Bachelor of Science in Software Engineering
May 2022

Express.js and Ktor web server performance A comparative study

Isac Glantz
Hampus Hurtig

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

This thesis is submitted to the Faculty of Computing at Blekinge Institute of Technology in partial fulfilment of the requirements for the degree of Bachelor of Science in Software Engineering. The thesis is equivalent to 10 weeks of full time studies.

The authors declare that they are the sole authors of this thesis and that they have not used any sources other than those listed in the bibliography and identified as references. They further declare that they have not submitted this thesis at any other institution to obtain a degree.

Contact Information:
Author(s):
Isac Glantz
E-mail: isac.glantz.ig@gmail.com

Hampus Hurtig
E-mail: hampus.hurtig@outlook.com

University advisor:
Doctoral student Michel Nass
Department of Software Engineering

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

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

Abstract

As more and more companies use the internet to grow their businesses and sales, it is crucial to have a fast and responsive site that keeps customers on the site. Hence, comparing two web frameworks with respect to response time is vital, as it is a significant part of delivering the page. The comparison will help developers to choose between Express.js and Ktor.

Our research shows how the two frameworks, Ktor and Express.js, compare in response times for static and dynamic pages for a set of concurrent users. The comparison will explain how the frameworks' response times change when having a different number of concurrent users and delivering static vs. dynamic content.

An experiment with Locust was conducted to obtain the data needed to show the differences in response time for the two frameworks. Additionally, a literature study was conducted to find the best way to structure the servers, design the tests, and find information on how the frameworks should perform.

We found that Express.js has an overall better response time than Ktor. At the same time, it was found that the Object Relational Mapper used with Ktor affected the result more than the Object Relational Mapper used with Express.js. Hence, we conclude that Express.js is the better choice, but since both frameworks had low response times, we would say that even Ktor is a valid choice.

Keywords: Performance, Concurrency, Web frameworks, Kotlin, JavaScript

Contents

List of Figures

Listings

List of Tables

Glossary

"top" command A shell program that provides a real-time view of processes or threads running on a UNIX-like operating system such as Linux and Mac OS and can provide information about current CPU and memory usage [49]. 12

average In this thesis average refer to mean average. iii, v, 5, 6, 19–22, 25, 26, 28, 29

bounce rate Percentage of users that leaves the website after only viewing one page. 6, 29

Chrome's V8 A JavaScript engine that executes JavaScript code. V8 can execute JavaScript code outside of a web browser, making it possible to write and run JavaScript even for servers [5]. 2, 5, 41

Embedded JavaScript A templating engine for JavaScript that in our testing is combined with the Express.js framework to generate dynamic pages. 3, 9

Express.js "... a minimal and flexible Node.js web application framework that provides a robust set of features for web and mobile applications." [12] that in our thesis is used to manage requests for the JavaScript server. i, iv, 1, 4, 5, 7–10, 12–17, 19–27, 29, 41, 42

full stack When a developer can do both backend and frontend programming [48]. 26

JavaScript JavaScript (JS) is a lightweight, interpreted, or just-in-time compiled programming language mostly used when developing for the web [32]. 4, 5, 7, 13, 16, 26, 30

JetBrains A development company that is mainly known for its powerful IDEs and is also the company that created the Kotlin programming language, as well as the Ktor framework and the Exposed ORM used in this thesis. 5, 10, 16

Kotlin An open-source statically typed programming language that targets the Java Virtual Machine, Android, JavaScript and Native that is 100% interoperable with the Java programming language, which means that Kotlin code can call/interact with Java code and Java code can call/interact with Kotlin code [19]. 5, 7, 9, 10, 13, 15–17, 26, 29, 41

Ktor A framework developed by JetBrains that is used to "... easily build connected applications – web applications, HTTP services, mobile and browser applications." [21] in the Kotlin language. i, iv, 4, 5, 7–10, 12–17, 19–27, 29, 41, 42

Locust A load testing framework where it is possible to define user behavior with Python code and simulate a given number of concurrent users making requests to a system. i, 6, 7, 11–13, 15, 17, 19, 25, 28

lorem ipsum Dummy text without meaning often used when developing websites before real content is created. 9

Model View Controller An architectural pattern that separates an application into three main logical components to create projects that are scalable and extensible. 3, 9

Node.js A JavaScript runtime built on Chrome's V8 JavaScript engine [13]. 4, 5, 10, 16, 17, 26, 41

Object Relational Mapper An Object Relational Mapper is a layer between object-oriented code and a relational database such as MySQL or Postgres. The ORM provides a less complex way of accessing data in a SQL database without the need of manually writing SQL queries. i, 3, 9

response time The time it takes from user interaction until the server response is received by the client. i, iii, 4–7, 12–14, 16, 17, 19–30

REST API Representational state transfer application programming interfaces, or REST APIs or RESTful APIs, is a type of API that is structured according to the constraints of the REST architectural style [17]. The APIs uses HTTP requests to make requests to a server, and the server responds with some sort of data, often JSON. 13, 14, 30

route A defined HTTP path to get the requested data or page. 11–13

shell script A computer program designed to run on the Unix command-line. 12, 25, 26, 28

switch A network device that connects devices within networks, often Local Area Network (LAN), and forwards package to and from those devices. A switch only sends data to the intended device [4]. 11, 41

templating engine A templating engine replaces variables in a template file with real values and transforms the template into a HTML file that can be sent to a client. v, 1, 9, 25, 26, 30

Acronyms

CSV Comma-Separated Values. 12

EJS Embedded JavaScript. 9, 25, 41

IDE Integrated Development Environment. 1, 16

LAN Local Area Network. 2, 11

ms Milliseconds. 17, 19–22, 24–26

MVC Model View Controller. 9, 11, 15–17, 19

ORM Object Relational Mapper. i, iv, v, 9, 10, 24–27, 30, 42

Chapter 1

Introduction

1.1 Background

More and more time is spent on the internet in modern times, as can be seen in [10] where 87% of the survey participants said that they had used the internet daily for the past three months before the survey was conducted. According to Eurostat [11], some of the most common activities of internet users in the European Union were sending and receiving emails, finding information, reading online news and participating in social networks 2017. In the same year, 8 out of 10 users in the EU searched online for information about goods and services. This led to more companies increasing their presence on the internet by having a website or using social media [11]. The websites are used to provide different information and functionalities to their customers or business partners according to Eurostat [11]. Velocity consultancy [7] states that it is essential for companies to have a website, as it, according to them, improves credibility, can help increase sales, and improve customer experience. As shown in the Eurostat survey [11], users use the internet to gather information. By having a website, Velocity [7] says that companies can increase their sales, as potential customers can find information about the company and make the decision to use their services or products instead of competitors. Speed is essential to get users to stay on the website, as long page load times and poor response times to user actions create a bad user experience according to Cloudflare [6]. Cloudflare [6] also states that waiting for the content to load becomes frustrating for users and can cause them to leave the site or the application altogether, since users are likely to close the window or click away if a page does not load within a few seconds. This shows the importance of having high-performance web pages. This thesis will compare one state-of-practice web server framework, Express.js, with one less common web framework, Ktor, to see how they compare in response time. Response time is vital for web pages because this is the time it takes from the user interaction until the server results arrive in the user's browser again. If the response time is high, the risk that users leave the site increases.

We decided to compare Express.js and Ktor with a comparative experiment. Express.js was chosen as it is one of the most popular web frameworks, while Ktor can be the next rising star. The creators of Express.js describe it as "... a minimal and flexible Node.js web application framework that provides a robust set of features for web and mobile applications." [12]. The framework is used to write server-side JavaScript that runs on the Node.js runtime. Node.js' runtime is built on

Chrome's V8 JavaScript engine [13]. Express.js was first released in November 2010, and Node.js was initially released for Linux in 2009, according to Mozilla [31]. Ktor is a framework developed by JetBrains that is used to "... easily build connected applications – web applications, HTTP services, mobile and browser applications." [21] in the Kotlin language, also developed by JetBrains. Among the frequently asked questions about Kotlin it says that Kotlin is an open source statically typed programming language that targets the Java Virtual Machine, Android, JavaScript and Native and that Kotlin is 100% interoperable with the Java programming language, which means that you can easily call Kotlin code from Java and Java code from Kotlin [19].

1.1.1 Scope

For this thesis, we focus on the mean average response time of the selected frameworks. In addition, some data for CPU utilization and memory will be used to give some context of how much of the system resources are used. We will conduct 5 tests total where each test consists of four runs for each framework, and the runs are divided by the set of concurrent users which are 50, 100, 150, and 200. Each run takes 5 minutes to get as good average as possible without taking up too much time, since there are a total of 40 runs to be run.

As we are only looking at the response time, we do not factor in the loading of the page or the need to fetch pictures, JavaScript files, CSS files, etc. We will not provide any data about total requests made as it should not have any effects on the result. In addition, we will not test how many requests per second the frameworks can handle in total if we would send requests without any limitation.

Chapter 2

Method

2.1 Research questions

RQ1: How do the frameworks compare in response time with a given set of concurrent users?

Concurrency enables web servers to handle multiple users simultaneously, generally desired when creating a website. All modern web frameworks are built to have concurrency, but they differ in implementation and have different performances. It is essential that when the number of concurrent users increases, the server can still have a low response time, as a higher response time will increase the bounce rate [28, 38]. The question is to see how a set of 50, 100, 150, and 200 users impacts the response time of each framework.

This question will be answered by doing five tests with the previously mentioned set of concurrent users on both frameworks. The first test is to let the sets of users send request to all the 4 routes. The four other tests will be like the first but only sending requests to one of the four available routes. The tests will be made with Locust, which can send requests to one or more routes of the web server.

RQ2: How do the selected web frameworks compare in terms of average response time while serving dynamic pages?

RQ2.1: Are there any differences when using small vs. large dynamic pages?

Dynamic pages are more common these days, as they allow rapid changes in content on the site [18, 26]. The downside is that it takes time for the server to load the needed data and render the page before it can be sent to the client. Rendering will slow down the response time. This question is set to see how the frameworks compare response time when loading small and large dynamic pages (Table: 2.2).

Here we will answer the question by sending a request to both frameworks to get the dynamic pages. Again, Locust will be used to send the requests.

RQ3: How do the selected web frameworks compare in terms of average response time when serving static pages?

RQ3.1: Are there any differences when using small vs. large static pages?

Static pages were the most common in the early history of the web [26]. The server does not have to work more than getting a page from the file system and then sending it back to the requester when delivering static pages. Static pages have the disadvantage that they cannot be easily changed and that if two or more pages are using the same code, they must be written once for each page [26]. This question is set to compare how the frameworks compare in response time when serving small and large static pages (Table: 2.2).

The question will be answered by sending requests to both frameworks with Locust to get the static pages.

2.2 Literature study

To support the implementation choices and the result of our experiment, we conducted a literature study. We have looked at the most common way to structure the frameworks and how to design the test to obtain a relevant and accurate test result. Lastly, we found some data that indicate what the results could look like.

