Ida Kilpeläinen

# Value of WebAssembly in Full-Stack Applications

Metropolia University of Applied Sciences

Bachelor of Engineering

Information and Communication Technology

Bachelor’s Thesis

14 May 2021

## Abstract

| | |
|---|---|
| Author: | Ida Kilpeläinen |
| Title: | Value of WebAssembly in Full-Stack Applications |
| Number of Pages: | 64 pages + 4 appendices |
| Date: | 14 May 2021 |
| Degree: | Bachelor of Engineering |
| Degree Programme: | Information and Communication Technology |
| Professional Major: | Software Engineering |
| Instructors: | Tommi Lundell, R&D Squad Group Leader<br>Juha Kämäri, Lecturer |

WebAssembly is a new web standard designed by The World Wide Web Consortium. WebAssembly is another type of code that can be run in browsers, besides JavaScript. It can be compiled from high-level programming languages such as C/C++, C#, and Rust. The main goal of WebAssembly is to increase the performance of web applications running in browsers, nearing native speeds.

This study was done for Nokia Solutions and Networks, a business unit of Nokia Corporation. The purpose of this thesis is to give an overview of WebAssembly as a standard and how it could be utilised for better performance in web applications. This is done by introducing the usage of WebAssembly, informing of the web applications it has already been used in and presenting a study made for this thesis. The application created to study the possible performance increase is a full-stack application developed with React.js and Node.js. This not only enables the studying of performance, but also allows the studying of the ease of use of WebAssembly modules in a React application.

The function execution times were measured with two variables: One for the actual time it takes to finish a function and other for all the excess time surrounding the call of the function, such as data pre-processing. These were measured programmatically. Also, Performance Tools provided by browsers were observed to see the call stack of the functions.

The initial hypothesis was that WebAssembly would indeed increase the performance of the created application. The results suggest otherwise. Even though the functions themselves did finish faster, the excess time in WebAssembly functions was remarkable, making the total time greater than of JavaScript functions. The main reason for this is the wrong kind of application WebAssembly was implemented to.

The results of this study show that WebAssembly as it is, is not applicable in all kinds of applications. The application it is to be implemented in should be chosen with care. WebAssembly is only at its first version and needs to be developed to get the best possible outcome.

Keywords: WebAssembly, JavaScript, Rust, machine language

## Tiivistelmä

| | |
|---|---|
| Tekijä: | Ida Kilpeläinen |
| Otsikko: | WebAssemblyn tuoma hyöty full stack -sovelluksessa |
| Sivumäärä: | 64 sivua + 4 liitettä |
| Aika: | 14.5.2021 |
| | |
| Tutkinto: | Insinööri (AMK) |
| Tutkinto-ohjelma: | Tieto- ja viestintätekniikka |
| Ammatillinen pääaine: | Ohjelmistotuotanto |
| Ohjaajat: | Tutkimus ja tuotekehitys –tiimin johtaja Tommi Lundell<br>Lehtori Juha Kämäri |

WebAssembly on uusi web-standardi, konekieli, joka mahdollistaa muiden ohjelmointikielien kuin JavaScriptin ajamisen selaimessa. WebAssembly-koodi voidaan kääntää lukuisista ohjelmointikielistä, kuten C/C++, C# ja Rust. Sen päällimmäinen tarkoitus on parantaa web-sovellusten suorituskykyä. WebAssembly-ohjelman suoritusnopeus voi olla jopa yhtä hyvä kuin perinteisten käännettyjen ohjelmien.

Tämä tutkimus on tehty Nokia Solutions and Networks -yritykselle, joka on osa Nokia konsernia. Opinnäytetyön tarkoitus on esitellä WebAssembly-standardi ja kuinka sitä voitaisiin hyödyntää web-sovellusten nopeuttamisessa. WebAssemblyn käyttöä ja sitä jo käyttäviä sovelluksia tuodaan esille. Tutkimusta varten kehitettiin full stack -websovellus React.js sovelluskehyksellä ja Node.js ajoympäristöllä, jotta samalla voitaisiin tutkia jo luodun WebAssembly moduulin käytön helppoutta React-sovelluksessa.

Funktioiden suoritusaika mitattiin kahdella eri muuttujalla: Pelkän funktion suorittamiseen menevä aika ja funktiokutsun ympärillä suoritettavien prosessien, kuten datan esiprosessointiin kulunut aika. Nämä ajat mitattiin ohjelmallisesti. Myös selainten tarjoamia analyysityökaluja, joista voidaan nähdä esimerkiksi funktion kutsupino, käytettiin tulosten selkeyttämiseksi.

Alustava hypoteesi oli, että WebAssembly lisäisi luodun sovelluksen tehokkuutta. Tuloksista kuitenkin ilmeni, ettei tämä pitänyt paikkaansa. Vaikka WebAssembly-funktioiden suorittaminen itsessään oli nopeampaa, funktiokutsuun tarvittavat prosessit veivät huomattavasti aikaa. Tämän takia WebAssembly-funktioiden kokonaisaika oli suurempi kuin JavaScript-funktioiden. Pääsyy tähän on se, että luotu sovellus oli vääränlainen WebAssembly-implementaatiolle.

Tutkimuksen tulokset näyttävät, ettei WebAssembly tällaisenaan ole sopiva kaikenlaisiin sovelluksiin. Sovellus, johon WebAssembly halutaan yhdistää pitää valita tarkasti. Nykyinen versio WebAssemblystä on vasta ensimmäinen ja sitä pitää jatkokehittää parhaimman tuloksen saavuttamiseksi.

Avainsanat: WebAssembly, JavaScript, Rust, konekieli

## Contents

List of Abbreviations

1 Introduction 1

2 Compilers, Interpreters and Assembly 2

2.1 Introduction to Compilers and Interpreters 2
2.2 Just-In-Time Compiling in Browsers 4
2.3 Assembly Language and WebAssembly in Compilation Process 6
2.4 Target Languages for Web; asm.js vs Wasm 8

3 WebAssembly Web Standard 9

3.1 WebAssembly Core Specifications Overview 9
3.1.1 Design Goals of WebAssembly 10
3.1.2 WebAssembly Concepts 11
3.2 Security Concerns Surrounding WebAssembly 13

4 WebAssembly in Practice 15

4.1 Embedding Interfaces 15
4.1.1 JavaScript API and Web API 15
4.1.2 WebAssembly System Interface (WASI) 19
4.2 Web Frameworks Working with WebAssembly 20
4.3 Web Implementations Enabled by WebAssembly 24

5 Project 27

5.1 Application 28
5.2 Project Building 32
5.2.1 Full-Stack Application with React.js and Node.js 32
5.2.2 MariaDB and Running Database Server in Docker 33
5.2.3 Rust and Tools: wasm-pack and wasm-bindgen 35
5.2.4 Svelte and Usage with React 37
5.3 Data Manipulation Functions 38
5.3.1 Sorting Data 38
5.3.2 Filtering Data by Key 42
5.3.3 Filtering Data by Map Bounds 45
5.4 Improvements and Problems 47

5.5 Project Results 48

6 Summary and Conclusion 56

References 58

Appendices

Appendix 1: React application configurations for WebAssembly modules

Appendix 2: Running MariaDB server inside a Docker container with Docker-compose

Appendix 3: Sequence Diagram of Fetching in the Application

Appendix 4: Sequence Diagram of Sorting in the Application

## List of Abbreviations

| | |
|---|---|
| API: | Application Programming Interface. A set of definitions and protocols for interactions between multiple software applications. |
| CLI: | Command Line Interface. Command line program for operating system functions. |
| CPU: | Central Processing Unit. A portion of a computer that retrieves and executes instructions. |
| DOM: | Document Object Model. A programming interface for HTML documents. |
| HTLM: | Hypertext Markup Language. Standard markup language for Web pages. |
| HTTP: | Hypertext Transfer Protocol. An Application-layer protocol for transmitting hypermedia documents e.g. HTML. |
| IDE: | Integrated Developed Environment. A software for building applications. |
| IR: | Intermediate Representation. The data structure or code used internally by a compiler or virtual machine to represent source code. |
| ISA: | Instruction Set Architecture. An abstract model of a computer. |
| JIT: | Just-In-Time. Used for example with Just-In-Time compiling. |
| JS: | JavaScript. Programming language for Web applications. |
| JSON: | JavaScript Object Notation. Format for storing and transporting data. |

| | |
|---|---|
| NPM: | Node Package Manager. Package manager, allowing the installing of libraries etc., for applications using JavaScript. |
| POSIX: | The Portable Operating System Interface. A set of formal descriptions that provide a standard for the design of operating systems. |
| QML: | Query Markup Language. User interface markup language. Similar to CSS and JSON. |
| UI: | User Interface. The point where human interacts with the computer. |
| URL: | Uniform Resource Identifier. Unique address of a web resource. |
| SQL: | Structured Query Language. Standard programming language for relational databases. |
| WASI: | WebAssembly System Interface. Defines a system interface which provides access to the underlying operating system. |
| YAML: | Acronym from "YAML Ain't Markup Language. A Human-readable data-serialisation language. |

# 1 Introduction

JavaScript was for a long time the only programming language runnable on browsers, but this changed in 2017. This is when a new web standard, WebAssembly, was introduced. WebAssembly was created to get better performance on the web for cases such as video games, video editing, 3D rendering and music production.

This thesis was done for Nokia Solutions and Networks, a business unit of Nokia Corporation. The main focus was to study the possible performance increase that can be gained by using WebAssembly in a web application sorting a large dataset. A full-stack application was created to observe the simplicity using a WebAssembly module in a ready application. This is to see if it is worth to change some heavy processing functionality in the existing web applications with a WebAssembly module.

WebAssembly is a machine language for a virtual machine. This may be a difficult concept to understand so the second chapter of this thesis introduces the process from source code to target code. This includes interpreters, compilers, and assembly, and where does WebAssembly fit in this process. The third chapter gives an overview on the design and concepts of WebAssembly. The following chapter explains the usage of WebAssembly in JavaScript, non-web implementations and standalone runtimes. Also, ready-made frameworks and web applications using WebAssembly are presented.

The user interface and building blocks of the application developed for the present study will be introduced in Chapter 5, together with describing the implemented data manipulation functions, sorting, and filtering, and the functions behind them. Possible improvements and encountered challenges are discussed after that. Chapter 6 presents the results found with visualisations. The final chapter summarises the study, what was learned about WebAssembly and its usage, and the conclusion to the question: Should WebAssembly replace some functionalities that require high performance?

# 2 Compilers, Interpreters and Assembly

In order to comprehend the necessity of WebAssembly, it is imperative to understand how written code is inputted to the computer. This chapter describes the process.

## 2.1 Introduction to Compilers and Interpreters

Interpreters and compilers are programs designed to translate high-level programming languages into machine languages. These two programs do it in different ways: Interpreters translate the code line-by-line and compilers convert it into object code and store it. [1]

Compiling and executing a program takes time. Nevertheless, after that the program runs efficiently since it has already been translated. Examples of different compiler types are:

- Source-to-source compiler (transpiler). Translates one high-level language to another, for example Emscripten (C/C++ to JavaScript).
- Bytecode compiler. Translates a high-level language into an intermediate representation.
- Assembler. Translates human-readable assembly language into machine code. [1]

An interpreter performs the same work as a compiler (Figure 1) but since an interpreter does not have to translate the whole program at once, it is quick to get up and running. Examples of different types of interpreters are:

- Bytecode interpreter. Translates a high-level language into bytecode.
- Threaded code interpreter. Like bytecode interpreter but uses pointers instead of bytes.

  Abstract syntax tree interpreter. Transforms the code into an optimised abstract syntax tree. [1]

The various stages of the compiling and interpreting processes are shown in the figure below. These stages can be grouped into two phases: analysis phase and synthesis phase.

