the results from the tests seemed accurate with regard to the knowledge gained from the in dept research about the influence it was seen as a valid test type and example implementation. This was a large evaluation process and the last time activity 5 in the DSRM process was performed.

The graphs in the schematic was used to evaluate the results of the test types. If the results was in some way unexpected or peculiar then a reasoning behind these were done.

4

Influences of a WBAF

This chapter presents the results of the partial study to identify the influences described in section 3.2, and the compiled research of each of these. Table 4.1 illustrates the results of the first part of the influence study. The languages that are marked as bold in the table are the ones that fulfilled the criteria of being Applicable For Building WBAF and Top List Found of Any Package Manager. Furthermore, the package managers and Most Popular Fullstack Framework found for these languages are presented in the table as well.

Table 4.1: Result of the partial study to compile the applicable programming languages to derive from. AGR - Active GitHub Repositories, AFBW - Applicable For Building WBAF, TLFAPM - Top List Found of Any Package Manager, MPFF - Most Popular Fullstack Framework

Language	AGR [28]	AFBW	TLFAPM	PM	MPFF
Javascript	323,938	Yes	Yes [71]	NPM [70]	Sails [86]
Java	222,852	Yes	Yes [64]	Maven [51]	Spring [88]
Python	164,852	Yes	Yes [79]	PyPI [78]	Django [20]
CSS	164,585	No	-	-	
PHP	138,771	Yes	Yes [77]	Packagist [76]	Symfony [91]
Ruby	132,848	Yes	Yes [11]	Gems [83]	Rails [82]
C++	86,505	Yes	No	-	
C	73,075	Yes	No	-	
Shell	65,670	Yes	No	-	
C#	56,062	Yes	Yes [73]	Nuget [72]	.NET MVC [56]

Table 4.2 visualize the result of the next part of the influence study, i.e the inspection of the top lists of the package managers. All possible influences found in any of the top list are presented in the leftmost column. The remaining columns illustrates whether each possible influence was found in that specific package manager top list. The influences marked in bold are the ones found in more than half, i.e three, and was thus progressed to the final part of the influence study.

The results of the final part of the influence study are presented in table 4.3. The influences that are marked in bold were found in more than half of the fullstack frameworks derived from the first part of the influence study. These are the influences that will be concerned in the schematic, and the test types will be engineered

Table 4.2: Result of the inspection of the top lists of package managers.

Influence	NPM	Maven	PyPI	Packagist	Gems	Nuget
Internationalization				●	●	
Serialization		●	●	●	●	●
O*M	●	●	●	●	●	●
Template engine	●	●	●	●	●	●
Mail client	●	●		●	●	
HTML Parser	●	●	●		●	
Javascript executer					●	
HTTP client	●	●	●	●	●	●
SSL	●	●	●	●	●	
Cache client	●	●	●		●	
Authentication	●	●	●	●	●	●
Token generator	●	●	●	●	●	●
Websocket	●				●	●
Async			●	●	●	●

to cover them. The influence called Request routing will be added to the seven influences that this partial study resulted in, the reason behind this is that a WBAF will, by definition 2.2, always contain this influence and is thus the most used influence of them all. In order to engineer test types that actually will benchmark these influences, one must know what they actually are. In the following sections the required knowledge about each influence will be compiled, which will lay a foundation for the test types.

Table 4.3: Result of the inspection of the contents of fullstack frameworks.

Influence	Sails	Spring	Django	Symfoni	Rails	.NET MVC
Serialization		●	●	●	●	●
O*M	●	●	●	●	●	●
Template engine	●	●	●	●	●	●
Mail client		●	●	●		
HTML parser				●		
HTTP client	●	●				
SSL						
Cache client	●	●	●	●	●	
Authentication	●	●	●	●		●
Token generator	●	●	●	●	●	
Websocket	●	●	●		●	●
Async		●				

4.1 Request routing

Hyper Text Transfer Protocol (HTTP) is the transfer protocol that is used throughout the entire world wide web to transfer data between all its nodes. In order for a web backend application to determine what task it should perform for a specific HTTP message it receives, request routing functionality is implemented [41].

The first part of a HTTP message is called the Request-Line and it contains the method token, the request URI and the protocol version used in the message [36], see listing 4.1 for a visualization. The request routing will use the two first parts of the Request-Line, i.e the mehod token and the request URI, to determine what to do for said request. In practise this means that the method and the URI will be mapped to a certain function. With the example used in listing 4.1, GET and /login would together be used to determine which function should be excecuted. In order to do this, the WBAF must somehow store all the routes that is created at the application start up. When a user then connects to a certain route, the WBAF will do a lookup to see which function should be executed for that specific route.

Listing 4.1: Template and example of the Request-Line of a HTTP message

```
% Technichal template for the Request-Line
Method SP Request-URI SP HTTP-Version CRLF

% An example of the Request-Line
GET /login HTTP/1.1
```

The request routing functionality is something that is essential for every web application, and it would be redundant to write this functionality for each application, thus it's implemented in every WBAF. However, how the different WBAF's implement request routing differs, and how efficiently this is done has an influence on responsiveness and load scalability [41].

4.2 Serialization

Serialization is the process of converting the state of an object instance into a binary or textual form, this is most commonly done in order to persist it into storage or transport it over a network [95]. The latter use case is realized when a WBA and a client are exchanging data. Serialization is done according to a serialization format, which in HTTP is specified by the Content-type header. The value of this header is called an Internet media type, and it is an identifier that indicates what type of data the message contains [36]. There is an extensive amount of different media types allowed, where some of the most common are text/xml, application/json or text/csv.

4.3 Template Engine

In order to make a web page dynamic, it's creation must be controlled by a template engine on either the client with the help of Javascript, or by the server. In some cases both options are used. A template engine processes a template together with data, often from a data source, and produces a HTML page ready to be returned to the requesting client, as shown in figure 4.1.

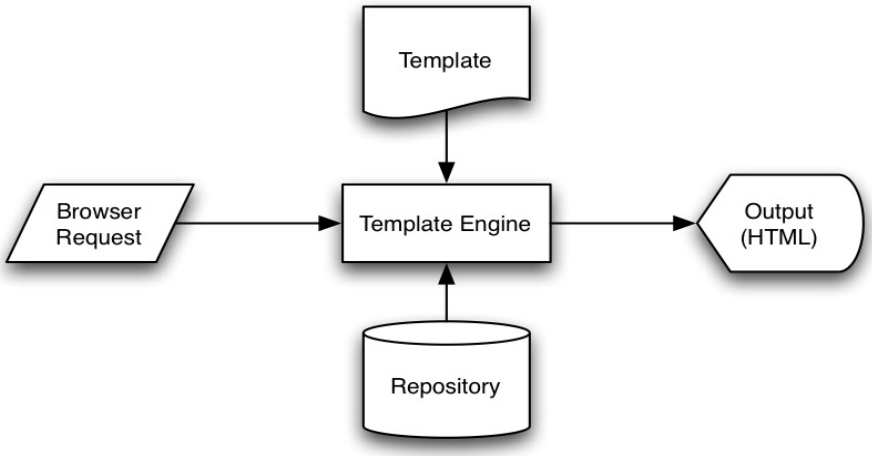

Figure 4.1: A template engine processes a template and data to HTML

Template engines doesn't only inject data into placeholders, other common features include variable declaration, string escaping, looping and more.

4.4 O*M

If a database is used in a web application then it is the web backend application's task to communicate with that database. To be able to communicate with the database and exchange data with it some type of functionality needs to be implemented in the WBA. Not only does the WBA need to set up a secure connection between the database and the WBA it also needs to map the data that is interchanged with the database. These actions can be abstracted by an Object * Mapper (O*M) component. The O*M is primarily responsible for abstracting the most common database interactions such Create, Read, Update and Delete (CRUD) by mapping objects written in backend code to data in a database. Backend platform objects that represents data from a database are referenced as entity objects. The * in the O*M represents the type of the database. O*M components are often used when interacting with relational databases, i.e object relational mapper (ORM), which virtually are SQL based databases. Recently other forms of databases have started to grow in popularity, some of them are Document, Key-Value and Graph based databases. These are often casually grouped together as NoSQL databases. The main difference between SQL and NoSQL is that the schema for SQL databases

is static while NoSQL databases usually have dynamic schema [59]. If an O*M component is present in the WBAF then it's probably one of the largest influences for that WBAF. As with every other influence, a WBAF doesn't need to include an O*M and it's only fullstack WBAFs that does include one out of the box.

4.4.1 ORM

As stated above, ORM is the most popular type of O*M and because of the complex nature of a relational database, this deserves a deeper explanation of what the ORM does to help developers.

To map a single table or column from a database to an entity object on the different programming platforms is an uncomplicated and straightforward implementation as long as there exists a valid connector between the platform and the database. But in a relational database the tables of data are usually related to other tables in more or less complex ways, where the relations are other than just one-to-one and multiple tables could inherit data columns from the same data table. On top of these complex relations a program will usually have entity objects representing data from multiple tables in it's runtime memory making the relation from the data in the database to the data in the program memory even more intricate. The more complex the data relations are the more complex the CRUD queries will be. It is for these complex environments that ORM can be really helpful. By abstracting the actual database queries and giving the programmer an entity object representing the data to work with instead [46].

ORMs can have different types of features and functions depending on how complex the implementation of the ORM is. Bauer and King categorize ORMs into four different categories in their book Hibernate in Action [14]. The categories are as follows:

Pure Relational
: When the application itself is designed in a similar way as the relational database storing the data for the application. Stored procedures are the main entry point from the application to the database which move some of the workload from the applications business layer to the database.

Light object mapping
: Database entities are manually mapped to objects in the application. The mapping is hidden from the business layer of the application using different design patterns.

Medium object mapping
: When the application design does not need to represent the design of the relational database. The SQL statements for interacting with the database are generated at build time or at runtime. SQL queries are usually specified in a platform specific way e.g. object-oriented. Stored procedures are not common in these types of ORM.

Full object mapping
: Supports complex data relations such as inheritance, composition, polymorphism and persistence by reachability. Persistence by reachability mean that if a transient object X reference another transient object Y then persisting X will also persist Y. Full object mapping also implements the data retrieving strategies lazy and eager fetching and different caching strategies that the application can take advantage of. Some well known Full object mapping ORM's are Entity Framework from Microsoft [58] and Hibernate used by JPA [29].

These four categories abbreviates the data fetching and storing procedures in different ways and to varying extent. An ORM with full object mapping puts more strain on the application performance with all the mapping and templating logic. An ORM of the pure relational type on the other hand, moves most of the work to the database server thus adding next to none strain on the application performance regarding the CRUD operations.

4.5 Cache client

An important technology in Computer Science is the cache technology. Caching is used in both hardware and software components such as HDD's, CPU's and web browsers. To cache data is to temporarily store the data in a more accessible and faster way than the original location that the data is primarily stored at. In order to develop high performance web applications caching is a key aspect to reach that goal. In web development caching isn't just a means to access the data faster, it can also help to make data available even if the connection to the WBA is lost. This ability increases the robustness and fault tolerance of the web application.

In web development there are generally two types of caching, output caching and application data caching [62]. As the name indicate, application data caching caches data in the memory within the application. This cache type is useful when trying to speedup the logic executed on the web server. It is typically used to reduce the reads and writes to/from disk or database thus minimizing the time the application will spend on round trips outside the running program. Output caching caches dynamic pages and user controls on cache-capable devices so that the pages and user controls only executes on the first request thus making subsequent requests faster.

Having the cache in the local memory gives supreme speed when accessing the cached data thus making a larger portion of the time spent on executing the logic instead of fetching and pushing data to and from storage. The fact that local cache uses the same memory as the application leads to a number of problems; sharing the memory means that the cache and the application will affect each others performance, executing the application over multiple threads gives references issues to the cache and if the application's memory gets flushed the cache is lost as well. However, there are alternative storage solutions that are both fast, reliable and not dependent on the application memory. These alternative cache providers often stores the data in a key/value, in-memory fashion. This simple storage usually

makes it faster to retrieve the data than it would have been going through all the layers down to the database and back, even thou the data might need to travel over the network wire. But if the speed loss of having to retrieve the cached data over the network are to great there is the possibility add a smaller local cache for data that are accessed frequently. Having a cache provider outside of the application also improves the scalability of the application by making it possible to increase the cache performance without decreasing the application's performance [81, 52, 63, 61].

When the cache becomes full there are different ways to replace the old data in the cache with new data. Some of these cache replacement policies are; LRU (Least Recently Used) policy which replaces the object that where used the longest time ago, LFU (Least Frequently Used) policy keeps the objects that are accessed more times and replaces those that aren't accessed as much. There are also policies that concerns the size of the data objects which keeps small objects and replaces bigger ones and some concern the time the data have been in the cache by either deleting data from the cache after a predefined period of time or prioritize the data by how long they have left before they expire [99, 80].

4.6 Authentication

Authentication is the process of determining if an individual is who he or she claims to be [48, 98]. In the context of a WBA, this usually means to match a username and password to a local database or a third party authentication method. In this influence the second strategy of the two mentioned will be examined, as the process of looking up an entity in a database is already concerned in section 4.4, consequently the O*M test type will concern that performance.

4.7 JSON Web Token

JSON Web Token (JWT) is a means of representing claims to be transferred between two parties, this allows users to obtain a token by providing their username and password. This token will then let the users fetch protected resources without providing their user name and password repeatedly [102]. For this method to work, a way of signing and verifying tokens is required. The process of signing tokens are done by encrypting either a secret or private key with a string or an object called a payload which identifies the user, often the username or a combination of the username and the password. Verifying the tokens encrypted with a secret key is done with the token itself and the secret key, this process outputs the provided payload. Ultimately, verifying the tokens encrypted with private keys is done with the token and a public key, this process also outputs the payload input for signing the token. These two architectures are illustrated in figure 4.2.

The process of encrypting and decrypting the tokens is done according to a JSON Web Algorithm (JWA) [44]. One vital part of the JWA is the cryptographic hash

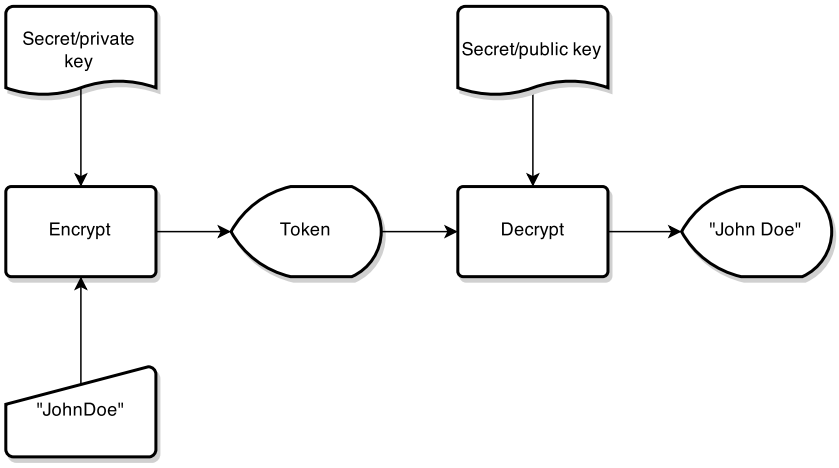

Figure 4.2: Illustration of how a string is encrypted and decrypted using either a secret or private/public key architecture.

function. What characterizes these hash functions is that they are practically impossible to invert, meaning to recreate the input data from the hash value alone [93]. In the context of JWT this means that the secret or public key is required to acquire the payload from the token. The other vital part that constitutes the JWA is the digital signature or message authentication code (MAC), i.e the format of the token.

Hash-based Message Authentication Codes (HMACs) enable one to use a secret to generate a Message Authentication Code (MAC). RSASSA is a digital signature algorithm [96] defined according to the standard in [37] which produces a digital signature using the private/public key architecture. The Elliptic Curve Digital Signature Algorithm (ECDSA) provides for the use of Elliptic Curve cryptography, which is able to provide equivalent security to a private/public key architecture but using shorter key sizes and with greater processing speed. The Internet Engineering Task Force (IETF) have supplied a standard for how JWT should be implemented [44]. In this standard a list of JWA's available for encrypting is presented, as well as a priority level on its implementation. This list can be seen in table 4.4.

Table 4.4: JWA's to use in JWT specified by IETF.