2.2.1 Search strategy

When conducting the literature study, the following types of literature were used:

1. Literature from databases such as IEEE, Google Scholar, and Diva.
2. Documentation for specific languages or frameworks.
3. Gray literature such as blogs, websites.

The following keywords were used when searching for literature:

- JavaScript
- Express.js
- Kotlin
- Ktor
- Web framework
- Web server
- Performance
- Response time
- Concurrency

2.2.2 Criteria

The following criteria were used to find relevant and reliable literature.

Databases	Only articles that are no older than five years, that are in a conference paper or that are peer-reviewed will be used. We ensure that the authors are trustworthy by checking their profession and if they have written about the subject before
Documentation	Must be about one of the frameworks.
Gray literature	Should not be older than three years, as it is not as reliable and often does not have much to back them up, if not being backed up by at least one more article that is newer than three years.
Overall	The literature has to include some comparison, test, or fact of how the frameworks or language should be set up, built, or tested to be as fair as possible when testing. It also must focus on one of Express.js, Ktor, or their respective language.

Table 2.1: Criteria to find relevant literature.

2.3 Experiment

An experiment was carried out to answer the research questions given, in which two different web servers were built as similarly as possible. Here, our course of action is described.

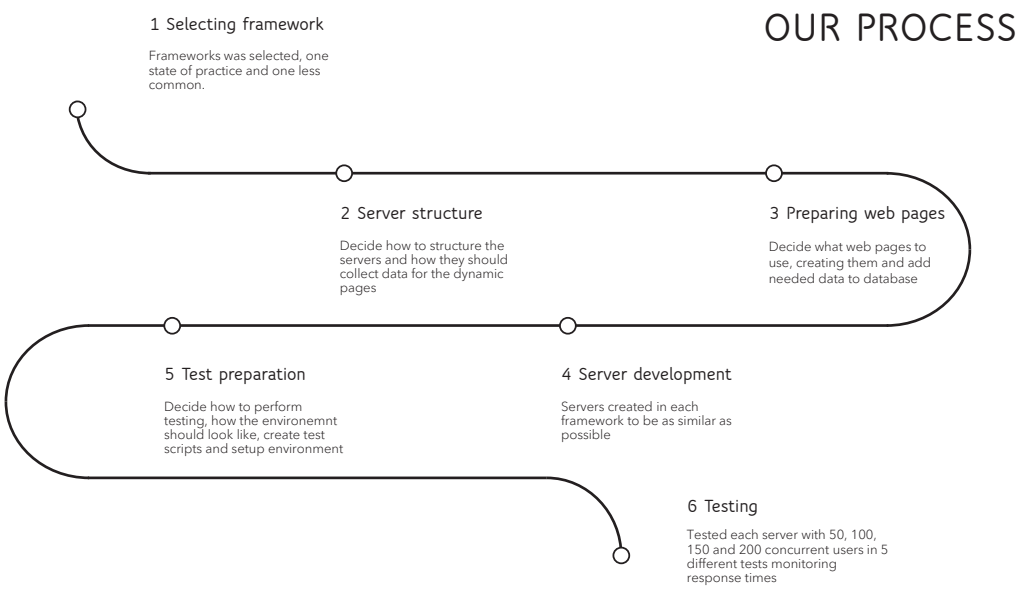

Figure 2.1: Outline of the experiment process.

2.3.1 Selecting frameworks

The two frameworks used to develop the servers were Express.js and Ktor. These frameworks were selected because the study planned to compare a state-of-practice framework with an upcoming framework that may have the potential to be popular in a few years, as there is often a lack of research on the upcoming frameworks. Since there are many languages or frameworks to choose from, two languages that seemed to be the most exciting or familiar were selected. Express.js was selected because it is a popular language that was ranked number three as the most popular framework by programmers in the 2021 Stack Overflow developer survey and number four by professional developers in the same survey [41], and we have prior knowledge developing servers in the framework. Ktor was selected because Kotlin is a growing language with great support that has real potential to be the next big thing, as it is used by large companies like Google, Netflix, Twitter, etc. [22]. Google also went so far as adopting Kotlin as the main language to use to develop Android apps in 2019 [15]. The two selected frameworks are used to handle requests from a client, but since they cannot render dynamic web pages on their own, each framework needs to be combined with a templating engine. The templating engine of choice with Express.js was Embedded JavaScript (EJS), a top rated templating engine to use when developing a web server with Express.js, which was also familiar to us. Apache Freemarker was chosen as the templating engine for Ktor since it was recommended on the Ktor web page and seemed similar to EJS.

2.3.2 Server structure

The goal was to create the two servers as identical as possible to minimize the risk of affecting performance due to the internal structure of the applications. For both servers to match, it was decided that the servers should be structured according to the Model View Controller (MVC) design pattern, which is a widespread way of structuring web applications. It was also decided that both servers would use an Object Relational Mapper (ORM) to manage database connections. The usage of ORM was decided mainly because no easy way was found to make it possible to connect Kotlin to a database directly using Kotlin libraries and create SQL queries manually. Java libraries could have been used, but we wanted to keep it as native to Kotlin as possible. The ORM used with each respective framework was Sequelize for Express.js and Exposed for Ktor.

For the dynamic pages, we decided to store the content in a Postgresql database instead of keeping the content in variables or files, since this is the most common way to store dynamic content in a real application.

2.3.3 Preparing web pages

Before the content stored in the shared database could be decided, the pages used in the tests had to be created. A open source web page template was used to save development time, and content added to the template to create a page was generated using a lorem ipsum placeholder text generator. The generated text was then used

to replace the text in the template to get as much data as we wanted. All tests used the same underlying web page but with different amounts of text for the small and large pages to match with the table below (Table: 2.2). Dynamic small and static small have the same content, and the same is true with the dynamic large and static large page, but it seems like the different sending techniques results in different sizes. This will not make any difference as we only compare dynamic large against dynamic large and static large against static large for the two frameworks. The static pages had the generated text directly in the HTML file. Dynamic pages fetched the content from the database and rendered it into the page before returning the result to the user. The size of the pages was decided by going to around 20 pages and by using the built-in network tool in Chrome we looked on the size of the HTML files and then we made an approximate of what was a small and large page. We found that a small page is about 6.5 KB while a large page is around 21 KB.

In the table below, it can be seen how significant the different pages were and how the content was loaded.

Page	Size	Rendering
Dynamic large	21.7 KB	data fetched from database and rendered into the final page
Dynamic small	6.25 KB	data fetched from database and rendered into the final page
Static large	20.2 KB	complete page loaded from disk and returned
Static small	5.88 KB	complete page loaded from disk and returned

Table 2.2: Web pages that will be delivered under testing.

The database used in the experiment was a Postgres SQL database accessed by the ORM:s in the respective server. The same database was used for both servers to ensure that the configuration and data were the same. Using the same database ensures that the only difference related to the database will be the ORMs and their internal implementation. For the Express.js server, an ORM called Sequelize was used, as it was the first ORM that appeared when searching for "Express.js ORM" and it is also popular with more than 1.25 million weekly downloads from the Node.js package manager, npm, [33] and 1.4 million weekly downloads from the Yarn package manager [50]. For the Ktor server, Exposed was used as it was one of the first ORM that was found when searching "Ktor ORM" and is developed by JetBrains that are the developers of both Kotlin and Ktor. The data for the dynamic pages were saved in two different tables in the database connected with relations, so the database would need to perform some processing before returning the data. This was decided because data for dynamic pages often need to be collected from multiple database tables in real applications.

2.3.4 Server development

Both servers have four different routes that could be accessed by clients: /static/small, /static/large, /dynamic/small and /dynamic/large. As the routes implies, the /static route serves both large and small static pages, and the /dynamic route serves the small and large dynamic pages. Since the servers are structured according to the MVC design pattern, requests for the dynamic pages are received by the Controller, asking the Model to get the needed data from the database. The data are then used to render the view into the final page that is returned to the client. For static files, the complete file is read from the disk and returned to the client when the Controller gets the request for a certain static page.

2.3.5 Test preparation

With the servers developed, it is time to test them. It was decided to use Locust [27] to test the servers. Locust is a load testing tool that was selected for its ease of use as was found in the literature study (Chapter: 4). In addition, we had prior knowledge of using the tool. It is easy to create virtual concurrent users with Locust, as it is only a class in Python that needs to be defined that can be given tasks (methods) to perform. Normally, users are made to create requests to a server to simulate a concurrent load.

It was also decided that the servers should run individually on the same computer, with as few programs running in the background as possible so that the other server or programs would not affect the test result. Requests should be sent from another computer running Locust over LAN using a wired connection through a network switch. This setup would ensure that the servers are tested with the least possible interference. If the same computer runs the servers and sends requests, both Locust and the servers would require hardware usage simultaneously, possibly hurting the server's performance. It would also not be realistic to run both the server and Locust on the same machine, as no real requests would have been sent over the network. The drawback of sending requests over the network is that it can affect the result if the network equipment does not have enough capacity to handle all traffic. By only sending the requests over the switch, the risk of interference is minimized, since only the two computers were connected. In addition, the switch can forward network packages directly from one computer to the other without passing through the router that handles packages from other devices or the internet, minimizing interference. Having the data only going through the switch is important, since we want to test the web server performance and not how well our network handles the data.

With the setup used for the testing, the network should not affect the results, since high-performance equipment was used that is capable of handling up to 1 Gbps of traffic for all devices (Table: A.2).

The computer running the servers had an Intel Core i3 processor with 4 hyper-threaded cores with 16 GB of memory (Table: A.1). The computer had a fresh install of the Ubuntu-based operating system Pop!_OS [42] to ensure that as few

programs were running as possible. If the original OS used on that machine had been used, many programs would run in the background, possibly affecting the test results.

A shell script was created to collect server metrics during the testing. The script ran the "top" command each second during the test and then filtered the data to only collect the CPU and Memory metrics for the tested server and output it in a Comma-Separated Values (CSV) file. Locust-generated metrics were used to obtain information about response times from the server to the client.

2.3.6 Testing

A total of five test cases were created. There were four tests, testing only one route at a time, and the last test randomly sent requests to all available routes. Each test case was run against the Express.js and Ktor servers, respectively, for 5 minutes. Each test was run four times to obtain metrics on how the servers handle concurrency, each time with a different number of simulated concurrent users, 50, 100, 150, and 200, per test. The set of users was decided as pre-tests showed that the test system could handle those amounts. The test of randomly sending requests to all endpoints was created since it would be a more typical workload mixing different pages since not all pages on a real server will have the same content and all users request the same page. This test was designed to have the same probability of accessing each endpoint. Otherwise, it is possible to set weights in Locust, so one endpoint is more likely to be called than another. However, in this test, it is the same probability to simulate different types of pages with different sizes being accessed.