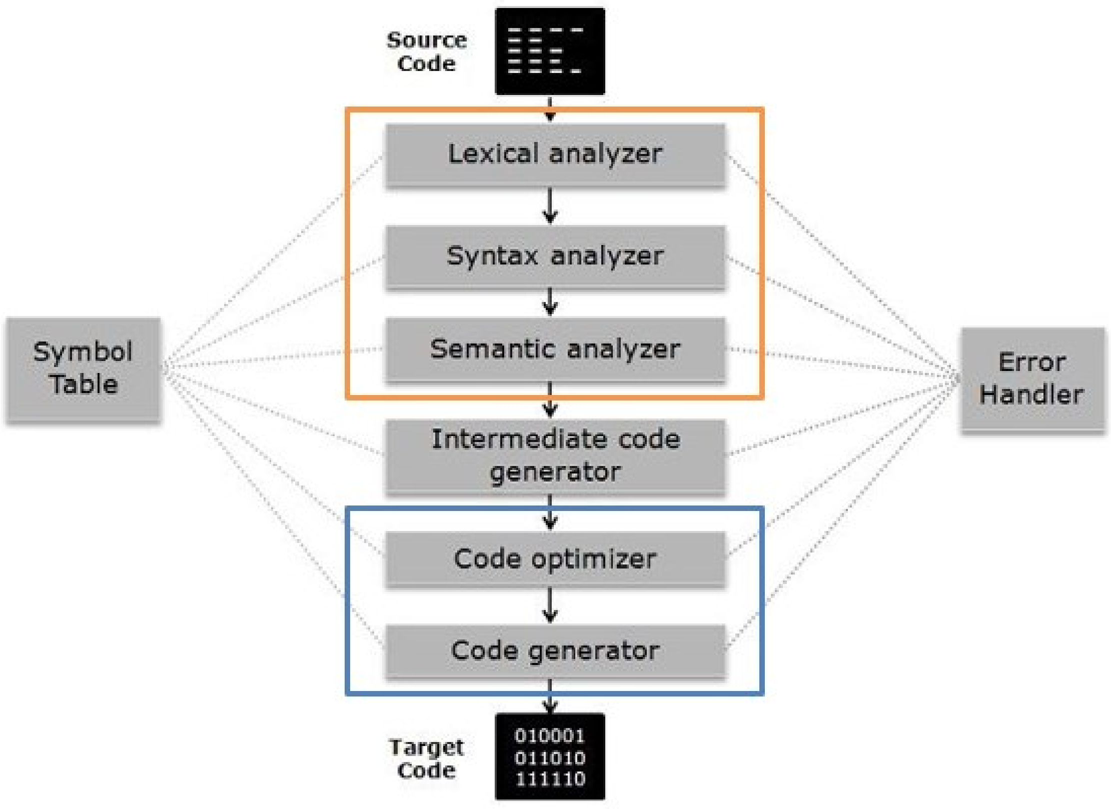


Figure 1. Phases of compiler. [2]

The analysis phase consists of feeding code to a lexical analyser, a semantic analyser, and a syntax analyser. The code optimiser and generator stages belong to the synthesis phase. After the analysis phase, the syntax and semantics of the code have been checked against the grammar and rules of the programming language. The intermediate code is generated and, in the synthesis phase the redundant code is eliminated and the target code for a particular machine is created. Table 1 below shows the comparison between a compiler and an interpreter in different areas. [2]

Table 1. Comparison between a compiler and an interpreter.

| Comparison | Compiler | Interpreter |
|---|---|---|
| Input | Entire program | Single line of code or instruction |
| Output | Intermediate object code. | No intermediate object code |
| Working mechanism | The program is compiled before execution | Program is compiled and executed simultaneously |
| Speed | Faster at runtime | Slower at runtime |
| Memory | Requires more memory | Requires less memory |
| Errors | Display errors after compilation | Display error of each line one by one |
| Programming languages | C, C++, C#, Typescript | PHP, Python |

As is shown in Table 1 above, the differences are major, and a compiler seems like the logical solution for being more effective. Still, an interpreter is more convenient during program development because it is fast to execute and debugging is usually easier, since the errors contain references to the code lines that caused them. The first JavaScript engines used by browsers were interpreters, but since then they have been replaced by Just-In-Time compilers (JIT compilers). [2]

## 2.2 Just-In-Time Compiling in Browsers

Just-In-Time compilers attempt to combine the benefits of compilers and interpreters. JIT compilers in browsers work with a profiler, a new part added to the JavaScript engine around 2008. The profiler monitors and collects code execution data. JavaScript engine used by Chrome is Google’s V8 and Firefox uses Mozilla’s SpiderMonkey. [3; 4]

Just-In-Time compilers usually consist of a baseline compiler and an optimising compiler. When a piece of code is executed multiple times, the profiler marks that piece of code as warm. After a piece of code has become warm, it is moved to the baseline compiler. The baseline compiler creates a stub for this part of code. When a part of the code is very hot, the profiler will send it off to the optimizing compiler. This will create another, even faster, version of the code that will also be stored. Figure 2 below shows a representation of the V8 JavaScript engine's workflow. [4]

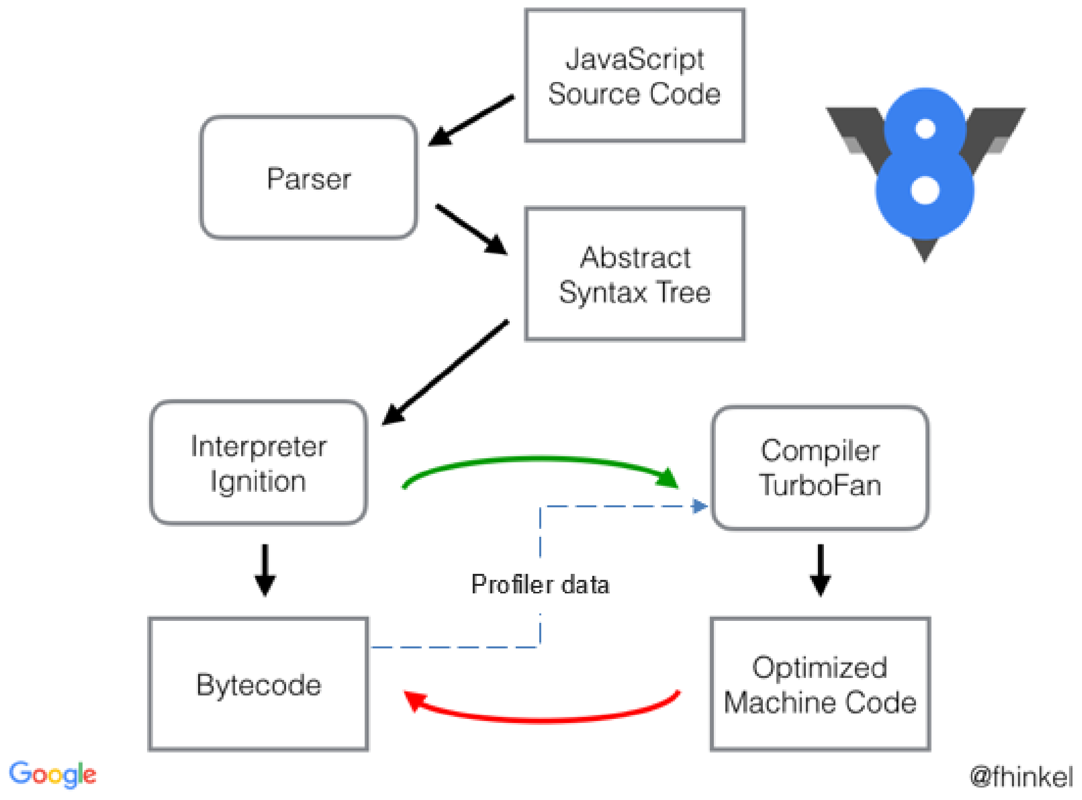
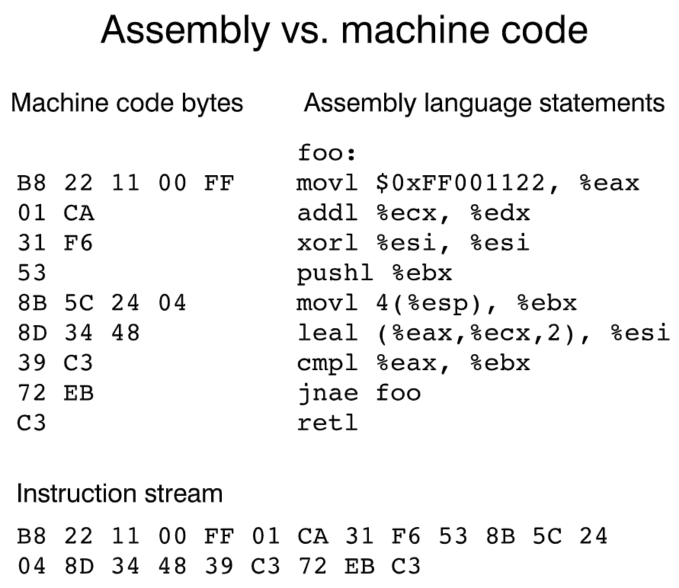


Figure 2. Workflow of the V8 JavaScript engine. [5]

As seen in Figure 2 above, first the parser generates an abstract syntax tree, a representation of the structure of the source code. Then the interpreter (which is called Ignition in V8) generates bytecode from the syntax tree. The generated bytecode is monitored by the profiler. Eventually the bytecode is given to the optimising compiler, TurboFan, which turns it into optimised machine code which is then executed by the computer. If the optimised machine code fails a dynamic check it must be deoptimized and transformed back to bytecode. [5]

## 2.3 Assembly Language and WebAssembly in Compilation Process

Assembly language is a low-level programming language designed for exactly one specific computer architecture. Assembly language instructions are written in symbolic code, which is more understandable for humans, compared to machine language's binary format. Below (Figure 3) is an example of the Intel's x86 instruction set architecture (ISA) and the corresponding instructions in hexadecimal. [6]

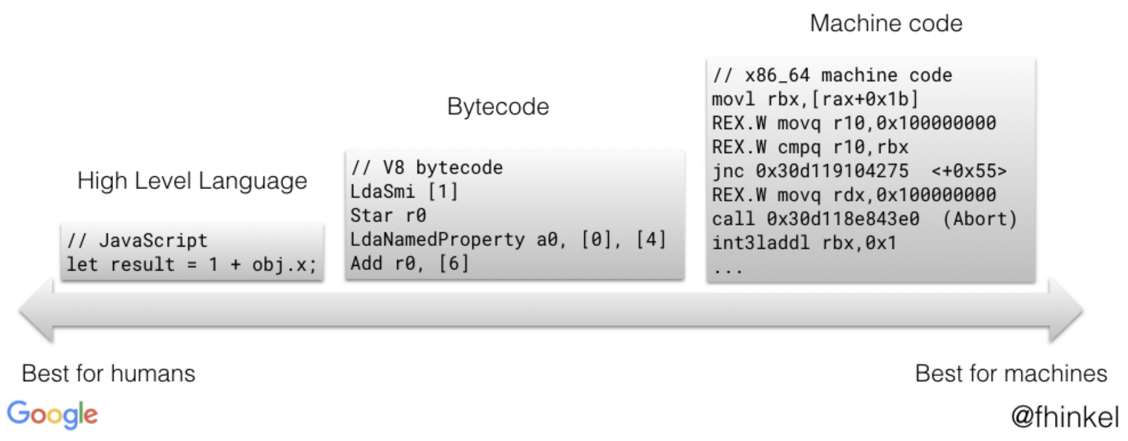


Assembly vs. machine code

```
Machine code bytes    Assembly language statements

                      foo:
B8 22 11 00 FF        movl $0xFF001122, %eax
01 CA                 addl %ecx, %edx
31 F6                 xorl %esi, %esi
53                    pushl %ebx
8B 5C 24 04           movl 4(%esp), %ebx
8D 34 48              leal (%eax,%ecx,2), %esi
39 C3                 cmpl %eax, %ebx
72 EB                 jnae foo
C3                    retl

Instruction stream

B8 22 11 00 FF 01 CA 31 F6 53 8B 5C 24
04 8D 34 48 39 C3 72 EB C3
```

Figure 3. Assembly vs. machine code [7]

Intermediate representation (IR) is a representation of a program, that is not high-level nor machine language but an abstract machine language. Without IR, each high-level language would need their own translator for each of the assembly languages. To solve this without creating several different translators, compilers add an IR layer to the compiling process. Bytecode is an example of an IR and it is used in the V8 JavaScript engine (Figure 4). [8]



```
// JavaScript
let result = 1 + obj.x;
```

```
// V8 bytecode
LdaSmi [1]
Star r0
LdaNamedProperty a0, [0], [4]
Add r0, [6]
```

```
// x86_64 machine code
movl rbx,[rax+0x1b]
REX.W movq r10,0x100000000
REX.W cmpq r10,rbx
jnc 0x30d119104275   <+0x55>
REX.W movq rdx,0x100000000
call 0x30d118e843e0  (Abort)
int3laddl rbx,0x1
...
```

Figure 4. V8 bytecode [5]

WebAssembly (Wasm) brings another block to the chain (Figure 5). Delivering code to be executed on the end user's machine across the web is difficult, since the machine architecture may not be known. This is where WebAssembly comes in useful. It is a machine language for a virtual machine and its purpose is to compile code to real architectures. [9]

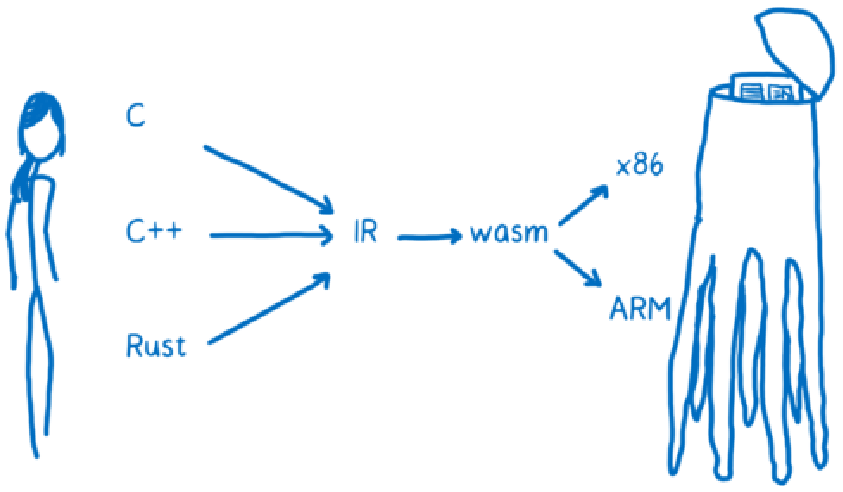


Figure 5. WebAssembly in the compiler process chain. [9]

The browser downloads the WebAssembly and now, the translation to the target machine's assembly code is more efficient by Wasm being closer to the assembly than for example JavaScript. The specifications of Wasm and in-depth information are discussed in Chapter 3. [9]

### 2.4 Target Languages for Web; asm.js vs Wasm

Asm.js and WebAssembly are both ways to generate code runnable on a browser. The development of asm.js was dismissed in 2014 and is now deprecated. So why does WebAssembly exist?

