JÖNKÖPING UNIVERSITY
School of Engineering

A Performance comparison Between ASP.NET Core and Express.js for creating Web APIs

MAIN FIELD: Computer Science
AUTHOR: Oliver Karlsson
SUPERVISOR: Peter Larsson-Green
JÖNKÖPING: July 2021

This final thesis has been carried out at the School of Engineering at Jönköping University within computer science. The author is responsible for the presented opinions, conclusions, and results.

Examinator: Anders Adlemo
Handledare: Peter Larsson-Green
Scope: 15 hp (bachelor's degree)
Datum: 2021-07-25

Mail Address:	Visiting Address:	Phone:
Box 1026	Gjuterigatan 5	036-10 10 00 (vx)
551 11 Jönköping		

Abstract

Modern web applications are growing in complexity and becoming more widely used. Using frameworks to build APIs is a popular way for both hobby developers and businesses to speed up development time and save costs. With this dependence on frameworks to be the foundation for potentially large applications comes the need to understand their performance qualities and which areas they are best suited for.

This study compares the performance of the two similarly popular frameworks ASP.NET Core and Express.js, when used together with a MySQL Database to build Web APIs. This was done by building two different API implementations in each framework, one employing a RESTful approach and the other using the new querying language GraphQL. Experiments were run where the peak CPU usage, peak memory usage and response times were measured.

The results of the experiments were that in a RESTful API, ASP.NET Core is faster at serving requests during lower loads whereas Express.js outperforms ASP.NET Core when faced with a higher amount of concurrent requests that fetch a lot of data. In a GraphQL API Express.js was able to perform similarly or better in all cases in terms of response times and resource usage compared to ASP.NET Core.

Keywords

API, Framework, REST, GraphQL, HTTP, SQL, ASP.NET Core, Express.js, Operating System, CPU, Memory, Response times.

Contents

1 Introduction

1.1 Background

The use of Application Programming Interfaces(APIs) when creating mobile and web applications is an effective way for an application to have remote access to data or other devices, but because of this there is a greater need for performance than before. APIs are commonly found in IoT(Internet Of Things) devices where things such as common household items or vehicles are able to exchange information or be controlled by a user through the internet. They are also used to create Single Page Applications which are applications that loads a single web document with the necessary HTML, CSS and JavaScript. These static pages then fetch the content that should be displayed through a Web API.

Frameworks can be viewed as "semi-complete" applications that provide a specific set of structures and functionalities[1] that developers can extend and use to build complete software. Frameworks encourage reusing code, allowing for quick implementation of functionality thereby cutting development times and costs[2]. This makes them a very attractive solution to developers and companies that want to get started quickly.

Selecting a framework for a specific project often means having to consider multiple factors: which platform can it support and what devices will the userbase use to access it. What will the expected size and complexity of the project be, does it need to be easily scalable to accommodate fluctuations in user traffic? One must also consider the knowledge required to use the framework for its intended goal. Is the developer team already familiar with the language and style it uses, will they need time to learn or should someone with the knowledge be hired?

ASP.NET Core and Express.js are two open-source Web frameworks that can be used to develop APIs. Express.js was selected for this study because of its high popularity[3], its simplicity and because it uses JavaScript to develop software which is a popular language for front-end development. ASP.NET Core is just under Express.js in popularity[3] and according to the Stack Overflow developer survey done in February 2020 it is the most preferred web framework by professional developers[4].

1.2 Problem description

The amount of web applications that rely on APIs for data is increasing. ProgrammableWeb is an information and news source regarding APIs, they also provide an API directory used to catalogue and document API usage. In 2019 they reported that they added 168 APIs to their directory each month resulting in over 2000 new APIs each year on average[5]. With applications growing in complexity and the number of users on the internet steadily increasing, it can make it harder for a developer to decide which technology is aptly used for a projects desired outcome.

Response times and loading times on websites and applications greatly impact user experience and a large portion of this is the server performance. In 2018, BBC reported that for every extra second a page took to load, around 10% of users left[6]. COOK is a company that sells frozen meals through their website. They thought that slow loading times were having a negative effect on their business. A case study was done and showed that reducing the average load time by 850 milliseconds increased the amount of users making a purchase by 7%, and increased customer engagement by 10%[7].

Erlandsson & Remes[8] tested and compared the performances of REST, SOAP and GraphQL on an API built in .NET Core. They found that in most test cases GraphQL performed worse than REST and SOAP regardless of database type used, while REST had the best performance across the board. The cases where GraphQL outperformed REST and SOAP were when a 100 rows were requested in a MySQL database, and when only a single row was requested in a MongoDB database. This shows that there could be a possibility that if a specific framework performs better under certain conditions it would be more effective to make use of a GraphQL architecture.

With an increase in applications that depend on APIs and the growing number of people using the internet. The importance of choosing technology that is able to provide users with the performance that they want becomes greater. This study will focus on comparing the performance of ASP.NET Core and Express.js using a REST(REpresentational State Transfer) architecture, and the new querying language GraphQL. The reason for choosing these architectures is because most Web APIs use REST or an architecture close to it, while GraphQL is very new and not widely used[9]. GraphQL has been shown to require less effort compared to REST when implementing the same functionality [10] and knowing it's performance in different frameworks may therefore be a point of interest to both new and experienced developers.

1.3 Limitations

This study will only be focusing on the performance of each framework and how they compare to each other. To ensure that the results are not influenced by constantly changing variables such as internet speed, consistency and reliability, both APIs will be hosted offline and accessed through a local area network connected by ethernet with a latency of less than 1ms. The database will also be located on the same machine hosting the APIs to remove any latency that would occur if it were hosted on a separate machine.

Both APIs will only be implemented together with a MySQL database. MySQL has consistently been a popular database amongst developers[11–14] for its ease of use, high performance and scalability[15]. To reduce the scope of the study the database will only be used to fetch data from and therefore the performances when writing to a database will not be evaluated in this study.

Because of its high performance and customizability the APIs will be hosted on a Linux machine. The clients that send the requests will be run on a Windows machine, this is since that Windows is the most used operating system for desktops and laptops[16].

1.4 Aim & Research Objectives

The aim of this study is to measure, evaluate and compare the performance of the two web application frameworks ASP.NET Core and Express.js when used to develop Web APIs, to establish how fast they are able to serve requests, and how much computer resources they require. The results from this study will be of interest to any developer looking to develop an API in either of these frameworks together with a RESTful or GraphQL architectural design and a relational database.

This study will provide answers to the following questions:

- What are the differences between an ASP.NET Core Web API and an Express.js Web API in CPU usage, memory usage and response times when implemented with a RESTful architecture?
- What are the differences between an ASP.NET Core Web API and an Express.js Web API in CPU usage, memory usage and response times when implemented with GraphQL?

2 Theoretical Framework

2.1 Link between research questions and theory

ASP.NET Core and Express.js are two frameworks with similar popularity that can be used for the same goals. Each framework has its own architecture and way of doing things that affect its CPU usage, memory usage and how fast it is able to process requests. Making them well suited for a comparative study.

2.2 Representational State Transfer

The RESTful architectural design is a specific style for implementing APIs. Introduced in the year 2000 by Roy Fielding, it defines a set of specific constraints for how an API should function and behave. These constraints are meant to help improve an applications reliability, scalability and performance[27]. Because of its popularity within HTTP based applications it is commonly confused with HTTP, but an application can in theory be RESTful as long as it follows the 6 principles of REST.

2.2.1 The Six Principles of REST

Client-Server: A client-server architecture means a separation of concerns between the two parts. Each system is responsible for different tasks, e.g. the server is responsible for the manipulation of data whereas the client takes information from the server and displays it to the users.

Stateless: Every request that is sent to the server should contain all the necessary information to complete the request. All requests made to the server are independent of each other and the server should not store any information about the clients session.

Cacheable: It is required that the responses to a request state whether it is allowed to be cacheable or not. If a response is cacheable then the client may save the response for later use.

Uniform Interface: A uniform interface between all components makes the overall system architecture much more simple and easier to handle if all parts follow the same rules of communication between each other. REST has four different constraints to achieve this. The first is that each resource should have a unique identifier. The second is that resources should be manipulated through representations. The third principle states that all messages must be self-descriptive. The fourth principle is that hypermedia must be the engine of application state.

Layered System: Dividing the system into layers where each layer cannot see beyond the layer with which they are interacting. This allows each part of the system to easily be replaced or extended without affecting the rest of the system.

Code on demand: The sixth and final principle is optional. This principle lets clients download functionality in the form of scripts. This allows clients to extend their own functionality without needing to implement it on their own. But it is optional because clients may not be able to download or run this code so REST cannot rely on it.

2.2.2 Request & Response

As stated previously, one of the constraints for a uniform interface is that each resource should have a unique identifier. This comes in the form of Uniform Resource Identifiers(URIs), the example seen in Figure 1 is a Uniform Resource Locator(URL) which is a type of URI. it contains the protocol used to communicate and the path to a specific resource, in this case it is a user with the ID 1. The response to an HTTP GET request to this URL would in this case be in the JSON format, and look like the example seen in 2.

```
https://localhost/user/1
```

Figure 1: Example URL

```
{
    "user": {
        "id": 1,
        "name": "John Doe",
        "age": 30,
        "address": "Example"
    }
}
```

Figure 2: RESTful HTTP GET Response

2.3 GraphQL

Developed by Facebook and released as open source in 2015, GraphQL is a querying language designed for APIs. Unlike a traditional API which consists of multiple endpoints to help separate requests and resources, an API built with GraphQL only consists of a single endpoint that validates and executes incoming queries to fetch any resource[17].