Shorthand	Digital Signature or MAC Algorithm	Requirement
HS256	HMAC using SHA-256	REQUIRED
HS384	HMAC using SHA-384	OPTIONAL
HS512	HMAC using SHA-512	OPTIONAL
RS256	RSASSA using SHA-256	RECOMMENDED
RS384	RSASSA using SHA-384	OPTIONAL
RS512	RSASSA using SHA-512	OPTIONAL
ES256	ECDSA using P-256 curve and SHA-256	RECOMMENDED
ES384	ECDSA using P-384 curve and SHA-384	OPTIONAL
ES512	ECDSA using P-521 curve and SHA-512	OPTIONAL
none	No digital signature or MAC	REQUIRED

4.8 WebSocket

WebSocket (WS) is a transport protocol that communicates over TCP just as HTTP. The difference is that the communication is bidirectional in WS as opposed to HTTP's client initiated communication. When having a duplex communication stream through a single socket the server have the ability to send information to the client without having the client ask for it. A WS connection is initiated with a client HTTP request asking the server to change the communication protocol to WS, see header in listing 4.2. If the web server accepts the change it responds with a HTTP response confirming the WS connection, see header in listing 4.3. The key field in the headers are the upgrade field which specifies which protocol the request wants to switch to and the response have agreed to switch to. After the initial HTTP handshake the communication is bidirectional until either the server or the client close the connection, see figure 4.3.

Listing 4.2: Request header initiating a WebSocket communication

```
GET ws://localhost:49840/api/Websockets HTTP/1.1
Host: localhost:49840
Connection: Upgrade
Upgrade: websocket
Sec-WebSocket-Version: 13
Sec-WebSocket-Key: BncdOQYi+SsCBz9liavXGA==
```

Listing 4.3: Response header accepting a WebSocket communication

```
HTTP/1.1 101 Switching Protocols
Upgrade: Websocket
Server: Microsoft-IIS/8.0
X-AspNet-Version: 4.0.30319
Sec-WebSocket-Accept: m+No5WV18dwpWEUf5lbsai4buGo=
Connection: Upgrade
Date: Wed, 06 May 2015 11:49:37 GMT
```

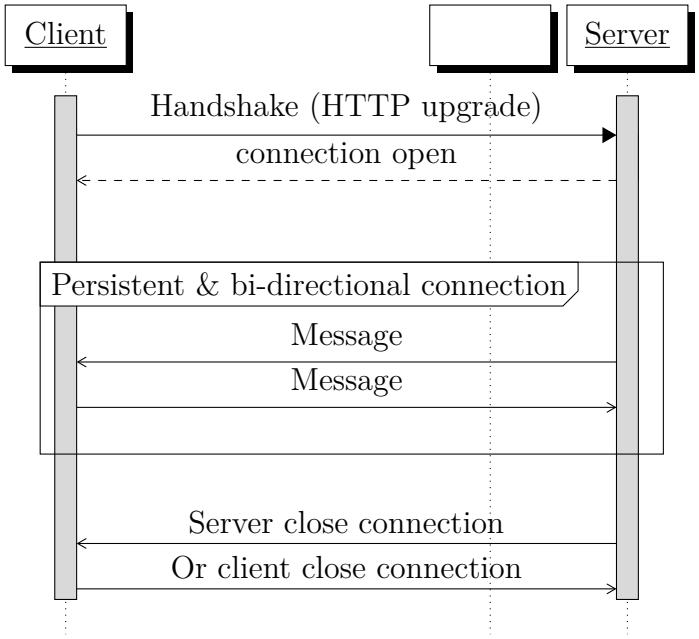

Figure 4.3: WebSocket sequence diagram.

A common scenario when describing WS is a chat application. When a user writes a message in a chat room the message get sent to the server and with WS the server can broadcast this message to all the user's in the same chat room without them having to request a message. If this was done with pure HTTP then each user would have to continuously ask the server if there are any new messages. This polling generates a lot more network traffic and it wouldn't be a real time chat.

5

Schematic

In this chapter the engineered schematic is presented. The schematic contains a set of test types and charts that is to be used to compare and evaluate the result of the test types.

5.1 Environment and tools

The test types sections in this schematic describe how to implement and execute the test type and then compare and evaluate the results. However, there are some prerequisites that needs to be discussed before these can be done. The environment that is recommended to use during these test types is illustrated in figure 5.1. It is optional to put the database server on the same machine as the one hosting the test types, however it better resembles real world use cases to put it on a standalone machine, especially when proper scaling is to be considered. Naturally, the benchmarks should be performed from a separate machine from the server hosting the test types, otherwise the network latency would be disregarded. The network should be configured with the aim to mitigate interference as much as possible, the optimal scenario would be on a local network limited to the nodes in figure 5.1.

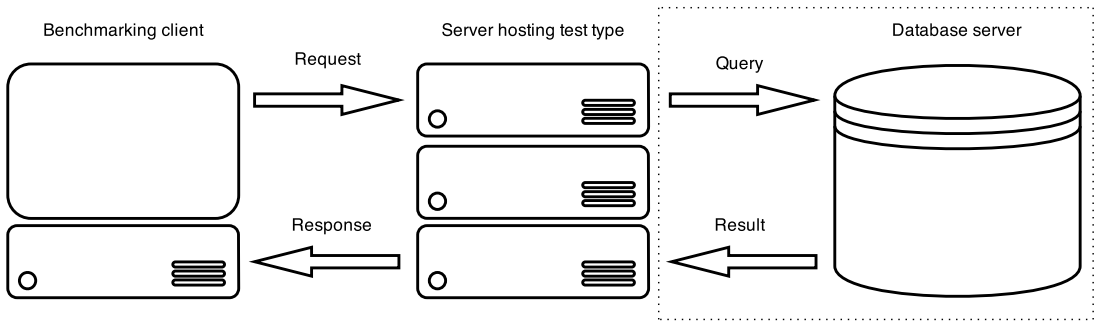

Figure 5.1: The environment prerequisite for executing the test types in the schematic.

On the benchmarking client it is allowed to use any benchmarking tool of choice, however for convenience two choices is suggested. Apache benchmark (ab) [9] and wrk [100] both provide the required features to perform all the benchmarks in this

schematic, except WebSockets. The requirements for the benchmarking tool is provided beneath.

R1 The benchmarking tool should be able to send HTTP GET requests.

R2 The amount of concurrent connections should be able to be set.

R3 The duration of the benchmark should be able to be set.

R4 The benchmarking tool should be able to provide the statistics for average response time and throughput.

5.2 Concurrences

All test types in this schematic are to be benchmarked with the amount of concurrent requests in table 5.1. Further more, the test types either have a set of variants or variable values that it is is going to benchmarked with. This means that the amount of times each test type will be benchmarked is; the concurrency values times the variant or variable values. The concurrency values in the table 5.1 are those that are recommended to use. However, as with everything else in this schematic, those are interchangeable to values better suiting the user.

When the number of concurrent requests are equal to 1 it is called Pmax. With only one concurrent request running, that request won't have to share resources with other requests, thus it represents the maximum performance for said test type. This is used to find out which is the best possible response time for each influence per variant or variable value. Pmax is not used when discussing throughput as a single concurrent request doesn't put enough load on the WBA to utilize the full capacity of the CPU.

Table 5.1: The values of concurrences to use in the test types

Concurrent requests	1 (Pmax)	10	100

5.3 Charts

In the subsection Comparison and evaluation of each of the test types in section 5.4, charts are going to be used to visualize results and simplify comparisons. Because these charts will be occurring in each test type, it would be inconvenient to describe their usage in every test type section. Therefor the common usage of each chart is compiled in this section. Conclusions unique for each test type is presented in the Comparison and evaluation subsection of each test type.

5.3.1 Throughput Line Chart

The Throughput Line Chart (TLC) displays the throughput capabilities of an influence and how well the level of throughput is continued as concurrency is increased, i.e its load scalability with regard to throughput. The chart applies the metric request per second on the y-axis and the variable of corresponding test type on the x-axis, as illustrated in the example chart in figure 5.2. Each plot in the chart will represent the different levels of concurrences in table 5.1 except Pmax. Finally, this chart should be plotted for each WBAF that is being evaluated.

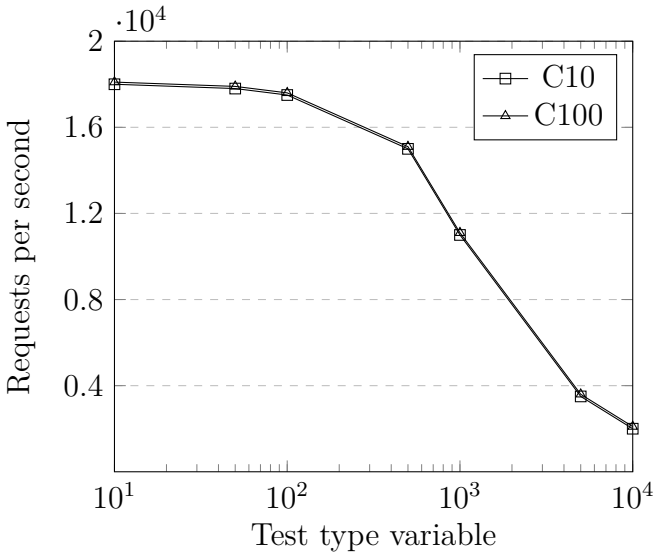

Figure 5.2: Example TLC showing throughput with regard to single test type variable for different concurrency levels

There are some conclusions that can be drawn from the Throughput Line Chart on its own. This chart will give an indication on how well the influence of the WBAF scales with regard to throughput. If the plots for all concurrency levels are federated, i.e it looks like all plots are merged, then there are no concerns on the load scalability of the throughput. This phenomenon is illustrated in figure 5.2. However, if the requests per second is declined as concurrency is increased then there is a load scalability issue. In most cases where this chart is employed, all plots will decline as the variable is increased.

5.3.2 Throughput Comparison Line Chart

The objective of the Throughput Comparison Line Chart (TCLC) is to supply a visual comparison of the different WBAF's from the Throughput Line Charts in a unified chart. This chart is built from the plots of the concurrency level from each TLC representing the expected maximum load in the WBA, as illustrated in figure 5.3.

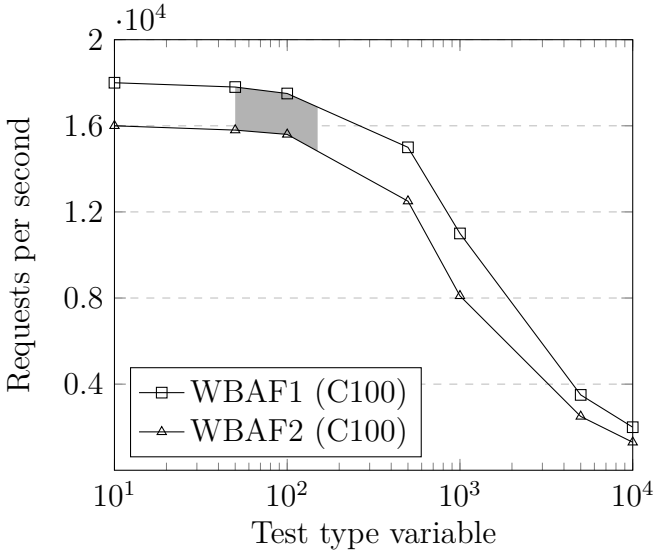

Figure 5.3: Example TCLC illustrating a comparison of the throughput with regard to single test type variable for two WBAF's

A comparison can now be done by defining the expected range of the test type variable, and then examining the difference in throughput for provided WBAF's. In figure 5.3 this is exemplified by marking the area between 50, which is the least expected variable value, and 150 which is the maximum expected variable value. In this example WBAF1 (C100) provides a better throughput than WBAF2 (C100) in the defined area.

5.3.3 Throughput Bar Chart

Much like the chart TLC the Throughput Bar Chart (TBC), figure 5.4 measure the load scalability with regard to throughput. The difference is that each value of the concurrency variable will plot a number of values in the chart. The number of values that will be plotted for each concurrency load is dependent of the number of variants that the test type will exercise. When changing the concurrency level an additional number of values will be plotted. These new values are grouped together with the previous values in sets of the different variants.

The conclusions that can be drawn from this chart is similar to the Throughput Line Chart i.e if the bars height in each set are notably declining when the concurrency increases then there are issues with the load scalability with that specific variant of that specific influence. The height difference between the sets doesn't say anything from a scalability perspective but gives an indication on how the performance differs between the different variants of the influence.

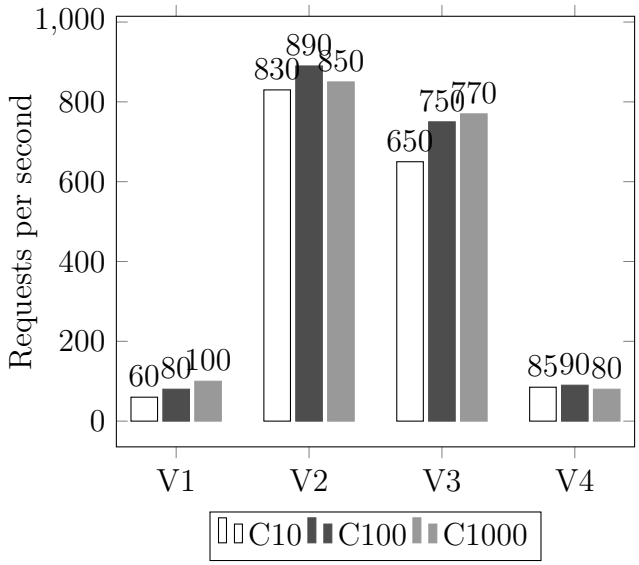

Figure 5.4: Example TBC showing throughput with regard to multiple variants of a test type for different concurrency levels

5.3.4 Throughput Comparison Bar Chart

When the expected maximum load in the WBA have been determined the corresponding results from the TBC's 5.4 will be plotted in a TCBC, see figure 5.5 for the WBAF's that are to be compared.

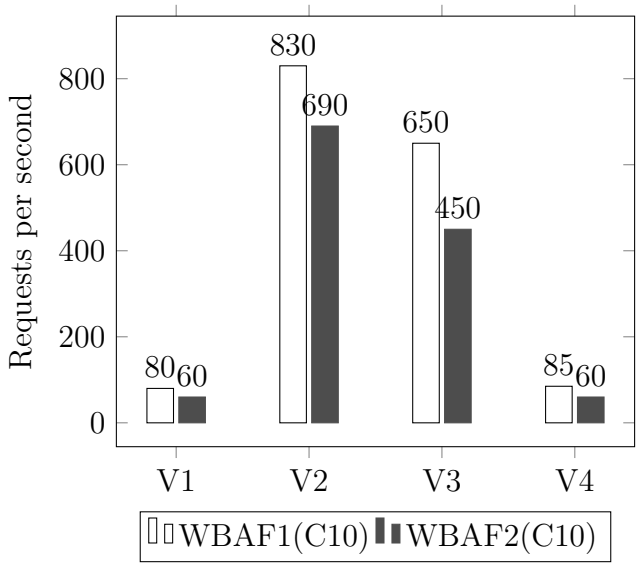

Figure 5.5: Example TCBC illustrating a comparison of the throughput with regard to multiple variants test type for two WBAF's

The conclusions that can be drawn from this chart is simply that the WBAF with higher bars have better throughput for the variants of the exercised influence.

5.3.5 Response time Line Chart

In the Response time Line Chart (RLC) the response time of an influence in combination with its load scalability regarding response time will be visualized. Average latency will be applied on the y-axis and the variable of corresponding test type on the x-axis. The plots represents the different levels of concurrences in table 5.1 including Pmax. An example RLC is illustrated in figure 5.6.

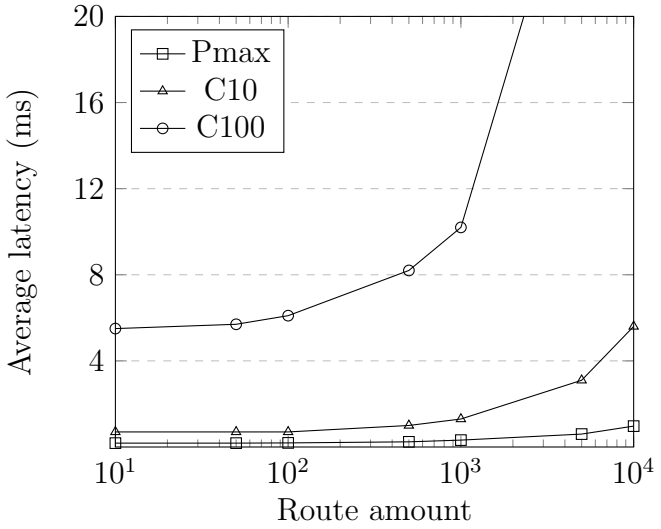

Figure 5.6: Example RLC illustrating response time with regard to single test type variable for different concurrency levels

Similar to the TLC chart described in 5.3.1, this chart will also give an indication on how well the influence scales, however here with regard to response time. Unlike throughput, it's not feasible to achieve an absolute load scalability regarding response time as described in section 2.1. As shown in figure 5.6 the response time will increase as concurrency is increased.