Asm.js is a strict subset of JavaScript, a set of definitions providing good performance characteristics. The main idea is to use JavaScript more strictly, making the program execution more effective. As seen in Figure 6 below, using asm.js did decrease the execution times but still using WebAssembly has benefits over asm.js. [10]

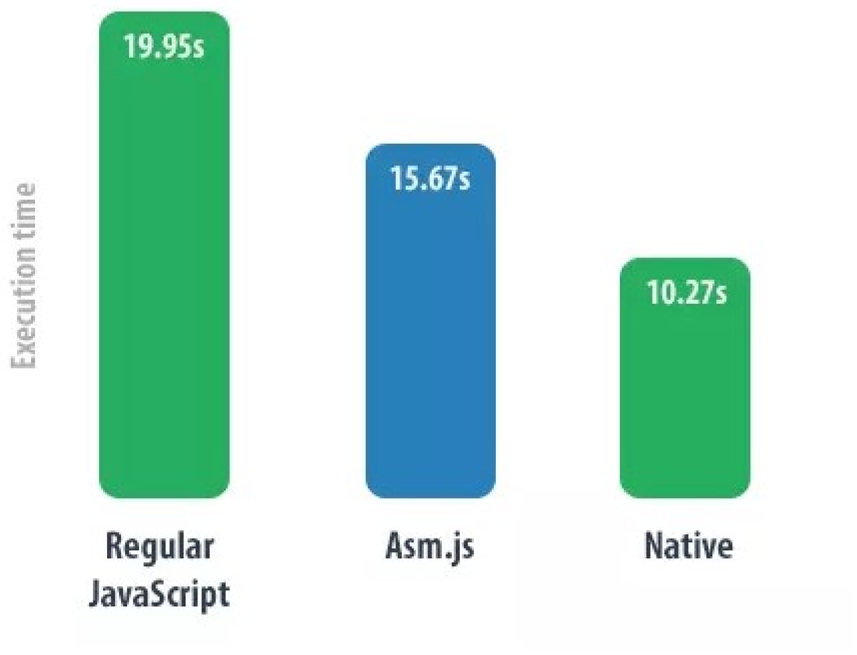


Figure 6. JavaScript vs asm.js vs C Fibonacci benchmarks. [10]

The benefits of WebAssembly are that Wasm binary format can be natively decoded much faster than JavaScript can be parsed, decreasing the loading time. Wasm also enables the use of more CPU features, such as 64-bit integers which can quadruple the speed of hashing and encryption algorithms. It has been designed together by all the major browsers, making it and its performance more predictable. In addition, Emscripten, a toolchain for generating Wasm, has been optimised and improved comparing to generating asm.js. [11]

# 3 WebAssembly Web Standard

Designed and maintained by The World Wide Web Consortium (W3C), the WebAssembly (Wasm) was announced an official web standard in December 2019. Wasm is a low-level code format, and its main goal is to enable the development of high-performance Web applications. [12]

Essentially it is a virtual instruction set architecture, an ISA specifies what the machine is capable of, it is an interface between the hardware and the software. This means that WebAssembly can be embedded into many different environments. [12; 13]

## 3.1 WebAssembly Core Specifications Overview

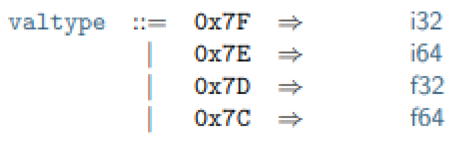
The core WebAssembly standard is described in “WebAssembly Specification” by WebAssembly Community Group. The newest release of the documentation was released in March 12, 2021. The documentation covers the structure, validation, and execution of the WebAssembly language, as well as in-depth look into the binary and text format. This section only scratches the surface of the documentation.

### 3.1.1 Design Goals of WebAssembly

The design goals of Wasm, defined in WebAssembly Specification are as follows: fast, safe, and portable semantics, also efficient and portable representation.

Wasm code is executed with near native code performance, in a memory-safe, sandboxed environment preventing data corruption or security breaches. It is hardware-, language-, and platform-independent, meaning it can be compiled on all modern architectures. [14, p. 1]

WebAssembly is comprised of two formats: a human-readable text format (.wat) and a binary format (.wasm). Both map to a common structure. For example, in the binary format the i32 value type is represented as hexadecimal 0x7F, which binary is 0111 1111 (Figure 7).



```
valtype  ::=  0x7F  ⇒      i32
           |  0x7E  ⇒      i64
           |  0x7D  ⇒      f32
           |  0x7C  ⇒      f64
```

Figure 7. Wasm value types in hexadecimal and text format. [14, p. 110]

The binary format is fast to transmit due to its compactness. Wasm code is modular so it can be divided into smaller parts which can be transmitted, cached, and consumed separately. It is efficient since it can be decoded, validated and compiled in single pass: The input is processed only once. The decoding, validating and compiling process can be started as soon as possible, before seeing all data. The beforementioned process can be split into many independent parallel tasks. WebAssembly code is also portable since it makes no architectural assumptions. [14, pp. 1-2]

### 3.1.2 WebAssembly Concepts

WebAssembly code formats are based around the following concepts:

- Values,
- instructions,
- traps,
- functions,
- tables,
- linear memory,
- modules,
- embedder.

WebAssembly provides four basic value types, integers, and floating numbers, each in 32- and 64-bit width. The 32-bit integers also serve as Booleans and memory addresses. Wasm code consists of sequences of instructions. The instructions manipulate values and are executed in order. Instructions fall into two main categories: simple and control instructions. Simple instructions perform basic operations on data and control instructions alter the control flow. Certain instructions may produce a trap, this is a way to abort execution and the traps are reported to the outside environment. [14, pp. 3-4]

A table in Wasm instructions is an array containing references to functions. The table allows emulating function pointers while not revealing the references' memory addresses. The Wasm module can call these functions by passing the index to an "call_indirect" operation that only then calls the function. In the code example below (Listing 1), a function pointer has been made of function "add" and it is called in "pointer_test" function with "call_indirect". The information after the “elem” keyword describes that the “add” function resides in the first index of the table. [15]

```
(func $add (type $t0) (param $p0 i32) (result i32)
     get_local $p0
     i32.const 2
     i32.add)
(func $invoke_function_pointer (export "pointer_test) (type $t2) (result i32)
     i32.const 10
     i32.const 0
     i32.load offset=1024
     call_indirect (type $t0))
(table $T0 2 2 anyfunc)
(elem (i32.const 1) $add)
```

Listing 1. Function pointers and call_indirect in WebAssembly.

WebAssembly has a linear memory which is contiguous, mutable array of bytes. The memory is initialised with a specific size, but it can be grown dynamically. A program using the WebAssembly module utilizes the linear memory to load and store values at any byte address. The memory is accessed with “load” and “store” instructions. [14, pp. 3-4]

In the code block below (Listing 2) is a C program that can simply set and get the value of a global variable and below it is the same code in Wasm. On the last line of the Wasm code is the definition of a global variable, following its type (i32) and memory address (1024) and the initial value is zero. In the “setVariable” function, “i32.store offset=1024” stores the value given as a parameter to memory address 1024. Whereas the “getVariable” function loads the value from memory address 1024.

```
// C code
int variable;

WASM_EXPORT
void setVariable(int param) {
     variable = param;
};

WASM_EXPORT
int getVariable() {
     return variable;
};

// Wasm code
(module
     (type $t0 (func))
     (type $t1 (func (param i32)))
     (type $t2 (func (result i32)))
     (func $setVariable (export "setVariable") (type $t1) (param $p0 i32)
          i32.const 0
          get_local $p0
          i32.store offset=1024)
     (func $getVariable (export "getVariable") (type $t2) (result i32)
          i32.const 0
          i32.load offset=1024)
     (memory $memory (export "memory") 2)
     (data (i32.const 1024) "\00\00\00\00"))
```

Listing 2. WebAssembly memory load and store.

WebAssembly binary appears as a module that consists of definitions for functions, tables, linear memories, and global variables. One module per file, example of a module in listing above. An implementation of WebAssembly is typically embedded into a host environment. More on embedding is covered in Chapter 4. [14, p. 4]

## 3.2 Security Concerns Surrounding WebAssembly

Since WebAssembly executes in a sandboxed environment, it is separated from the host runtime and the sandbox cannot be escaped without using appropriate application programming interfaces (API). Each Wasm module is accountable to its embedding’s security policies: Same-origin policy within web browser and POSIX (The Portable Operating System Interface) security model on a non-web platform. These features enhance the security of a WebAssembly application. [16 ]

> POSIX.1-2008 defines a standard operating system interface and environment, including a command interpreter (or “shell”), and

common utility programs to support applications portability at the source code level. [17]

There has been some concern surrounding the memory model of WebAsssembly. This has been tried to take into consideration by isolating memory addresses by bounds of the ArrayBuffer providing a boundary [16]. The block of memory is fully contiguous, this means that every pointer between 0 and maximum memory is valid, whereas the memory of a natively run program has many gaps, unmapped pages. If a malicious program tries to read or write to an unmapped page, the malicious program halts. Native programs also use address space layout randomisation for making it harder for an attacker to target a particular memory address as seen in Figure 8 below. Whereas the memory layout of WebAssembly can be deduced from the compiler and program.

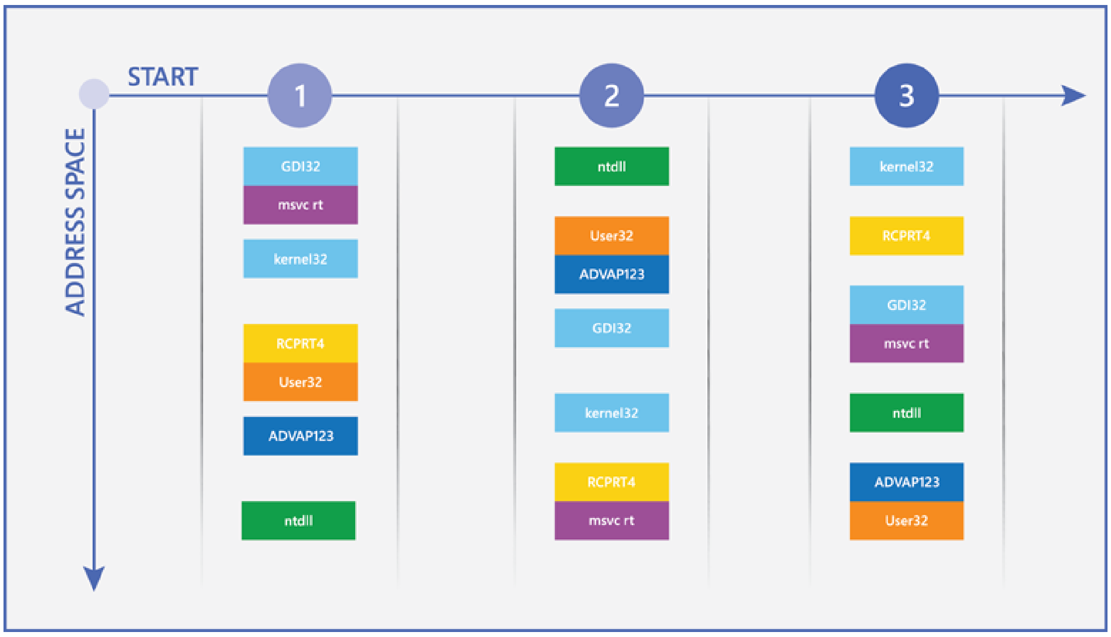


Figure 8. Example of an address space layout randomisation. [18]

There are numerous security concerns, addressed in a research paper “Everything Old is New Again: Binary Security of WebAssembly”. The conclusion of the paper is as follows:

> Overall, our findings are a call to arms for further hardening the WebAssembly language, its compilers, and ecosystem, making the promise of a secure platform a reality. [20]

This thesis does not go into too much detail in the issue.

# 4 WebAssembly in Practice

This chapter introduces different ways of using WebAssembly. Either with embedding interfaces or ready-made frameworks. The final section presents applications running in production made with or ported into WebAssembly.

## 4.1 Embedding Interfaces

WebAssembly has three different embedding interfaces: JavaScript API for accessing Wasm, Web API for expanding the JavaScript API for broader use and WASI API for non-web use-cases. [21]

### 4.1.1 JavaScript API and Web API

The JavaScript API enables accessing WebAssembly from within JavaScript. The API defines JavaScript classes and objects, including methods for validation, compilation, and instantiation, some of these can be seen in Table 2 below. Also, classes for representing and manipulating imports and exports are provided. WebAssembly also provides a Web API which expands the JavaScript API, this defines for example the “initiateStreaming” method (Table 2). The Web API is designed to integrate WebAssembly with the broader web platform. [21; 22]

Table 2. Wasm JavaScript integration, most used references.

| Object or Function | Description |
|---|---|
| Global | A global variable, accessible from Wasm and JS. |

| | |
|---|---|
| Module | Contains stateless Wasm code, has already been compiled. |
| Instance | Stateful, executable instance of a Module. |
| instantiateStreaming | Function for compiling an instantiating Wasm code. |
| Memory | Resizable ArrayBuffer, holding raw bytes of memory. |
| Table | Resizable typed array of function references. |

First the loading of a Wasm module into JavaScript is explained with examples. In the C program below, the function simply returns the given integer. The same function in WebAssembly text format can be found below in the C program (Listing 3).

```
// C code
int main(int value) {
  return value;
}

// Wasm code
(module
  (type $t0 (func (param i32) (result i32)))
  (func $main (export "main") (type $t0) (param $p0 i32) (result i32)
    get_local $p0))
```

Listing 3. Simple C and Wasm programs, returning the given integer.

The created WebAssembly module can then be loaded into JavaScript with the “WebAssembly.instantiateStreaming” function. This is the most efficient and optimised way to load Wasm code [23]. The code example below simply prints the given value, which is returned from the WebAssembly function, in this case the integer “123”.

```
WebAssembly.instantiateStreaming(fetch('simple.wasm')
).then(obj =>
      console.log(obj.instance.exports.main(123)      // print 123
));
```

Listing 4. Load Wasm module into JavaScript with “instantiateStreaming” and print the returned value.