2.3.1 Schema

The most common and important GraphQL component is the schema object. GraphQL used its own syntax for defining these schemas and it is referred to as either SDL(Schema Definition Language) or IDL(Interface Definition Language). A schema definition contains two major parts, the type and all the fields related to it. There are five different scalar types that are natively supported by GraphQL and which are Int, Float, String, Boolean and ID, with non-nullable fields being denoted by a ! following the scalar definition.

```
{
    type User {
        id: ID!
        name: String!
        age: Int
        address: String
    }
    type Query {
        user(id: ID!): User
    }
}
```

Figure 3: GraphQL type schema definition

Figure 3 above shows the definition of two different types, an object type and a query type. The object type `User` is made to represent an object that exists in the database and should define all of the fields related to that object that are returnable. The `Query` type is what defines a valid entrypoint to the server. It defines the name of the query, all arguments necessary to find the perform the operation and the type of the object that it will respond with. These queries then get matched to resolver functions that are responsible for carrying out the tasks related to their respective queries.

2.3.2 Query

```
  {
    user(id: "1"){
        name
        age
    }
}
```

```
 {
    "user": {
        "name": "John Doe",
        "age": 30
    }
}
```

Figure 4: GraphQL Query Request & Response

Figure 4 above shows a basic example of a GraphQL query request and response. This query will fetch the `name` and `age` of a user with the id of 1. Note that even though the schema above in Figure 3 was defined to also include a field for address and ID, it was not included in the query. This is how GraphQL handles overfetching, instead of the server returning all of the data related to that object it allows the client to specify in the query itself exactly which fields are needed. While this method does create more overhead for the HTTP request itself, it has also been shown to greatly reduce the size of the responses in cases where REST has to hit multiple endpoints to fetch the same amount of data[18].

2.3.3 Execution flow

The GraphQL query goes through three different phases once it reaches the server. In the first phase the query is parsed into an abstract syntax tree(AST). The second phase validates the AST against the created schemas where it checks for the correct syntax and if the provided fields are defined. The third phase is execution. The runtime will go through the AST where it will invoke the correct resolvers, collect all the results and return the JSON[19].

2.4 ASP.NET Core

ASP.NET Core(Active Server Pages .NET Core) is an open-source web framework for creating web applications in C# released in 2016. It is a re-design of ASP.NET and was developed by Microsoft and the .NET Foundation. ASP.NET Core applications run on Windows, Linux, macOS and Docker, as opposed to ASP.NET that only supported Windows[20]. It can be both synchronous and asynchronous and uses a multithreaded approach to handling many requests concurrently. This means that each connection to the application by a client is allocated its own thread with a dedicated amount of RAM.

2.4.1 Threads

A thread is the execution context of a process. When the application process is started, the Common Language Runtime(CLR) will create one foreground thread to execute application code. A single process may have multiple threads active in the background to execute code related to the process[21]. Figure 5 below shows the flow of a request through an ASP.NET Core application. The request will be assigned an available thread in the thread pool. The code related to the request will then be executed until it comes across an operation that it needs to wait for, at which point a continuation will be scheduled and the thread will be returned to the thread pool. If the thread pool has no available threads to handle a new request, then a new thread may be spawned. But simply spawning a new thread is not always the solution, a thread needs regular access to resources in order to progress. The situation where a thread is unable to have regular access to shared resources and CPU time is referred to as thread starvation.

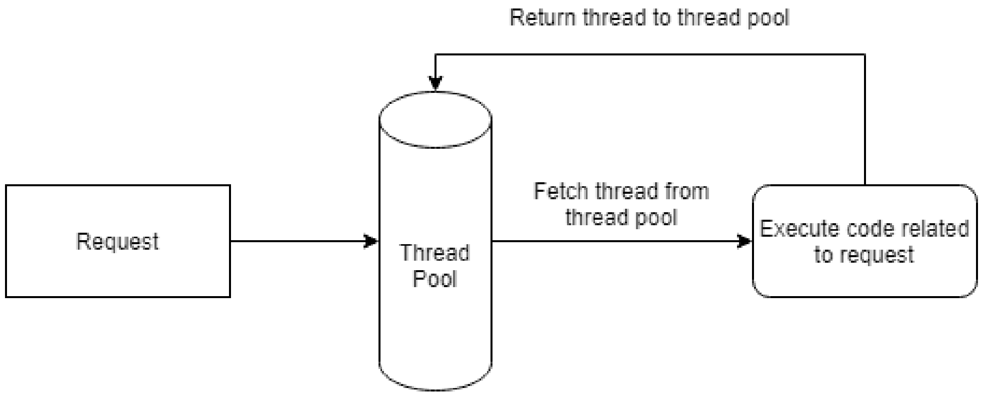

Figure 5: Thread pool flow

2.4.2 Tasks & Async/Await

Tasks are a key component to achieving a task-based asynchronous pattern in .NET[22]. It represents an operation that does not return a value, instead it returns the promise of a result in the future. Tasks are scheduled on the thread pool, with no control over which thread it should specifically execute on.

Asynchronous I/O calls are implemented through the async/await pattern. Functions are defined with the `async` keyword which allows the use of the keyword `await` inside the function. Figure 6 below shows a code example of what the `await` keyword does[23]. It first checks to see if the thing it is awaiting is already done. If it is done then it simply gets the results of the action, if it is not done then it will create a continuation to this action and then immediately return so that the thread can be returned to the thread pool and perform other work. Once the awaited part is complete the continuation can be scheduled on any available thread.

```
var $awaiter = something.GetAwaiter();
if (!$awaiter.IsCompleted)
{
    _state = 42;
    AwaitOnCompleted(ref $awaiter);
    return;
}
Label42:
$awaiter.GetResult();
```

Figure 6: await procedure

2.4.3 C# Compiler

C# Uses its own open-source compiler named Roslyn[24]. It works by converting all C# code into intermediate language(IL) which then gets stored inside an assembly(either an DLL or a EXE). The IL then gets executed by the .NET Core virtual machine CoreCLR. It does this by loading the IL during runtime which then gets compiled into native CPU instructions by a just-in-time(JIT) compiler[25]. As of .NET Core 3.0 the JIT compiler features tiered compilation[23]. This is a compromise to achieve faster start-up times instead of spending too much time optimizing code compilation. Methods first get compiled with a fast-pass that applies very few optimizations to the code. But as the methods are executed again and again, more and more optimizations are made to code quality.

2.4.4 Memory Management

Memory management in ASP.NET Core works by placing objects into one of the 3 different generations: 0, 1 or 2(see Figure 7). Objects placed in the lower generations will be attempted to be freed more often than those in the higher generation. The lifetime of objects will decide which generation they are placed in. Objects created during web requests are short lived and will often remain in the lower generations, while objects such as application level singletons live for the duration of the application and so they eventually end up in generation 2[26].

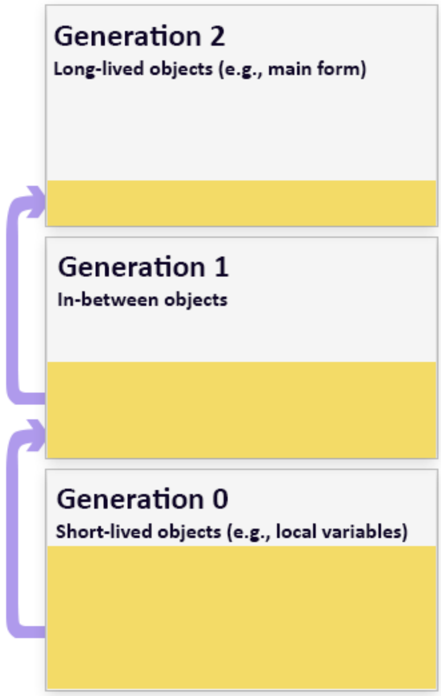

Figure 7: .NET Memory Generations

As objects are created they get allocated to generation 0. When generation 0 is full the Garbage Collector will run where it will remove all unreachable objects from the heap. All objects that are reachable are then promoted to generation 1. This process is repeated for each generation when they become full. When the garbage collection process happens, it will start on the latest full generation and working back down to generation 0[27].

2.5 Express.js

Express.js is an open-source web-application framework for Node.js. It was released in 2010 as a way to make it easier and quicker to develop web applications and APIs in Node.js[28]. Express.js runs on google's V8 JavaScript engine and describes itself as a "Fast, unopinionated, minimalist web framework". Node.js is a single threaded, event-driven execution environment for JavaScript. Multiple studies evaluating the usability of Express.js and Node.js have been done. Both Greiff & Johansson[29] and Lund[30] concluded that a large reason for its popularity is because it allows the use of JavaScript for both front-end work and back-end.

2.5.1 Event Loop

The way Node.js implements concurrency is through the event loop provided by the library libuv[31] that is written in C. libuv also provides support for asynchronous I/O by making use of the Operating System kernel, libuv manages its own thread pool that is used for operations that are too heavy to run in the event loop thread such as file system operations or DNS functions[32].

The event loop is the single threaded part of Node.js and is the thread where all the user written code is executed. This means that every request made to the application will be processed one at a time. When it comes across operations that are potentially blocking, it will offload them to the system kernel if possible. When the kernel has finished processing it will notify the Node.js application that will then queue a callback function(a function that is passed into another function as a parameter) onto the queue to be executed.

The event loop has six different stages it has to pass before it can begin a new iteration or "tick"[33]. Figure 8 below shows the six different stages of the loop, note that the *idle, prepare* phase is only used internally and it is not possible to have any direct influence on the duration or execution of this phase.