5.3.6 Response time Comparison Line Chart

The Response time Comparison Line Chart (RCLC) aims to simplify the comparison of the response time capabilities of an influence. This chart consists of the plots representing the expected maximum load, i.e expected concurrency level, from each of the RLCs, in a unified chart as illustrated in figure 5.7.

In order to do the comparison of the WBAF's in the context of interest, the expected range of the test type variable must be defined. In the example figure 5.7 the range of interest is between 50 and 150 variable value units. In the defined area WBAF1 proves to supply a better response time than WBAF2.

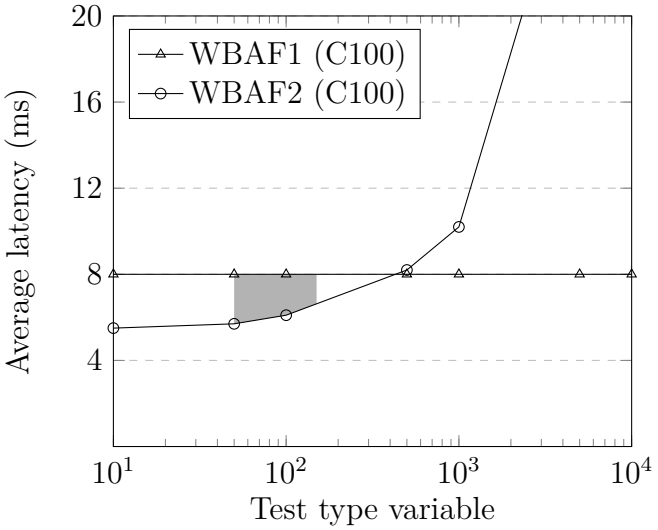

Figure 5.7: Example RCLC illustrating comparison of the response time with regard to test type variable for two WBAF's

5.3.7 Response time Bar Chart

Like the RLC, 5.3.5, the Response time Bar Chart (RBC) displays the response time of an influence in combination with its load scalability regarding response time. Average response time will be applied on the y-axis and the variant groups are on the x-axis. The chart will give an indication on how well the influence scales, with regard to response time. The bars represents the average response time for the different levels of concurrences grouped by the different variants. Figure 5.8 illustrates an example of such a chart.

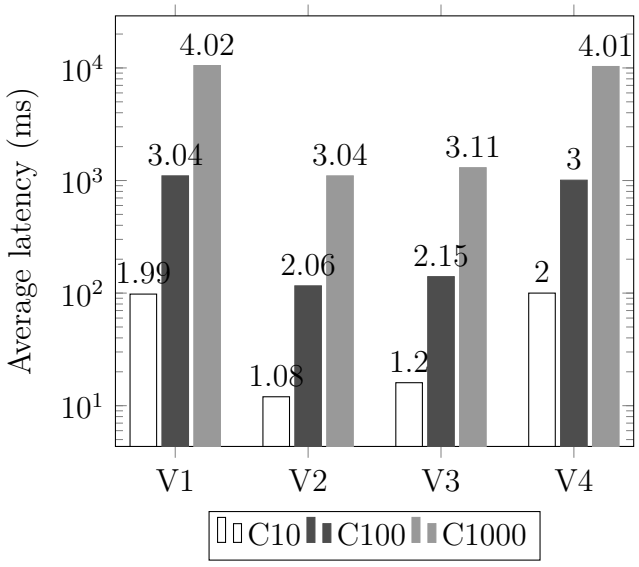

Figure 5.8: Example RBC showing response time with regard to multiple variants of a test type for different concurrency levels

5.3.8 Response time Comparison Bar Chart

The Response time Comparison Bar Chart (RCBC) compares two or more WBAFs with regard to the response time capabilities of an influence. The expected maximum concurrent connections are chosen as the plots from each of the RBCs. Figure 5.9, illustrates a comparison between two WBAFs.

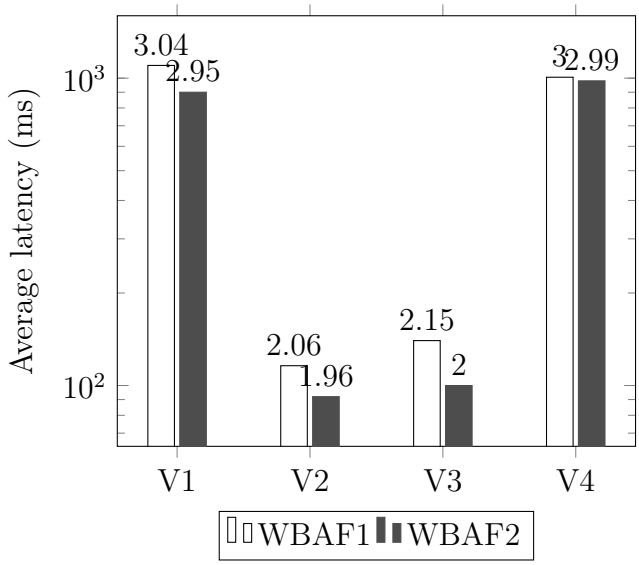

Figure 5.9: Example RCBC comparing two WBAFs by showing response time with regard to multiple variants of a test type

In each variant group each WBAF have one bar that can be directly compared to the others for each type of variant.

5.4 Test types

The test types are derived from the WBAF influences from chapter 4. This does not mean that these are the only influences that are relevant to test, rather that these are the most popular and most used in development according to the influence study.

For each test type a description is supplied that explains the test type in running text, some test types requires a longer description in order to resolve any ambiguity, and some test types are rather straight forward which requires a lesser description. In addition to the description, the test types also include a set of requirements. In some cases these requirements are on a rather low implementation level to ensure that the implementation is carried out as intended. Very small implementation changes can have huge impact on performance, e.g reading a file for every request instead of preloading it into memory during application startup.

Furthermore the test types also contain a variables or variants section. A test that is using variables is running the exact same test but with different values for something

that has an impact on its performance. Each test type supplies a recommendation for the values to benchmark the test type with. These recommendation values are on a logarithmic scale for all test types in order to test a wider spectrum. Some values may not be applicable to a real use case, i.e an application with 10k routes. However, by displaying the difference on such a large scale it becomes more apparent that this variable actually have an impact on performance, the TCLC and RCLCs is later used to delimit the result to an actual use case. The test types that are using variants on the other hand are completely different tests, but in the context of the same influence. Variants can be different use cases within an influence, i.e the O*M test type, or it can be different methods to conduct said influence altogether, i.e the JSON Web Token test type. Like the variables, these variants are recommendations and if any of them don't fit the context or any variant is found missing, simply add or remove said variant.

A running text describing how the benchmarking should be performed when the implementation is finished can be found in the execution paragraph of the test type. This should be very convenient for all test types if the requirements have been followed correctly.

Lastly, each test type contains a comparison and evaluation paragraph. Here it is described which charts to use to visualize and evaluate the results. In section 5.3 more details is given on how to interpret the results and which conclusions can be drawn from each chart.

5.4.1 Request Routing

The WBAF stores routes with pointers to its corresponding functions to execute. The process of looking this function up for a provided route is essentially all the work the Request routing influence will do upon a request. The number of employed routes in the WBA will have an influence on how fast the request routing will do this lookup. In order to benchmark this influence, a routing schema with completely unique routes will be employed, with the amount of unique routes being the variable for this test type.

The creation of the routes will be done programatically during the WBA startup time, uuid version four will be used to create these routes to ensure that they are unique [33]. Adding the route that will be requested during this test type will be done halfway through adding the filler routes. The reason for this is that if the look up process loops through the collection of routes without any shuffling, it will either always find the route on the first iteration, because it was added prior to the filler routes, or it will find it on the last iteration because it was added after the filler routes. By adding it halfway through it is ensured that the WBAF's employing this look up algorithm won't get any unfair advantage or disadvantage.

Requirements

R1 The WBA should be able to run with an integer option.

R2 During application startup, a set of filler routes should be created. The amount of routes is derived from the option in **R1**.

R3 The format of the filler routes URI should be `/request-routing/`, followed by a version 4 uuid [33], followed by a parameter and ended with another version 4 uuid. An example of such an URI is `/request-routing/f66453fe-f04c-4ab7-a288-34ffd0d90eb3/9999/248ded9e-1ac6-41d2-9496-b6b03e448f14`.

R4 Each URI from **R3** should be combined with a GET, POST, PUT and DELETE method token to create a route.

R5 Four specific routes with the URI `/request-routing/Hello/:parameter/World` and with the method tokens in **R4** should be created. These routes are to be added after half of the filler routes in **R2** have been added, as illustrated in figure 5.10.

R6 The response of all routes should be an empty `200 OK` message.

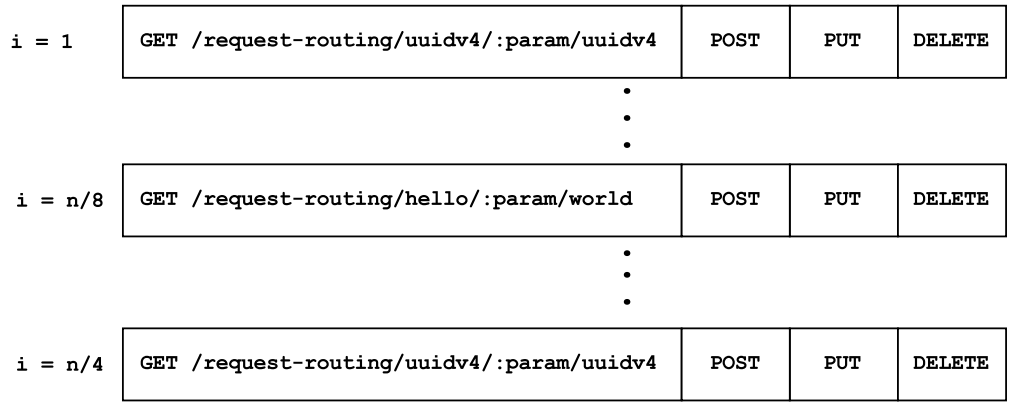

Figure 5.10: Illustration of when to add the routes in **R5**.

Variables

The variable to be used in the benchmarks is the amount of routes employed in the WBAF, and the different values can be seen in table 5.2. The variables are on a logarithmic scale to test a broader spectrum.

Table 5.2: The values of variables to use in request routing benchmark

Route amount	10	100	1000	10000

Execution

Perform the benchmarking for each combination of the values in table 5.2 and the concurrences in table 5.1 on the GET route in **R5** for each WBAF.

Comparison and evaluation
After having performed the experiments for all combinations, the results are going to be plotted in four chart types. The x-axis in these charts are going to be on a logarithmic scale due to the values of the variable used in this test type. Start with plotting the TLCs for all WBAF's as described in section 5.3.1. In addition to the conclusions mentioned in the section 5.3.1, another can be drawn by analyzing the differential quotient of the plots. If all plots in one chart is declining, like in figure 5.2, then the route lookup growth rate is dependent on the the route amount. A scenario where this isn't the case is if the WBAF employs a hash table for storing routes.

After having plotted and analyzed the TLCs plot the TCLC as described in section 5.3.2. The range of interest is represented as the expected route amount employed in the application in this test type. Comparing the request routing influence of the WBAFs should then be a convenient task by simply analyzing the marked area of interest.

Having evaluated the throughput and load scalability regarding that aspect, evaluating response time and the ability of the WBAF to sustain an acceptable response time as concurrency increases now remains to complete the evaluation of the influence request routing. Consequently, the next chart that will be plotted is the RLC, instructions is to be found in section 5.3.5. In contrast to the TLCs, if the plots see an increase in this chart as opposed to a decline, then the route lookup growth rate is dependent on the the route amount.

Finally a RCLC will be plotted in order to make the response time comparison of the influence of each WBAF more convenient, instructions are described in section 5.3.6.

5.4.2 Serialization

The influence serialization is a key feature when it comes to exchanging data between the server and the client. To test the WBAF performance in serializing data this test will, for each request, serialize predefined data and then deserialize the data. The amount of data that needs to be serialized will determine how much time the serialize task will take, consequently data size influences serialization and will therefor be the variable in this test type. The data that will be serialized will be read from predefined files. This will be done during application startup and stored in memory, consequently no file system I/O will affect the test results. Finally, a 200 OK message will be sent as a response. The response should be empty to minimize work not related to test.

Requirements

R1 A file for each value in table 5.3 should be created.

R2 Each file in **R1** should be filled with corresponding size of data.

R3 The format of the data in **R2** should be in chosen media type, e.g XML, JSON or CSV.

R4 The data in **R2** should not be prettified [18].

R5 The data in **R2** should have a spread usage of the data types Array, Object, Null, Boolean, Integer, Float and String.

R6 The files in **R1** should have an equal amount of data types described in **R5**.

R7 The WBA should during application startup read the files in **R1** and store those strings in a key value store, where the key is its corresponding filename.

R8 A GET route with the URI `/serialization/:size` should be created.

R9 The route in **R8** should match its parameter `:size` to the key value store in **R7**, and serialize that string to an in memory object, and deserialize it back to a string afterwards.

R10 The response of the route in **R8** should be an empty 200 OK message.

Variables
Kilobytes of serialized data in the format of use is the variable to be used in this test type. The different values are also here on a logarithmic scale in order to test a wider spectrum. The values can be found in 5.3.

Table 5.3: The values of variables to use in request routing benchmark

Data size (kB)	10	100	1000	10000

Execution
Perform the benchmarking on the route in **R8**, where the parameter represents a value of the file size variable in table 5.3, for each combination of variable values and concurrences in table 5.1.

Comparison and evaluation After having performed the experiments for all combinations, the results are going to be plotted in the same chart types as in the request routing test type, TLC, TCLC, RLC and RCLCs. Like in the request routing test type, these charts should apply a logarithmically scaled x-axis to cope with the file size values.

In the charts related to throughput the request per second should decline as the file size grows, as the file size growth increases the amount of work that needs to be done per request. Similarly the response time will increase with an increased file size.

5.4.3 O*M

The WBAF influence O*M described in section 4.4 abstracts the functionality that is used to interact with a database. The actions that will be performed in order to exercises this influence is a variable number of concurrent create, find one, find all and update operations on a relational database. The delete operation will not be tested due to it's complexity and similarities to the other operations. The test needs to be run four times on each concurrency level, one run for each type of operation.

The database will be initialized at start up with the same schema and data for each test. Whether the database is of a relational, e.g MySQL or MSSQL, or non-relational type, e.g MongoDB or CouchDB, doesn't matter for this test. In order to not get favourable results in the comparison the database in use should be same for the different WBAFs.

Requirements

R1 A predefined schema for data entries for the chosen database shall exist.

R2 An initialization procedure that empties and fills the the database from requirement **R1** with the correct amount of data entries shall be executed at application start up.

R3 The id's of the data entries from **R2** shall be indexed into an array during application start up.

R4 Four different routes routes shall be added to the WBA, one route for each of the following operations, create, find one, find all and update.

R5 The route that receives the create request shall add a predefined data entry through the O*M to the database.

R6 The route that receives the find one request shall select a random id from the array of id's and use the O*M to fetch the data entry that corresponds to that id.

R7 The route that receives the find all request shall use the O*M to retrieve all the data entries from the database.

R8 The route that receives the update request shall select a random id from the array of id's and use the O*M to update some fields in the data entry that corresponds to the selected id.

R9 The response of all routes shall be an empty `200 OK` message.

Variants

The O*M variants that will be benchmarked are different query types, these can be found in table 5.4.

Table 5.4: The variants in the O*M benchmark

Query type	Create	Find one	Find all	Update

Execution
Perform the benchmark for each combination of the variants in table 5.4 and the concurrency level in table 5.1. The variant executed is dependent on which URI that the benchmark is run on in accordance to requirement **R4**. To avoid the need of resetting the database between the tests, execute the create operation variant last.

Comparison and evaluation When all combinations of the test type have been executed the results shall be plotted in the charts; TBC 5.3.3, TCBC 5.3.4, RBC 5.3.7 and RCBC 5.3.8. The chart variants are the same as displayed in the Variants table 5.4 in this section.

5.4.4 Template Engine

As mentioned in section 4.3, a template engine uses two ingredients to create an HTML page, data and templates. Thorough research about template engine functionality and existing benchmarks [94, 97] have unveiled that escaping strings, looping through collections and including partial views provide an honest understanding about the performance of a template engine. These influences will all be performed once per object instance in the test data, therefor the data length will determine the workload of this test type, i.e the data length is the variable of this test type.