A WebAssembly module is directly compiled and instantiated from a streamed underlying source with the “instantiateStreaming” function. If it is not possible to use the streaming methods, the module can be loaded without streaming with “WebAssembly.compile” or “WebAssembly.instantiate” functions (Listing 5).

```
fetch('simple.wasm').then(response =>
     response.arrayBuffer()
).then(bytes =>
     WebAssembly.instantiate(bytes)
).then(results => {
     console.log(results.instance.exports.main(123));
});
```

Listing 5. Loading of Wasm module with “instantiate” function

The non-streaming functions do not directly access the byte code. This means that the response must be turned into an ArrayBuffer before compiling or instantiating the Wasm module as seen in the code block above. [24]

The memory instance is a resizable ArrayBuffer or a SharedArrayBuffer and it is created with the “WebAssembly.Memory” constructor. The constructor’s parameters are initial size, a maximum size, and a shared property. WebAssembly pages are used as the units of initial and maximum properties as seen in Listing 6 below. One page is 64KB. [25]

```
let memory = new WebAssembly.Memory({initial:10, maximum:100, shared:false});
WebAssembly.instantiateStreaming(fetch(‘simple.wasm’)m {js: mem:memory}})
      .then(obj => {
            new Uint32Array(memory.buffer)[0] = 42;
            console.log(new Uint32Array(memory.buffer)[0]);   //print 42
            memory.grow(1);
});
```

Listing 6. WebAssembly memory in JavaScript

In the code block above (Listing 6), first a memory with size of 640KB is created. An integer is written directly into the first index of linear memory and it can be read from this index. The memory can be grown with “Memory.prototype.grow” function, where the argument is specified in WebAssembly pages, in the example the memory is grown with 640KB.

Tables are a way to store function references in the Wasm instructions. More information on tables can be found in Section 3.1.2 WebAssembly Concepts. The WebAssembly code below (Listing 7) defines a table with two elements: Function references to “thirteen” and “fortytwo”. These functions simply return the integers 13 or 42. The functions’ references can be fetched in JavaScript with “tbl.get(0)”. The second set of parentheses, as seen in the example code, invoke the function. [26]

```
(module
  (func $thirteen (result i32) (i32.const 13))
  (func $fortytwo (result i32) (i32.const 42))
  (table (export "table1") anyfunc (elem $thirteen $fortytwo))
)

let table1 = results.instance.exports.table1;
console.log(table1.get(0)());        // 13
console.log(table1.get(1)());        // 42

let table2 = new WebAssembly.Table({initial:1, element:"anyfunc"});
table2.set(0, table1.get(0));        //sets “thirteen” to table2
console.log(table2.get(0)();         // 13
```

Listing 7. WebAssembly tables in JavaScript.

The WebAssembly.Table is created with “element” and “initial” members. Element represents the type of the value to be stored in the table, where “anyfunc” is the only accepted value at the moment. “Initial” is the initial number of elements in the table. A table’s “set” function can set an exported WebAssembly function to a table, as seen in the code example above. [26]

Global variable instances are accessible from JavaScript and can be imported and exported across one or more WebAssembly module instances (Listing 8).

```
const global = new WebAssembly.Global({value:'i32', mutable:true}, 123);
console.log(global.value)            //123
global.value = 42;
console.log(global.value)            //42
```

Listing 8. WebAssembly globals in JavaScript

WebAssembly global variable instances can be created in JavaScript with the “WebAssembly.Global” constructor. The constructor takes two parameters, an object containing a value type (i32, i64, f32 or f64) and a Boolean value determining the mutability of the variable, and the initial value of the global. The object's "value" property can be used to get and set the value of the global variable. [27]

### 4.1.2 WebAssembly System Interface (WASI)

WASI is a family of APIs that define a system interface which provides access to the underlying operating system e.g. the usage of files and networking [28]. Figure 9 below describes the architecture of WASI API. WASI Libc is a C standard library for Wasm programs. It is built on top of WASI system calls and it provides POSIX-compatible C language APIs such as filesystem manipulation, memory management and time. [29]

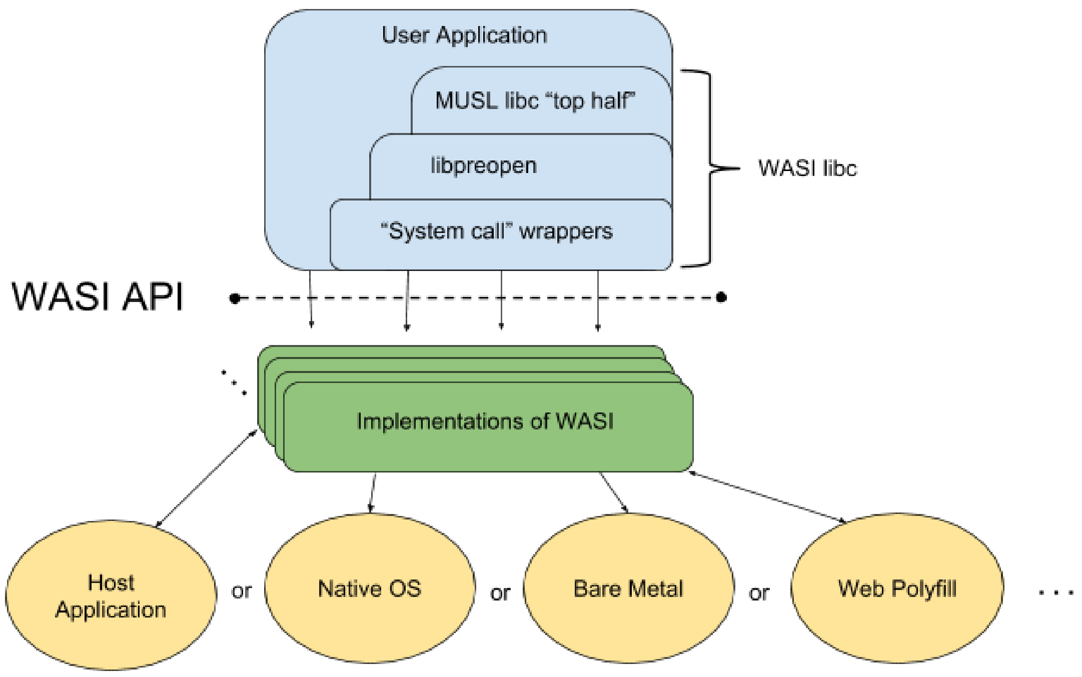


Figure 9. WASI Software Architecture. [28]

WASI can be used with multiple engines, such as Node.js, Wasmer and Wasmtime. Wasmer and Wasmtime are standalone runtimes for WebAssembly. Wasmer can be embedded in different languages, such as Rust, C/C++, Go and Python, and be used as a library. Wasmer provides a package called

“@wasmer/wasi” for using WASI in JavaScript, Listing 9 below contains an example of this. [26]

```
const fs       = require("fs")
const { WASI } = require("@wasmer/wasi")
const nodeBindings = require("@wasmer/wasi/lib/bindings/node")

const wasmFilePath = "./helloworld.wasm"

let wasi = new WASI({          // Instantiate a new WASI Instance
      args: [wasmFilePath],
      env: {},
      bindings: {
            ...(nodeBindings.default || nodeBindings),
            fs: fs
      }
})
let bytes = new Uint8Array(fs.readFileSync(pathToWasmFile)).buffer

let wasmModule = await WebAssembly.compile(wasmBytes);
let instance = await WebAssembly.instantiate(wasmModule, {
      ...wasi.getImports(wasmModule)
      });

wasi.start(instance)           // Start the WASI instance
```

Listing 9. Using WASI in JavaScript with Wasmer. [31]

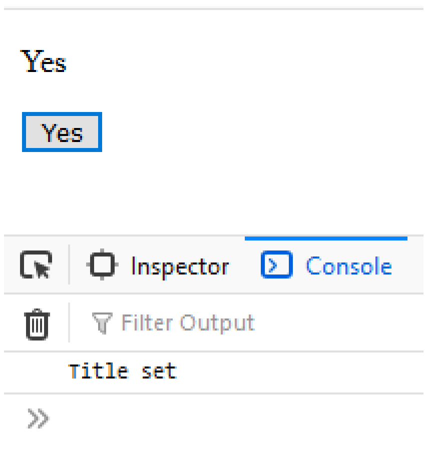
In the code above, a WASI instance is instantiated and after this, a Wasm instance is created. The exported WebAssembly function “_start” calls an imported WASI function that writes to the operating system’s standard output (stdout). Wasmer provides various features for users and developers. It supports multiple compilers, caching of compiled Wasm modules and monitoring of computation time and resources [32].

## 4.2 Web Frameworks Working with WebAssembly

There are different front-end frameworks that allow you to create web applications without having to write JavaScript, such frameworks are

- Yew, written with Rust,
- Blazor, written with C#,
- Qt, written with C++,
- Vugu, written with Go Language.

A simple example code is provided using all the frameworks. The sample application fills an html paragraph tag with the word “Yes” and prints “Title set” to console, after the button is clicked (Figure 10).



```
Title set
```

Figure 10. Simple sample web application.

Yew framework is written with Rust and it supports multi-threading and JavaScript npm packages [33]. Yew uses Wasm-pack and Wasm-bindgen for compiling, building, and using Wasm code [34]. More on these tools in Section 5.2.3. The sample code below (Listing 10) shows an enum, a struct (Listing 16) and an implementation of the struct. The “MyComponent” struct implements the “Component” trait which describes the lifecycle of a component: e.g. create, view and update methods.

```
struct MyComponent {
     link: ComponentLink<Self>,
     title: String,
}
enum Msg {
     SetTitleToYes,
}

impl Component for MyComponent {
     type Properties = ();
     type Message = Msg;
     fn create(_: Self::Properties, link: ComponentLink<Self>) -> Self {
          Self { link, title: "" }
     }
     fn update(&mut self, msg: Self::Message) -> ShouldRender {
          match msg {
               Msg::SetTitleToYes => {
                    self.title = "Yes"
                    ConsoleService::log("Title set");
               }
          }
          true
     }
     fn view(&self) -> Html {
          let setTitle_callback = self.link.callback(|_| Msg::SetTitleToYes);
          html! {
               <div>
                    <p>{ self.title }</p>
                    <button onclick=setTitle_callback>{ "Yes!" }</button>
               </div>
         }
    }
}
```

Listing 10. Sample code written with Yew framework.

Blazor is a feature of Microsoft's .net framework. Blazor applications are based on Razor components and the components have a file extension ".razor". Razor is a combination of HTML and C# as can be seen in the code example below (Listing 11). The @code block enables the use of C# fields, properties, and methods to a component. Blazor has two hosting models: WebAssembly hosting model and Server hosting model. With Blazor Wasm, the code runs in the browser on the client and with Blazor Server, the C# code runs on the server and SignalR library is used for passing information between browser and server. [35; 36]

```
<div>
     <p>@Title</p>
     <button @onclick="SetTitleToYes">Yes!</button>
</div>

@code {
    [Parameter]
    public string Title { get; set; }

    private void SetTitleToYes()
    {
          Title.set(“Yes”)
          Console.WriteLine("Title set”)
    }
}
```

Listing 11. Sample code written with Blazor framework.

Qt is a framework for creating graphical user interfaces and cross-platform applications, supporting platforms such as: Windows, Linux, macOS and mobile platforms [37]. For Qt, WebAssembly is just another target platform [38]. Qt provides different application frameworks such as Qt Quick and Qt Widgets [39]. Qt Quick is the standard library for writing QML applications. QML is a user interface specification and programming language and its syntax is similar to JSON. Qt Quick offers a set of controls, such as buttons, containers, and text fields. The sample code below (Listing 12) is only QML but usually C++ is also required for more complex applications. Qt uses Emscripten toolchain for compiling to WebAssembly. [40; 41]

```
Rectangle {
     Property alias settitle: title
     Button {
          onClicked: {
               item.settitle.text = “Yes”
               console.log(“Title set”);
          }
     }
     Label {
          id: title
          text: “”
     }
}
```

Listing 12. Sample code written in Qt Quick’s QML language.

Vugu is written in Go language and is inspired by the VueJS JavaScript framework. Official Vugu documentation describes Vugu more as a library rather than a framework. The files contain HTML, styling, and code sections (Listing 13) and the .vugu files can be used as components. Go code can be moved to a separate file and import the methods in the .vugu file. [42; 43]

```
<div>
     <p vg-content=”c.Title”> </p>
     <button @click='c.SetTitleToYes ()'>Yes!</button>
</div>

<script type="application/x-go">
import "log"

func (c *Root) SetTitleToYes () {
     c.Title = “Yes”
     log.Print(“Title set”)
}

type Root struct {
    Title string `vugu:"data"`
}
</script>
```

Listing 13. Sample code written in Vugu framework.

It is not guaranteed that the code samples work as-is, since some unnecessary code is removed for clarity and the code is not tested. The main point of this section is to give a small example of the technology already available for WebAssembly development and how it has enabled the use of other languages than JavaScript in web development.

## 4.3 Web Implementations Enabled by WebAssembly

WebAssembly has brought multiple desktop applications to the web. AutoCAD by Autodesk is a computer-aided design and drafting application. Google Earth (Figure 11) was first released as a native application because rendering required a lot of time since Google Earth renders the whole world in real-time. The applications were partially ported using the old code. This is notable: reutilizing the already written and constantly changing code to run in a browser. [44; 45]

Figure 11. Snapshot of Google Earth running in a browser.