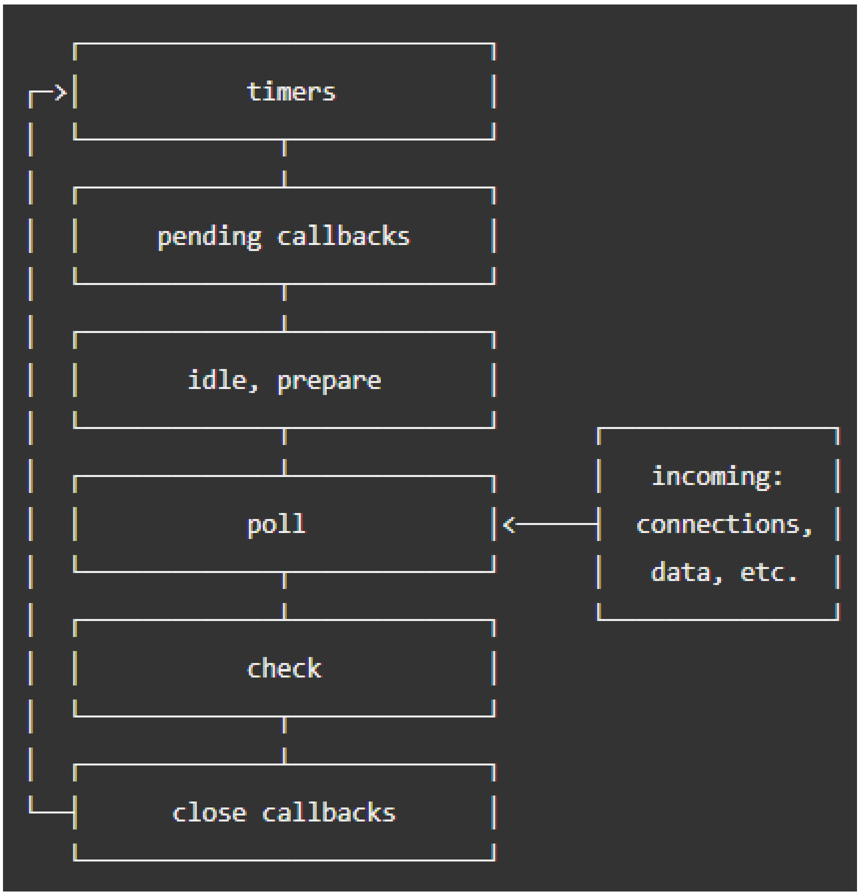

Figure 8: Node.js Event-loop

Timers: Anything scheduled by `setTimeout()` and `setInterval()` is set to be executed here. The scheduled events will be called as early as possible but can be delayed by certain events such as OS scheduling or other callbacks.

Pending callbacks: This phase is responsible for executing callbacks that are registered with system operations such as accessing files, or a network operations.

Poll: This is the phase where requests are received and all of the user-written JavaScript will be executed, until it comes across something that can be delegated to either the system kernel or the libuv thread pool. The poll phase is responsible for two things. First it calculates how long it should block and poll for I/O operations and then it executes any callbacks that exist in the poll queue. It will keep executing callbacks until the poll queue has been exhausted or the system-dependant hard limit has been reached.

Check: If any callbacks are scheduled with setImmediate(), they will be executed here. This timer is used for when a user wishes to execute a callback immediately after the poll phase has ended. If the poll phase becomes idle and scripts have been queued with setImmediate() the event loop will continue to the check phase instead of waiting.

Close callbacks: Here is where all callbacks for close events will be executed, such as closing web sockets. This is where the event loop is cleaning up and preparing for the next tick.

2.5.2 V8 JavaScript engine

The V8 JavaScript engine written in C++ and was originally created for the Chrome web browser[34], but it is also employed by Node.js to run server side JavaScript. V8 will parse the code into an abstract syntax tree(AST). The AST will then get compiled into bytecode by their interpreter Ignition[35] and then executed once. Further executions and optimizations to code are done by the just-in-time compiler TurboFan[36]

2.5.3 Memory Management

V8 is also responsible for managing the memory in Node.js. It does this by dividing its heap into two different generations: new space and old space. The new space contains objects that have recently been allocated, while the old space contains objects that have survived multiple garbage collection cycles[37].

V8 makes use of an optimization method called **Pretenuring**. A pre-tenured object means that it gets directly allocated in the old space. Through special tags called *mementos* that get allocated together with an object on the heap, they encode a textual representation of the code that created the object and records the information into a special map. The encoded data is then used for future optimization of compilers and predicting objects' lifetimes[37].

Asynchronous Callbacks

Callbacks in JavaScript are functions that are passed to other functions as arguments[38] and are commonly used to achieve asynchrony. Figure 9 below shows an example of how a callback is passed, and subsequently called to return a result. *foo()* will call upon function *bar()* and passes along a function that takes another parameter called result. Once *bar()* has finished the work necessary to produce a result, it will simply call on the callback like any normal function, passing the result back to *foo()* that will process the returned results.

```
function foo(){
    bar(function(result) {
        alert(result)
    })
}

function bar(callback){
    var result = 1
    callback(result)
}
```

Figure 9: Callback example use

3 Method and Implementation

This study focuses on a quantitative comparison between ASP.NET Core and Express.js. The data is collected by running experiments that are designed to fetch data, thus this study will not evaluate performance when writing data to the database. The reason for choosing experiments as the method to evaluate these technologies is because this study focuses on measuring performance, and experiments are able to verify if a system can complete a specific task with a reasonable use of resources [39]. For the client the relevant metric is the individual response time for each request, and for the server it is the memory and CPU utilization of the application when processing requests. Each test case will vary in how many parallel requests are sent, how much data is requested as well as the complexity of the query to the database.

3.1 Work Process

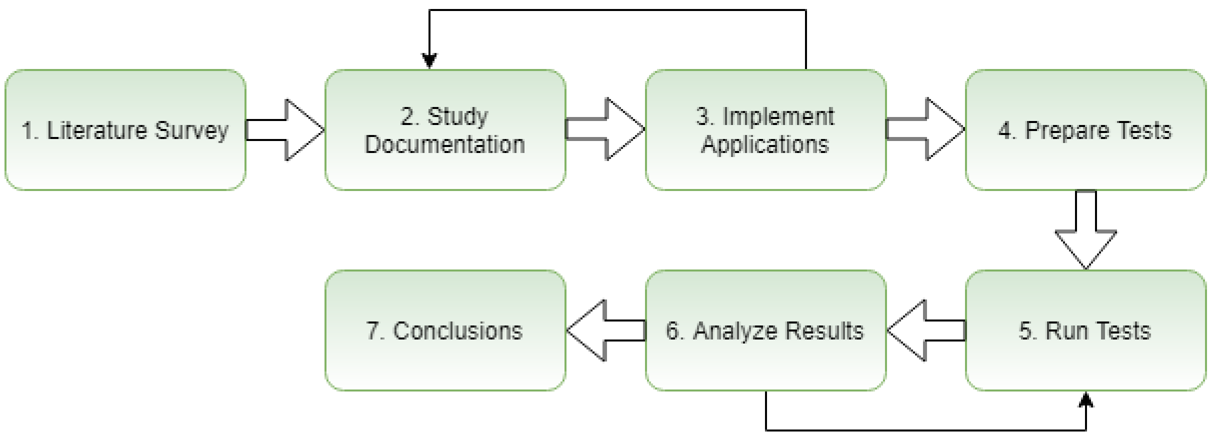

Figure 10: Work Process

Figure 10 above shows the work flow throughout the study. A literature survey was first conducted to gain information about previous research done on the topic and to gain a deeper knowledge about the different technologies involved in the study. As expected, studies done on different frameworks using RESTful architectures were much more common than comparisons with declarative data fetching methods such as GraphQL.

The implementation of the applications was done iteratively together with studying the official documentation for both frameworks to learn how they should be used. After the tests and test data had been prepared the experiments were conducted while collecting the data needed to answer the research questions. The tests were run for 10 iterations each and the applications were restarted between each test case. The decision for 10 iterations was because similar research[29] was performed with only 5 iterations and found that there was very little fluctuations in the metrics between iterations and was enough to provide accurate data. The test cases for REST were completed and analysed first before moving on to GraphQL.

3.2 Data Collection

Server

Because both applications run inside a single process, they can easily be monitored by the top command that is native to Linux that shows information about active processes. Combining top with the grep command makes it so that the top will run in intervals with grep recording the output from top to a desired .txt file. Supplying grep with a process name will make it so that only the information of a specific process is saved. Naturally the top command will only run at 3 second intervals, but running it with the -d flag allows the interval to be manually set. For these tests the interval was set to 0,1 seconds, resulting in 10 updates per second. Both necessary commands can be found in Appendix B.

Client

Apache JMeter v.5.4.1[40] is used to send parallel requests and to record response times on the client. Each test is implemented as a separate test plan where the endpoint, http method and the number of parallel requests are specified. The results from each test will be saved in a CSV format for easy parsing. The tests are run with the command `jmeter -n -t <test_name>.jmx -l <output_file>.jtl`. The `-n` flag specifies that the test should be run in CLI(Command line interface) mode for optimal performance and proper thread initialization. Running the tests in CLI mode is recommended because otherwise JMeter may not initialize all threads quickly enough and therefore the intended load will not be achieved.

3.3 Test Design

The test cases for REST are defined by three variables: number of parallel requests, the number of rows requested from the database and the number of joins between tables performed. Parallel requests will be limited to 100, 250 and 500. Rows requested will be 1, 50 and 100 while table joins are 0 and 1. This means that REST and GraphQL will have 18 test cases each, for a total of 36 test cases. A full table of all combinations can be seen in the table below.

Test Case	Requests	Rows	Joins
1	100	1	0
2	100	50	0
3	100	100	0
4	100	1	1
5	100	50	1
6	100	100	1
7	250	1	0
8	250	50	0
9	250	100	0
10	250	1	1
11	250	50	1
12	250	100	1
13	500	1	0
14	500	50	0
15	500	100	0
16	500	1	1
17	500	50	1
18	500	100	1

These three variables make the test cases fair to both frameworks as they capture results from scaling by adding more requests and by changing the amount of data fetched by each request. The domains set for the variables also reflect a lower, medium and very high load. 500 requests per second is around 43 million requests per day. This compared to the English Wikipedias 29 million unique devices accessing the website each day in January 2021[41] means a web server sustaining a large amount of traffic.