In order to test the template engine performance of a WBAF, a route will be setup that renders an HTML string from the templates and the data. Both the templates and the data will be cached in run time memory, as reading from file system would take a long time and would skew the results. The parameter for said route will determine the number of objects that should be included in the data, i.e the data length.

Requirements

R1 The data should be an array of objects.

R2 The objects in **R1** should have two string properties.

R3 One property of the objects in **R2** should be a randomized string not requiring escaping. Example: `"Lorem ipsum dolor sit amet"`.

R4 One property of the objects in **R2** should be a randomized string requiring escaping. Example `"<p>Lorem ipsum</p> dolor sit <p>amet</p>"`

R5 There should be a partial template which should render both properties in **R2** within `<p>` tags as illustrated in figure 5.11.

R6 There should be a main template which should include the `<html>` and `<body>` tags as illustrated in figure 5.11.

R7 The main template should render the partial in **R5** for each object in **R1** as illustrated in figure 5.11.

R8 All templates should be precompiled if possible.

R9 A GET route with the URI `/template-engine/:dataLength` should be created.

R10 The route in **R9** should use the template engine to render the HTML with the data in **R1** and templates in **R6** and **R5**.

R11 Caching the rendered HTML is prohibited.

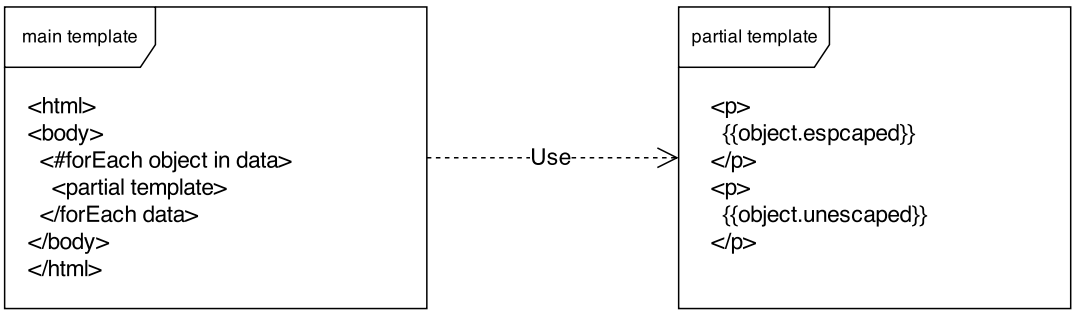

Figure 5.11: Illustration of how the two templates in the test type should be constructed.

Variables

The length of the data array that is to be combined with the templates to render the HTML is the variable to be used in these benchmarks. Aligned with previous variable usages, these values are also on a logarithmic scale and can be found in table 5.5.

Table 5.5: The values of variables to use in template engine benchmark

Data length	10	100	1000	10000

Execution

Perform the benchmarking on the route in **R9**, where the parameter represents a value of the data length variable in table 5.5, for each combination of variable values and concurrences in table 5.1.

Comparison and evaluation For comparison and evaluation the TLC, TCLC, RLC and RCLCs will be plotted with logarithmically scaled x-axis to cope with the

data length values. In this test type there are no real use to identify an variable range in the TCLC and RCLCs to mark the area between them. The reason for this is that the variable, data length, is not conveniently translated into something that can be expected from the WBA that is to be built. Naturally, as data length grows the throughput will decline and vice verse for response time.

5.4.5 JSON Web Token

JWT combines, as described in section 4.7, a payload, a secret or private key and a JWA to sign a token. Naturally, the security differs depending on which JWA is chosen, with the trade off of performance. In the table 4.4 that is from the JWT specification from IETF, an implementation requirement level is specified for each JWA. Meaning that HS256 and none will be included in all JWT implementations, and RS256 and ES256 in a majority. Therefor these four JWA's act as standard variants in this test type, however removing or adding JWA's of choice is encouraged.

In this test type a token will be signed and verified according to a specific JWA. The creation and reading the keys from files will naturally be done prior to the request execution to avoid non JWT related work to be benchmarked.

In cryptographic functions the key length is an important security parameter. Academic and private organizations provide recommendations and mathematical formulas to calculate the minimum key size requirement for a specific year. BlueKrypt is an organization that have compiled these papers and publications [12] on the Internet, which present the minimum key size for all publications for a specific input year. In table 5.6 the relevant columns for the keys used in this type is presented with the input year of 2015. Both HS256 and ES256 have hard requirements that a 256 bit key should be used, however these columns are also included in table 5.6 to illustrate that these JWA's satisfies recommended securities for 2015. The RSA key for RS256 have no hard requirement on size, therefor the maximum value for any method is going to be used to ensure proper security, which is 2048 bits. If chosen to include other JWA's than the standard ones described in this test type, be sure to do research about the required key size, as it will influence the outcome of the benchmarks.

Table 5.6: By BlueKrypt, calculated key sizes in bits safe for use in 2015 by using methods from different publications.

Method	RSA	Elliptic Curve	HMAC
Lenstra / Verheul [3]	1613	154	163
Lenstra Updated [2]	1245	156	156
ECRYPT II [21]	1248	160	160
NIST [67]	2048	200	200
ANSSI [6]	2048	200	200
BSI [13]	1976	224	224

As mentioned earlier HS256 have a hard requirement on key length. The reason is that if a key is shorter than its block size, in bits, then it is padded with a non randomized character, usually zeroes until block size is reached. If the key is too long then its cut off, the length of the hash output is still the block size [34]. Therefore the secret key will be 256 bits long as HS256 is using the SHA-256 cryptographic hash function. The string will also be randomized in order to achieve complexity of the string, which ensures its quality. The ES256 JWA uses a P-256 curve, which with the same analogy as HS256 has a requirement on a 256 bits string [35].

Requirements

R1 The secret key should be a randomized 256 bits long string.

R2 The private RSA key should be a 2048 bits long .pem file.

R3 The public RSA key should be a .pem file derived from the private key in **R2**.

R4 The private elliptic curve key should be a 256 bits long .pem file with a P-256 curve.

R5 The public elliptic curve key should be a .pem file derived from the private key in **R4**.

R6 The common payload for all algorithms should be a string with the value `"John Doe"`.

R7 The .pem files in **R2**, **R3**, **R4** and **R5** should be read from the file system during application startup in UTF8 encoding.

R8 A GET route with the URI `/jwt/hs256/` should be created.

R9 The route in **R8** should sign a token with the secret key in **R1** and the payload in **R6** using the HS256 JWA.

R10 The route in **R8** should verify the signed token in **R9** with the secret key in **R1**.

R11 A GET route with the URI `/jwt/rs256/` should be created.

R12 The route in **R11** should sign a token with the private key in **R2** and the payload in **R6** using the RS256 JWA.

R13 The route in **R11** should verify the signed token in **R12** with the public key in **R3**.

R14 A GET route with the URI `/jwt/es256/` should be created.

R15 The route in **R14** should sign a token with the private key in **R4** and the payload in **R6** using the ES256 JWA.

R16 The route in **R14** should verify the signed token in **R15** with the public key in **R5**.

R17 A GET route with the URI `/jwt/none/` should be created.

R18 The route in **R17** should sign a token with the payload in **R6** but without any key using the none JWA.

R19 The route in **R17** should verify the signed token in **R18** without using any key.

R20 The response of all routes shall be an empty `200 OK` message.

Variants
The different JWA's are the variants to benchmark this test type with, the ones supplied in table 5.7 is recommended by IETF and is thus used as default in this schematic.

Table 5.7: The variants in the JSON Web Token test type

JWA	HS256	RS256	ES256	none

Execution
Perform the benchmark for each combination of the variants in table 5.7 and the concurrency level in table 5.1, where each route represent a different variant.

Comparison and evaluation
Comparison and evaluation will be done by plotting the TBC, TCBC, RBC and RCBCs. The comparison should be convenient after having done this, especially if the JWA of choice is known beforehand. Verifying a token is something that will be done for every request to a protected resource for a WBA, therefor the responsiveness of this influence are of utmost importance. Limiting the routes to only verifying the tokens, and doing the signing in application startup, could be done to measure verifying only.

5.4.6 Cache client

The WBAF influence that can affect the users perceived feeling about the WBA's performance the most is the cache described in section 4.5. To test this influence the WBAF's performance to access and retrieve data from a cache provider will be measured.

The cache will be initialized at start up with the data that will be used in the test. In order to not get skewed results in the comparison the cache provider in use should be same for the different WBAF's.

Requirements

R1 The cache provider used should be an external cache that stores the data outside of the WBA's memory.

R2 An initialization procedure that empties and fills the cache from requirement **R1** with the data entries shall be executed at application startup.

R3 A route with an integer parameter that routes to a function that retrieves data from the cache shall exist.

R4 The route that receives the cache request shall retrieve a variable representing the number of cache entries the WBA shall fetch from the external cache.

R5 The data that is retrieved from the cache shall be made as a single get operation. Batch operations are not allowed due to that this test objective is to exercise the connection to the cache provider.

R6 The response of the route **R3** shall be an empty `200 OK` message.

Variables
The variable to be used in this test type is the number of data entries that shall be fetched from the external cache provider. How many request that shall be performed is listed in table 5.8.

Table 5.8: The number of data entries that shall be fetched from the external cache in Cache benchmark

Number of get's	10	100	1000

Execution
Perform the benchmark for each combination of the variable in table 5.8 and the concurrency level in table 5.1. The variable executed is the parameter on the route in requirement **R3**.

Comparison and evaluation
After having performed the experiments for all combinations, the results are to be plotted in the charts TLC, TCLC, RLC and RCLC. If the variables in the table 5.8 are used, then the x-axis should be logarithmically scaled to enhance the visibility.

In the charts related to throughput the request per second should decline as the number of gets increases, because the amount of requests that the WBA performs on the cache provider per request increases as well. Similarly the response time will increase with a growth in number of gets.

5.4.7 Websockets

WebSockets enable bi-directional communication as described in section 4.8. In this test type connections will first be established via handshaking, and then messages will be sent to the WBA. The length of these messages will act as the variable in

this test type. The work that is to be done by the WBA for each incoming message from the benchmarking tool is simply to echo said message.

In order to perform the benchmarks in this test type, a tool with WebSocket testing functionality is required. The tools mentioned in 5.1 is not sufficient for this task, therefor a tool called Thor [74] is recommended. However, any tool that meet the requirements for benchmarking this test type is naturally allowed.

Requirements

R1 The WBA should be able to receive and establish a WebSocket connection with a client.

R2 The WBA should be able to receive a UTF8 message on the opened connection in **R1**.

R3 Upon receiving a message in **R2**, the same message should be sent back to the sender, i.e the benchmarking tool.

Variables

The length of the messages to be sent to the WBA by the benchmarking tool is the variable in this test type, and these values can be found in table 5.9.

Table 5.9: The length of each message to be sent by the benchmarking tool.

Message length	10	100	1000	10000

Execution

Perform the benchmarking on the index route with the WS protocol for each combination of message length and concurrences in table 5.1.

Comparison and evaluation

Plot the TLC, TCLC, RLC and RCLCs for comparison and evaluation with a logarithmically scaled x-axis to cope with the message length values. Try to identify a variable range that is expected in the WBA that is to be built, and use that in the TCLC and RCLCs.

5.4.8 Authentication

After an extensive research about the authentication influence, it became apparent that this influence is redundant to benchmark. The reason for this is that a vast majority of the work in this influence is done by a third party vendor, i.e Facebook, Google or an internal LDAP server. Combined with the fact that all test types includes the request routing influence, the actual work left for this influence that is done on the client (WBA) side is considered insignificant.

6

Express.js vs .NET MVC

The most plausible use of the engineered schematic in this thesis is to compare a set of WBAF's with building a specific application in mind. However, this running example will take a different approach and compare two WBAF's without a specific application in mind. The basis of this comparison will instead be to evaluate a new technology on the market in an actual real world use case. Furthermore, this chapter will also act as a complement to the schematic to supply a better understanding about its usage.

6.1 Use case

The comparison of the two frameworks in this use case, Express.js and .NET MVC, are of interest and have been requested by two organzations. Both of these organizations employ the .NET MVC framework in production environment as of today, but with the industry of web development evolving in a rapid pace, the ability to stay ahead and evaluate new technology are of utmost importance. Express.js is the most popular WBAF for Node.js, with Node.js being the new player on the market and the platform of interest to the aforementioned organizations. The first release of Node.js was published in 2009, and five years later Node.js is the third most popular project on GitHub [90]. Several large scale enterprises have been adopting Node.js as their runtime environment for web applications, both for fresh development projects and as substitute for solutions in use [69]. Naturally this draws interest to organizations within the industry, but its performance regarding load scalability and response time needs to be evaluated before an adoption can be considered. Express.js is a micro WBAF, consequently extensions as packages will have to be used in order to perform the test types in the schematic. Each package chosen is always the most popular package for the task in each test type.

.NET MVC is the latest framework that have been used for building applications within these two organizations. It's developed by Microsoft and is based on their .NET platform [53]. This framework is of the fullstack kind and therefor most of the test types can be performed without additional packages, in the few cases in which this is required the same principle as Express.js have been applied, i.e the most popular packages is used.

Because this running example will act as a complement to the schematic, the requirements demanded from the organizations have been limited in order to not deviate from the standard usage of the schematic. Consequently there will be no conflicts of interest and these rather generic results can be used not only by both of the organizations, but also for a broader audience reading this thesis. The concurrency levels that have been chosen for these benchmarks is, beyond Pmax, C10 and C100. In charts where concurrency is to be limited to one value, i.e TCLC, C10 was chosen as it best resembles the scope of use in both organizations. Further specification is supplied per test type in this chapter.

6.2 Specifications

The test types were run with the environment setup described in section 5.1. The benchmarking client was a MacBook Air 2013 with a 1.7GHz Intel quad core i7 processor and 8GB RAM with a memory speed of 1600MHz running OS X 10.2, and the tool used for benchmarking was wrk [100]. The server hosting the test types was running 64-bit Windows 8.1 with a 2.4GHz Intel dual core i5 processor with hyperthreading enabled, the memory capacity was 8GB with a memory speed of 1066 MHz and the SATA hard drive have a spindle speed of 5400rpm. The web server used for hosting the .NET MVC test types was IIS 8.0, and the Express.js test types was hosted using the Node.js v.0.12.2 built in web server. The database server was hosting MySQL 5.6.24 on Windows 7 with a 3.3GHz Intel quad core i5 processor, 8GB RAM with a memory speed of 1333MHz and a SATA6 Gbit/s hard drive with spindle speed of 7200rpm. All devices were interconnected with a router and wires supporting 1000BASE-T Ethernet connection [103].

6.3 Test types

All Express.js test types were implemented using clustering [68] to utilize all the cores on the machine. No extra work was required for the .NET MVC implementations to utilize all the cores. During the benchmarks the Resource monitor program attached to Windows 8.1 was used to verify that all cores were utilized to the fullest. The source code for all test types can be found on GitHub [45], any non trivial task will be further described in each test type section.

All test types in the schematic was benchmarked with the concurrency levels, variants and variables recommended by the schematic. Each combination of influence, concurrency and variant or variable were run over three rounds, where the mean of these rounds acts as the result for said combination. Furthermore, each round was benchmarked for 60 seconds and a non included warm up benchmark for 10 seconds was performed prior to each combination in order to allow lazy-initialization to execute and just-in-time compilation to run. The total combinations of influence, concurrency and variant or variable for all test type resulted in an amount of 75

combinations. As these were all run three times for two frameworks, combined with the warm up time, effective benchmarking time resulted in close to eight hours.

The results for each round were manually documented in an excel document from the output of wrk in the terminal. When a request from wrk exceeded 2000 ms, a timeout error was thrown as a response. This mostly only occurred when using a concurrency level of 100 combined with a high variable value, i.e serializing 10 000 kB of JSON data. However, when these timeout errors appeared the average response time results became skewed, consequently these results were left out of the charts. This is why some plots aren't completed in the response time charts. The results for throughput wasn't skewed by these errors, thus they aren't affected.

6.3.1 Request Routing

With request routing being an influence which requires a very low amount of work on its first variable value, i.e 10 routes, this benchmark will give an indication about the best possible throughput and response times possible for each WBAF. Naturally another influence that requires a low amount of work may surpass the request routing results due to network and operating system inconsistencies.