Real-time audio manipulation on the web has been nearly impossible because of the latency caused by using JavaScript. Now WebAssembly has enabled the use of natively high-performance programming languages on the web and thus audio manipulation on the web is possible. A music production software “Soundation” (Figure 12) announced over 300% performance improvement with the implementation of WebAssembly Threads. [46]

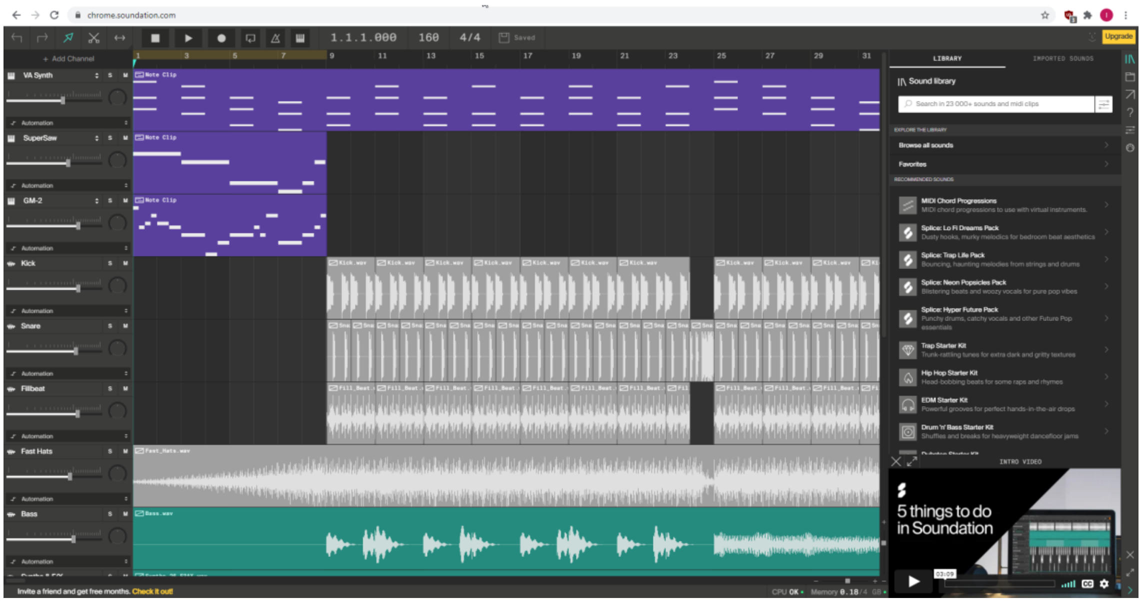
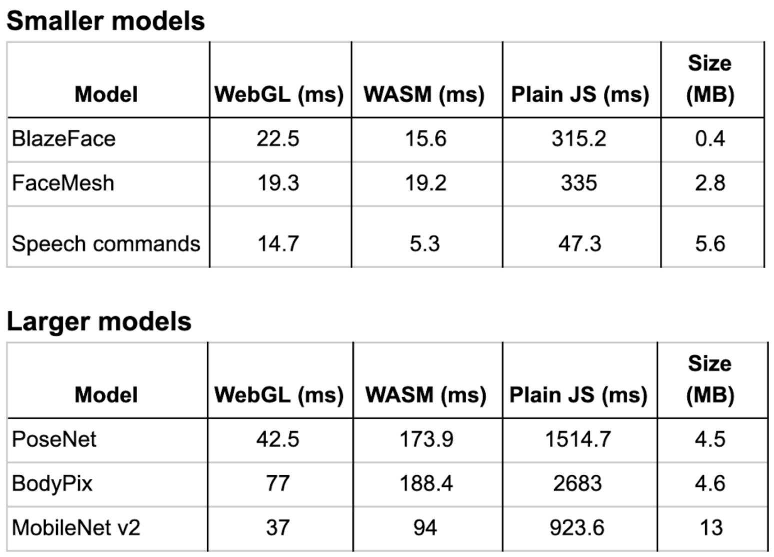


Figure 12. Snapshot of Soundation running in a browser.

Tensorflow is an open source platform for machine learning. WebAssembly is perfect for this kind of work because of the CPU intensive, or heavy processing tasks [47]. In Figure 13 below is a table of inference times from TensorFlow and one can see that using Wasm made the program 10-30 times faster than the plain JavaScript. It is still recommended to use WebGL for training models in the browser. [48]



**Smaller models**

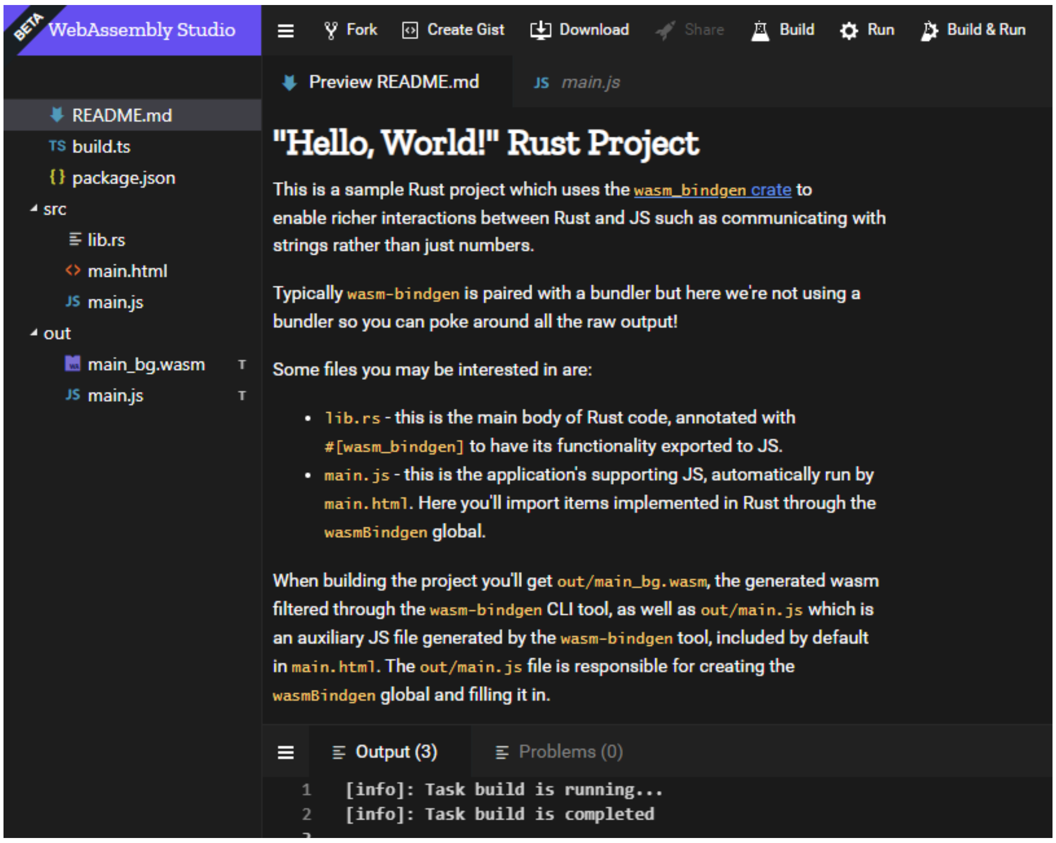
| Model | WebGL (ms) | WASM (ms) | Plain JS (ms) | Size (MB) |
|---|---|---|---|---|
| BlazeFace | 22.5 | 15.6 | 315.2 | 0.4 |
| FaceMesh | 19.3 | 19.2 | 335 | 2.8 |
| Speech commands | 14.7 | 5.3 | 47.3 | 5.6 |

**Larger models**

| Model | WebGL (ms) | WASM (ms) | Plain JS (ms) | Size (MB) |
|---|---|---|---|---|
| PoseNet | 42.5 | 173.9 | 1514.7 | 4.5 |
| BodyPix | 77 | 188.4 | 2683 | 4.6 |
| MobileNet v2 | 37 | 94 | 923.6 | 13 |

Figure 13. Inference times of TensorFlow using different models. [44]

A WebAssembly online playground called WebAssembly Studio is an online IDE (integrated development environment), compiling and running WebAssembly with Rust, C++, and C (Listing 14) [49]. It was also used when writing this thesis, helping with testing code examples.



"Hello, World!" Rust Project

This is a sample Rust project which uses the `wasm_bindgen` crate to enable richer interactions between Rust and JS such as communicating with strings rather than just numbers.

Typically `wasm-bindgen` is paired with a bundler but here we're not using a bundler so you can poke around all the raw output!

Some files you may be interested in are:

- `lib.rs` - this is the main body of Rust code, annotated with `#[wasm_bindgen]` to have its functionality exported to JS.
- `main.js` - this is the application's supporting JS, automatically run by `main.html`. Here you'll import items implemented in Rust through the `wasmBindgen` global.

When building the project you'll get `out/main_bg.wasm`, the generated wasm filtered through the `wasm-bindgen` CLI tool, as well as `out/main.js` which is an auxiliary JS file generated by the `wasm-bindgen` tool, included by default in `main.html`. The `out/main.js` file is responsible for creating the `wasmBindgen` global and filling it in.

```
[info]: Task build is running...
[info]: Task build is completed
```

Figure 14. Snapshot of WebAssembly studio.

WebAssembly has also enabled the development of web 3D games and a comprehensive list of different projects and applications using WebAssembly have been gathered to “Made with WebAssembly” website [50].

# 5 Project

The web application in question is a full stack application, and the main point is to measure the performance of different ways of sorting and filtering a data set of over one hundred thousand elements. The chosen implementation may seem overly complicated for simply measuring performance but the co-operation between a JavaScript (or React) application and WebAssembly is also analysed. The question is whether rewriting parts of web applications with WebAssembly gives a big enough performance boost to warrant the extra work.

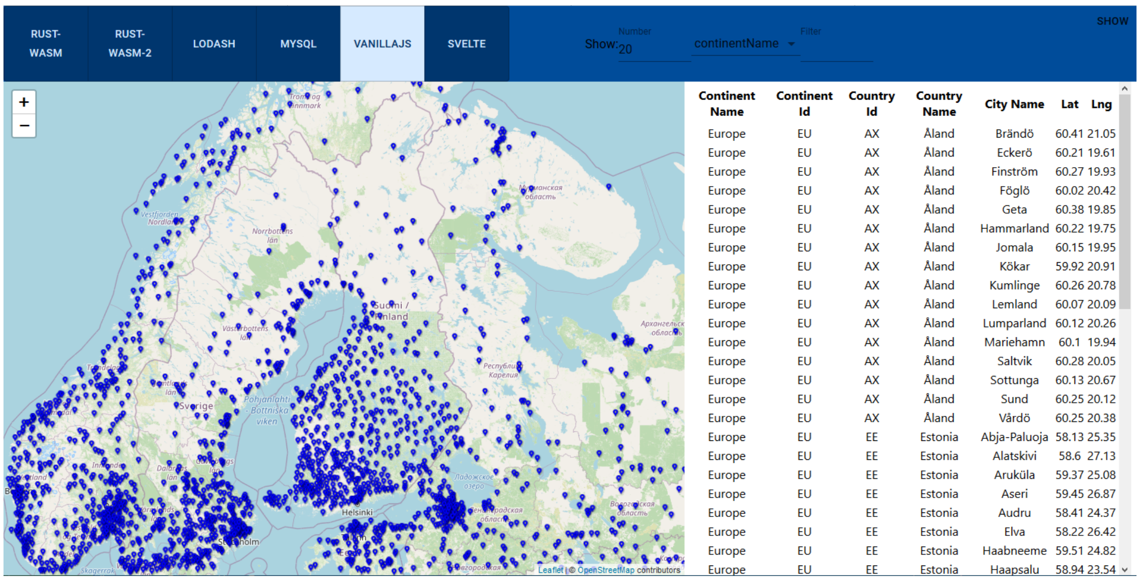
## 5.1 Application

This section introduces the main components of the application and visualises it with images. In this thesis, the ways of sorting and filtering the data are called “sorters”. All the sorters and short descriptions are shown in Table 3 below.

Table 3. Sorters and short descriptions.

| Sorter | Programming language | Description |
|---|---|---|
| rust_wasm | Rust to Wasm | Data is given as a function parameter. |
| rust_wasm_2 | Rust to Wasm | Data is stored in a global variable. |
| vanilla_js | JavaScript | Pure JavaScript with a custom compare function. |
| lodash | JavaScript | JS library that uses JavaScript’s built-in sort. |
| svelte | JavaScript | Pure JavaScript with a custom compare function. |
| sql | Structured Query Language | Uses SQL statements for data manipulation. |

The application consists of three main components: A map, a table and a navigation panel that can be seen in Figure 15 below. It is to be noted that due to lack of time, the styling of the user interface of the application was not a priority.



| Continent Name | Continent Id | Country Id | Country Name | City Name | Lat | Lng |
|---|---|---|---|---|---|---|
| Europe | EU | AX | Åland | Brändö | 60.41 | 21.05 |
| Europe | EU | AX | Åland | Eckerö | 60.21 | 19.61 |
| Europe | EU | AX | Åland | Finström | 60.27 | 19.93 |
| Europe | EU | AX | Åland | Föglö | 60.02 | 20.42 |
| Europe | EU | AX | Åland | Geta | 60.38 | 19.85 |
| Europe | EU | AX | Åland | Hammarland | 60.22 | 19.75 |
| Europe | EU | AX | Åland | Jomala | 60.15 | 19.95 |
| Europe | EU | AX | Åland | Kökar | 59.92 | 20.91 |
| Europe | EU | AX | Åland | Kumlinge | 60.26 | 20.78 |
| Europe | EU | AX | Åland | Lemland | 60.07 | 20.09 |
| Europe | EU | AX | Åland | Lumparland | 60.12 | 20.26 |
| Europe | EU | AX | Åland | Mariehamn | 60.1 | 19.94 |
| Europe | EU | AX | Åland | Saltvik | 60.28 | 20.05 |
| Europe | EU | AX | Åland | Sottunga | 60.13 | 20.67 |
| Europe | EU | AX | Åland | Sund | 60.25 | 20.12 |
| Europe | EU | AX | Åland | Vårdö | 60.25 | 20.38 |
| Europe | EU | EE | Estonia | Abja-Paluoja | 58.13 | 25.35 |
| Europe | EU | EE | Estonia | Alatskivi | 58.6 | 27.13 |
| Europe | EU | EE | Estonia | Aruküla | 59.37 | 25.08 |
| Europe | EU | EE | Estonia | Aseri | 59.45 | 26.87 |
| Europe | EU | EE | Estonia | Audru | 58.41 | 24.37 |
| Europe | EU | EE | Estonia | Elva | 58.22 | 26.42 |
| Europe | EU | EE | Estonia | Haabneeme | 59.51 | 24.82 |
| Europe | EU | EE | Estonia | Haapsalu | 58.94 | 23.54 |

Figure 15. Snapshot of the application.