The Locust tests were designed to send one request and then wait 2 seconds before sending the next request to simulate a real user action as they often do not go to the next page directly after the page loaded, but at the same time, we did not want it to wait too long, as we are trying to stress test the frameworks. Both Locust and the shell script were started simultaneously when the tests were performed. When Locust starts, it has 0 users to send requests, so it was decided to add 10% of the total number of users each second until the desired number of users was reached. For example, this would result in adding 5 users each second until 50 was reached for the test with 50 concurrent users. Adding 10% of the total number of users until the desired number of users was reached would quickly reach the desired number of users but without overloading the server and the client by instantly creating all users.

During the tests, we used the metrics generated by Locust to obtain statistics on response times and to see if there were any errors. We looked at the mean average response time that was calculated by Locust. The response time is interesting because it shows how well the server handles the load on the server for a given set of users. If the server cannot handle all requests, the response time will increase. A shell script was used to collect CPU and memory metrics from the server to see how different amounts of requests affect the hardware usage of the servers.

Chapter 3

Related work

There is some research made that relates to our research on different web frameworks in the state-of-practice programming languages for web development, such as Express.js (JavaScript), Flask (Python), .Net (C#), etc. While searching, we had a hard time finding any papers on the upcoming languages web frameworks such as Ktor (Kotlin), Amber (Crystal), dotweb (Golang), etc. The existing research that we found generally does not take response times into account as performance metrics and is usually not compared to the web frameworks of up-and-coming languages. They are also generally not taking up the performance of delivering static and dynamic pages. Instead, they are frequently using an API that delivers raw data and are looking at the performance with regard to CPU utilization and memory usage.

In a study by Sai Sri Nandan Challapalli et al. [2], they presented a comparative analysis of the two web frameworks, Express.js and Flask. When it comes to the method, they conducted two tests in which they compared the performance in the form of response time and requests per second of the two web frameworks. Locust and Autocannon were used to separate the two tests. Locust was used to focus on requests per second and Autocannon was used to obtain the response time of the two servers. They concluded that Express.js is better than Flask in all tests and that Express.js is better at handling multiple concurrent users sending requests than Flask. The experiments were less comprehensive than those made in this paper, as they only ran the test on each framework for 30 seconds with ten concurrent users and they only had one route that simply returned "Hello World". The result shows that Express.js is better than Flask, another popular framework. This gives an indication that Express.js should be very optimized and perform well.

In another article by Hardeep Kaur Dhalla [9], a comparative experiment is executed between the two web frameworks Spring Boot (Java) and the.net core. Two similar REST APIs were built for each framework. In the experiment, he was looking for the response time for each of the main four HTTP requests (GET, POST, PUT, and DELETE); as the number of concurrent users increased, he used JMeter to conduct the load on each HTTP request of the two servers. The tests were carried out three times, each time on a different computer with a different processor and amount of memory (RAM) while having the same operating system (Windows 10) to see if it had any impact on the result. His work differs from the experiment in this paper mainly because we use different frameworks altogether. He uses a different tool, uses a REST API instead of serving a web page, and is even testing on several machines.

There is also a study by Lakshmi Prasanna Chitra et al. [3], where they compare Express.js with .Net. In their experiment, they built an REST API in each framework. The APIs were then tested to see which had the best performance in terms of throughput in bytes per minute. The tests were divided into throughput, throughput in IO-intensive situations, and throughput in CPU-intensive situations. JMeter was used with a range of 10 to 1000 concurrent users on each part. Some differences exist from our paper, mainly that our experiment is focused on the response time instead of the throughput. They are also comparing Express.js against .Net, whereas we are comparing against Ktor.

In a paper by Magnus Greiff et al. [16], they compare the two frameworks, Symfony (PHP) and Express.js. They discuss how the setup process differs in the two frameworks and what features are common and different. Two comparative experiments were also performed to see the difference in concurrency and high amount of data transfers. A comparison between CPU utilization was then made for each experiment. Bombardier was used to test concurrency, and Wget (a gnu project tool to retrieve content and files from web servers [14]) was used to fetch a large amount of data from the servers. This paper focuses on the CPU utilization of the two servers, while we focus more on the response time. The paper also compares Express.js with Symfony, while we compare against Ktor. However, it is still strongly related to our paper, as both compare two web frameworks' performance and use Express.js as one of the frameworks. The result on this paper shows that Express.js is also better than Symfony on all tests but especially on higher amounts of requests.

Chapter 4

Literature study

4.1 Literature

The following table lists all the literature that has been used during the literature study.

Creating a new Ktor project [25]	Official documents for installing and getting started with Ktor. Here, it is shown how to create a new Ktor project with the help of IntelliJ IDEA. It also shows how to write a "Hello World!" program.
Kotlin with Ktor [1]	An article about setting up a Ktor server from scratch.
Application structure [23]	Official documentation on how a Ktor project should be structured. It draws connections with how its structure is very similar to how other frameworks like ASP.NET and Ruby on Rails are doing their MVC structures with minor differences.
Installing Express.js [36]	Documentation from the official Express.js organization, with simple-to-follow instructions on how to install Express.js.
Hello world, Express.js [35]	Code example on writing a "Hello world!" application by the official documentation.
How To Setup An Express.js Server [8]	An article explains how the Express.js setup process works and gives some arguments as to why it is a popular framework.
Developer Survey 2021 [41]	A survey by stack overflow where over 80,000 developers participated. The article shows statistics on the popularity and usage of programming languages, frameworks, development environments, etcetera.
Choosing the right load testing tool [30]	This article compares the load testing tools JMeter, Locust and Goos.
Load Testing Alternatives [39]	Comparison and pros versus cons of the three testing tools Locust, Selenium and Gatling.
Express.js with MVC [29]	Article showing how to structure Express.js to use MVC.

Benefit of using MVC [46]	Explains the parts of MVC, how they work and what it is good at.
Why MVC? [40]	Walks through the main parts of MVC and its pros and cons.
Performance comparison between Node.js and Python [2]	A comparative experiment in which response time, requests per second, and total requests are taken into consideration.
Performance comparison between Node.js and .NET [3]	A comparative experiment in which the throughput is compared in bytes per second for different scenarios in Express.js and .Net.
Web framework benchmarks [45]	Benchmarking different web frameworks to compare how many requests per second each framework can handle when serving plain text to clients.
Node.js Event Loop [34]	An explanation of why not to block the event loop in the official documentation of Node.js. Also, an explanation of how concurrency works in Node.js.
Coroutines basics [24]	Official documentation about coroutines in Kotlin. A way to create concurrency in Kotlin.

Table 4.1: Literature used in the literature study.

4.2 Literature comparison

In references [1, 23, 25] they are showing how a Ktor application is set up. In [23] they are mostly going into detail about how the application should be structured and that it is a MVC-like structure. While in [1, 25] they show an example code of how to get up and running from scratch. However, there is a difference where [25] shows how to get started with the help of IntelliJ IDEA (an IDE by JetBrains [20]). In contrast, [1] shows that it is as easy to setup the server without the help of IntelliJ IDEA.

The following articles show the installation process and simple hello world example from the official Express.js documents [35,36]. In article [8] a very similar installation and setup process is shown. It also takes into account that Node.js has a large community, which is also confirmed in [41], and that it could be beneficial when learning, as it is easy to find help when you are having problems.

By wrapping an Express.js server with an MVC structure, you can take the application to a high level as it makes it easier to distinguish the logic from the pages and fetching of the data, which makes the code more maintainable [29]. Article [46] agrees that an MVC structure goes along with JavaScript when using Asynchronous Method Invocation. Asynchronous Method Invocation allows the developer to build a fast-loading web application [46]

MVC have three different layers: Model, Views and Controller [29, 40, 46]. The

Model is the data layer and has the task of fetching and storing data. The View is the interface layer that displays the current data to the user. The Controller is the logic layer that handles requests from the user by connecting the Model to store or fetch data and then giving back the View to the user. Both [40] and [46] give the advantages of using MVC. Some of the advantages that they agree on are ease of modification, fast development, high cohesion, and the ability to have multiple Views for a Model. While [46] lists more advantages, such as support for test-driven development, ease of planning and maintenance, [40] addresses some disadvantages, such as challenging code navigation and the need to maintain consistency between MVC layers when changing in one of the layers.

Both literature [2, 3] makes a comparison of Express.js performance. In both papers, Express.js is found to be better than the other framework. Where they differ is what they have tested, [2] shows that Express.js can handle many requests in a short time and that the response time is much better than Flask. In [3], it was found that Express.js is great at IO operations, while CPU-intensive work was worse than .NET.

Multiple web frameworks are compared in terms of the number of requests they can handle per second in [45]. They compare frameworks when serving plain text to clients. It is clear from their data that Ktor should be able to handle more requests than Express.js. Ktor placed 153 with about 760 000 requests per second handled in their testing, while Express.js placed 307 with 138 000 requests per second handled. In the data, they also show the latency of the servers, showing that Ktor had a latency of 7.2 Milliseconds (ms), while Express.js had a latency of 74.7 ms. Their tests have been performed using server-grade hardware with 14 core hyperthreaded CPUs with 32 GB of RAM and 10 GBit/s network connections [43]. They also use three different servers, one application server for the framework server, one for the database, and one for sending requests to the server. For the plain text tests, they tested with 256, 1,024, 4,096, and 16,384 concurrent users [44]. According to [37], Express.js is not suitable for creating a highly performant application, also shown by [45].

Both Node.js and Kotlin have ways to create concurrent execution [24, 34]. Coroutines are the way to create concurrency in Kotlin. A coroutine is very similar to a thread, since it executes a block of code [24]. The difference between a thread and a coroutine is that the coroutine can run on any CPU thread, while a logical thread is bound to a particular CPU thread. Node.js runs an event loop that takes care of the main execution. When an I/O operation is needed, it assigns the task to a thread in the Worker group that is made up of a few threads [24]. As Node.js uses a small number of threads, it can handle many concurrent clients at the same time, as it will not need to spend time on the overhead that many threads bring [34]. "...Node.js is fast when the work associated with each client at any given time is 'small'." [34].

The focus of [30] and [39] is to find a good load testing application for web applications. Both find that Locust is a great tool to use. The main difference between the two articles (with regard to Locust) is that [39] sets Locust as the main choice as it can create many concurrent users spread over several cores for a better test,

while [30] does not set it as the best but still recommends it for its great monitoring.

Chapter 5

Results

This segment shows the result of the experiment and the literature study.

The literature study found that the frameworks have a straightforward setup process and that both are recommended to use the MVC structure for fast development. Some code examples were found, which were considered and used as reference when developing the applications. Locust was found to be an excellent tool to create concurrent users and retrieve data and metrics. One of the literature also showed that Ktor should perform better than Express.js, while another said that Express.js is very efficient when having a smaller amount of work to do.

It can be seen that Express.js has a better response time than Ktor on all sets of virtual users when looking at the response times in the concurrency test (Figure: 5.1). The difference in response time is about six and ten ms in all cases. Express.js indicates a linear growth, whereas Ktor seems to indicate an exponential growth in response time, but more tests with higher and lower amounts of concurrent users are needed to prove this.