Looking at the plots for both 10 and 100 concurrent requests in figures 6.1 and 6.2 shows that Express.js have a throughput advantage of 1200-1000=200 requests per seconds when employed with 10 routes. Similarly, by looking at the plots which gives the best response time values for both WBAF's, i.e Pmax, in figures 6.3 and 6.4 it can be seen that Express.js has a 7-4=3 ms base response time advantage. This means that when both these WBAF's are employed with only a few amount of routes Express.js can handle 200 more requests per second. Additionally when experiencing a load of only one concurrent request, i.e Pmax, Express.js handles each request averagely 3 ms faster than .NET MVC.

In .NET MVC the small throughput increase with the route increase in figure 6.1 is likely due to a small inconsistency in the network or os. It can also be seen that the number of routes won't affect the performance before 1000 routes is employed in the WBA. The Express.js results in figure 6.2 suggest that its route lookup growth rate isn't dependent on the amount of routes employed in the WBA, as the requests per second never declines as the amount of employed routes in the WBA is increased. This means that Express.js scales better with the number of routes employed.

The area of interest, which is marked as grey in figures 6.5 and 6.6, was defined between 50 and 250 routes, this combined with the concurrency level of interest 10, gives Express.js the upper hand in throughput which can be seen in figure 6.5. However, figure 6.6 suggests that .NET MVC provide a better average latency in the same area.

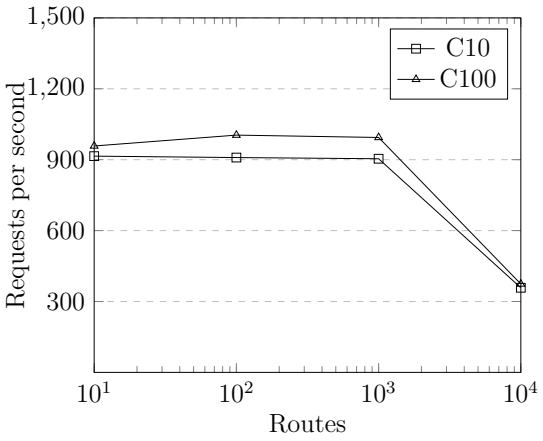

Figure 6.1: Request routing .NET TLC

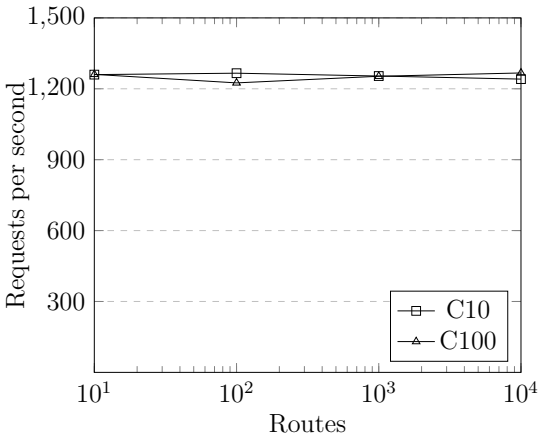

Figure 6.2: Request routing Express.js TLC

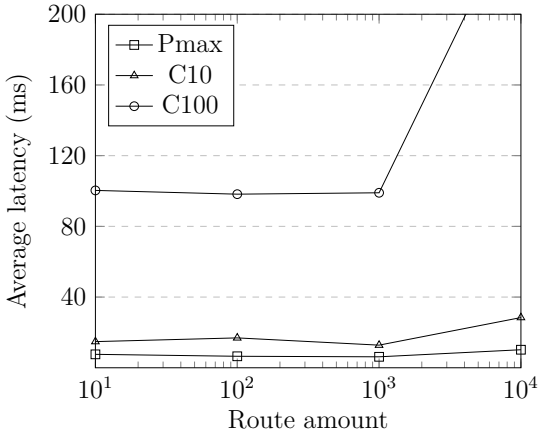

Figure 6.3: Request routing .NET RLC

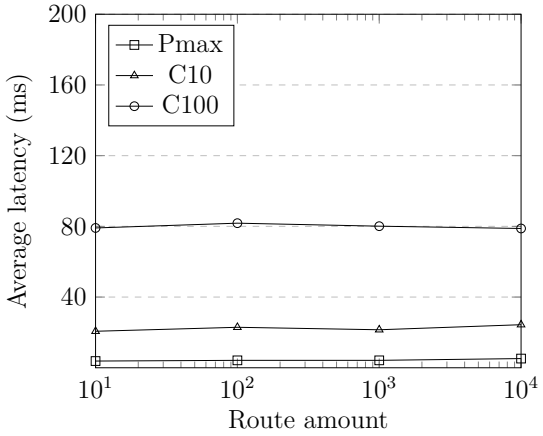

Figure 6.4: Request routing Express.js RLC

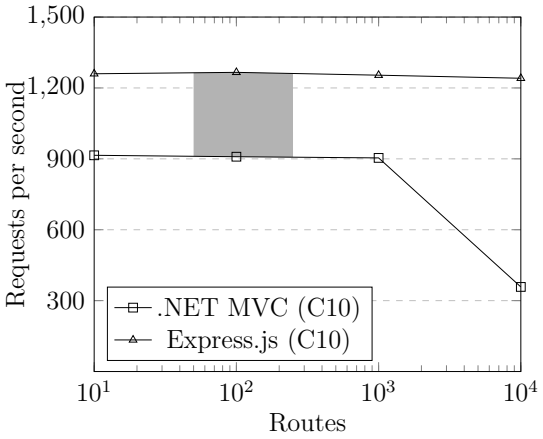

Figure 6.5: Request routing C10 TCLC

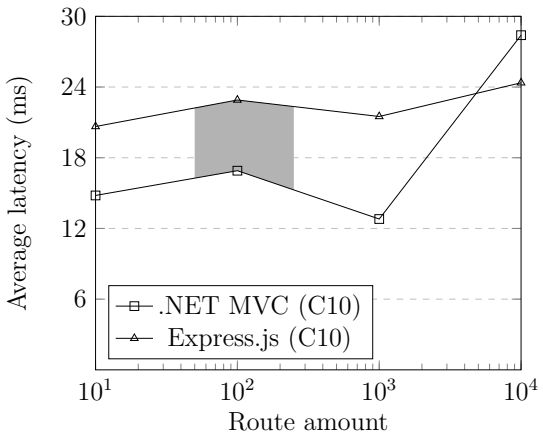

Figure 6.6: Request routing C10 RCLC

6.3.2 Serialization

For the serialization test type JSON was used as the format in which to serialize and deserialize the data. Unlike request routing, the performance in the serialization test type declined as the variable, i.e data size, was increased for both WBAF's. This is obvious as more data means more work, whereas in request routing it was dependent on the the route lookup algorithm. Figures 6.7 to 6.10 supports this statement as a massive decline in throughput and increase in respone time can be seen for each tenfold increase in data size for both WBAF's.

As illustrated in figures 6.9 and 6.10, when benchmarking with 10 concurrent requests timeout errors started to appear at 10 000 kB data size for both WBAF's, and when benchmarking with 100 concurrent requests they started to appear at 1000 kB data size already. As described in section 6.3 these results are skewed, thus the plots are cutoff. These two charts show that serializing data takes longer time for .NET MVC than Express.js. When looking at the plots which loads the WBA with one request at a time, i.e Pmax, it can be seen that Express.js serializes and deserialize 1000 kB data 110/40=2.75 times faster than .NET MVC. These results were somewhat expected and a theory to why this huge difference could be that Javascript objects are more lightweight than the .NET MVC objects. Meaning that when deserializing the string describing the objects, the parser will instantiate many objects, and if this process is faster in Javascript then the results will reflect that advantage.

10 and 100 kB delimits the area of interest for this test type, where the results in figures 6.11 and 6.12 suggest a clear advantage for Express.js in both throughput and response time.

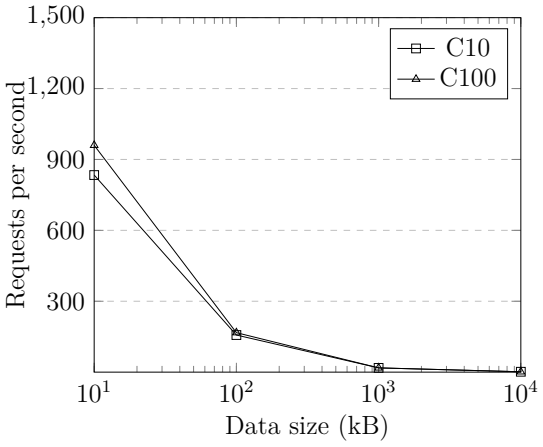

Figure 6.7: Serialization .NET MVC TLC

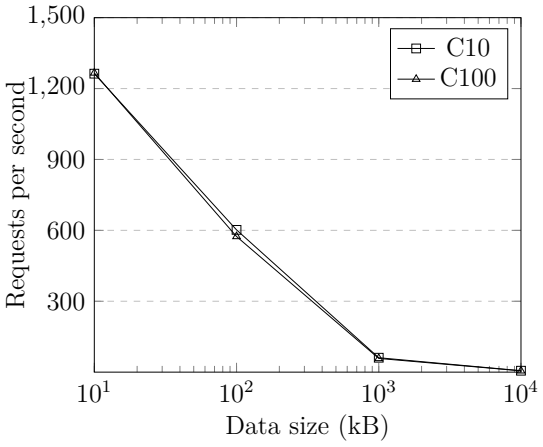

Figure 6.8: Serialization Express.js TLC

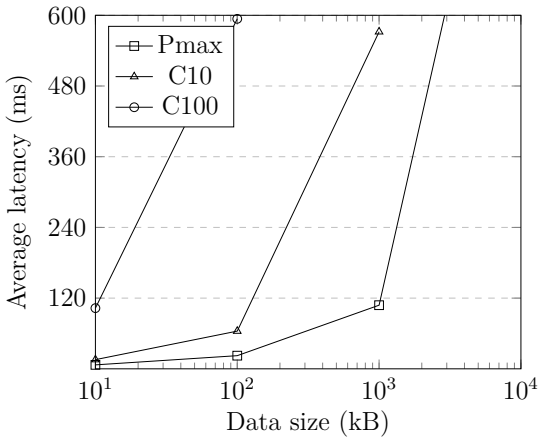

Figure 6.9: Serialization .NET MVC RLC

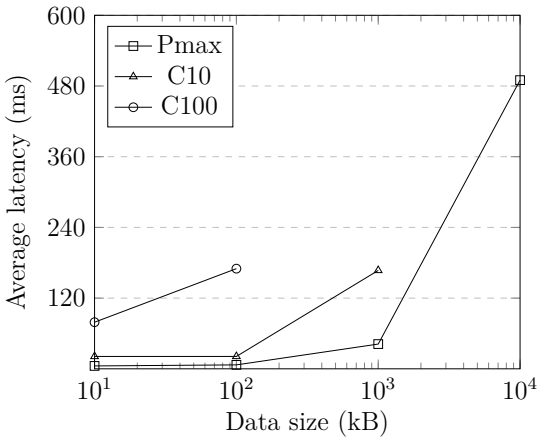

Figure 6.10: Serialization Express.js RLC

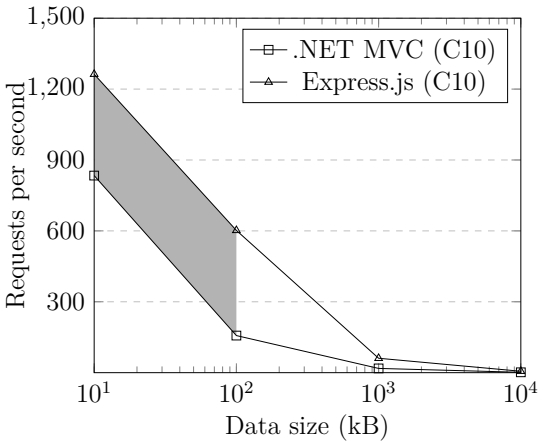

Figure 6.11: Serialization C10 TCLC

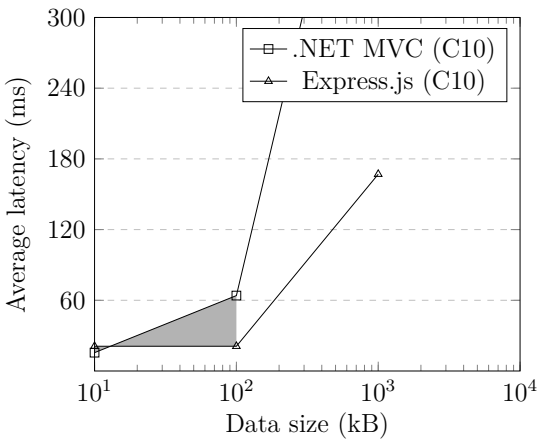

Figure 6.12: Serialization C10 RCLC

6.3.3 O*M

The O*M test type is the only test type that uses a three tier architecture, all other test types are limited to the client and one server. A MySql instance were used on the database server. The O*M packages used for this test type were Entity Framework [58] for .NET MVC and Sequelize [87] for Express.js.

In this test type four variants are being benchmarked instead of a changing variable, thus the bar charts are used to fit this purpose. These results are very interesting in several aspects, e.g for .NET MVC in figure 6.14 the throughput of the Find one, Update and Create variants is far less for C10 than for C100. A theory to why this occurs could be because .NET is using a threaded programming paradigm in combination with the communication being synchronous. In .NET MVC a thread is created for each request that it receives, and the thread is disposed after the response have been sent for that request, thus 10 threads are the maximum number of threads used for C10. In all test types not using an external process, i.e this one

and cache, follows the rule; if the number of concurrent requests are more than the number of cores in the CPU, then the full capacity of the CPU is used and thus the throughput results are aligned for the different concurrences. However, in O*M the communication is performed to the database, and because that communication is synchronous, each thread will be slept when performing a query and won't awake before a result is received from the database. This means that it is possible that so many threads are put to sleep so that the number of threads in work are less than the number of cores in the CPU, meaning that the full capacity of the CPU isn't used. This phenomena is illustratated in figure 6.13, where only three out of the eight threads are awake. If a quad core processor is used in this case, one core is not doing anything. When the concurrency is increased to 100 instead of 10, that possibility becomes far less and the CPU capacity is better employed. The reason to why this isn't so apparent in the Find All variant is that the large result set needs to be parsed to objects, which requires the threads to be alive for a longer duration, thus mitigating the possibility of the number of awake threads falling below the number of cores in the CPU.

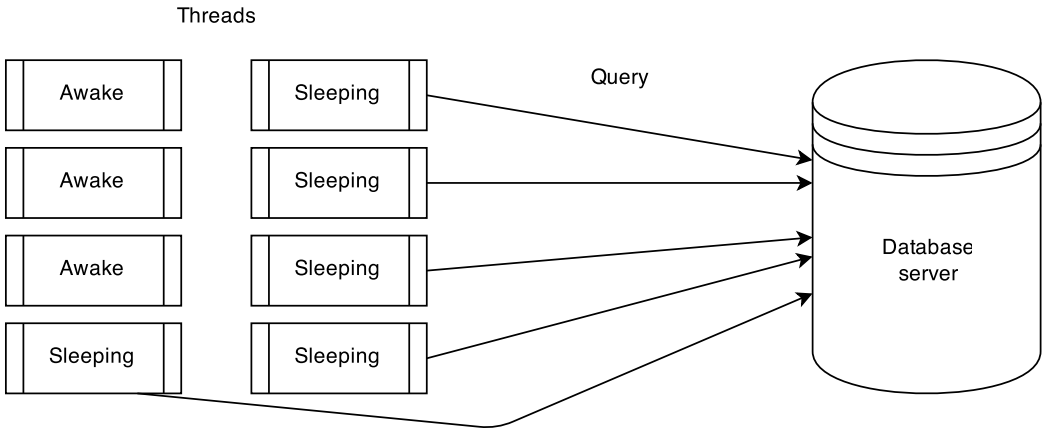

Figure 6.13: Illustrating a case of not utilizing full CPU capacity in the O*M test type as only three threads are awake.

Express.js uses an event driven programming paradigm with asynchronous communication, which according to the results in figure 6.15 seem to provide better throughput for lower concurrences when a database server is used. Note that in the Express.js chart for throughput, i.e figure 6.15, the results are more even between the concurrences.

Another interesting point can be seen in figure 6.18, where .NET MVC provide better throughput than Express.js for the Find All variant. The initial theory was that Express.js would provide better throughput in this variant too, as Express.js advantage for instantiating objects have been clarified in the serialization test type. One reasonable explanation to why this isn't the case is that Sequelize, the O*M plugin used for Express.js, is doing something inefficient and is thus visible for larger data sets.