The map shown in Figure 15 is populated with markers when a sorter is chosen in the navigation panel and the table of cities is filtered by the bounds of the map. The map, markers and table are updated when the map is moved. The navigation panel contains buttons for choosing the wanted sorter (Figure 15). Clicking one of these buttons calls a function to fetch the cities from the database. Depending on the chosen sorter, the fetching is either done straight from the React application or using a fetching function implemented in an external module as show in Figure 16 below. A more comprehensive diagram is shown in Appendix 3.

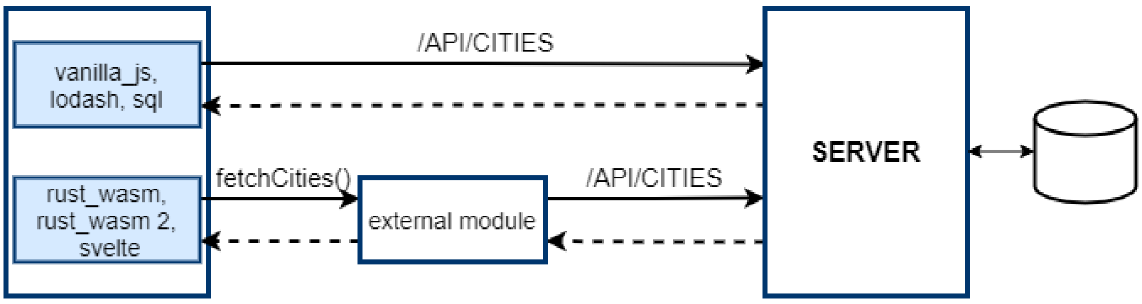


Figure 16. Diagram of fetching in the application.

The chosen sorter defines the used functions for sorting and filtering, a sequence diagram to visualise this is shown in Appendix 4. Figure 17 below shows a simpler diagram for sorting, depending on the chosen sorter.

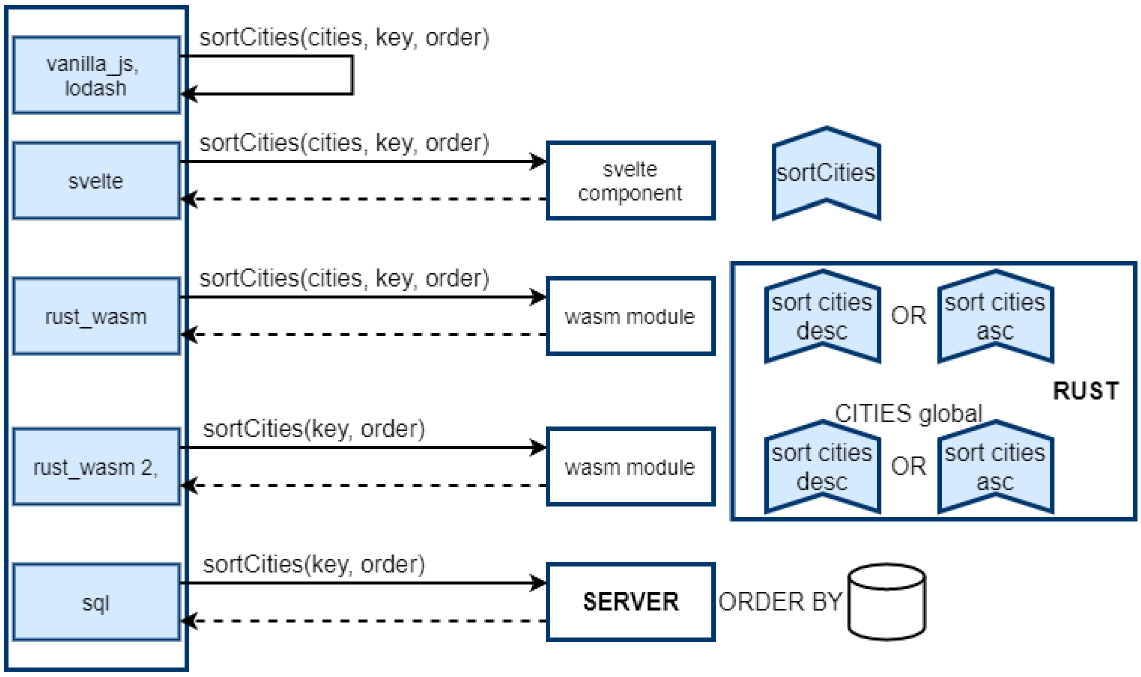
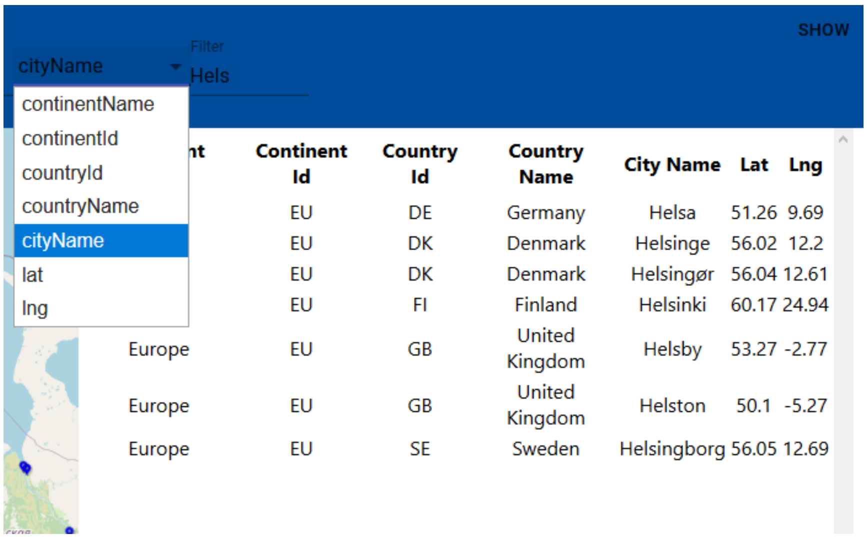


Figure 17. Diagram of sorting in the application.

The navigation panel contains means for filtering the table. First, choosing a key to be filtered by and then writing the wanted city, country etc. In the Figure 18, all the cities beginning with “Hels” is updated to the table.



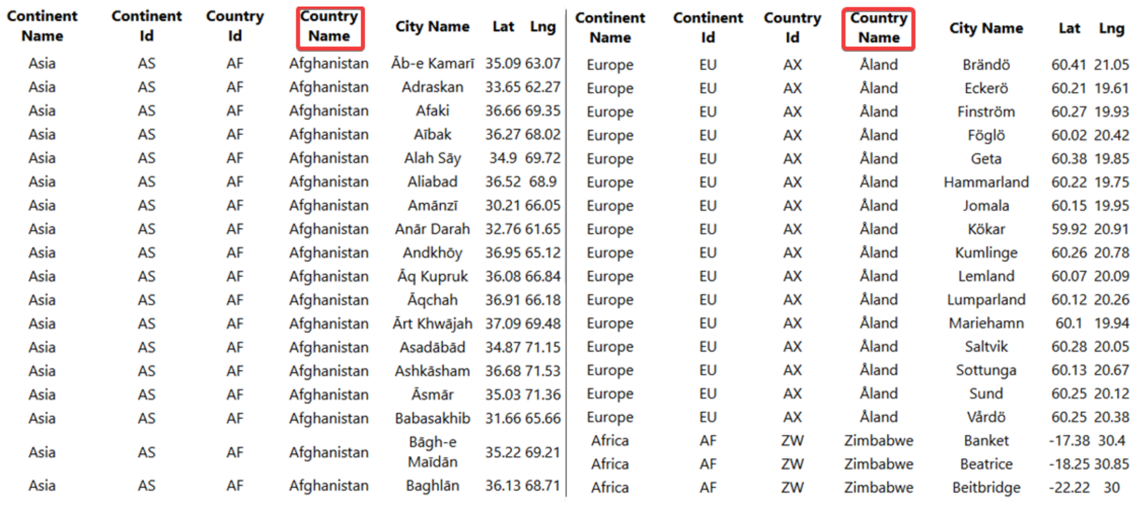
cityName

Filter: Hels

| nt | Continent Id | Country Id | Country Name | City Name | Lat | Lng |
|---|---|---|---|---|---|---|
| | EU | DE | Germany | Helsa | 51.26 | 9.69 |
| | EU | DK | Denmark | Helsinge | 56.02 | 12.2 |
| | EU | DK | Denmark | Helsingør | 56.04 | 12.61 |
| | EU | FI | Finland | Helsinki | 60.17 | 24.94 |
| Europe | EU | GB | United Kingdom | Helsby | 53.27 | -2.77 |
| Europe | EU | GB | United Kingdom | Helston | 50.1 | -5.27 |
| Europe | EU | SE | Sweden | Helsingborg | 56.05 | 12.69 |

Figure 18. Filtering the data by city name.

The whole list of cities is sorted by the clicked table header and the table is not populated only by the map bounds, until the map is updated (moved) again. Figure 19 shows the table sorted by country name in ascending and descending order. If the same header is clicked again, the table will be sorted in opposite order.



| Continent Name | Continent Id | Country Id | Country Name | City Name | Lat | Lng |
|---|---|---|---|---|---|---|
| Asia | AS | AF | Afghanistan | Āb-e Kamarī | 35.09 | 63.07 |
| Asia | AS | AF | Afghanistan | Adraskan | 33.65 | 62.27 |
| Asia | AS | AF | Afghanistan | Afaki | 36.66 | 69.35 |
| Asia | AS | AF | Afghanistan | Aībak | 36.27 | 68.02 |
| Asia | AS | AF | Afghanistan | Alah Sāy | 34.9 | 69.72 |
| Asia | AS | AF | Afghanistan | Aliabad | 36.52 | 68.9 |
| Asia | AS | AF | Afghanistan | Amānzī | 30.21 | 66.05 |
| Asia | AS | AF | Afghanistan | Anār Darah | 32.76 | 61.65 |
| Asia | AS | AF | Afghanistan | Andkhōy | 36.95 | 65.12 |
| Asia | AS | AF | Afghanistan | Āq Kupruk | 36.08 | 66.84 |
| Asia | AS | AF | Afghanistan | Āqchah | 36.91 | 66.18 |
| Asia | AS | AF | Afghanistan | Ārt Khwājah | 37.09 | 69.48 |
| Asia | AS | AF | Afghanistan | Asadābād | 34.87 | 71.15 |
| Asia | AS | AF | Afghanistan | Ashkāsham | 36.68 | 71.53 |
| Asia | AS | AF | Afghanistan | Āsmār | 35.03 | 71.36 |
| Asia | AS | AF | Afghanistan | Babasakhib | 31.66 | 65.66 |
| Asia | AS | AF | Afghanistan | Bāgh-e Maīdān | 35.22 | 69.21 |
| Asia | AS | AF | Afghanistan | Baghlān | 36.13 | 68.71 |

| Continent Name | Continent Id | Country Id | Country Name | City Name | Lat | Lng |
|---|---|---|---|---|---|---|
| Europe | EU | AX | Åland | Brändö | 60.41 | 21.05 |
| Europe | EU | AX | Åland | Eckerö | 60.21 | 19.61 |
| Europe | EU | AX | Åland | Finström | 60.27 | 19.93 |
| Europe | EU | AX | Åland | Föglö | 60.02 | 20.42 |
| Europe | EU | AX | Åland | Geta | 60.38 | 19.85 |
| Europe | EU | AX | Åland | Hammarland | 60.22 | 19.75 |
| Europe | EU | AX | Åland | Jomala | 60.15 | 19.95 |
| Europe | EU | AX | Åland | Kökar | 59.92 | 20.91 |
| Europe | EU | AX | Åland | Kumlinge | 60.26 | 20.78 |
| Europe | EU | AX | Åland | Lemland | 60.07 | 20.09 |
| Europe | EU | AX | Åland | Lumparland | 60.12 | 20.26 |
| Europe | EU | AX | Åland | Mariehamn | 60.1 | 19.94 |
| Europe | EU | AX | Åland | Saltvik | 60.28 | 20.05 |
| Europe | EU | AX | Åland | Sottunga | 60.13 | 20.67 |
| Europe | EU | AX | Åland | Sund | 60.25 | 20.12 |
| Europe | EU | AX | Åland | Vårdö | 60.25 | 20.38 |
| Africa | AF | ZW | Zimbabwe | Banket | -17.38 | 30.4 |
| Africa | AF | ZW | Zimbabwe | Beatrice | -18.25 | 30.85 |
| Africa | AF | ZW | Zimbabwe | Beitbridge | -22.22 | 30 |

Figure 19. Sorting the data by country name.