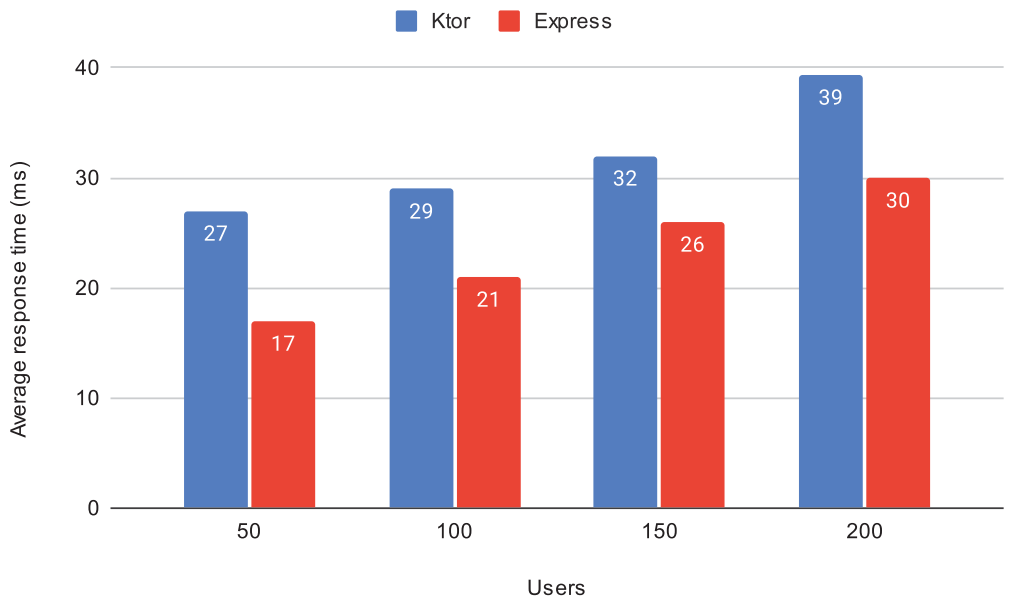

Figure 5.1: Average response time per request when sending requests to all routes. (Lower is better)

When comparing the frameworks for the small dynamic page test (Figure: 5.2), it can be seen that Ktor has around three times the response time than Express.js. Ktor has a response time of 35 ms when having a load of 50 users and ranges up to 71 ms when having 200 users. Express.js ranges from 13 to 19 ms in the same range.

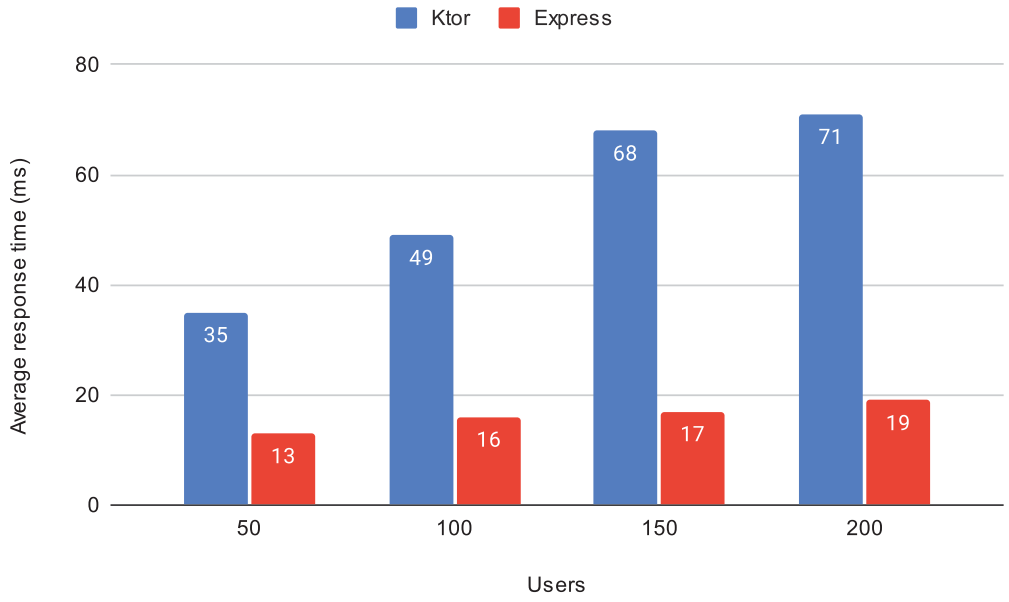

Figure 5.2: Average response time per request when testing small dynamic pages. (Lower is better)

It can be seen that Express.js keeps the response time low when looking at the response times for the large dynamic page test (Figure: 5.3). Ktor can be seen decreasing in response time when going from 50 to 100 users, but then having a large increase when looking at 150 users.

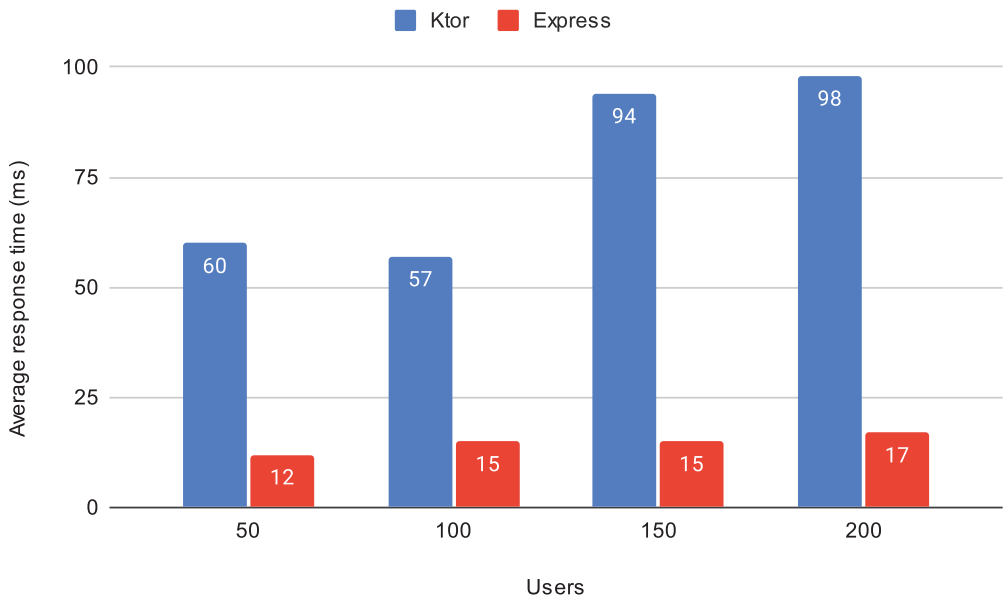

Figure 5.3: Average response time per request when testing large dynamic pages. (Lower is better)

In the small static page test (Figure: 5.4), Ktor is mostly slower than Express.js, but when having 50 concurrent users, it is a few ms faster. It can also be seen that the response time in Ktor grows faster than Express.js. Express.js decreases by one ms when going from 50 to 100 concurrent users.

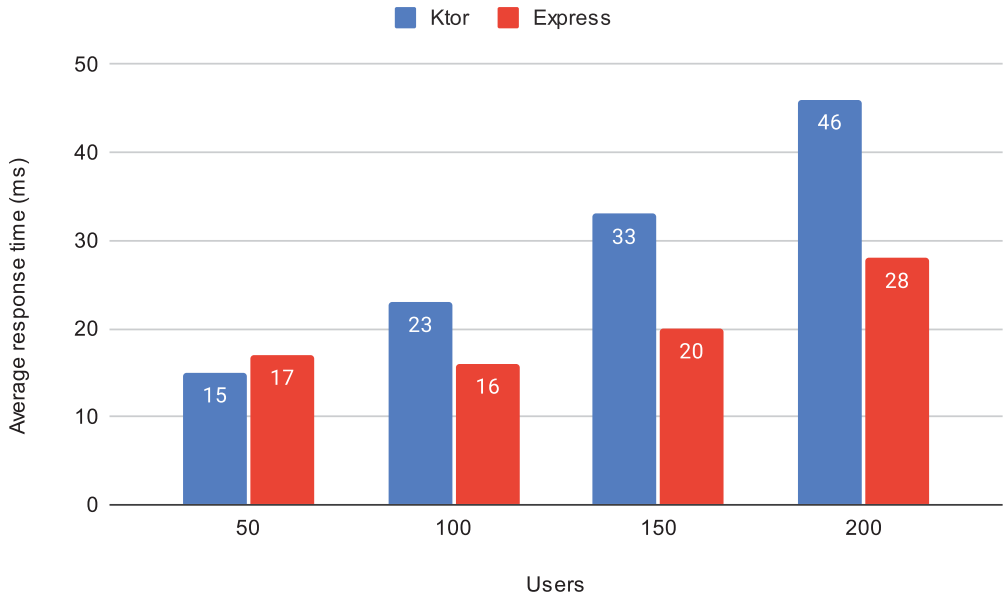

Figure 5.4: Average response time per request when testing small static pages. (Lower is better)

For the large static page test (Figure: 5.5), Express.js is found to have nearly exactly the same response time as in the small static page test (Figure: 5.4) where only 50 and 100 users have a slight difference of 2 ms, respectively, 1 ms. Ktor also has nearly the same result as the previously mentioned test when it comes to 50 and 100 users, but adds 10 ms in both the 150 and 200 user tests.

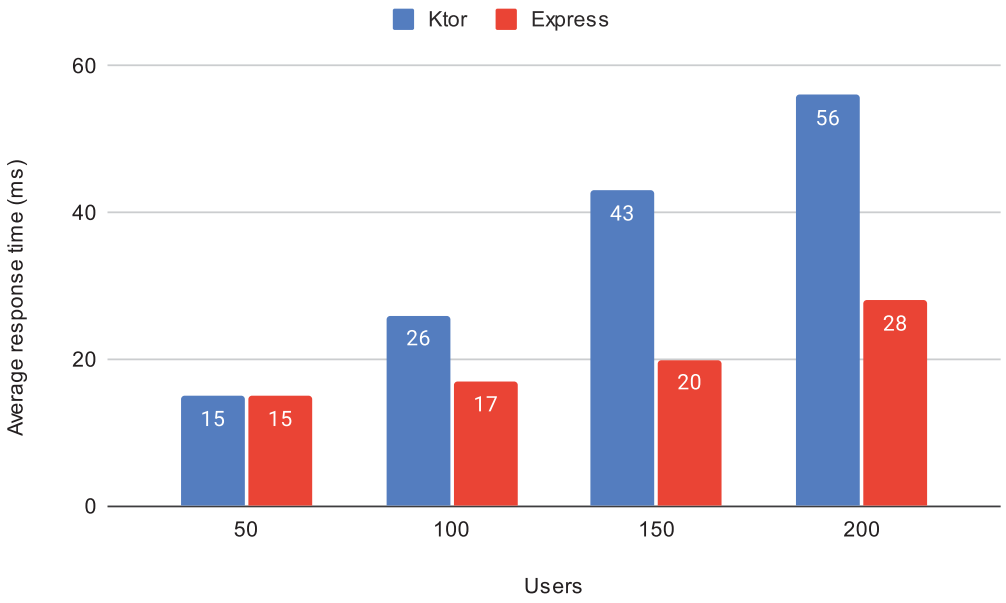

Figure 5.5: Average response time per request when testing large static pages. (Lower is better)

RQ1: *How do the frameworks compare in response time with a given set of concur- rent users?*

For **RQ1**, it can be seen that Express.js had a better response time than Ktor, with some exceptions where Ktor has a few milliseconds lower or the exact same response time. When looking at the different sets of users, it shows that Ktor has a higher growth of response time when increasing the number of users than Express.js has.