3.4 Database

3.4.1 Schema

The database schema resembles a simple forum containing four different tables. These tables store information about users, threads, comments and thread categories. This schema provides realistic relations between the tables and allows the use of both simple and advanced queries to be used when fetching data. Any effect the schema design would have on performance, will be irrelevant for this study. This is because both applications are using the same schema and the same data. The full queries for generating the tables can be found in Appendix A and an ER diagram of the schema can be seen below in Figure 11.

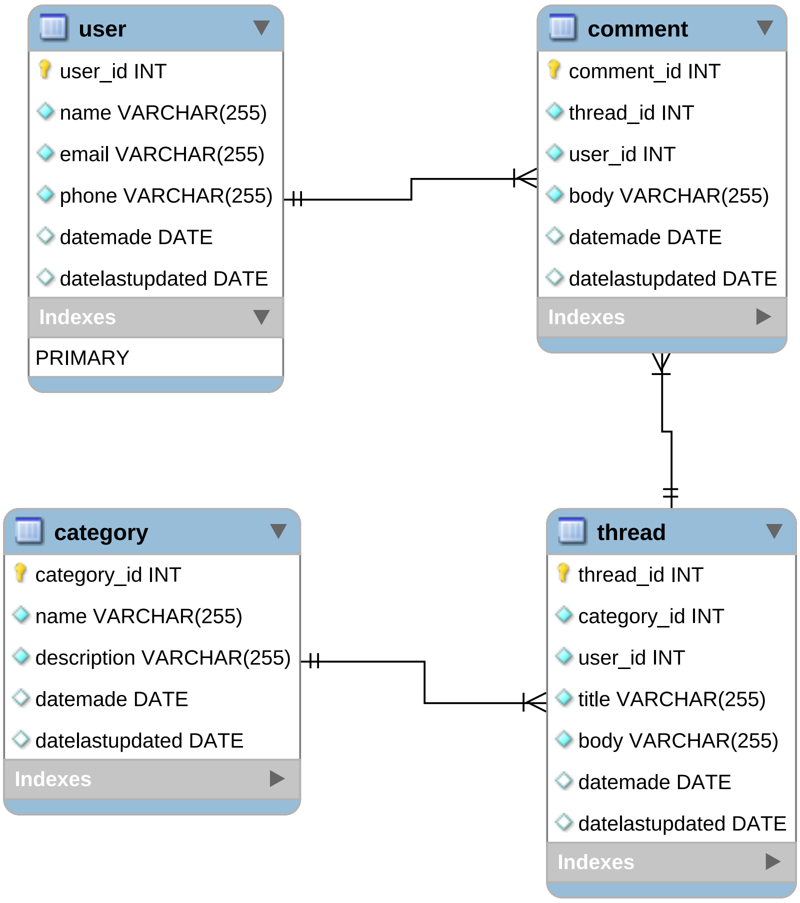

Figure 11: Entity Relationship Diagram

3.4.2 Data

A small Node.js application was developed to fill the database with large amounts of data. It uses the NPM package faker[42] to generate mock data which is then inserted into the database. Faker allows multiple types of data to be generated from a specific seed making it very easy to always generate the same data. The application generates 1000 users and categories and threads. The table that is used for the experiments is the comment table, this will contain 200 comments for each user resulting in 200 000 total rows to emulate a reasonably large database. All relevant code for the application can be seen in Appendix C.

3.5 Implementation

This section will briefly go over the implementation details of each framework. Both frameworks are implemented with the minimum amount of dependencies needed and will be built with stable Long Term Support(LTS) versions. The reason for this is to reduce the amount of variables that could impact the results such as third-party libraries or experimental changes in newer versions. The only optimizations made to each framework is that they are both implemented with asynchronous I/O calls and that they use MySQL connection pooling. This is to prevent any possibility of the database running out of available connections and affecting the experiment results. The connection pool uses the default amount of pools which is 100.

Both REST and GraphQL will be implemented in the same application through different endpoints for both frameworks. This is unlikely to make any impact on the performance due to the application itself exposing a small amount of endpoints whereas real-world examples expose a much larger amount of endpoints. The RESTful endpoints were done entirely with native means and will not be discussed in this section. The source code applications can be seen in their repositories found in Appendix D. Figure 12 below shows a simplified diagram of the data flow of the applications.

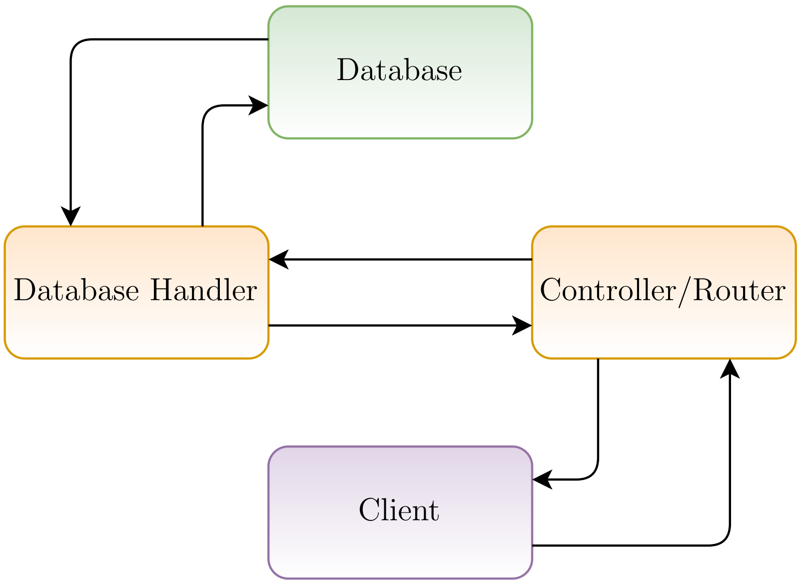

Figure 12: Application Data Flow

3.5.1 API Specification

The main table that will be queried for all test cases is the `comment` table. Each RESTful API will expose the four following endpoints:

- `/comment/:id`
- `/comments/:limit`
- `/user/:id/comment`
- `/user/:id/comments/:limit`

The queries that need to be defined for the GraphQL API are the following:

- `commentById(id: ID!): comment`
- `commentsByLimit(limit: Int!): [comment]`
- `commentByUserId(id: ID!): comment`
- `commentsByUserId(id: ID!, limit: Int!): [comment]`

Fetching a single row from the table will be done by specifying an ID for the comment. Fetching of multiple rows will be done by providing a limit to how many comments should be included in the query result set. Performing queries where a join is to be used, the comment table will be joined with the user table on a specific users ID. Fetching a single row here is then done by passing the ID of a specific user as the argument, which will then retrieve the very first comment made by that user. Multiple rows are fetched through a user ID and a limit which will then fetch a list of comments made by the user.

3.6 ASP.NET Core

The application runs on version 3.1.17 LTS and is created in the .NET Core CLI by running the dotnet new webapi command. This command will create all the necessary files required. MySQL Connector/NET is developed by MySQL themselves and is used to enable communication with the database. It is deployed in production by running the command dotnet publish --configuration Release and then starting the application by through dotnet <projectname>.dll.

To achieve a full asynchronous flow in the application, functions will be defined with the async keyword and the return type is specified as Task<IActionResult>.

3.6.1 GraphQL

Because GraphQL is not yet native to ASP.NET Core the NuGet package GraphQL[43] was installed for this study. The application needs three things besides the HTTP controller in order to fully process GraphQL requests. The first thing is a GraphQLQuery object that is the base model for all received queries, it contains the operation name, query name and all the variables. All of the class objects that are to be handled within the GraphQL endpoint need to be defined as Types. This is done by creating a class that inherits from the ObjectGraphType class. This class will map all fields of the base class to an IdGraphType so that they can be parsed. The last thing needed is the query schema. This defines all the available queries, their return types and what should be done once a specific request is received.

3.7 Express.js

Runs on Express.js version 4.17.1 and Node.js version 14.17.0. This application is created with by running npm init in the command line and will only create the necessary package.json file for the project. Because Express.js has no defined way of doing things, it was implemented very minimally to make it a fair comparison. The application mainly consists of three things, a database handler that handles all the SQL queries, a router for the REST endpoints and a router for the GraphQL endpoint. Express.js also needs to be explicitly told to run in Production which is done through environment variables. The max amount of allowed memory is also set to allow the application to use as much of the available memory it needs.

- NODE_ENV=production
- NODE_OPTION=–max_old_space_size=8192

The asynchronous I/O calls in Express.js are made possible by using callback functions that are called upon once the database has finished fetching the data.

3.7.1 GraphQL

For GraphQL to be implemented in an Express.js application two npm packages are required. The package `graphql`[44] enables schema building through the object `buildSchema`. The schema defines all the types of queries that are to be accepted as valid, as well as the types of objects and data that each query will contain and what they will return.

The package express-graphql[45] provides the `graphqlHTTP` object, this object is the route handler and will receive any incoming HTTP requests made to the defined GraphQL endpoint. Once the request is received it is forwarded to the appropriate resolver. Each resolver is defined inside a root object that will match the query to the correct resolver. This resolver then parses the necessary data such as any included arguments and forwards the request.

GraphQL in Node.js is required to return either a direct value or a promise that in the future will resolve to a value. Therefore all GraphQL resolvers are specified to return a *Promise* object. Figure 13 shows a GraphQL resolver than returns the promise that a database result will eventually be found.

```
comments: ({limit}) => {
    return new Promise((resolve, reject) => {
        db.getComments(limit, function(result){
            resolve(result)
        })
    });
}
```

Figure 13: JavaScript Promise

3.8 Environments

3.8.1 Client

- OS: Windows 10 Home 64Bit
- CPU: Intel(R) Core(TM) i7-8750 CPU @ 2.2GHz
- RAM: 16GB

3.8.2 Server

The chosen Linux Distribution for hosting the APIs is Ubuntu 20.04 Desktop. This is due to its popularity in web hosting, long time support and because it supports all versions of .NET Core and Node.js. The server is a completely fresh install of Ubuntu where all the previous data and operating systems have been removed. The only software installed were MySQL to host the database and Visual Studio Code along with the necessary SDKs and runtimes needed to develop and run the applications.