Furthermore, when comparing the response time in figure 6.19, Express.js has the

advantage over .NET MVC in all variants, including Find all. Another interesting point is why .NET MVC takes so long to do updates compared to Express.js. The high throughput and low response time for update in Express.js negate any theory that the MySql database would be a bottleneck. One would think that the update results for .NET MVC would be more similar to the Create variant, as in the results for Express.js.

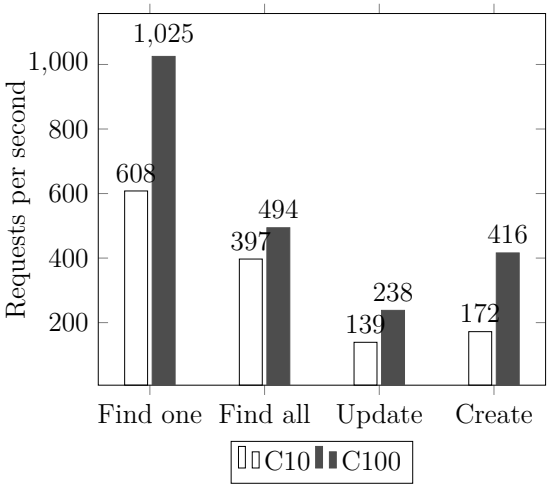

Figure 6.14: O*M .NET MVC TBC

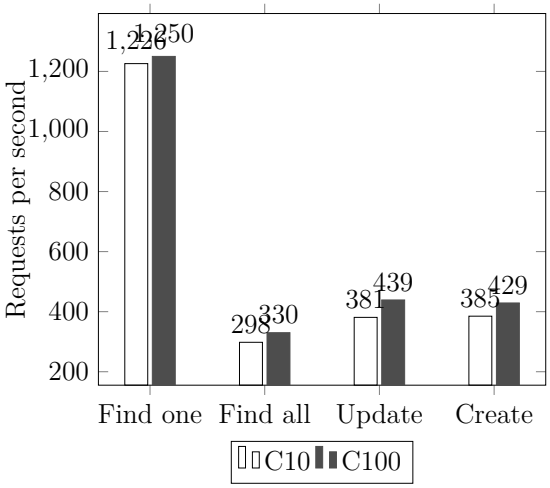

Figure 6.15: O*M Express.js TBC

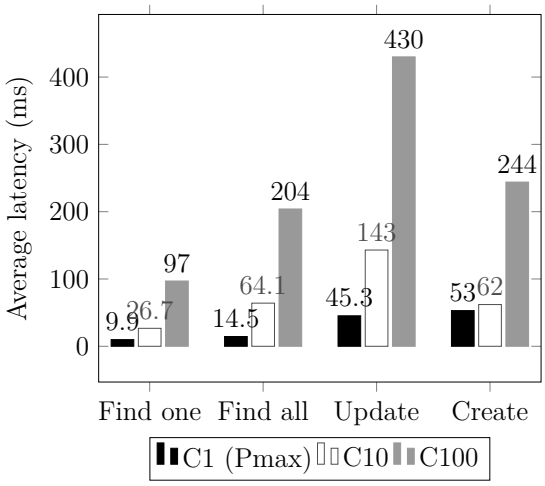

Figure 6.16: O*M .NET MVC RBC

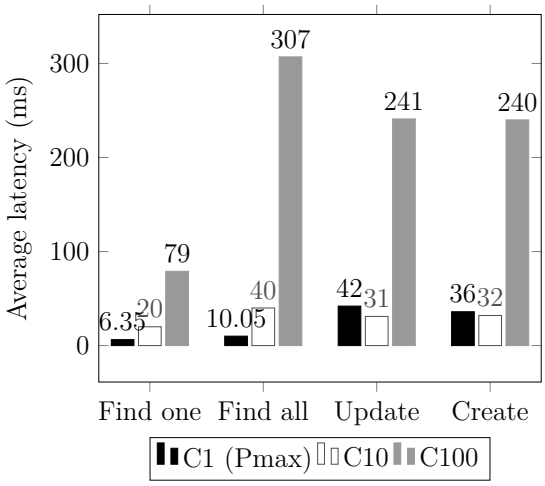

Figure 6.17: O*M Express.js RBC

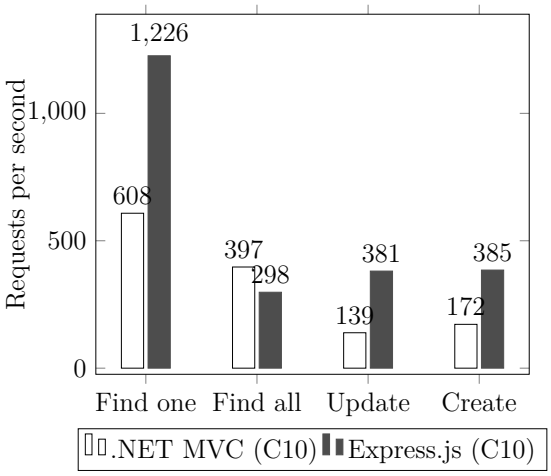

Figure 6.18: O*M C10 TCBC

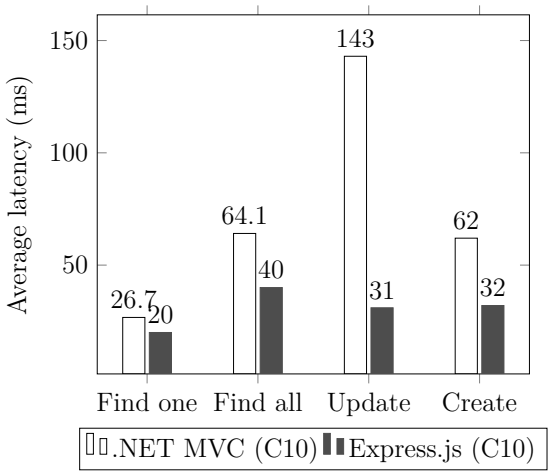

Figure 6.19: O*M C10 RCBC

6.3.4 Template engine

The charts suggest that this influence behave very similar for both WBAF's, with the initial throughput being higher for Express.js as expected due to its higher base throughput. In figure 6.24 it can be seen that the plots intersect at data length around 100. Furthermore, both the charts for throughput and response time suggest that rendering HTML pages is an heavy task. Both .NET MVC and Express.js see their throughput halved for a variable value of 10, compared to the same variable value in request routing.

The response time charts, i.e figures 6.22 and 6.23, have seen an indentation to their plots to better visualize the C100 plot, which is a single dot as higher data length values yielded timeout errors during the benchmarks. Both WBAF's behave almost identically when concurrency is increased, however the charts suggest that Express.js scales better with load.

Defining an area of interest for the TCLC and RCLCs was skipped in this test type as recommended from the Comparison and Evaluation paragraph in section 5.4.4. It doesn't matter to much as Express.js will outperform .NET MVC for all data length values, more so when less rendering is required.

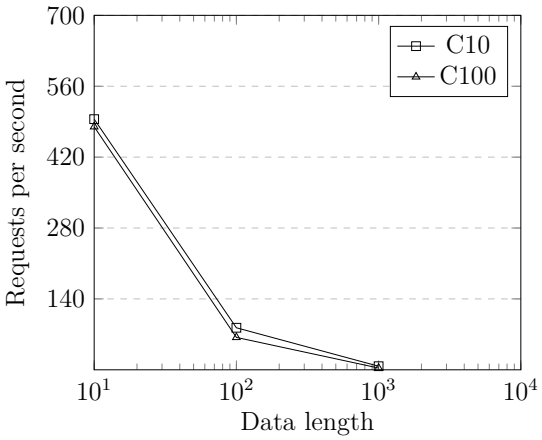

Figure 6.20: Template engine .NET MVC TLC

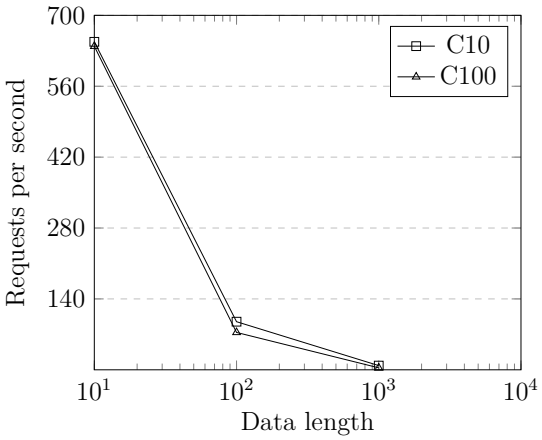

Figure 6.21: Template engine Express.js TLC

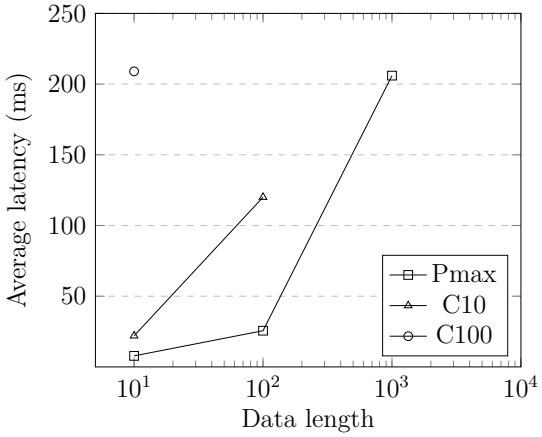

Figure 6.22: Template engine .NET MVC RLC

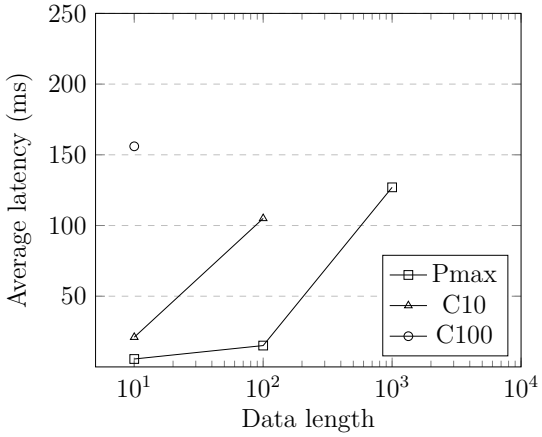

Figure 6.23: Template engine Express.js RLC

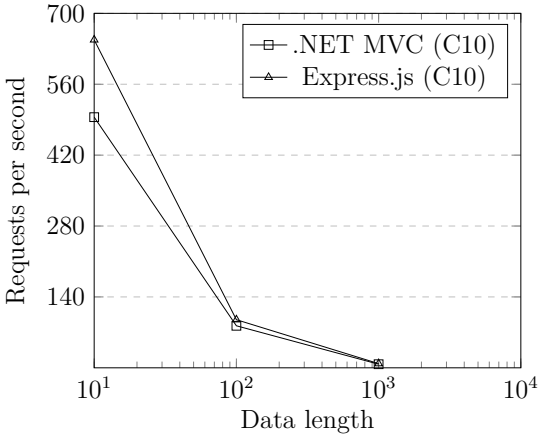

Figure 6.24: Template engine C10 TCLC

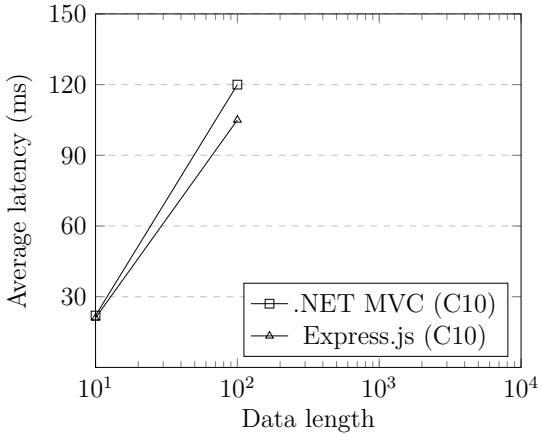

Figure 6.25: Template engine C10 RCLC

6.3.5 JSON Web Token

From the source code for this test type [45] it can be seen that some keys are being read from .pem files. How to generate these files from only the description in section 5.4.5 can be a non trivial task, therefor the commands on how to do it in openssl [75] have been provided. The recommended variants and their corresponding key sizes from section 5.4.5 in the schematic was used.

RSA private:
```
$ openssl genrsa -out private.pem 2048
```
RSA public:
```
$ openssl rsa -in private.pem -out public.pem -outform PEM -pubout
```
EC private:
```
$ openssl ecparam -out privateec.pem -name prime256v1 -genkey
```
EC public:
```
$ openssl req -new -key privateec.pem -x509 -nodes -out cert.pem
```

All charts suggest that encrypting and decrypting with the HS256 and none JWA's is very fast for both WBAF's, as both the throughput and response time results are close to the ones for request routing with 10 routes. However, for the RS256 and ES256 JWA's the performance starts to drop, which is expected as these techniques require more work due to the private/public key architecture. The results in the figures 6.30 and 6.31 suggest that .NET MVC have a huge advantage when it comes to decrypting and encrypting with the RS256 algorithm, and Express.js have an advantage when using ES256. These results are somewhat unexpected, one would expect that the performance would decline with roughly the same portion. A possible explanation might be that these two aren't the best possible implementations of said JWA, i.e the ones made by the package authors.