RQ2: *How do the selected web frameworks compare in terms of average response time while serving dynamic pages?*

RQ2.1: Are there any differences when using small vs. large dynamic pages?

The result of **RQ2** is that the average response times for dynamic pages clearly shows that Ktor is around three times higher than Express.js. When serving small pages, Express.js had a higher response time than when serving large pages. Ktor got the opposite effect, where the response time of large pages increased.

RQ3: *How do the selected web frameworks compare in terms of average response*

time when serving static pages?

RQ3.1: Are there any differences when using small vs. large static pages?

As a response to **RQ3**, it can be concluded that Express.js shows response times around 150% better than Ktor when serving static pages for 100 to 200 users. For the static test with 50 users, Ktor is faster or has the same response time as Express.js. Furthermore, the response times for Express.js does not seem to change when serving small or large static pages. Ktor, on the other hand, has a significant incline when serving a larger number of users.

Chapter 6

Discussion and Analysis

In this part, the experiment and the result are discussed and analyzed. We will discuss and analyze how the result could be interpreted and what could have affected it.

It is very clear from the experiment that Express.js has a better response time than Ktor in almost all tests. However, in the literature study, it was found that Ktor should be faster than Express.js in latency and requests per second. Even if we do not look at the latency or the request per second as Tech empower does in [45], it could have an indication of which has the better response time, as the latency is part of the response time. Request per second could indicate how quickly a request is processed so that the next one can be processed. There are also differences from our experiment and inconsistency in their benchmarking. For example, they did not use an ORM for Ktor, whereas they did use an ORM with Express.js. The fact that they differ in implementation could make the comparison unfair and may have given Ktor an advantage. On the other hand, it was also found in the literature study that Express.js can be very effective when the workload for each contemporary work is small. This is because Express.js always has a small group of active threads to assign work to, meaning that there will be no overhead of creating new threads before the processing of the request can start. Moreover, if the workload for each thread is small, the task will complete quickly, and thereby the thread can quickly process a new task.

When serving small static pages to 50 users, it was found that Ktor had a lower response time than Express.js (Figure: 5.4). Therefore, Ktor's better response time could indicate that it is better at serving static pages at 50 or a smaller number of users. The static large-page test further shows this, as Express.js and Ktor have the same response time at 50 users (Figure: 5.5). The reason could also be that when doing the 50 user test for small static, the request from the Express.js server did not get as much CPU time as it did for the other tests. As can be seen in the next test with 100 users, it was one ms faster. It can also be seen that Ktor has a lower CPU utilization for the static tests (Figures: A.7, A.9), which will be discussed below.

An ORM was used for each framework to handle the connection to the database. This could have impacted the results, as each framework has an ORM that is made for the respective language, and the implementations are most certainly not the same. So, the SQL code to the databases could be a bit different, or one of the implementations could be faster at creating the code than the other. Again, this could have affected the dynamic page results, as the dynamic part was to fetch data

from the database made through the ORM. After further investigation of the ORMs, we found that the ORM (Sequelize) for Express.js only took 3 ms for dynamic large pages, while Ktors ORM (Exposed) took 106 ms for the same page (Table: 6.1). It was also found that Ktor makes two requests to the database (Listing: A.2), while Express.js only makes one request (Listing: A.1). This could also be the reason why the ORM for Ktor takes longer time, since it needs to make two connections to the database. When the second query is performed, it may take a while before the database can process it, even if it is unlikely to happen in our test cases. Given the huge difference in the time it takes for the ORMs to finish, it may be possible that Sequelize caches the result from the database, while Exposed for Ktor does not. It may also be the database that caches the result of the query from Express.js since it is a JOIN operation that combines the data from two tables.

In addition to the ORM it could also be that the FreeMarker templating engine used by Ktor is less optimized than EJS used by Express.js. Our investigation shows that for the first request, EJS takes 10 ms to generate the page, while Freemarker takes 158 ms. But after a few requests, both templating engines take about 1 ms to create the complete HTML file and send it to the requester, which can be seen as an indication that it should not have any impact on the overall result in our tests (Table: 6.1). Caching could also be involved here, as the templating engines only have long execution times the first few times.

The investigations were not made on the same machine as the main tests (Table: A.4), but the application setup was the same, so it should still indicate that the ORM of Ktor is significantly slower than the one in Express.js.

	ORM (ms)	Templating (ms)
Express.js	3	1
Ktor	105	1

Table 6.1: Average execution time from investigation of templating engines and ORM

Locust and the shell script were started simultaneously to obtain as much data and accurate data as possible when the tests were started. Unfortunately, this might have caused the server to get a high response time (800 to 1400 ms) in the first few seconds of some of the tests. As a result, part of the test showing inaccurate data could cause some of the tests to have a higher average response time than they should have. However, the tests ran for five minutes and a few thousand total requests were made, so the high response time at the beginning of the test should not have much impact. In our investigation after the tests, we could see that both Express.js and Ktor initially had high execution times for the ORMs (Table: 6.2), but that after a few requests the time would decrease and settle (Table: 6.1). The cause of the initially high response times for the ORM may be that the system does not think the application needs that much CPU time, as it is not doing much before the first requests come in. This means that it will take longer before the CPU starts to process the first request, leading to high response times.

	ORM (ms)	Templating (ms)
Express.js	234	10
Ktor	979	158

Table 6.2: First execution time from investigation of templating engines and ORM

In the data obtained by the shell script that collects CPU and memory usage of the application, it can be seen that Ktor has less CPU utilization than Express.js when serving static pages (Figure: A.9), but the memory usage is around three times as high for Ktor (Figure: A.10). Ktor still has a longer response time than Express.js for all test cases except for the tests with 50 users. This may indicate that Ktor just isn't as effective as Express.js or that the implementation for the Ktor server is ineffective, resulting in more processing.

When serving dynamic pages, Ktor's CPU utilization is approximately three times as high on average for small pages and about three and a half times as high for large dynamic pages compared to Express.js (Figure: A.5), but it still shows about three times as long response time for small dynamic pages and 5,2 as long for dynamic large. Express.js keeps its memory usage low while Ktor more than doubles its usage from the static test (Figure: A.6). This may, in addition to what has been discussed earlier, indicate that the overhead of managing the coroutines in Kotlin increases the response times.

Since JavaScript was made in 1995 [47] it has gotten a large community [41]. The release of Node.js in 2009 made it possible to go full stack with just JavaScript and some HTML, this has made the community grow even larger. Therefore, it is easy to find documentation and other resources about both JavaScript and Express.js as was concluded in the literature review (Chapter: 4). In contrast, Kotlin is still relatively new and has a much smaller community [41]. The authors of this paper found it especially difficult to find documentation and resources when developing the Ktor server, which could be the fact that Ktor has an even smaller community.

Chapter 7

Validity threats

Inefficiently written software

As we are not experts in either of the two frameworks or languages, the servers probably do not have the best implementation, which could lead to less performance than expected.

Different sending techniques for static files

When sending static files from the Ktor server, it goes through the same function as dynamic pages and therefore may take some time to find if the page needs to render something before being sent. This is not how the Express.js server sends the static files. Instead, it sends the file directly without going through any checks.

Hardware

The experiment was carried out in a low-tier personal computer setup that is not as good or effective as the server hardware on which the web servers should be deployed. Therefore, the result could be different from what it would actually have been when deployed on a dedicated server.

ORM

As the ORM for Ktor appears to have a more significant impact on response time than Express.js, it could be seen as an unfair comparison.

Caching

It could be that the frameworks cache the pages, especially those created dynamically, to keep the response time low and therefore give an inaccurate result. It may also be that one framework caches the result, while the other does not. In addition, it may be that the database caches the result of the JOIN made by the ORM used for the Express.js server, but not the results of the two separate queries made by the ORM used in Ktor.

Human error

All tests have been started manually and may not be started at the same time. It would have been better if the shell script and the Locust tests had been started and stopped automatically. A large amount of data has been handled, so it could be that some data could be wrong or mismatched.

High response times on start

When starting the tests, we sometimes got high spikes on the response times. As talked about in "Discussion and Analysis", this could have to do with the program having a cold start. This could have resulted in the final average being slightly higher than it should.

Chapter 8

Conclusions

For a company to keep its valuable customers on its website, it needs a fast and responsive website. Response time is part of how fast the customer gets the requested page and is therefore a crucial metric to consider when building the next web server. This thesis shows that the frameworks differ in response time and that Express.js has the lowest overall response time. However, even if Express.js has a better response time, we would not say that the response time of Ktor is too high, as all response times have been way below the two seconds before many users start to leave. If a developer already knows Kotlin, Express.js may not be worth learning, as the performance of Ktor will have little to no impact on the bounce rate of the site.

The experiment clearly showed that Express.js has a lower response times than Ktor on all sets of concurrent users when using the specific tech stack in this thesis, especially for 150 and 200 concurrent users.

On average, Express.js was found to have a response time four times as low as Ktor when delivering small pages and five times as low when delivering large pages. Ktor had a significant disadvantage in the dynamic tests that could have impacted the final result.

When delivering static pages, Express.js and Ktor both had low response times at 50 users, but Ktor had 50 % longer response times on the other amounts of users.

Even if Ktor had shown an equal response time to Express.js, we would still have recommended Express.js, as we noticed that it was much harder to find resources for Ktor. In addition, Express.js has a larger community, as was found in the literature study, making it much easier to find information and help when developing. This opinion might change in the next few years if Kotlin and Ktor grows to the potential that it has.

Chapter 9

Future Work

There are many topics within the performance of web frameworks that could be explored. Below are some of the topics that could be further investigated.

The way in which the templating engine affects performance in response time could also be explored, as it takes time to put everything in place before it can be delivered to the user. There are often many different engines available in web frameworks for rendering, and they are different in how to write the templates and in performance.

Having the frameworks send SQL queries to the database without using an ORM. Sending the SQL queries will show more of how the frameworks perform, as it will not have the performance hit of the ORM.

How sending all the data and files needed to render the page affects the performance of the web frameworks is another topic that could be explored. Almost every web page has some styling, images, and JavaScript that must be sent separately.

A REST API in each framework could be performance tested. Today, many web servers are divided into two servers, a REST API server that handles everything with the database and a server that renders and sends the web pages.

A test of how many requests per second the frameworks can handle before they start throwing errors could also be done. Requests per second will show the framework's capabilities of handling many requests simultaneously.