More on the used sorting and filtering functions and the working mechanics behind them is to be found in Section 5.3 Data Manipulation Functions.

## 5.2 Project Building

This section introduces the used programming languages, libraries and technologies used in the project.

### 5.2.1 Full-Stack Application with React.js and Node.js

The front end of the application was done with React.js, a popular JavaScript library that is used for building user interfaces [51]. To be able to use WebAssembly modules in a React application, the default configurations must be changed. (Appendix 1)

The map in the application is implemented with Leaflet, a JavaScript library for interactive maps [52]. It was clear early in the development, that rendering the markers is too slow. A WebGL drawing library, PixiJS solved this. A React wrapper had to be created for PixiJS and the usage is simple as seen in the code block below (Listing 14). [53; 54]

```
<div> {markers.length ? <PixiOverlay markers={markers} /> : null } </div>
```

Listing 14. Usage of PixiJS React wrapper in the application.

Material-UI is a component library for building React applications, providing extra properties for components and reducing the need of styling since the components implement Google’s Material Design [55]. During the development it was noticed that it lowered the rendering speed, so for example the table in the application is HTML because it must be able to render thousands of rows.

Node.js JavaScript runtime environment is used for back end. Node.js is built on Chrome’s V8 JavaScript engine and it makes it possible to run JavaScript on the machine as a standalone application [56]. Express.js is a web framework and it provides the methods to specify which function is called for a particular HTTP verb and URL pattern [57]. This application programming interface (API) provides a way to access the data from the client side of the application. The server side of the application consists only of one JavaScript file, containing the interaction between database and server, an example can be found in Listing 25.

### 5.2.2 MariaDB and Running Database Server in Docker

MariaDB server is a popular open-source database server. For this project it is set up running inside a Docker container using a MariaDB docker image found in Docker Hub [58]. Docker is a containerisation technology, allowing the packaging and running an application in an isolated environment, a container. A container is a runnable instance of a Docker image, which is a set of instructions for creating the Docker container. [59]

The database consists of four tables: continents, countries, cities, and performance. All tables but performance, are populated with data from JSON files, containing all continents, countries, and largest cities of the world. The cities table contains over 128 000 elements. Figure 20 below shows the design of the relational database tables of continents, countries, and cities.

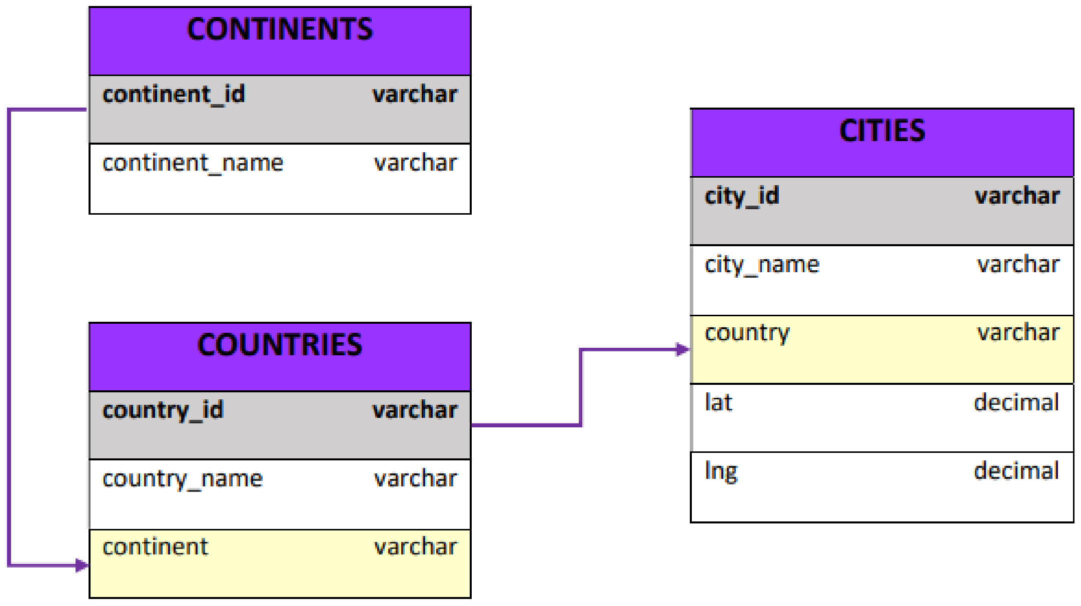


Figure 20. Design of continents, countries, and cities tables.

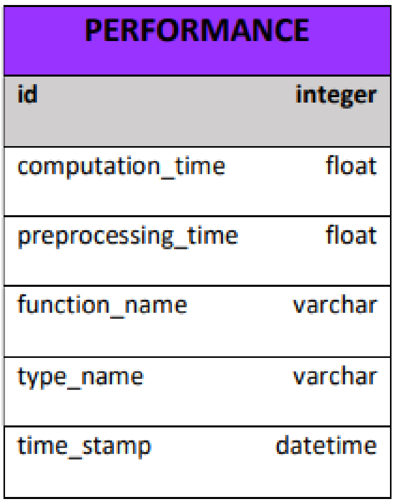
A JavaScript script handles the population, first checking if the JSON files exist locally, if not the data is fetched from different URL addresses. After the data is acquired, it is inserted into the database.

Every time the container is destroyed, the tables and data are deleted. A data dump can be created to back up the database tables and data with mysqldump client utility. The docker exec command runs a new command in a running container, in the first example the command is mysqldump. The MariaDB command-line client can be opened with the second command. Now, sql commands, such as SELECT, INSERT INTO and DELETE can be used. (Listing 15).

```
docker exec db_container mysqldump -u <user> -p<password> <db> > data-dump.sql

docker exec -it sql -u <user> -p<password> <db>
MariaDB[db]> SELECT * from cities;
```

Listing 15. Docker exec for mysqldump and sql commands.

Docker compose can use a YAML file to configure the container and use the data dump file to populate the tables upon the creation of the container (Appendix 2). Figure 21 below shows the design of the table used in storing the measured performance data.



| PERFORMANCE | |
|---|---|
| **id** | **integer** |
| computation_time | float |
| preprocessing_time | float |
| function_name | varchar |
| type_name | varchar |
| time_stamp | datetime |

Figure 21. Design of the performance table.

The performance table has multiple columns for performance time, as shown in Figure 20. This is because for some types of sorting, for example for SQL or WebAssembly and Rust, the data processing takes extra time. This changes the results. More on this in upcoming chapters.

### 5.2.3 Rust and Tools: wasm-pack and wasm-bindgen

Rust is a programming language designed to be fast, efficient, and reliable. It guarantees memory- and thread-safety with its rich type-system and ownership model [60]. Rust was chosen for this project due to the many tools provided for working with Rust and WebAssembly. Rust uses “structs” for structured data (Listing 16).

```
#[derive(Debug, Serialize, Deserialize, Clone)]
pub struct City {
    continentName: String,
    continentId: String,
    countryId: String,
    countryName: String,
    cityName: String,
    lat: f64,
    lng: f64,
}
```

Listing 16. Definition of a City struct in the application.

Wasm-bindgen and wasm-pack are tools developed by The Rust and WebAssembly Working Group and were used in this project for the integration of Rust, WebAssembly and JavaScript.

Wasm-bindgen is a Rust library and CLI tool that creates the bindings between Wasm modules and JavaScript. It can also import JavaScript functionality in to Rust such as DOM manipulation, console logging, performance monitoring and fetching. The functions marked with “#[wasm_bindgen]” attribute, are accessible from JavaScript and Rust. In the code example below (Listing 17), the “sort_ascended_by” function sorts the city list by key which are provided as parameters. The code for sorting is further inspected in Section 5.3.1 Sorting Data. In JavaScript the WebAssembly module is loaded from the bundle and the functions of the module are ready for use. [61]

```
// IN RUST
#[wasm_bingen]
pub fn sort_ascended_by(_key: &str, js_object: &JsValue) -> JsValue {
      …
      JsValue::from_serde(&cities).unwrap()
}

// React application's package.json
      "dependencies": {
            …
            "rust-wasm": "file:./rust-wasm/pkg"
      }

// IN JAVASCRIPT
const wasm = await import("rust-wasm");
const sortedCities = await wasm.sort_ascended_by("cityName", cities);
```

Listing 17. Adding “rust-wasm” Wasm module and usage in JavaScript.

Wasm-pack is a build tool which automates the WebAssembly package build process [62]. It provides commands for creating, building, testing, and publishing a Rust Wasm project. The “wasm-pack new <project name>” command creates a new Rust Wasm project which can then be built with “wasm-pack build” command, it creates a pkg folder with the WebAssembly binaries and needed JavaScript glue code. (Figure 22)

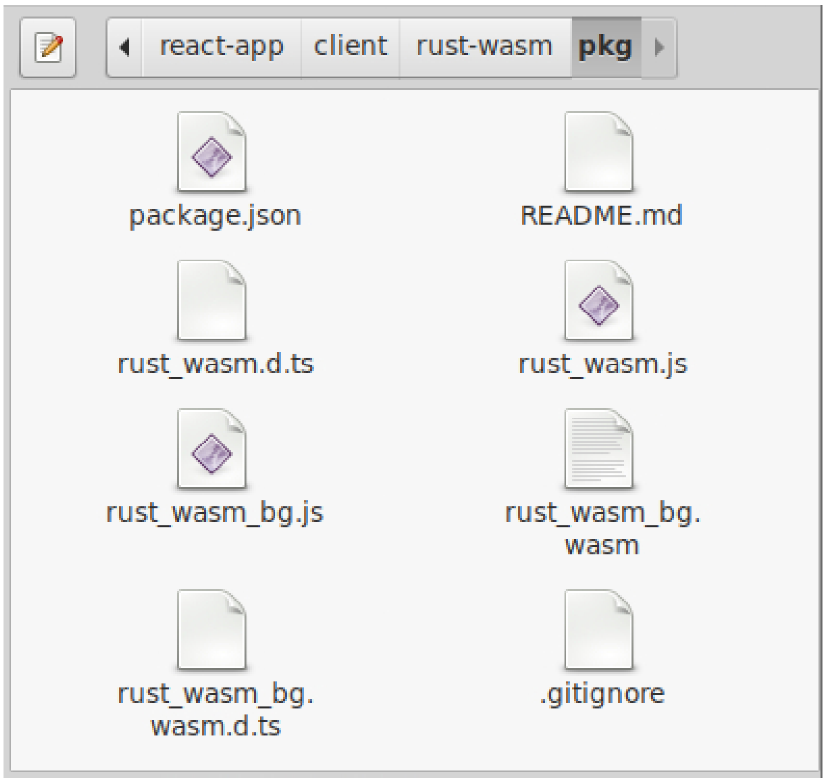


Figure 22. Contents of the pkg folder created by wasm-pack.

Wasm-pack allows the publishing of the created packages to the npm registry with “pack” and “publish” commands. The generated Wasm code can be tested in different environments, such as Node, Firefox, or Chrome, with “wasm-pack test” command. [63]

### 5.2.4 Svelte and Usage with React

Svelte is a fairly new approach to building user interfaces. It is a component framework like React, but it runs at build time. Svelte converts components into highly efficient imperative code that surgically updates the DOM. React uses a technique called virtual DOM diffing, which is comparing the upcoming UI changes to the existing UI and identifying the needed changes [64]. This reduces framerate and strains the browser’s garbage collector. [65]

Svelte was added to this project out of curiosity to use it with a React application. When the Svelte application is built, a “dist” directory is created that contains the compiled, optimised JavaScript code. The optimised code can then be used in a React application as shown in Listing 18.

```
/* React application’s package.json */
     “dependencies”: {
          …
          “svelte-component”: “file:./svelte-component/dist/index.js”
     }

// JAVASCRIPT
const svelte = await import(“svelte-component”);
const sortedCities = await svelte.sortCities(“cityName”, cities, “asc”);
```

Listing 18. Adding and usage of “svelte-component” in React application.

This module does not need a dedicated file loader like the .wasm files since the code is JavaScript. It can be straightforwardly used after adding it to the React application’s “package.json” file (Listing 18).

## 5.3 Data Manipulation Functions

The data manipulation functions, sorting and filtering, have been implemented with Rust, JavaScript, and SQL. The data is fetched from the database via an API. The React application uses Axios library for fetching the data. Wasm-bindgen provides a way of fetching data with Rust by using the web-sys crate. There are two functions implemented in Rust, one stores the fetched cities into a global function and returns the data to the React application after this, another only returns the data.

### 5.3.1 Sorting Data

Even though the built-in sorting functions use different algorithms it is appropriate to use these for the sake of research. Very few developers want to write their own sorting algorithms and thus use the built-in one. JavaScript's built-in sort function uses different algorithms depending on the browser in use. Mozilla's SpiderMonkey uses merge sort and Chrome's V8 uses Timsort, which is an adaptive merge sort variant. [66; 67]

The sort function in JavaScript converts the elements to strings and compares the strings if no compare function is not provided [68]. This is problematic when dealing with numbers, "10" as string comes before "2". Custom compare function below fixes this problem, it also considers if the elements must be sorted in ascending or descending order (Listing 19). This same function is used in "vanilla_js" and "svelte" sorters.

```
cities.sort((a,b) => {
     let x = order === "desc" ? b[key] : a[key];
     let y = order === "desc" ? a[key] : b[key];
     return x < y ? -1 : x < y ? 1 : 0;
});
```

Listing 19. JavaScript sort used in the project application.

Lodash is the most popular JavaScript utility library. It provides commonly used functions for working with arrays, numbers, and objects. Lodash uses the

JavaScript’s sort function under the hood. But as the function below shows, the usage is easier than that of pure JavaScript. [69]

```
import _ from "lodash";
_.orderBy(cities, [(city) => city[key]], [order])
```

Listing 20. Lodash sort with key and order.