- OS: Ubuntu LTS 20.04 Desktop
- CPU: Intel(R) Core(TM) i5-4670K CPU @ 3.4GHz
- RAM: 8GB 1333MHz
- Storage: 120GB SSD

3.9 Validity & Reliability

Both APIs will be using a relational database to store data, using the same original data and the same logic for accessing data. This means that the results given by the tests will only be influenced by the performance of each framework.

Each test will be done in 10 iterations and the results will then be used to calculate average values for each test case. This is done to filter out any anomalies that could occur such as background processes using resources and to reduce the significance of any fluctuations in measured values.

4 Empirical Data

This chapter will present tables containing raw data that was collected from running all the test cases. Response times were measured in milliseconds while memory and CPU are in percentages.

4.1 REST

REST - CPU Usage (%)		
Case	ASP.NET Core	Express.js
1	111,28	66,35
2	265,81	90,54
3	310,18	101,36
4	96,59	61,89
5	266,53	107,00
6	327,55	108,91
7	145,64	109,01
8	314,72	125,64
9	346,00	137,00
10	133,29	108,46
11	313,00	126,92
12	337,90	137,37
13	205,47	150,54
14	333,90	147,48
15	348,80	147,00
16	194,99	140,65
17	327,00	144,55
18	350,89	142,55

Table 1: REST CPU Usage

REST - Memory Usage(%)		
Case	ASP.NET Core	Express.js
1	1,38	0,84
2	1,53	0,88
3	1,56	0,97
4	1,46	0,78
5	1,54	0,87
6	1,54	0,88
7	2,00	0,89
8	1,63	1,04
9	1,73	1,16
10	2,09	0,87
11	1,64	1,12
12	1,73	1,16
13	2,82	1,04
14	1,52	1,19
15	1,75	1,32
16	2,69	1,17
17	1,66	1,18
18	1,77	1,34

Table 2: REST Memory Usage

REST - Mean Response Times(ms)		
Case	ASP.NET Core	Express.js
1	78,4	104,0
2	137,0	122,0
3	207,4	137,2
4	74,6	99,0
5	134,9	121,1
6	197,4	135,1
7	83,8	132,0
8	255,9	184,2
9	400,2	246,4
10	82,1	149,5
11	257,0	195,1
12	418,5	247,9
13	120,0	204,8
14	474,9	319,6
15	681,7	385,9
16	126,3	192,6
17	486,3	313,7
18	709,9	369,7

Table 3: Mean REST Response Times

REST - Minimum Response Times (ms)		
Case	ASP.NET Core	Express.js
1	64,7	93,0
2	107,5	99,3
3	126,4	89,9
4	67,3	89,4
5	88,5	98,0
6	115,4	90,8
7	57,1	87,8
8	74,5	92,1
9	103,2	108,7
10	50,8	97,3
11	107,8	97,9
12	168,7	95,2
13	54,9	107,5
14	135,9	122,7
15	145,9	132,7
16	70,5	102,7
17	126,2	124,3
18	226,1	119,6

Table 4: REST Minimum Response Times

REST - Maximum Response Times (ms)		
Case	ASP.NET Core	Express.js
1	100,3	120,2
2	162,8	152,2
3	277,5	188,0
4	83,3	111,5
5	170,4	153,4
6	236,9	183,9
7	108,7	173,2
8	389,1	271,4
9	497,1	381,2
10	114,2	194,9
11	390,5	287,1
12	592,3	388,6
13	182,3	301,6
14	812,4	502,6
15	966,8	637,5
16	183,9	275,8
17	826,6	499,6
18	989,7	675,8

Table 5: REST Maximum Response Times

4.2 GraphQL

GraphQL - CPU Usage (%)		
Case	ASP.NET Core	Express.js
1	263,85	99,04
2	345,28	124,74
3	368,91	152,45
4	229,73	105,00
5	337,10	132,64
6	355,73	144,64
7	323,01	142,01
8	370,00	163,36
9	389,00	163,00
10	349,91	147,45
11	365,00	161,53
12	379,00	161,45
13	350,55	168,45
14	383,00	155,55
15	382,00	163,90
16	354,00	164,55
17	375,00	163,90
18	382,00	166,36

Table 6: GraphQL CPU Usage

GraphQL - Memory Usage(%)		
Case	ASP.NET Core	Express.js
1	2,50	0,88
2	2,97	0,97
3	3,17	1,14
4	2,53	0,87
5	2,94	0,97
6	3,11	1,16
7	3,56	0,95
8	3,86	1,31
9	4,51	1,39
10	3,08	0,98
11	3,93	1,29
12	4,36	1,41
13	4,34	1,23
14	4,85	1,53
15	5,36	1,62
16	4,64	1,29
17	4,53	1,55
18	5,23	1,60

Table 7: GraphQL Memory Usage

GraphQL - Mean Response Times(ms)		
Case	ASP.NET Core	Express.js
1	144,2	152,4
2	197,5	173,9
3	244,2	226,6
4	125,9	152,3
5	195,1	175,3
6	258,4	224,4
7	220,1	223,3
8	382,8	304,4
9	571,3	373,6
10	234,5	237,5
11	378,5	328,6
12	533,9	382,4
13	367,5	371,3
14	588,8	464,0
15	984,9	594,7
16	401,9	362,0
17	705,7	483,5
18	1003,5	584,6

Table 8: Average GraphQL Response Times

GraphQL - Minimum Response Times (ms)		
Case	ASP.NET Core	Express.js
1	110,2	145,9
2	129,6	137,6
3	168,1	162,2
4	87,5	143,7
5	115,7	137,9
6	135,0	159,6
7	97,8	170,3
8	130,9	165,9
9	153,2	174,6
10	109,0	183,8
11	130,7	188,1
12	191,8	177,0
13	89,3	210,4
14	165,8	188,9
15	177,3	161,5
16	144,6	193,6
17	202,1	185,9
18	253,6	173,4

Table 9: Minimum GraphQL Response Times

GraphQL - Maximum Response Times (ms)		
Case	ASP.NET Core	Express.js
1	168,4	164,4
2	228,7	214,9
3	301,8	300,5
4	144,9	165,8
5	222,1	217,9
6	294,1	299,0
7	274,5	271,8
8	516,7	421,4
9	689,3	545,6
10	298,5	288,6
11	533,5	447,0
12	679,0	558,5
13	523,2	480,7
14	1002,5	696,8
15	1289,5	938,4
16	542,0	462,4
17	1019,4	715,9
18	1273,2	921,7

Table 10: Maximum GraphQL Response Times

5 Analysis

This chapter will compare the average values calculated from the test iterations and compare them to each other. Each research question is divided into three sections, CPU usage, memory usage and lastly response times. The CPU and memory sections will be comparing the mean peak values whereas response times will look at overall mean time and mean minimum & maximum times. Because the CPU has 4 cores, the value for CPU usage will be between 0 and 400.

5.1 Research Question 1

5.1.1 CPU Usage

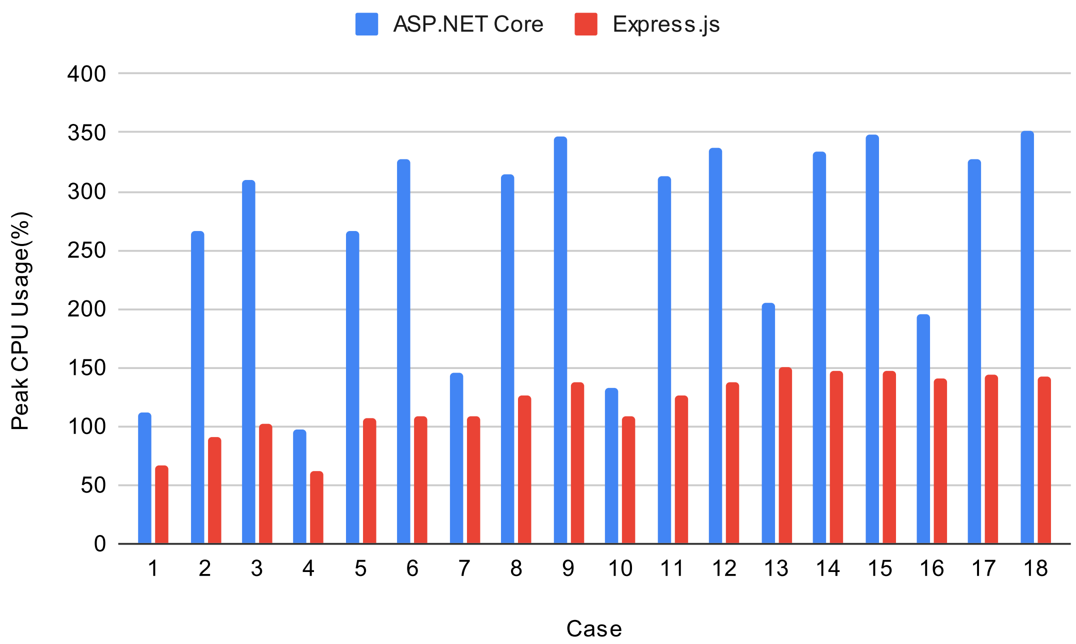

Figure 14: REST CPU Usage

As seen in figure 14 above, the CPU usage is very similar for both frameworks in cases 1, 4, 10, 13, 16. These are all cases where only a single row was requested from the database. ASP.NET Core however sees a much larger increase in CPU usage as the number of rows requested are increased than Express.js.