References

[1] Baeldung. (2021, 2) Kotlin with ktor. Accessed: 2022-05-05. [Online]. Available: https://www.baeldung.com/kotlin/ktor

[2] S. S. N. Challapalli, P. Kaushik, S. Suman, B. D. Shivahare, V. Bibhu, and A. D. Gupta, "Web development and performance comparison of web development technologies in node.js and python," in *2021 International Conference on Technological Advancements and Innovations (ICTAI)*, 2021, pp. 303–307. [Online]. Available: https://ieeexplore.ieee.org/document/9673464

[3] L. P. Chitra and R. Satapathy, "Performance comparison and evaluation of node.js and traditional web server (iis)," in *2017 International Conference on Algorithms, Methodology, Models and Applications in Emerging Technologies (ICAMMAET)*, 2017, pp. 1–4. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/8186633

[4] Cloudflare. What is a network switch? Accessed: 2022-06-15. [Online]. Available: https://www.cloudflare.com/learning/network-layer/what-is-a-network-switch/

[5] ——. What is chrome v8? Accessed: 2022-06-15. [Online]. Available: https://www.cloudflare.com/learning/serverless/glossary/what-is-chrome-v8/

[6] ——. What is site speed? Accessed: 2022-05-13. [Online]. Available: https://www.cloudflare.com/learning/performance/why-site-speed-matters/

[7] V. Consultancy. (2021, 11) Importance of website: Reasons why your business needs it. Accessed: 2022-05-13. [Online]. Available: https://www.velocityconsultancy.com/importance-of-website-reasons-why-your-business-needs-it/#:~:text=A%20website%20helps%20create%20brand,business%20apart%20from%20the%20competitors.

[8] R. Cruz. (2019, 7) How to setup an express server. Accessed: 2022-05-05. [Online]. Available: https://medium.com/@ralph1786/how-to-setup-an-express-server-5fd9cd9ae073

[9] H. K. Dhalla, "A performance comparison of RESTful applications implemented in spring boot java and MS.NET core," *Journal of Physics: Conference Series*, vol. 1933, no. 1, p. 012041, jun 2021. [Online]. Available: https://doi.org/10.1088/1742-6596/1933/1/012041

[10] E. C. Eurostat, *Digital economy and society in the European Union*. Publications Office, 2018, accessed: 2022-05-13. [Online]. Available: https://data.europa.eu/doi/10.2785/436845

[11] ——, *Digital economy and society in the European Union*. Publications Office, 2018, accessed: 2022-05-13. [Online]. Available: https://data.europa.eu/doi/10.2785/436845

[12] Expressjs. [Online]. Available: https://expressjs.com/

[13] O. Foundation. Accessed: 2022-05-13. [Online]. Available: https://nodejs.org/en/

[14] D. G. (2022, 3) What is the wget command and how to use it (12 examples included). Accessed: 2022-05-04. [Online]. Available: https://www.hostinger.com/tutorials/wget-command-examples/#What_Is_the_Wget_Command

[15] Google. https://developer.android.com/kotlin/first. Accessed: 2022-05-18. [Online]. Available: Android\T1\textquoterightsKotlin-firstapproach

[16] M. Greiff and A. Johansson, "Symfony vs express: A server-side framework comparison," BLEKINGE INSTITUTE OF TECHNOLOGY, Tech. Rep., 2019. [Online]. Available: https://www.diva-portal.org/smash/record.jsf?pid=diva2%3A1327290&dswid=1261

[17] R. Hat. (2020, 5) What is a rest api? Accessed: 2022-06-13. [Online]. Available: https://www.redhat.com/en/topics/api/what-is-a-rest-api

[18] C. Herath. (2021, 4) Web development in 2021: Dynamic vs. static vs. single-page architecture. Accessed: 2022-05-04. [Online]. Available: https://betterprogramming.pub/web-development-in-2021-dynamic-vs-static-vs-single-page-architecture-399d0c3defe6

[19] JetBrains. Faq (from kotlins official webpage). Accessed: 2022-05-14. [Online]. Available: https://kotlinlang.org/docs/faq.html

[20] JetBrains. Intellij idea. Accessed: 2022-05-06. [Online]. Available: https://www.jetbrains.com/idea/

[21] JetBrains. Ktor documentation v2.0.1. [Online]. Available: https://ktor.io/docs/welcome.html

[22] JetBrains. Why teach kotlin. Accessed: 2022-05-18. [Online]. Available: https://kotlinlang.org/education/why-teach-kotlin.html

[23] ——. (2022, 5) Application structure. Accessed: 2022-05-05. [Online]. Available: https://ktor.io/docs/structuring-applications.html

[24] ——. (2022, 03) Coroutines basics. Accessed: 2022-05-14. [Online]. Available: https://kotlinlang.org/docs/coroutines-basics.html

[25] ——. (2022, 5) Creating a new ktor project. Accessed: 2022-05-05. [Online]. Available: https://ktor.io/docs/intellij-idea.html

[26] B. Kalkman. (2021, 8) Static vs. dynamic websites: What are they and which is better? Accessed: 2022-05-04. [Online]. Available: https://rocketmedia.com/blog/static-vs-dynamic-websites

[27] Locust. An open source load testing tool. Accessed: 2022-05-18. [Online]. Available: https://locust.io/

[28] D. Low. (2022, 4) 6 reasons why website speed matters & how amazon would lose $1.6 billion if it slowed down. Accessed: 2022-05-04. [Online]. Available: https://www.bitcatcha.com/blog/6-reasons-why-website-speed-matters-how-amazon-would-lose-1-6-billion-if-it-slowed-down/

[29] I. Maboud. (2020, 10) What is the mvc, creating a [node.js-express] mvc application. Accessed: 2022-05-06. [Online]. Available: https://medium.com/@ipenywis/what-is-the-mvc-creating-a-node-js-express-mvc-application-da10625a4eda

[30] L. Miles. (2021, 2) Accessed: 2022-05-05. [Online]. Available: https://www.tag1consulting.com/blog/jmeter-vs-locust-vs-goose

[31] Mozilla. Express/node introduction - where did node and express come from? Accessed: 2022-05-14. [Online]. Available: https://developer.mozilla.org/en-US/docs/Learn/Server-side/Express_Nodejs/Introduction#where_did_node_and_express_come_from

[32] ——. (2022, 05) Javascript. Accessed: 2022-06-16. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/JavaScript

[33] npm. npm - sequelize. Accessed: 2022-06-14. [Online]. Available: https://www.npmjs.com/package/sequelize

[34] OpenJS Foundation. Don't block the event loop (or the worker pool). Accessed: Accessed: 2022-05-14. [Online]. Available: https://nodejs.org/en/docs/guides/dont-block-the-event-loop/

[35] ——. Hello world example. Accessed: 2022-05-05. [Online]. Available: https://expressjs.com/en/starter/hello-world.html

[36] ——. Installing. Accessed: 2022-05-05. [Online]. Available: https://expressjs.com/en/starter/installing.html

[37] A. Padmanabhan. (2022, 02) Express.js. Accessed: 2022-05-14. [Online]. Available: https://devopedia.org/express-js

[38] Pingdom. (2018, 1) Does page load time really affect bounce rate? Accessed: 2022-05-04. [Online]. Available: https://www.pingdom.com/blog/page-load-time-really-affect-bounce-rate/

[39] M. Rashed. (2020, 11) Load testing alternatives to jmeter: Locust, selenium, or gatling? Accessed: 2022-05-05. [Online]. Available: https://www.indellient.com/blog/load-testing-alternatives-to-jmeter-locust-selenium-or-gatling/

[40] Socratic Solution. (2017, 7) Why mvc architecture? Accessed: 2022-05-06. [Online]. Available: https://medium.com/@socraticsol/why-mvc-architecture-e833e28e0c76

[41] Stack Overflow. (2021) 2021 developer survey. Accessed: 2022-06-08. [Online]. Available: https://insights.stackoverflow.com/survey/2021

[42] System76. Welcome to pop!_os. Accessed: 2022-05-18. [Online]. Available: https://pop.system76.com/

[43] Tech Empower. (2021, 02) Web framework benchmark - environment. Accessed: 2022-05-12. [Online]. Available: https://www.techempower.com/benchmarks/#section=environment&hw=ph&test=composite

[44] ——. (2021, 02) Web framework benchmark - motivation & questions. Accessed: 2022-05-12. [Online]. Available: https://www.techempower.com/benchmarks/#section=motivation&hw=ph&test=composite

[45] ——. (2021, 02) Web framework benchmark - round 20. Accessed: 2022-05-11. [Online]. Available: https://www.techempower.com/benchmarks/#section=data-r20&hw=ph&test=plaintext

[46] A. Verma. (2020, 7) Benefit of using mvc. Accessed: 2022-05-06. [Online]. Available: https://www.geeksforgeeks.org/benefit-of-using-mvc/

[47] W3schools. Javascript history. Accessed: 2022-06-17. [Online]. Available: https://www.w3schools.com/js/js_history.asp

[48] ——. What is full stack? Accessed: 2022-06-17. [Online]. Available: https://www.w3schools.com/whatis/whatis_fullstack.asp

[49] J. Warner. top - display linux processes. Accessed: 2022-06-15. [Online]. Available: https://manpages.ubuntu.com/manpages/xenial/man1/top.1.html

[50] Yarn. Yarn package manager - sequelize. Accessed: 2022-06-14. [Online]. Available: https://yarnpkg.com/package/sequelize

Appendix A

Supplemental Information

Links to the code used in the experiment.

Express.js application

https://github.com/Hawhk/Examensarbete/tree/main/Express

Ktor application

https://github.com/Hawhk/Examensarbete/tree/main/ktor

Locust files

https://github.com/Hawhk/Examensarbete/tree/main/tests

Table creation and inserts for database

https://github.com/Hawhk/Examensarbete/blob/main/database.sql

Here we have some of the data in the form of graphs that were extracted in the tests that could have an impact on how the result could be interpreted.

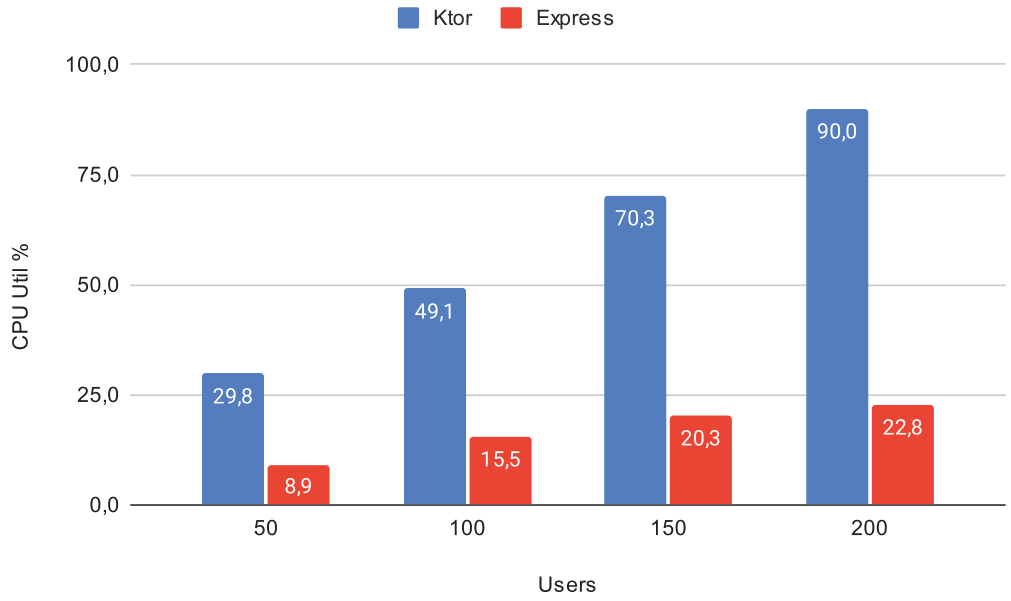

Figure A.1: CPU utilization for the concurrent tests on the servers.