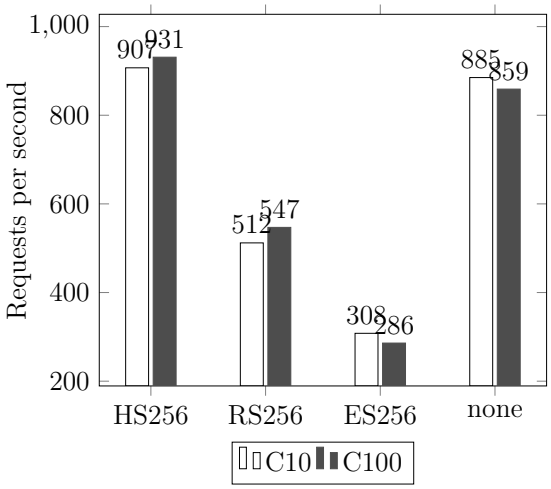

Figure 6.26: JSON Web Token .NET MVC TBC

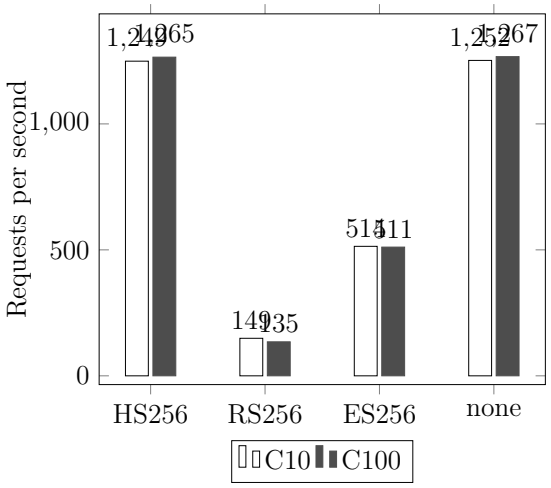

Figure 6.27: JSON Web Token Express.js TBC

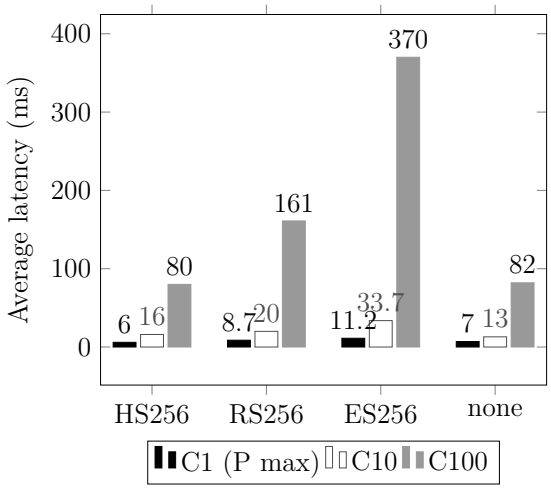

Figure 6.28: JSON Web Token .NET MVC RBC

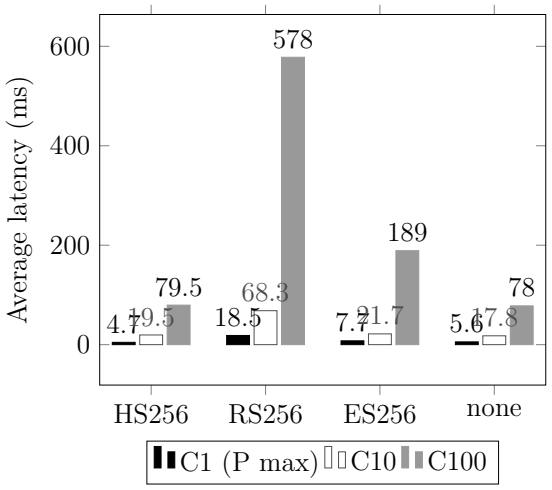

Figure 6.29: JSON Web Token Express.js RBC

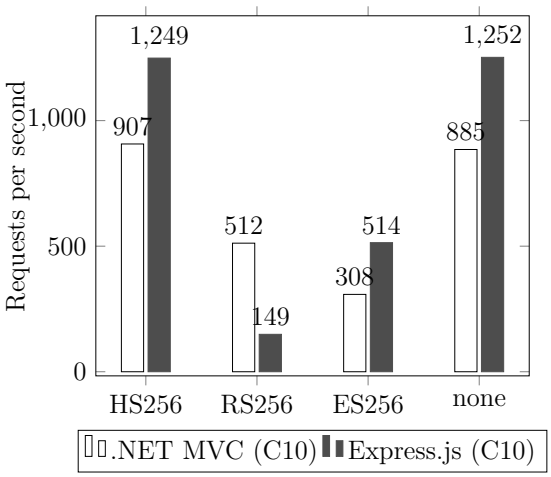

Figure 6.30: JSON Web Token C10 TCBC

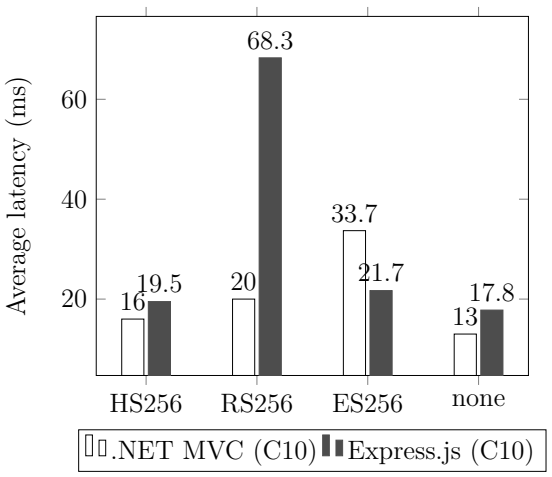

Figure 6.31: JSON Web Token C10 RCBC

6.3.6 Cache

Like O*M, cache relies on a third party, in this benchmark Redis [81] was used as a cache server. The redis instance was deployed on the same computer as the one hosting the cache test type. In the results in figure 6.32 a result appeared that haven't been seen before, the throughput declined as concurrency increased. The initial theory was that the cache server was the reason for this, however the results for Express.js in figure 6.33 negate that theory by showing aligned plots for the concurrency levels 10 and 100. The enormous difference in the results between the two WBAFs in this test type, which can be seen in figures 6.36 and 6.37, is because Express.js is communicating with the Redis server asynchronously, and this functionality wasn't provided in the most popular Redis client package for .NET MVC. This means that .NET MVC must wait for every request to be completed before it can start the next one, and Express.js can do all simultaneously. .NET

MVC would experience this poor performance for all the O*M variants as well if the number of times they would be performed would be increased. This however doesn't explain why the drop in throughput for the concurrency level 100 compared to 10, which is an unexpected result.

10 and 150 are the number of entries to be fechted that delimits the area of interest for this test type. The charts in figures 6.36 and 6.37 show a huge advantage for Express.js in these benchmarks for the defined area of interest, which as mentioned is due to the asynchronous I/O employed by Express.js. However, the charts suggests that this advantage is limited and that .NET MVC will eventually catch up to Express.js, and figure 6.37 shows that for response time it will eventually pass, i.e after 1000 GET's per request. If one looks at the derivative difference of the Express.js plot between 10 to 100 GET's and 100 to 1000 GET's, one can see that the effectiveness of the asynchronous communication starts to wear off. There could be a problem with Redis being loaded with that many GET's simultaneously. The .NET MVC plot is rising very linear through 10 to 1000 GET's, which is due to the communication being synchronous.

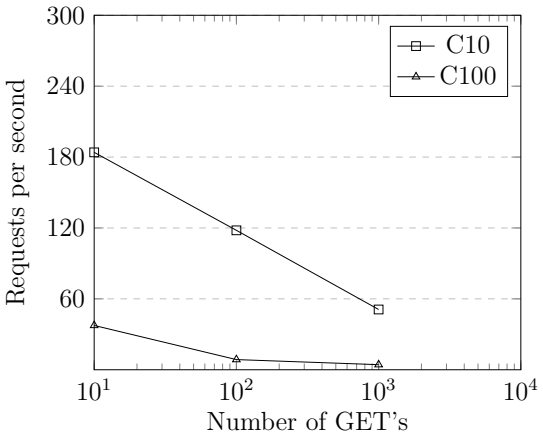

Figure 6.32: Cache .NET MVC TLC

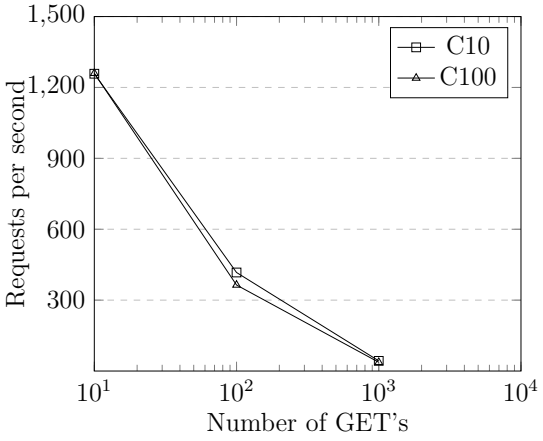

Figure 6.33: Cache Express.js TLC

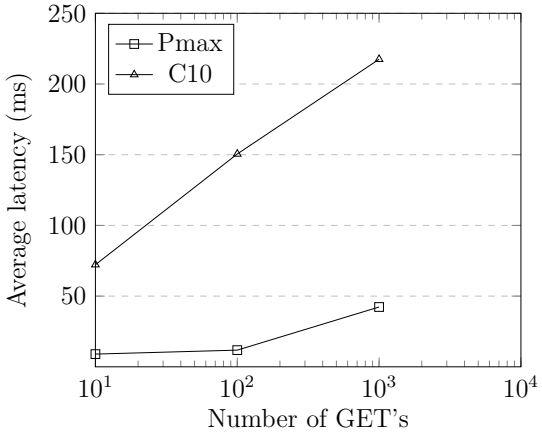

Figure 6.34: Cache .NET MVC RLC

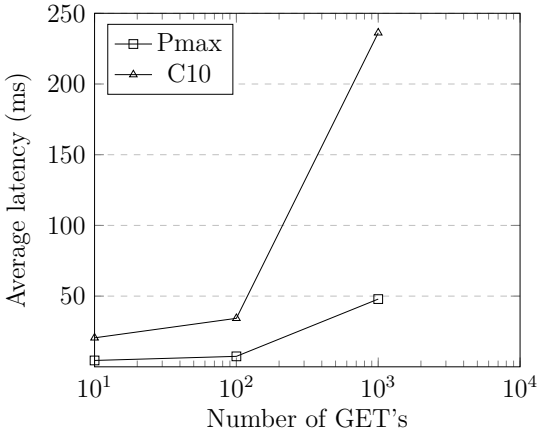

Figure 6.35: Cache Express.js RLC

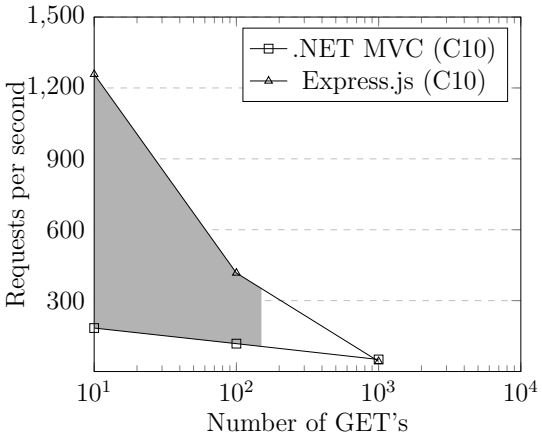

Figure 6.36: C10 TCLC

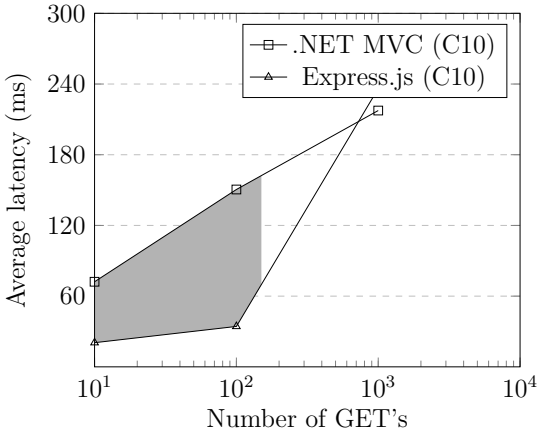

Figure 6.37: C10 RCLC

6.3.7 Websockets

Although an extensive amount of hours was spent on the issue, the .NET MVC websockets test type was unable to be successfully implemented to work with any benchmarking tool. The error appeared only when using a benchmarking tool, though the implementation worked fine when testing with a Chrome plugin called Simple Websocket Client. The implementation were successfully implemented in Express.js, and those benchmarks were successfully run. However, seeing as the schematic is to be used for comparisons means that the standalone Express.js results wouldn't give any value. Therefor it has been decided to not visualize these in charts as it would only confuse the reader.

7

Evaluation

Both a schematic and an example scenario have been provided in this thesis. The example scenario is implemented in such a way that it could and should be used as a complement to the schematic. This means that the environment and the implemented code may seem a bit simplistic and not as close to a real use case as one might have wanted. However, this is done by design. The implemented code was done as a general as possible to include all type of use of the schematic. The environment was set up on a local area network with only one router instead of the more realistic scenario including more routers over wider areas. This was done to get more control of the external factors which in the end results in clear and deterministic measured results from the tests.

In the example scenario the comparison is made between .NET MVC and Express.js. This might seem skewed because .NET MVC is classified as a fullstack framework while Express.js is a micro framework. When comparing WBAF's of different types the more lightweight framework needs not only to relay on itself but also on the packages that is needed to complement for the missing functionality that a heavier framework provides. While this might seem unfair the idea behind doing this is that while a comparison between frameworks of the same type would be easier and more straight forward the meaning of the example scenario is to be an aid and to show what and how it is possible to use the schematic. Another difference between .NET MVC and Express.js is that they use different communication paradigms for the I/O test types i.e. O*M and Cache, .NET MVC uses a synchronous communication and Express.js uses asynchronous communication. The result of this is that the asynchronous implementation will yield better result for all test types were this is utilized which means that Express.js will have a big advantage. However, to keep the simplicity and readability of the code and to follow the scenario guideline of using the most popular package when the need of an external package is needed the choice to give Express.js this advantages was accepted. Also, the actual measured results of the example scenario is not made to be a benchmark for the schematic and is therefore irrelevant for the result of this thesis.

During the execution of the example schematic it soon became evident that running all tests with the different concurrency level and variables would take a great deal of time. Not counting the time it takes to set up the test environment, develop the code for the test types and analyzing the results it would take approximately 7,5 hours to run the tests for one minute three consecutive times. This raises the question if

the execution of the tests could be streamlined in some efficient way. Even if it isn't in the scope of this thesis, some kind of automation of the testing procedure would be desirable to minimize the manually labour that needs to be performed to read and analyze the data.

On GitHub [45] the source code for the example scenario is provided. The source code is not meant to be seen as a perfect implementation of the test types but rather an aid to show how an implementation can look like. It is also a way to have transparency throughout the thesis so that each step can be back traced and reproduced by the users of the schematic.

7.1 Threats to Validity

To mitigate threats to the validity of this thesis one goal was that the resulting schema shall be able to be completely backed traced and a clear motivation and description should be given for the choices that were made in the different steps of constructing the schematic. However, it's impossible to remove all threats and here follows some of the threats that were identified.

Internal The steps to derive an influence was done in a process constructed by the authors thus the resulting influences does not have a scientific foundation and could be irrelevant.

When the test types were developed a research was conducted to see what variables effected the performance. Some variable can have, unintentionally, been disregarded that could have effect on the result and overrides the influence of the test type in question.

External The test environment is a Microsoft environment which is the same company that delivers the .NET platform. This could benefited one of the example WBAF more than the other. The test tool used could have performed it's operations in a way that does not represent desired behaviour.

The schematic is validated and exemplified by the same persons that wrote the description of the schematic. This may lead to the possibility of having to vague descriptions and motives for one or more of the test types because of missing information that is tacit knowledge for the authors.

Construct The implemented code in the example scenario is written in as performing way as possible at the time. However, senior developers in the example platforms can most likely implement the same functionality in an even more performing way.

8

Conclusion

In the web context, many identifications and expressions are characterised by some ambiguity in the terminology, the ambiguities comprise distinction issues between terms, and terms that are used for to many purposes. Therefor a terminology section was conducted to enable further discussion without repeatedly having to explain and define the context of use for each term. These terminologies have helped immensely during the course of this thesis, not only through making it easier to write, but also in conversation.

The first research question was; what influences the scalability and responsiveness of a WBAF? This is answered in chapter 4, because there is no definition or limit on how much functionality should be included in a WBAF, as it is up to the author of each WBAF to decide what to include, the task to define everything that affects its performance becomes an unfeasible one. Instead a study was conducted in order to define the most popular influences of a WBAF, and these results are marked as bold in table 4.3. The only influence found that after further inspection was decided not to be included as a test type in the schematic was authentication, as it became apparent that this test type would basically only benchmark a third party source.

The second research question was; how to compare scalability and responsiveness of WBAF's? The answer to this question is to, after having identified what affects the scalability and responsiveness of a WBAF, engineer metrics and methods to comprehend the expected results. Then perform isolated microbenchmarks on said influences in a homogeneous environment. All of these steps are described chapter 5 and exemplified in chapter 6, thus these two chapters together answers the second research question thoroughly.

Chapter 6 verifies that the produced schematic works for its intended purpose, i.e comparing WBAF's. A user will gain more knowledge about the scalability and responsiveness in the WBAF's benchmarked after having used the schematic. The plotted results in the form of charts made it easy to get an overview of the comparison on a per test type level. However, the schematic do lack a way of declaring a winner on both a per test type, and on a per WBAF level.

The running example was successfully implemented, additionally the results was prestented and discussed for all test types except websocket. The reason for this was that errors were thrown in the .NET MVC websocket test type during benchmarks,

the results for the Express.js was left out as it became redundant without any comparison.

After having done the running example, it became apparent that the schematic is not limited to testing WBAF's, even if that was the original intention. A third test suite could have been added, and this test suite could also been Express.js for instance, but with different packages chosen for the test types, or with a different operating system. Because of the extendability of the schematic, any block out of the technology stack can be swapped and a new benchmark can be run.

8.1 Future work

The schematic needs to be tested and further developed in close contact with potential users. As mentioned in section 7.1, the schematic as is might be hard to use as it has only been used by its authors. Therefor a case study could be performed in order to see if users are capable of developing the test types for new WBAF's and different languages. From this case study pros and cons could be gathered and translated into requirements that would improve the schematic.

The schematic itself is easily extensible, thus adding more test types could and should be done to suit a broader audience. The left out influences in table 4.3 would be a good start, as these are the next most fitting influences according to influence partial study.

Manually performing the benchmarks and registering the results was a cumbersome task, which for two WBAF's took eight hours of benchmarking time, i.e approximately two days for two people, to complete. Therefor this process should be automated, which would spare developer time.

The test types should also be made public, a GitHub repository might suffice as Techempowered did. The reason behind this is that it mitigates the risk of a test type being wrongly implemented if more people collaborate with their experiences. Additionally it saves developing time as each test type won't have to be totally developed from scratch for each user. Any configuration or modification could be done locally to the source code.

For the test types more variants or variables could be added. An example could be to add Delete to the O*M test type, or add data length to the cache test type. The latter would require a new chart type as a third dimension is introduced.

As mentioned the schematic do lack a way of declaring a winner on a per test type and WBAF level. The intention behind leaving this out was to keep it open for each user to evaluate the results in their own context, as every user would weight the result of each test type differently. This could however be implemented in the method that would declare the winner, i.e a set of weighing parameters for each test type in combination with some formula that outputs a score. It would be very convenient to present business people if they had something this easy to grasp, e.g

"WBAF 1 scored 87 and WBAF 2 scored 62 with the company's internal weighing parameters, therefor WBAF 1 have been chosen for the next investment".