5.1.2 Memory Usage

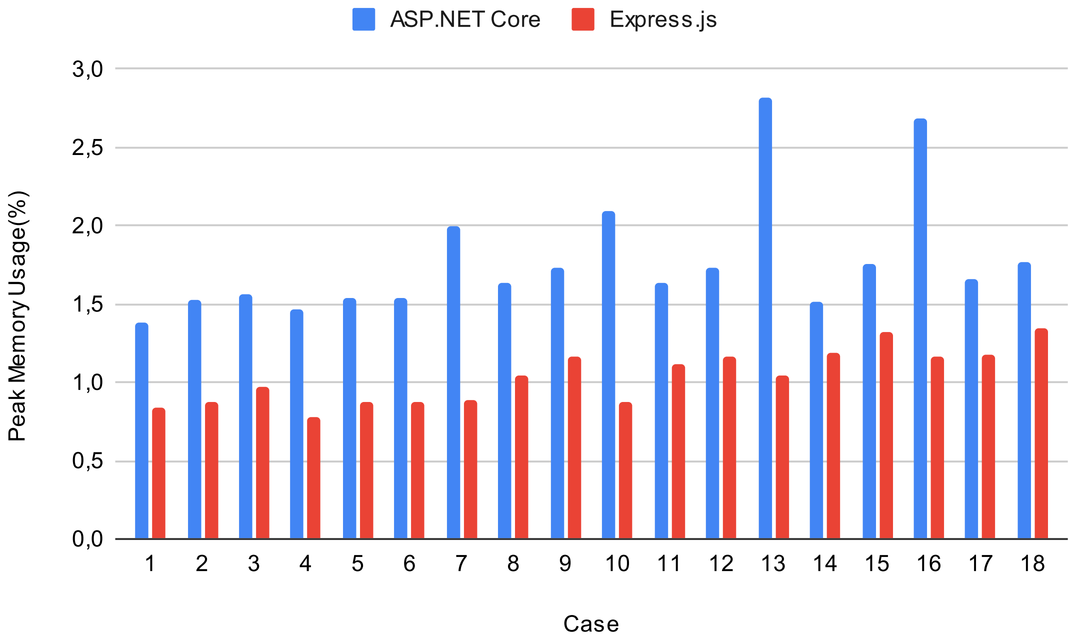

Figure 15: REST Memory Usage

Figure 15 shows that Express.js behaves similarly in cases 1-6 and showing larger increases for the rest of the test cases. When the amount of users are 250 and 500, ASP.NET Core sees an interesting pattern in its memory usage. All cases where only a single row is requested sees a much higher % of memory being used than the following cases for 50 and 100 rows with the same amount of users. But for cases 1-6, ASP.NET Core sees a very stable consumption of memory.

5.1.3 Response times

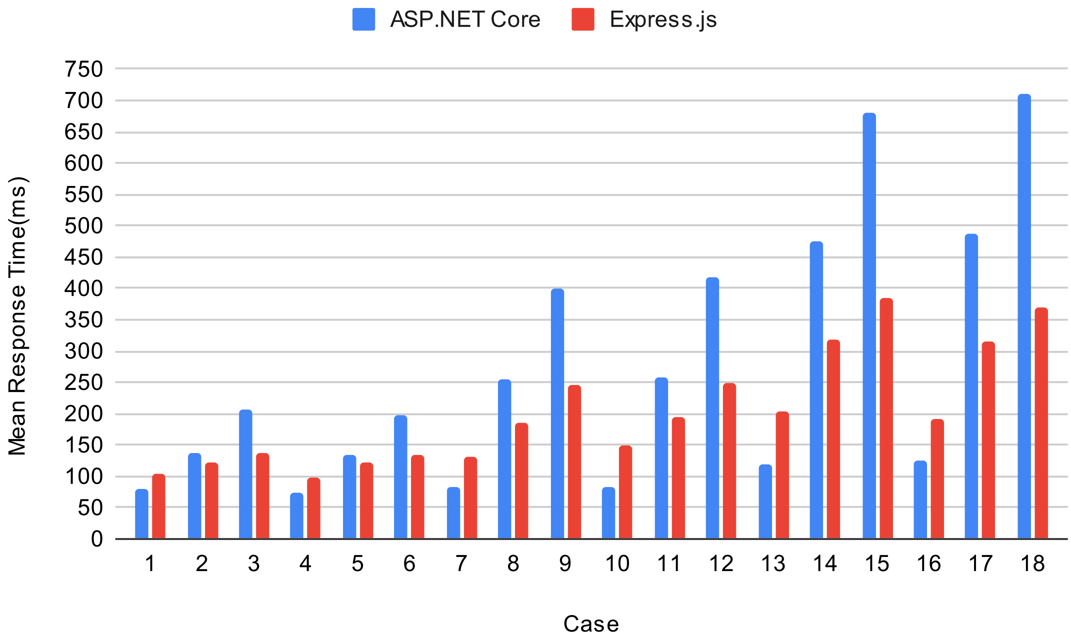

Figure 16: RESTful Response times in milliseconds

Express.js shows a linear increase in response to both an increase in the amount of users and the amount of rows requested from the database.
ASP.NET Core increases linearly mostly due to the amount of rows fetched. It does increase much more by an increase in users when the amount of rows fetched are 50 and 100. Cases (1,4) & (7, 10) where 100 and 250 users respectively, request a single row show very little to no change in response times at all. It does however show a smaller increase when going up to 500 users requesting a single row.
ASP.NET Core is also much more affected by an increase in the amount of rows requested than Express.js is.

All cases where ASP.NET Core had a higher average response time, it also has a much higher maximum response time, i.e. the absolute slowest response to a request.

In all test cases where a single row was requested, ASP.NET Core was shown to be consistently faster than Express.js(see Figure 16). This is also true for its minimum and maximum response times, ASP.NET Core was able to serve all requests faster than Express.js.

The opposite can be seen in cases where 50 & 100 rows were requested. Here ASP.NET Core displays both higher minimum and higher maximum, up to almost 2x during cases 14, 15 , 17 and 18.

5.2 Research Question 2

5.2.1 CPU Usage

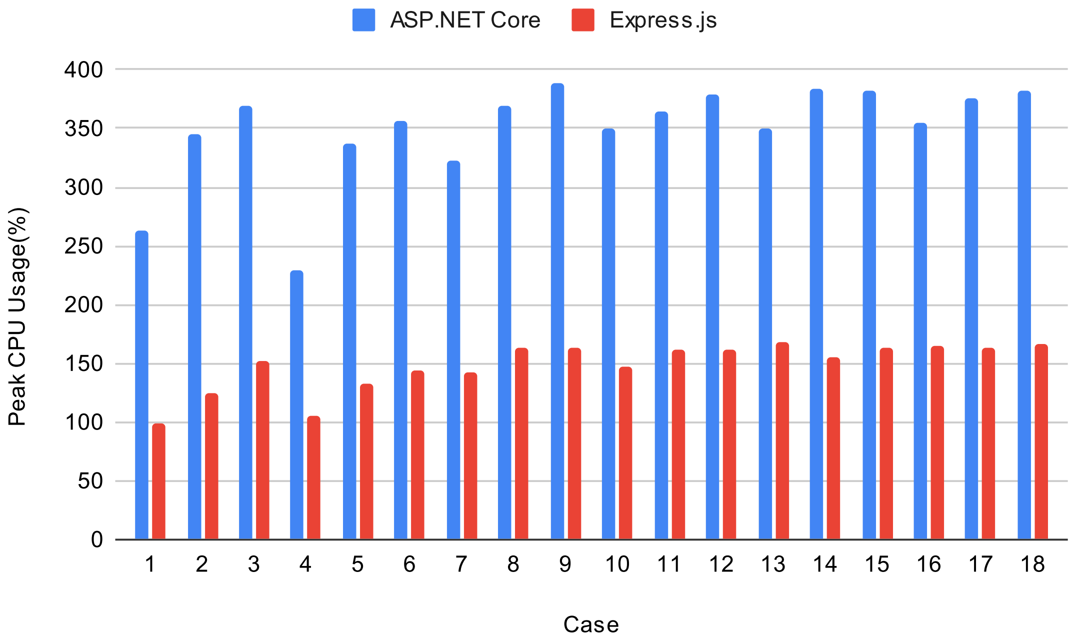

Figure 17: GraphQL CPU Usage

As shown in Figure 17 ASP.NET Core utilizes a very high amount of CPU even for the lowest loads. With all but two test cases using all 4 cores. For most test cases ASP.NET Core uses more than double the amount that Express.js uses.

5.2.2 Memory Usage

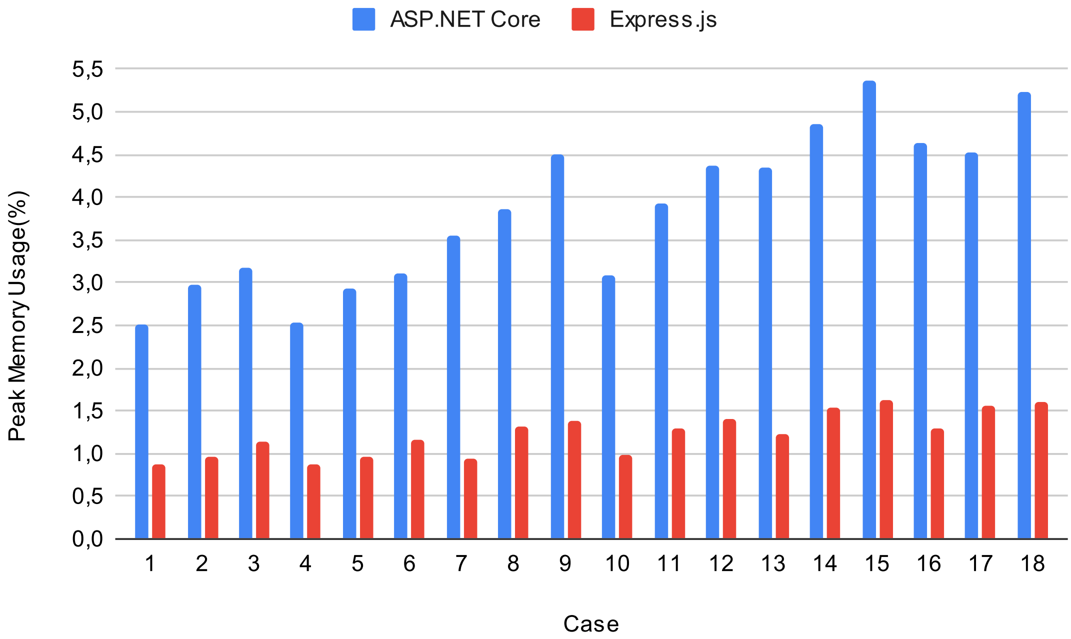

Figure 18: GraphQL Memory Usage

For all cases ASP.NET Core uses significantly more memory than Express.js does(see Figure 18). Express.js peaks as high as slightly above 1,5% where as ASP.NET Core uses almost 5,2% during the same test case. During the lowest loads ASP.NET Core uses roughly 2,8x the amount of Express.js, and up to 3,6x as much under high load.