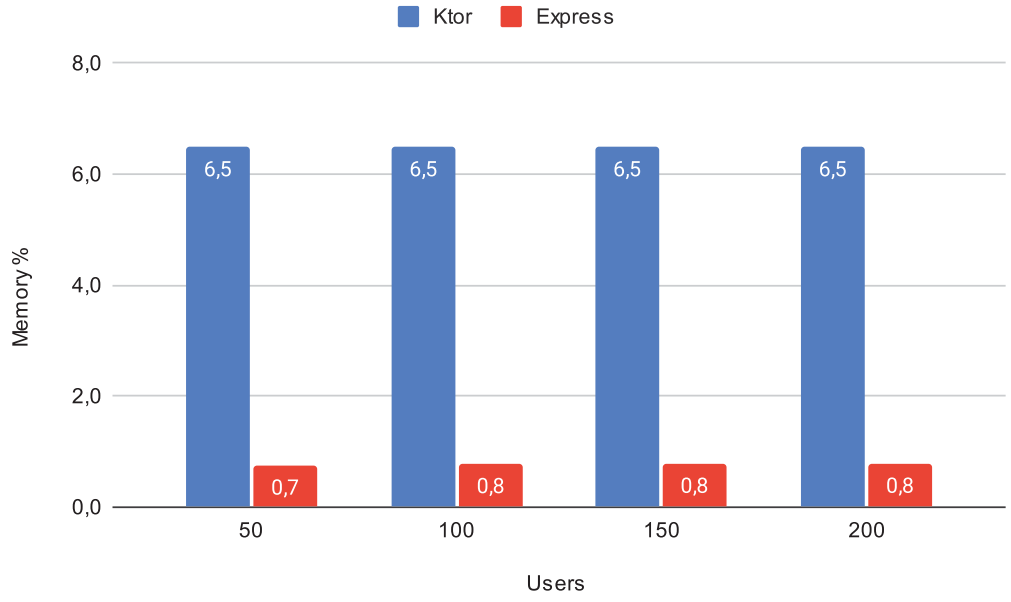

Figure A.2: Memory usage for the concurrent tests on the servers.

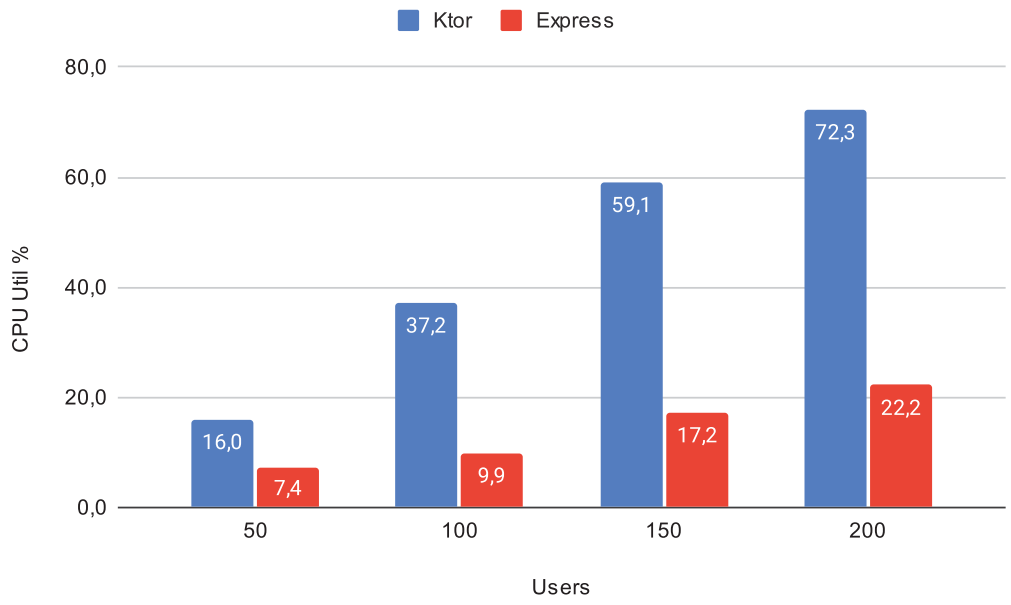

Figure A.3: CPU utilization for the dynamic small page tests on the servers.

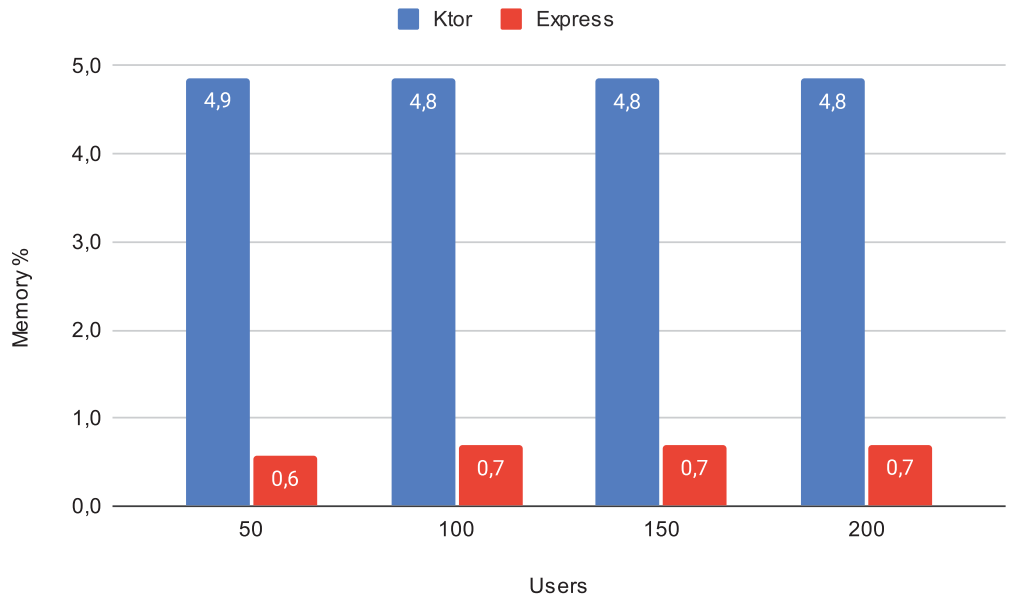

Figure A.4: Memory usage for the dynamic small page tests on the servers.

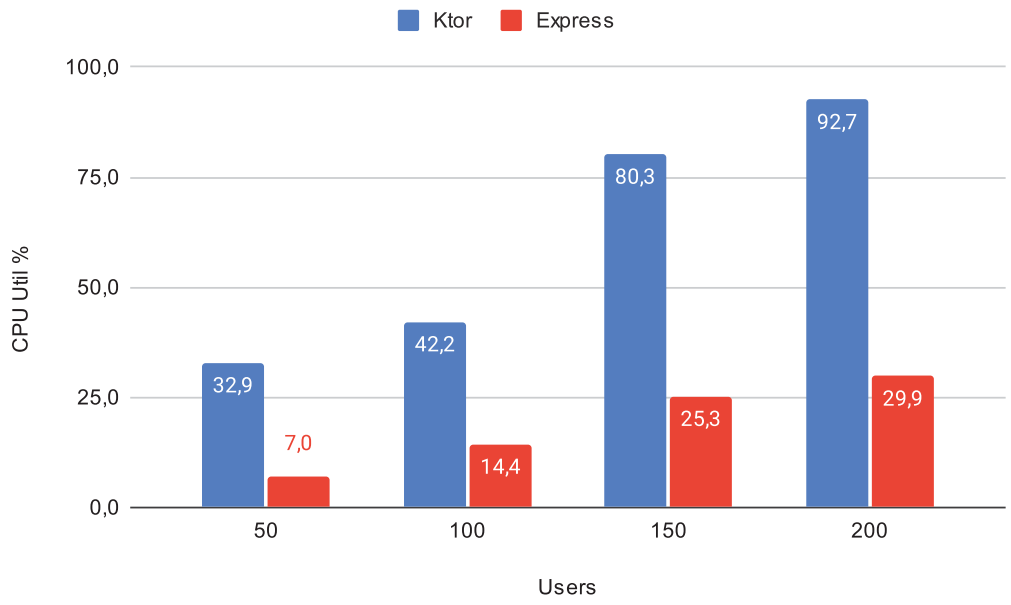

Figure A.5: CPU utilization for the dynamic large page tests on the servers.

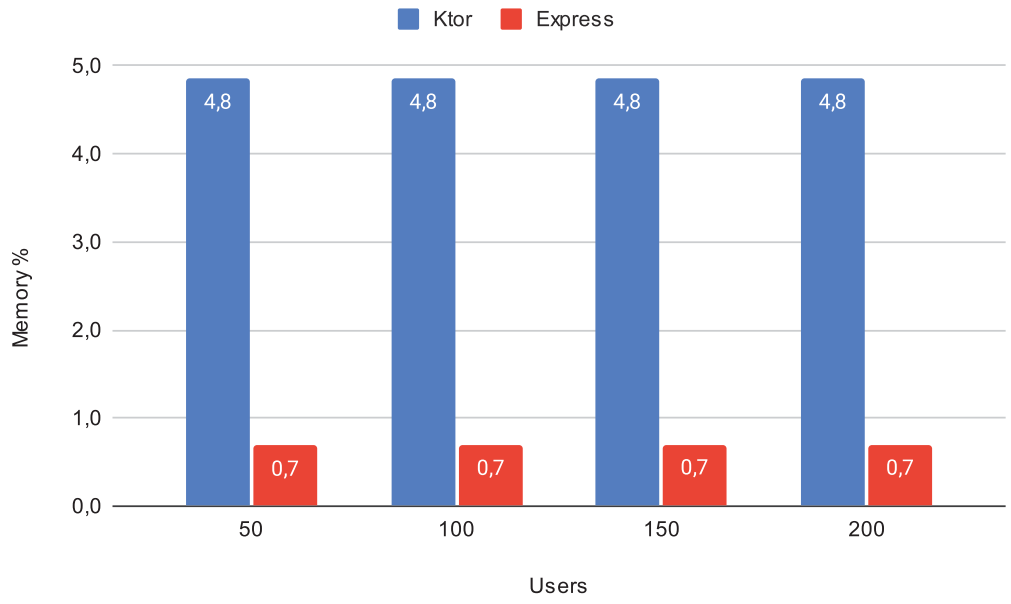

Figure A.6: Memory usage for the dynamic large pages tests on the servers.