The ultimate vision for the schematic is to build it as a software, a program that would integrate the benchmarking tool, result export and methods for declaring a winner. Visualizing the results could be done in any business intelligence tool, therefor implementing this would be wasteful. A benchmark would be set up from a configuration file with different routes and parameters that would be mapped to the different test types and variable/variant values, and different ports could be used for different WBAF's.

Bibliography

[1] A. Bondi. *Characteristics of Scalability and Their Impact on Performance*. AT&T Labs, New Jersey, 1994.

[2] A. K. Lenstra. *Key Lengths - Contribution to The Handbook of Information Security*. Citibank, N.A., and Technische Universiteit Eindhoven 2002. [2015-05-19].

[3] A. K. Lenstra, E. R. Verheul *Selecting Cryptographic Key Sizes*. J. Cryptology (2001) 14: 255–293, DOI: 10.1007/s00145-001-0009-4. 2001. [2015-05-19].

[4] A. Mesbah, A. van Deursen. *Migrating Multi-page Web Applications to Single-page AJAX Interfaces*. IEEE, Amsterdam, 2007.

[5] A. R. Hevner, S. T. March, J. Park. *Design Research in Information Systems Research*. MIS Quarterly, 28, 1, 75-105. 2004.

[6] Agence nationale de la sécurité des systèmes d'information. *Référentiel Général de Sécurité*. `http://www.ssi.gouv.fr/uploads/2015/01/RGS_v-2-0_B1.pdf` 2014. [2015-05-19].

[7] AngularJs. *AngularJS — Superheroic JavaScript MVW Framework*. `https://angularjs.org/`. 2015. [2015-03-11].

[8] Apache. *Welcome! - The Apache HTTP Server Project*. `http://httpd.apache.org/`. 2015. [2015-03-11].

[9] Apache. *ab - Apache HTTP server benchmarking tool*. `http://httpd.apache.org/docs/2.2/programs/ab.html`. 2015. [2015-05-19].

[10] B. Hauer. *What is Gemini?*. `https://groups.google.com/forum/#!topic/framework-benchmarks/p3PbUTg-Ibk`. 2013. [2015-04-01].

[11] BestGems. *Total Download Ranking*. `http://bestgems.org/total`. 2015. [2015-04-01].

[12] BlueKrypt. *Cryptographic Key Length Recommendation*. `http://www.keylength.com/en/compare/`. 2015. [2015-05-19].

[13] Bundesnetzagentur für Elektrizität, Gas, Telekommunikation, Post und Eisenbahnen. *Bekanntmachung zur elektronischen Signatur nach dem Signaturgesetz und der Signaturverordnung*. `http://www.bundesnetzagentur.de/SharedDocs/Downloads/DE/Sachgebiete/QES/Veroeffentlichungen/`

Algorithmen/2014Algorithmenkatalog.pdf?__blob=publicationFile&v=1 2014. [2015-05-19].

[14] C. Bauer, G. King *Hibernate in Actio*. Manning Publications Co. Greenwich, USA. ISBN: 1932394-15-X 2004.

[15] C. Click *The Art of (Java) Benchmarking*. http://www.azulsystems.com/events/javaone_2009/session/2009_J1_Benchmark.pdf. 2009. [2015-03-27]

[16] C. U. Smith, L.G. Williams. *Performance solutions: a practical guide to creating responsive, scalable software*. Boston, MA, Addision Wesley, 2002.

[17] ComputerHope. *Computer Hope's free computer help*. http://www.computerhope.com/. 2015. [2015-03-11].

[18] Curious Concept. *JSON Formatter & Validator*. http://jsonformatter.curiousconcept.com/#about. 2015. [2015-05-19].

[19] D. Esposito. *ASP.NET MVC Controllers and Conventions*. https://www.simple-talk.com/dotnet/asp.net/asp.net-mvc-controllers-and-conventions/. 2012. [2015-03-03].

[20] Django. *The web framework for perfectionists with deadlines*. https://www.djangoproject.com/. 2015. [2015-05-19].

[21] ECRYPT II. *ECRYPT II Yearly Report on Algorithms and Keysizes*. ICT-2007-216676, Katholieke Universiteit Leuven (KUL) 2012. [2015-05-19].

[22] ExpressJs. *Express - Node.js web application framework*. http://expressjs.com/. 2015. [2015-03-11].

[23] Flask. *Welcome | Flask (A Python Microframework)*. http://flask.pocoo.org/. 2015. [2015-03-24].

[24] Foldoc. *FOLDOC - Computing Dictionary*. http://foldoc.org/. 2015. [2015-03-11].

[25] G. Heiser *The Dos and Don'ts of Benchmarking*. http://haifux.org/lectures/311/BM_crimes.pdf. 2013. [2015-03-31]

[26] G. Linden. *Make data useful*. http://glinden.blogspot.se/2006/11/marissa-mayer-at-web-20.html.2006. [2015-03-09]

[27] G. Wilson, D. A. Aruliah, C. T. Brown, N. P. Chue Hong, M. Davis, et al. *Best Practises for Scientific Computing*. PLoS Biol 12(1): e1001745. doi:10.1371/journal.pbio.1001745, 2014.

[28] GitHub. *Build software better, together*. https://github.com/. 2015. [2015-05-19].

[29] Hibernate. *Hibernate ORM*. http://hibernate.org/orm/. 2015. [2015-03-23]

[30] Hot Frameworks. *Find your new favorite web framework*. http://hotframeworks.com/. 2015. [2015-05-19].

[31] Hot Frameworks. *Frequently Asked Questions*. http://hotframeworks.com/faq. 2015. [2015-05-19].

[32] I. Z. Schlueter. *Node.js Digs Dirt - about Data-Intensive Real-Time Applications*. http://video.nextconf.eu/video/1914374/nodejs-digs-dirt-about.2011. [2015-03-20].

[33] IETF. *A Universally Unique IDentifier (UUID) URN Namespace*. http://tools.ietf.org/html/rfc4122. 2005. [2015-05-19].

[34] IETF. *HMAC: Keyed-Hashing for Message Authentication*. http://tools.ietf.org/html/rfc2104 1997. [2015-05-19].

[35] IETF. *Elliptic Curve DSA for DNSSEC*. https://tools.ietf.org/html/draft-ietf-dnsext-ecdsa-07 2012. [2015-05-19].

[36] IETF. *Hypertext Transfer Protocol – HTTP/1.1*. https://tools.ietf.org/html/rfc2616. 1999. [2015-03-24].

[37] IETF. *RSASSA-PSS*. http://tools.ietf.org/html/rfc3447#section-8.1. 2003. [2015-03-24].

[38] J. Bixby. *Latency 101: What is latency and why is it such a big deal?*. http://www.webperformancetoday.com/2012/04/02/latency-101-what-is-latency-and-why-is-it-such-a-big-deal/. 2012. [2015-03-09].

[39] J. Bornholt *How Not to Measure Computer System Performance*. https://homes.cs.washington.edu/~bornholt/post/performance-evaluation.html. 2014. [2015-03-27]

[40] J. Harrell. *Node.js at PayPal*. https://www.paypal-engineering.com/2013/11/22/node-js-at-paypal/. 2013. [2015-05-15].

[41] J. Knupp. *What is a Web Framework?*. http://www.jeffknupp.com/blog/2014/03/03/what-is-a-web-framework/. 2014. [2015-03-02].

[42] K. Peffers, T. Tuunanen, M. Rothenberger, S. Chatterjee. *A Design Science Research Methodology for Information Systems Research*. Journal of Management Information Systems, vol. 24, no. 3, pp. 45-77. 2007.

[43] L. Bass, P. Clements, R. Kazman. *Software architecture in practice second edition*. Addision Wesley, 2003.

[44] M. B. Jones. *JSON Web Algorithms (JWA)*. http://tools.ietf.org/id/draft-ietf-jose-json-web-algorithms-08.html. 2012. [2015-05-19].

[45] M. Dosé, H. Lilja. *Source code to the example scenario*. https://github.com/mathiasdose/master-thesis/tree/master 2015. [2015-05-28].

[46] M. Fowler, D. Rice, M. Foemmel, E. Hieatt, R. Mee, R. Stafford *Patterns of Enterprise Application Architecture*. Pearson Education, Inc. Boston, USA. ISBN: 0-321-12742-0 2002.

[47] M. Koziarski. *Request per second*. http://www.therailsway.com/2009/1/6/requests-per-second/. 2009. [2015-03-31].

[48] M. Rouse. *What is authentication?*. http://searchsecurity.techtarget.com/definition/authentication. 2015. [2015-05-19].

[49] M. Woodside, G. Franks, D. C. Petriu. *The Future of Software Performance Engineering*. Washington, DC, USA, IEEE Computer Society, 2007.

[50] Mathias *Benchmarking Spray*. http://spray.io/blog/2013-05-24-benchmarking-spray/. 2013. [2015-03-31].

[51] Maven. *Maven - Welcome to Apache Maven*. https://maven.apache.org/ 2015. [2015-04-01].

[52] Memcached. *About Memcached*. http://memcached.org/about. 2015. [2015-05-05].

[53] Microsoft. *ASP.NET*. http://www.asp.net/mvc. 2015. [2015-05-19].

[54] Microsoft. *Visual C#*. https://msdn.microsoft.com/en-us/library/kx37x362.aspx. 2015. [2015-03-20]

[55] Microsoft. *Overview of the .NET Framework*. https://msdn.microsoft.com/en-us/library/zw4w595w.aspx. 2015. [2015-03-11].

[56] Microsoft. *ASP.NET MVC | The ASP.NET Site*. http://www.asp.net/mvc. 2015. [2015-03-20]

[57] Microsoft. *Home : The Official Microsoft IIS Site*. http://www.iis.net/. 2015. [2015-03-11].

[58] Microsoft. *Entity Framework*. https://msdn.microsoft.com/en-us/data/ef.aspx. 2015. [2015-03-24].

[59] Mongo DB. *NOSQL DATABASES EXPLAINED*. http://www.mongodb.com/nosql-explained. 2015. [2015-05-01].

[60] Mono. *Documentation | Mono*. http://www.mono-project.com/docs/. 2015. [2015-03-11].

[61] MSDN. *Cache Clients and Local Cache*. https://msdn.microsoft.com/en-us/library/ee790983(v=azure.10).aspx. 2015. [2015-04-21]

[62] MSDN. *ASP.NET Caching*. https://msdn.microsoft.com/en-us/library/xsbfdd8c(v=vs.140).aspx. 2015. [2015-04-21]

[63] MSDN. *ASP.NET MVC OUTPUT CACHING WITH WINDOWS APPFABRIC CACHE*. http://www.devtrends.co.uk/blog/asp.net-mvc-output-caching-with-windows-appfabric-cache. 2015. [2015-04-21]

[64] MVNRepository. *Top Projects*. http://mvnrepository.com/popular 2015. [2015-04-01].

[65] Nginx. *Nginx*. http://nginx.org/en/. 2015. [2015-03-11]

[66] NHibernate. *NHibernate: Home*. http://nhibernate.info/. 2015. [2015-03-24].

[67] NIST. *KEY MANAGEMENT*. http://csrc.nist.gov/groups/ST/toolkit/key_management.html 2015. [2015-05-19].

[68] NodeJS. *Node.js v0.12.3 Manual & Documentation*. https://nodejs.org/api/cluster.html. 2015. [2015-05-19].

[69] NodeJS. *Node.js in Industry*. https://nodejs.org/industry/. 2015. [2015-05-19].

[70] NPM. *npm*. https://www.npmjs.com/. 2015. [2015-04-01].

[71] NPM. *Most Starred Packages*. https://www.npmjs.com/browse/star. 2015. [2015-04-01].

[72] NuGet. *NuGet Gallery | Home*. https://www.nuget.org/. 2015. [2015-04-01].

[73] NuGet. *NuGet Gallery | Package Download Statistics*. https://www.nuget.org/stats/packages. 2015. [2015-04-01].

[74] Observe.it. *Thor - The WebSocket god of thunder*. https://github.com/observing/thor. 2013. [2015-05-19].

[75] OpenSSL. *OpenSSL Project*. https://www.openssl.org/. 2015. [2015-05-28]

[76] Packagist. *The PHP package archivist*. https://packagist.org/. 2015. [2015-04-01].

[77] Packagist. *Popular Packages*. https://packagist.org/explore/popular. 2015. [2015-04-01].

[78] Python. *PyPI - Python*. https://pypi.python.org/pypi. 2015. [2015-04-01].

[79] PyPI Ranking. *Find famous Python modules and authors*. http://pypi-ranking.info/alltime. 2015. [2015-04-01].

[80] Redis. *Using Redis as an LRU cache*. http://redis.io/topics/lru-cache. 2015. [2015-04-21]

[81] Redis. *Introduction to Redis*. http://redis.io/topics/introduction. 2015. [2015-05-05].

[82] Ruby on Rails. *Web development that doesn't hurt*. http://rubyonrails.org/. 2015. [2015-05-19].

[83] RubyGems. *RubyGems.org | your community gem host*. https://rubygems.org/gems. 2015. [2015-04-01].

[84] S. McCullough, *Breaking The Monolith: Using Node.js to Rewrite Groupon's Front-End by Sean McCullough*. http://vimeo.com/97666093. 2015. [2015-05-27].

[85] S. Silbert *Samsung reportedly not alone in cheating Android benchmarks.* http://www.engadget.com/2013/10/02/samsung-reportedly-not-alone-in-cheating-android-benchmarks/. 2013. [2015-03-31].

[86] Sails.js. *The web framework of your dreams.* http://sailsjs.org/#!/. 2015. [2015-05-19].

[87] Sequelize.js. *A promise-based ORM for Node.js and io.js.* http://docs.sequelizejs.com/en/latest/. 2015. [2015-05-28].

[88] Spring. *Let's build a better Enterprise.* https://spring.io/. 2015. [2015-05-19].

[89] Stack Exchange. *Stack Overflow.* http://stackoverflow.com/. 2015. [2015-05-19].

[90] StrongLoop. *Node.js is Enterprise Ready.* https://strongloop.com/node-js/nodejs-infographic/. 2015. [2015-05-19].

[91] Symfony. *Symfony is a set of reusable PHP components.* https://symfony.com/. 2015. [2015-05-19].

[92] T. C. Chieu, A. Mohindra, A. A. Karve, A. Segal. *Dynamic Scaling of Web Applications in a Virtualized Cloud Computing Environment* e-Business Engineering, 2009. ICEBE '09. IEEE International Conference on, 281-286, 2009.

[93] T. Fisher. *Cryptographic Hash Function.* http://pcsupport.about.com/od/termsc/g/cryptographic-hash-function.htm. 2015. [2015-05-19].

[94] TechEmpower. *Web Framework Benchmarks.* https://www.techempower.com/benchmarks/. 2015. [2015-05-19].

[95] Techopedia. *Serialization.* http://www.techopedia.com/definition/867/serialization-net. 2015. [2015-04-08].

[96] U.S. Department of Commerce. *Digital Signature Standard (DSS).* Information Technology Laboratory, National Institute of Standards and Technology, Gaithersburg 2013. [2015-05-19].

[97] V. Baryshev. *template-benchmark.* https://github.com/baryshev/template-benchmark. 2013. [2015-05-19].

[98] V. Beal. *What is authentication?, Webopedia.* http://www.webopedia.com/TERM/A/authentication.html. 2015. [2015-05-19].

[99] W. Ali, S. Shamsuddin, A. Ismail *A Survey of Web Caching and Prefetching.* Universiti Teknologi Malaysia, Johor 2011.

[100] W. Glozer. *Modern HTTP benchmarking tool.* https://github.com/wg/wrk. 2015. [2015-05-19].

[101] W. Koffel. *Choosing A Web Framework.* http://www.clearlytech.com/2013/12/01/choosing-web-framework/. 2013. [2015-03-03].

[102] w3.org. *Token Based Authentication – Implementation Demonstration*. http://www.w3.org/2001/sw/Europe/events/foaf-galway/papers/fp/token_based_authentication/. 2015. [2015-05-19].

[103] Webopedia. *1000Base-T (IEEE 802.3ab)*. http://www.webopedia.com/TERM/1/1000BaseT.html. 2015. [2015-05-19].

[104] Webopedia. *Webopedia: Online Tech Dictionary for IT Professionals*. http://www.webopedia.com/. 2015. [2015-03-11].

[105] WhatIs. *Webopedia: Online Tech Dictionary for IT Professionals*. http://whatis.techtarget.com/. 2015. [2015-03-11].

[106] X. Gan *Software Performance Testing*. Univesity of Helsinki, Finland 2006.