5.2.3 Response times

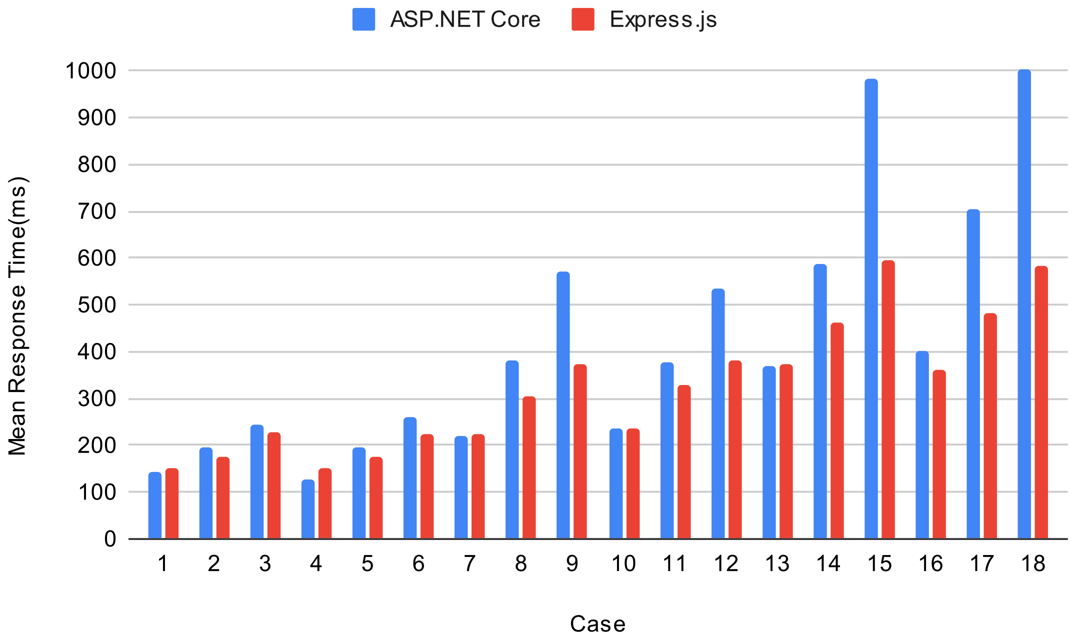

Figure 19: GraphQL Mean Response Times

Both frameworks show a linear increase in response times in response to both increased amount of data fetched from the database and the amount of users making requests. Express.js sees much smaller and steadier increases compared to ASP.NET Core that sees larger increases as the load is increased. The largest increases can be seen in Figure 19 during cases 15 and 18 for ASP.NET Core where 500 users request 100 rows.

6 Conclusions & Discussion

6.1 Findings

Research Question 1 - What are the differences between an ASP.NET Core Web API and an Express.js Web API in CPU usage, memory usage and response times when implemented with a RESTful architecture?

ASP.NET Core sees a much higher CPU usage compared to Express.js. The reason for this is due to the simplicity of the Node.js event loop. Because all code gets executed in the main thread it means that it is only executing the code for one request at a time. Whereas ASP.NET Core uses multithreading and is executing code for multiple requests at the same time, leading to a much higher utilization of the CPU.

ASP.NET Core is shown to be able to serve requests faster than Express.js as seen during cases 1, 4, 7, 10, 13 and 16. These are the test cases where only a single row was requested. But because of its high CPU utilization it also means that as requests start fetching and processing more data, it quickly reaches full usage of the CPU and becomes throttled in how much work it can do concurrently, whereas Express.js does not get affected as much due to the inherent delay that comes from the event loop. ASP.NET Core also sees a higher memory usage throughout all test cases, like the CPU usage this is because of the multithreading, it is processing multiple requests at the same time, therefore allocating more memory for objects and threads.

ASP.NET Cores memory usage during cases 7, 10, 13 and 16 is noticeably higher even though the amount of data fetched is lower than the following cases. This is due to each request only fetching a single row, meaning much less objects that get allocated than if more rows would be fetched. And because less objects are allocated, the garbage collector does not run as frequently because the heap does not get full as often, leading to more memory being consumed.

Research Question 2 - What are the differences between an ASP.NET Core Web API and an Express.js Web API in CPU usage, memory usage and response times when implemented with GraphQL?

Much of the behaviour noted in the RESTful APIs can also be observed here, except amplified due to the extra work done when processing GraphQL requests. Both frameworks show a similar speed in response times during lower loads and in contrast to REST there are no cases where ASP.NET Core is able to serve requests noticeably faster than Express.js. ASP.NET Core sees a large increase in response times as it reaches the higher loads such as cases 14, 15, 17 and 18. ASP.NET Core sees a much higher CPU usage for even the lowest loads and very quickly the usage becomes almost maxed out, leading to greater thread starvation. Express.js sees a significantly lower usage in memory, peaking at 1,6% compared to ASP.NET Cores peak of 5,3%.

An interesting thing that happens in the memory usage of ASP.NET Core here is that the memory usage starts low and grows will gradually increase for a couple of iterations until it reaches a steady state. The reason for this is that as more allocations take place, the frequency that the garbage collector will attempt free memory also increases. Combine this with the thread starvation and some requests taking very long to fulfill, it leads to many objects still being referenced as the garbage collector runs. This means that some objects will get promoted to the higher generations that get cleaned less often, leading to an increase in memory usage.

6.2 Implications

The results from this experimental study is an unbiased comparison between two different frameworks used for Web API development. Depending on an applications use case, expected load and desired performance the framework of choice varies. ASP.NET Core will work better for applications that are expected to perform CPU intensive work, and for simpler applications that fetch less data from a database. Express.js however is well suited for an I/O intensive application that will not perform many calculations. Keeping in mind that ASP.NET Core is very dependent on a powerful CPU so that it does not suffer from thread starvation and if CPU availability is limited, Express.js is a valid choice.

6.3 Limitations

While performing these tests on a local network makes the results more focused on the actual performances of the frameworks, it also lacks results that reflect real world scenarios where all requests are sent through the internet. Similarly these requests are only made to reading data from the database which means that it does not measure performance when data is being written.

Because GraphQL isn't native to either framework it means that third party solutions had to be used instead. This means that part of the difference in performance between these two implementations could be because of the implementations themselves instead and therefore not directly applicable to all versions of GraphQL for these frameworks.

6.4 Conclusion

As talked about in the introduction, there are many more factors other than performance that decide which framework is best suited for a specific project. But research and case studies show that performance can have a significant impact on the success of modern web applications.

The main goal of this study was to study and compare the performance of between ASP.NET Core and Express.js. This was done by building two applications that both make use of a RESTful architecture and the querying language GraphQL and then measuring their performance during different types of load. Two sets of experiments were conducted. The first set measured the applications performances when using a RESTful architecture. Here it was observed that ASP.NET Core is able to produce much faster responses during a lower load, but as the load increased Express.js was able to achieve faster response times while using less resources. During the second set the applications used the querying language GraphQL. Here the extra work needed per request in GraphQL made ASP.NET Core perform even worse during higher load due to suffering from thread starvation, whereas Express.js performed much more efficiently during all test cases.

The results of this study should be looked at with the limitations kept in mind. While the results do show that Express.js will perform better in most cases, it is not certain that it always will. Performing more calculations per request may change the results in favor of ASP.NET Core.

6.5 Future Work

These two frameworks are only two of very many frameworks that can be used to create Web APIs. The same tests could be done on different frameworks in order to get a better overview of their strengths and weaknesses. The same applies to hosting environments and databases. This study only evaluated frameworks running on Ubuntu and using a MySQL database. Perhaps one framework performs better in another environment such as Windows or Docker, or when using a non-relational database. Because this study also only focused on reading from the database, it can be expanded by measuring performance when writing to the database or managing different types of data.

It was observed that running Apache JMeter on non-CLI mode would provide very different results due to the client not initializing all requests as quickly as intended. This means that the variance in internet speed can have a large impact in a real world scenario and so performing them over different types if networks may be of interest.

7 References

[1] Douglas Schmidt and F. Buschmann. "Patterns, frameworks, and middleware: Their synergistic relationships". In: June 2003, pp. 694–704. ISBN: 0-7695-1877-X. DOI: 10.1109/ICSE.2003.1201256.

[2] Stefan Haefliger, Georg von Krogh, and Sebastian Spaeth. "Code Reuse in Open Source Software". In: *Management Science* 54.1 (2008), pp. 180–193. ISSN: 00251909, 15265501. URL: http://www.jstor.org/stable/20122369.

[3] *Stack Overflow Developer Survey - Web Frameworks*. Feb. 2020. URL: https://insights.stackoverflow.com/survey/2020#technology-web-frameworks-all-respondents2.

[4] *Stack Overflow Developer Survey - Most Loved, Dreaded, and Wanted Web Frameworks*. Feb. 2020. URL: https://insights.stackoverflow.com/survey/2020#technology-most-loved-dreaded-and-wanted-web-frameworks.

[5] Wendell Santos. *APIs show Faster Growth Rate in 2019 than Previous Years*. July 2019. URL: https://www.programmableweb.com/news/apis-show-faster-growth-rate-2019-previous-years/research/2019/07/17.