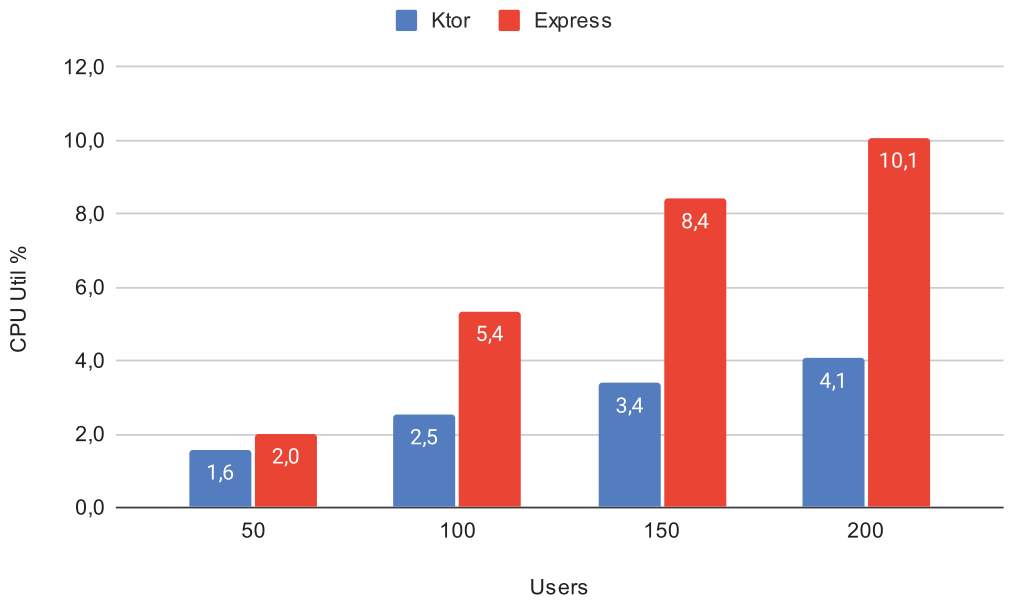

Figure A.7: CPU utilization for the static small pages tests on the servers.

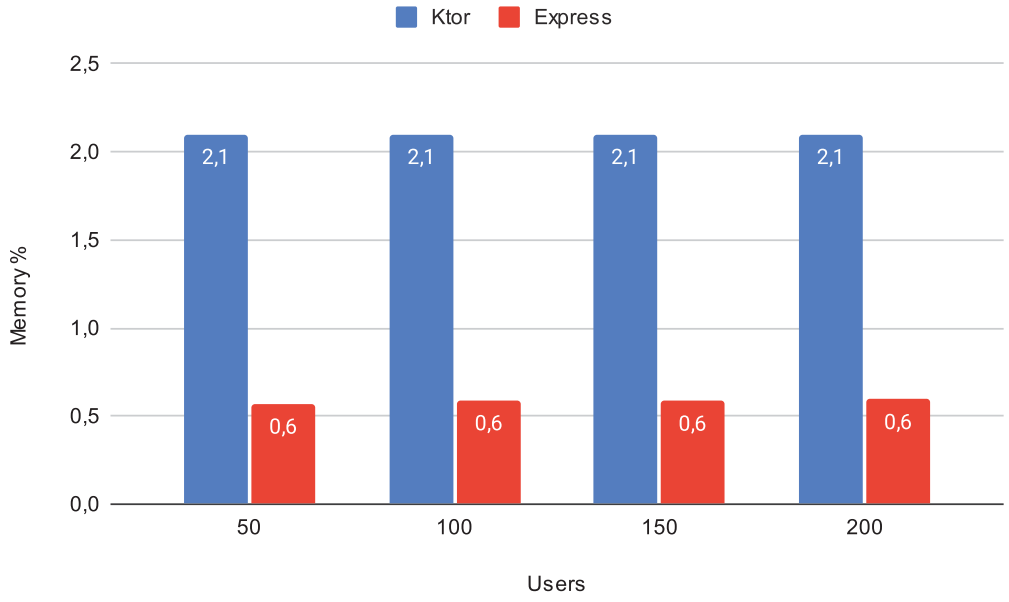

Figure A.8: Memory usage for the static small pages tests on the servers.

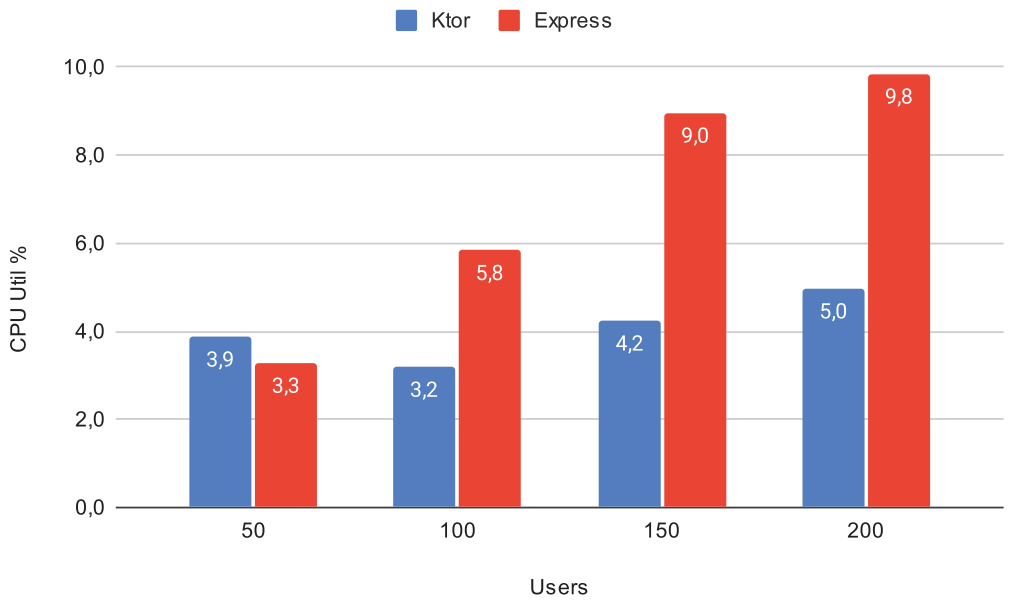

Figure A.9: CPU utilization for the static large pages tests on the servers.

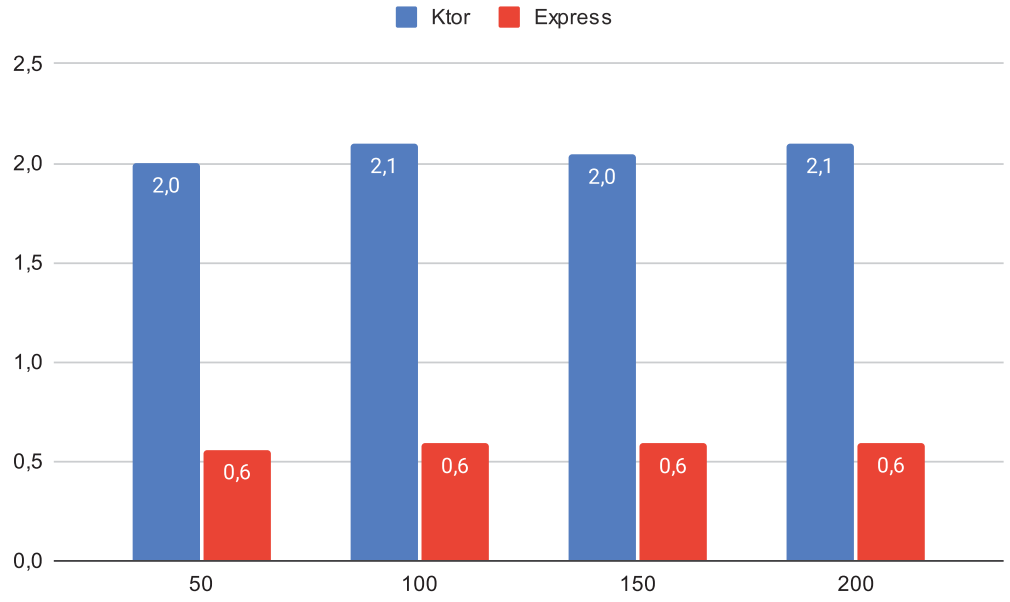

Figure A.10: Memory usage for the static large pages tests on the servers.

Setup of the computer used to host the servers.

Component	**Part**
CPU	Intel Core i3 10105
RAM	16GB
Storage	SSD
OS	Pop!_OS 20.04

Table A.1: Computer setup

Network devices

Device	**Name**	**Speed (Gbps)**	**Connection**
Server system	Intel® Ethernet Connection I219-V	1	Cable
Client system	Dell Adapter USB-C (DA300)	1	Cable
Network switch	Ubiquiti UniFi USW Flex Mini	1	Cable

Table A.2: Network setup

Versions of the programs and frameworks used.

Application	**Version**
Node.js	16.14.2
npm	8.5.0
Chrome's V8 engine	9.4.146.24-node.20
Express.js	4.16.4
EJS	2.6.2
Sequelize	6.17.0
OpenJDK	17.3
Kotlin	1.6.10
Ktor	1.6.8
Freemarker (io.ktor:ktor-freemarker)	1.6.8
Python3	3.10.4

Table A.3: Application versions

Setup of the investigation PC.

Component	**Part**
CPU	Intel Core i5 8300H
RAM	8GB
Storage	SSD
OS	Windows 11

Table A.4: Investigation computer setup

Here are the SQL queries made by the ORMs

```
SELECT "Article"."id", "Article"."header", "Article"."sub_header", "
    Article"."description", "Article"."date_posted", "Article"."
    posted_by", "Article"."picture_url", "Sections"."id" AS "Sections
    .id", "Sections"."header" AS "Sections.header", "Sections"."body"
     AS "Sections.body", "Sections"."article_id" AS "Sections.
    article_id", "Sections"."picture_url" AS "Sections.picture_url"
    FROM "articles" AS "Article" LEFT OUTER JOIN "sections" AS "
    Sections" ON "Article"."id" = "Sections"."article_id" WHERE "
    Article"."id" = 2;
```

Listing A.1: SQL query made by the Sequelize ORM in Express.js.

```
SELECT articles.id, articles."header", articles.sub_header, articles
    .description, articles.date_posted, articles.posted_by, articles.
    picture_url FROM articles WHERE articles.id = 2;

SELECT sections.id, sections."header", sections.body, sections.
    picture_url, sections.article_id FROM sections WHERE sections.
    article_id = 2;
```

Listing A.2: SQL queries made by the Exposed ORM in Ktor.

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