[6] Matthew Clark. *How the BBC builds websites that scale*. Jan. 17, 2018. URL: https://www.creativebloq.com/features/how-the-bbc-builds-websites-that-scale (visited on 09/19/2020).

[7] *Cook Case Study*. 2017. URL: https://blog.eggplantsoftware.com/case-studies/cook-increases-conversions-by-seven-percent-thanks-to-faster-load-time (visited on 09/19/2020).

[8] Pontus Erlandsson and Joakim Remes. "PERFORMANCE COMPARISON Between GraphQL,REST & SOAP". 2020.

[9] Wendell Santos. *Which API Types and Architectural Styles are Most Used?* Nov. 2017. URL: https://www.programmableweb.com/news/which-api-types-and-architectural-styles-are-most-used/research/2017/11/26.

[10] G. Brito and M. T. Valente. "REST vs GraphQL: A Controlled Experiment". In: *2020 IEEE International Conference on Software Architecture (ICSA)*. 2020, pp. 81–91.

[11] *Stack Overflow Developer Survey 2017*. URL: https://insights.stackoverflow.com/survey/2017#technology-_-databases (visited on 09/19/2020).

[12] *Stack Overflow Developer Survey 2019*. URL: https://insights.stackoverflow.com/survey/2018#technology-_-databases (visited on 09/19/2020).

[13] *Stack Overflow Developer Survey 2019*. URL: https://insights.stackoverflow.com/survey/2019#technology-_-databases (visited on 09/19/2020).

[14] *Stack Overflow Developer Survey 2020*. URL: https://insights.stackoverflow.com/survey/2020#technology-databases (visited on 09/19/2020).

[15] *What is MySQL?* URL: https://dev.mysql.com/doc/refman/8.0/en/what-is-mysql.html (visited on 09/19/2020).

[16] *Operating System Market Share*. URL: https://bit.ly/3i0W3r3 (visited on 09/18/2020).

[17] *GraphQL— A query language for your API*. URL: https://graphql.org/ (visited on 07/02/2020).

[18] Arnar Freyr Helgason. "Performance analysis of Web Services: Comparison between RESTful & GraphQL web services". 2017.

[19] Jonas Helfer. *GraphQL explained*. May 23, 2016. URL: https://www.apollographql.com/blog/graphql/basics/graphql-explained/.

[20] *What is ASP.NET Core?* URL: https://dotnet.microsoft.com/learn/aspnet/what-is-aspnet-core (visited on 07/28/2020).

[21] *C# Thread Class*. URL: https://docs.microsoft.com/en-us/dotnet/api/system.threading.thread?view=net-5.0.

[22] *C# Task Class*. URL: https://docs.microsoft.com/en-us/dotnet/api/system.threading.tasks.task?view=net-5.0.

[23] Stephen Toub. *Performance Improvements in .NET Core 3.0*. May 15, 2019. URL: https://devblogs.microsoft.com/dotnet/performance-improvements-in-net-core-3-0/.

[24] *roslyn*. 2021. URL: https://github.com/dotnet/roslyn.

[25] Mark J. Price. *C# 8.0 and .NET Core 3.0 - Modern Cross-Platform Development*. Fourth Edition. Packt, 2019. ISBN: 9781788478120.

[26] *Memory management and garbage collection (GC) in ASP.NET Core*. 2021. URL: https://docs.microsoft.com/en-us/aspnet/core/performance/memory?view=aspnetcore-5.0.

[27] *.NET Memory Management Concepts*. 2021. URL: https://www.jetbrains.com/help/dotmemory/NET_Memory_Management_Concepts.html#large-object-heap.

[28] *Express*. URL: https://expressjs.com/ (visited on 07/27/2020).

[29] Magnus Greiff and André Johansson. "Symfony vs Express: A Server-Side Framework Comparison". 2019.

[30] Niklas Lund. "Node.js vs PHP A comparison of server side technologies". 2017.

[31] *libuv*. 2021. URL: http://docs.libuv.org/en/v1.x/.

[32] *Thread pool work scheduling — libuv documentation*. URL: http://docs.libuv.org/en/latest/threadpool.html (visited on 05/21/2021).

[33] *The Node.js Event Loop, Timers, and process.nextTick()*. URL: https://nodejs.org/en/docs/guides/event-loop-timers-and-nexttick/.

[34] *What is v8*. URL: https://v8.dev/docs.

[35] *Firing up the Ignition interpreter*. URL: https://v8.dev/blog/ignition-interpreter.

[36] *Digging into the TurboFan JIT*. URL: https://v8.dev/blog/turbofan-jit.

[37] Georgiy Krylov et al. "The Evolution of Garbage Collection in V8: Google's JavaScript Engine". In: *2020 9th Mediterranean Conference on Embedded Computing (MECO)*. 2020, pp. 1–6. DOI: 10.1109/MECO49872.2020.9134326.

[38] *Callback function*. URL: https://developer.mozilla.org/en-US/docs/Glossary/Callback_function.

[39] Justin Zobel. *Writing for Computer Science*. 3rd. Springer Publishing Company, Incorporated, 2015. ISBN: 1447166388.

[40] *Apache JMeter*. 2021. URL: https://jmeter.apache.org/.

[41] *Wikimedia Statistics*. 2021. URL: https://stats.wikimedia.org/#/en.wikipedia.org/reading/unique-devices/normal%7Cline%7C2-year%7C(access-site)~mobile-site*desktop-site%7Cmonthly (visited on 02/03/2020).

[42] *faker - npm*. 2021. URL: https://www.npmjs.com/package/faker.

[43] *NuGet Gallery — GraphQL 3.3.2*. 2021. URL: https://www.nuget.org/packages/GraphQL/.

[44] *graphql - npm*. 2021. URL: https://www.npmjs.com/package/graphql.

[45] *express-graphql - npm*. 2021. URL: https://www.npmjs.com/package/express-graphql.

Appendix A Appendices

A SQL

```sql
CREATE TABLE `category` (
  `category_id` int NOT NULL AUTO_INCREMENT,
  `name` varchar(255) NOT NULL,
  `description` varchar(255) NOT NULL,
  `datemade` date DEFAULT NULL,
  `datelastupdated` date DEFAULT NULL,
  PRIMARY KEY (`category_id`)
) ENGINE=InnoDB AUTO_INCREMENT=1 DEFAULT CHARSET=utf8mb4;

CREATE TABLE `comment` (
  `comment_id` int NOT NULL AUTO_INCREMENT,
  `thread_id` int NOT NULL,
  `user_id` int NOT NULL,
  `body` varchar(255) NOT NULL,
  `datemade` date DEFAULT NULL,
  `datelastupdated` date DEFAULT NULL,
  PRIMARY KEY (`comment_id`),
  KEY `thread_id_fk` (`thread_id`),
  KEY `user_id_fk` (`user_id`)
) ENGINE=InnoDB AUTO_INCREMENT=1 DEFAULT CHARSET=utf8mb4;

CREATE TABLE `thread` (
  `thread_id` int NOT NULL AUTO_INCREMENT,
  `category_id` int NOT NULL,
  `user_id` int NOT NULL,
  `title` varchar(255) NOT NULL,
  `body` varchar(255) NOT NULL,
  `datemade` date DEFAULT NULL,
  `datelastupdated` date DEFAULT NULL,
  PRIMARY KEY (`thread_id`),
  KEY `category_id_fk` (`category_id`),
  KEY `user_id_fk` (`user_id`)
) ENGINE=InnoDB AUTO_INCREMENT=1 DEFAULT CHARSET=utf8mb4;

CREATE TABLE `user` (
  `user_id` int NOT NULL AUTO_INCREMENT,
  `name` varchar(255) NOT NULL,
  `email` varchar(255) NOT NULL,
  `phone` varchar(255) NOT NULL,
  `datemade` date DEFAULT NULL,
  `datelastupdated` date DEFAULT NULL,
  PRIMARY KEY (`user_id`)
) ENGINE=InnoDB AUTO_INCREMENT=1 DEFAULT CHARSET=utf8mb4i;
```

B Top & Grep Commands

```
top -d 0,1 | grep --line-buffered node >> node.txt
top -d 0,1 | grep --line-buffered dotnet >> dotnet.txt
```

C Faker.js App

```
const faker = require('faker')
const DbHandler = require('./dbhandler')

var db = new DbHandler()
faker.seed(1)

const iterations = 1001;

//create 1000 users & categories
for(i = 1; i < iterations; i++){
    console.log("Loop " + i)

    var name = faker.internet.userName()
    var email = faker.internet.email()
    var phone = faker.phone.phoneNumber()
    var datemade = faker.date.past()
    var lastupdated = faker.date.future()

    db.insertUser(name, email, phone, datemade, lastupdated,
     ↪  function(result){

    })

    var categoryName = faker.lorem.word()
    var description = faker.lorem.words()
    var datemade = faker.date.past()
    var lastupdated = faker.date.future()

    db.insertCategory(categoryName, description, datemade,
     ↪  lastupdated, function(result){

    })

}

//for each category, insert 1 thread
for(categoryId = 1; categoryId < iterations; categoryId++){
    var userId = 1000
    var title = faker.lorem.word()
    var body = faker.lorem.words()
    var datemade = faker.date.past()
    var lastupdated = faker.date.future()
```

```
    db.insertThread(categoryId, userId, title, body, datemade,
     ↪  lastupdated, function(result){

    })
    userId-=1
}

//For each user, insert 200 comments in threads 1-200
for(userId = 1; userId < iterations; userId++){
    for(threadId = 1; threadId < 201; threadId++){
    var body = faker.lorem.words()
    var datemade = faker.date.past()
    var lastupdated = faker.date.future()
        db.insertComment(threadId, userId, body, datemade,
         ↪  lastupdated, function(result){

        })
    }
}
```

D Application Repositories

https://github.com/oliverkarlsson59/bachelor-aspnet
https://github.com/oliverkarlsson59/bachelor-express