The Rust Standard Library offers the “Vec” core type. A vector is a growable array type and the struct provides sorting functions. Once the Rust code acquires the list of cities, it is serialised into a vector of structs. Sorting unstable is typically faster than stable sorting, which is why it is used in this application. The function “sort_unstable_by” is currently based on pattern-defeating quicksort and it is documented as follows:

> “…which combines the fast average case of randomized quicksort with the fast worst case of heapsort, while achieving linear time on slices with certain patterns. It uses some randomization to avoid degenerate cases, but with a fixed seed to always provide deterministic behavior.” [70]

Since Rust does not support the bracket notation for accessing the properties of a struct, matching is needed (Listing 21). This is almost the equivalent of the “switch-case” statement or multiple “if-else” statements.

```
match _key {
     "continentName" => …,      // function to sort by "continentName"
     "continentId" => …,
     "countryId" => …,
     "countryName" => …,
     "cityName" => …,
     "lat" => …,
     "lng" => …,
     _ => {}
}
```

Listing 21. Sorting by key in Rust with matching.

In the project, there are four different sorting functions written in Rust. Different functions are needed for ascending and descending sorting. Also, it seems that the less data is moved between WebAssembly and JavaScript, the faster the function is. So, there is one pair of implementations for passing the list of cities as parameters (rust_wasm), and another one with a global variable, where the data is stored before the function call (rust_wasm_2). Below are the functions implemented with the city list as a parameter (Listing 22).

```
pub fn sort_descended_by(_key: &str, js_objects: &JsValue) -> {
     let mut cities: Vec<City> = js_objects.into_serde().unwrap();
     match _key {
          "continentName" => cities.sort_unstable_by(|a, b|
               b.continentName.cmp(&a.continentName)),
          …
     }
     JsValue::from_serde(&cities).unwrap();
}

pub fn sort_ascended_by(_key: &str, js_objects: &JsValue) -> {
     let mut cities: Vec<City> = js_objects.into_serde().unwrap();
     match _key {
          "continentName" => cities.sort_unstable_by(|a, b|
          a.continentName.cmp(&b.continentName)),
          …
     }
     JsValue::from_serde(&cities).unwrap();
}
```

Listing 22. Sorting with the list of cities passed as a parameter (rust_wasm).

Values are compared with “cmp” or “partial_cmp” functions. The latter function is needed for float value types since they do not implement “Ord” trait. [71]. JsValue is a representation of a JavaScript object [72]. First, it is converted into a JSON string, the resulting string is passed into Rust and then parsed into given Rust value, which here is a vector of City structs. After the sorting, the vector must be converted back to a JsValue. These operations are time consuming functions and are discussed in Section 5.5.

```
pub fn sort_descended_by_global(_key: &str) -> {
     let mut result: Vec<City>;
     unsafe { res = CITIES.clone(); }
     match _key {
          "continentName" => result.sort_unstable_by(|a, b|
               b.continentName.cmp(&a.continentName)),
          …
     }
     JsValue::from_serde(&result).unwrap();
}

pub fn sort_ascended_by_global(_key: &str) -> {
     let mut result: Vec<City>;
     unsafe { res = CITIES.clone(); }
     match _key {
          "continentName" => result.sort_unstable_by(|a, b|
               a.continentName.cmp(&b.continentName)),
          …
     }
     JsValue::from_serde(&result).unwrap();
}
```

Listing 23. Sorting with a global variable (rust_wasm_2).

The cities are fetched into a global variable and cloned to another variable when needed as seen in the listing above. The cloning is necessary, so that the original list of cities is preserved. Even though the variable is defined as static, it can still be declared mutable. This is when an “unsafe” block is required. It should be made sure that modifications to a mutable static are safe with respect to other running threads [73].

SQL also provides its own means for sorting data by key with the “ORDER BY” keyword. It is not known which algorithm it uses. In Listing 24 below, the query “selectAll” selects all the cities in the database, chooses the needed columns and joins them with the right country and continent.

```
const selectAll = "SELECT continentName, continentId, countryId, countryName,
cityName, lat, lng FROM ((continents INNER JOIN countries ON
continents.continentId = countries.continent) INNER JOIN cities ON
countries.countryId = cities.country)";
```

Listing 24. The “selectAll” constant in the application.

The order of the sorting can be chosen with “asc” or “desc” keywords after the chosen column [74]. In the application the column name is the variable “sortBy” and the order of the sorting is “order” variable. These are provided in the parameters of the request, in the web address (Listing 25).

```
// http request URL localhost:3001/api/cities/sort/cityName/asc

app.get("/api/cities/sort/:sortBy/:order", async (req, res) => {
  const key = req.params.sortBy;
  const order = req.params.order;
  const q = `${selectAll} ORDER BY ${key} ${order}`; //template literal
  try {
    conn = await pool.getConnection();
    const result = await conn.query(q);
    res.json(result);
  } catch (err) {
    throw err;
  } finally {
    if (conn) conn.end();
  }
});
```

Listing 25. Sort with SQL in the application.

Some code examples in this chapter use template literals. This is a way of using embedded expressions in strings as shown in the code listing above [75]. After the sorted data has been acquired from the database it is sent to the client side of the application as a HTTP response.

### 5.3.2 Filtering Data by Key

In the application the data is filtered, and the table updated with each key press (Figure 18). JavaScript's "filter" method takes a callback function as a parameter to test each element of the given array and creates a new array with all the elements passing the test, it does not mutate the array on which it is called. Vanilla_js and svelte sorters use the same filtering function shown below. The JavaScript "filter" function loops through the list of cities and the "includes" function checks if the object's value includes the given value. [76; 77]

```
cities.filter((city) => city[key].includes(value));
```

Listing 26. JavaScript filter implementation in the application.

Lodash library provides the same functions as JavaScript, with a slightly different syntax.

```
import _ from "lodash";
_.filter(cities, (e) => _.includes(e[key], value));
```

Listing 27. Lodash filter implementation in the application.

In Rust, the vector to be iterated must be turned into an iterator with the “into_iter” function. This iterator provides a “filter” function which uses a function to determine if an element should be yielded, depending on whether the returned value is “true” or “false” [78]. The “collect” function transforms the iterator back into a collection [79].

The filtering functions of the application with Rust use the same matching as the sorting functions (Listing 21). The function given to “filter” checks if the given string contains a substring and passes the ones that do not. Rust provides a built in “contains” function for this. The results of the iteration are transformed back into vector of cities. Two functions are implemented, one for passing the list of cities as parameters, and another using a global variable as seen in Listing 28 below.

```
// CITIES IN PARAMETERS   rust_wasm
pub fn filter_cities(_key: &str, _value: &str, js_objs: &JsValue) -> JsValue {
      let cities: Vec<City> = js_objs.into_serde().unwrap();
      let res: Vec<City>;
      match _key {
            "continentName" => {
                  res = cities
                        .into_iter()
                        .filter(|city| city.continentName.contains(_value))
                        .collect::<Vec<City>>()
            }
            …
      }
      JsValue::from_serde(&res).unwrap()
}


// CITIES IN A GLOBAL VARIABLE  rust_wasm_2
pub fn filter_cities_global(_key: &str, _value: &str) -> JsValue {
      let cities: Vec<City>;
      let res: Vec<City>;
      unsafe { cities = CITIES.clone(); }
      match _key {
            "continentName" => {
                  res = cities
                        .into_iter()
                        .filter(|city| city.continentName.contains(_value))
                        .collect::<Vec<City>>()
            }
            …
      }
      JsValue::from_serde(&res).unwrap()
}
```

Listing 28. Rust filtering function implementations in the application.

SQL enables filtering with the “LIKE” operator. It is used in a “WHERE” clause to search for a specific pattern in given column [80]. The “%” wildcard represents multiple characters, meaning that it is acceptable to have multiple characters after “Hels” [80]. Listing 29 below shows the filtering in the application. The filtering query is paired with a “selectAll” constant (Listing 24) to acquire all the needed information.

```
// http request URL localhost:3001/api/cities/filter/cityName/Hels

app.get("/api/cities/filter/:filterBy/:value", async (req, res) => {
  const filterBy = req.params.filterBy;
  const value = req.params.value;
  const q = `${selectAll} WHERE ${filterBy} LIKE '${value}%'`;

  …       … // execute query and send result to client with HTTP request
});
```

Listing 29. SQL filtering in the application.

The column name is the variable “filterBy” and pattern to be searched is the “value” variable. These are provided in the GET request’s parameters, in the web address.

### 5.3.3 Filtering Data by Map Bounds

The data is filtered by the map bounds shown in the application. The Leaflet library provides functions for acquiring coordinates of the top right (northeast) and bottom left (southwest) corners (Listing 30). These coordinates are used to filter the data by checking if the city’s coordinates are within bounds. The filtering is done every time the map is moved.

```
const { southwest, northeast } = map.getBounds();
// "southwest" object
southwest: {
     lat: 65,
     lng: 25
}
// "northeast" object
northeast: {
     lat: 65,
     lng: 25
}
```

Listing 30. Acquiring the coordinates for map bounds.

JavaScript implementation uses “filter” function and the given callback function checks if the current city’s latitude and longitude are within bounds. This is done with “greater than” and “less than” comparison operators as shown in Listing 31 below.

```
const filterByBounds = (southwest, northeast) => {
     cities.filter(city) =>
          southwest.lat < city.lat &&
          northeast.lat > city.lat &&
          southwest.lng < city.lng &&
          northeast.lng > city.lng
     )
}
```

Listing 31. Filtering by bounds in vanilla JavaScript.

Once again, Lodash makes the syntax clearer with an “inRange” function, as can be seen in Listing 32 below. It checks whether the first parameter is between the second and third parameters “inRange(n, start, end)” [81].

```
import _ from 'lodash';
const filterByBounds = (southwest, northeast) => {
     _.filter(cities, (city) =>
                 _.inRange(city.lat, southwest.lat, northeast.lat) &&
                 _.inRange(city.lng, southwest.lng, northeast.lng)
     )
}
```

Listing 32. Filtering by bounds with Lodash library.

Rust provides a simple way of creating a range with “..” notation (start..end) [82]. For example “1..5” would mean all the values between one and five. The created range has a function “contains” which checks if the range contains the given value. The bounds are an array of floats instead of an object since the usage of JsValue should be minimised. This function is also implemented two times, with and without a global variable. In Listing 33 below, is the function that uses the global variable.

```
// rust_wasm_2 (with a global variable for cities)

pub fn filter_by_bounds(bounds: &[f64]) -> JsValue {
     let lat_range = bounds[0]..bounds[1];
     let lng_range = bounds[2]..bounds[3];
     let cities: Vec<City>;
     let res: Vec<City>;
     unsafe { cities = CITIES.clone(); }
     res = cities.into_iter()
          .filter(|city|  lat_range.contains(&city.lat)&&
                          lng_range.contains(&city.lng))
          .collect::<Vec<City>>();
     JsValue::from_serde(&res).unwrap()
}
```

Listing 33. Filtering by bounds in Rust.

SQL provides a “BETWEEN” operator which selects the values within given range, also accepting text or dates [83]. The bounds are acquired from the body of the HTTP POST request message. Listing 34 below shows how the bounds are acquired and the SQL query string used for filtering by bounds.

```
app.post("/api/cities/bounds", async (req,res) => {
      const northeast = req.body.northeast;
      const southwest = req.body.southwest;

      const q = `${selectAll} WHERE
            (lat BETWEEN ${southwest.lat} AND ${northeast.lat}) AND
            (lng BETWEEN ${southwest.lng} AND ${northeast.lng})`

      … // execute query and send result to client with HTTP request

});
```

Listing 34. Filtering by bounds with SQL.

One reason for implementing filtering by bounds to the application is to observe the effectiveness of rendering markers to the map.

## 5.4 Improvements and Problems

WebAssembly as a topic is comprehensive and hard to fully understand without previous understanding of machine languages and the low-level side of programming, especially for developers that are used to JavaScript. Programming with JavaScript requires less knowledge of the underlying mechanics since the JavaScript engines do most of the work behind the scenes. It took a long time to understand WebAssembly. Due to the lack of knowledge, the beginning of the project and the writing of this thesis, took a lot of time and perhaps even unnecessary steps were taken. For example, AssemblyScript, a language based on TypeScript and designed for WebAssembly, was considered instead of Rust, but after a while of research it was noticed that the support for objects between JavaScript and Wasm was lacking. Also, at first the application for the study was started using the Blazor framework. Later it was changed, decreasing the time for completing the study.

A part of this project was done with Rust, a previously unknown programming language for the author of this thesis. Even though C++ was somehow familiar, the mechanics of the Rust language: Traits, variable mutability, and lifetime, coupled with the restrictions brought by the wasm-bindgen tool, made the development challenging. Starting a project with a new language is always challenging, figuring out the package manager and best practices does not happen overnight. It must be noted that this could affect the results.

WebAssembly is a fairly new standard and even though almost all the browsers support it, the tools for building a WebAssemly application still have ways to go. Especially when more complex data types, such as objects, are needed. This raised the question: Was this simply a wrong project for implementing WebAssembly? The usage of JsValue slowed down the performance a great deal. This is likely to be fixed in the future.

> A JsValue doesn't actually live in Rust right now but actually in a table owned by the wasm-bindgen generated JS glue code. Eventually the ownership will transfer into wasm directly and this will likely become more efficient, but for now it may be slightly slow. [72]

It might be beneficial to test this project with Emscripten and C++. Perhaps Emscripten has a better optimised way for objects to be used. Also, the user actions during measuring should be automated, eliminating possible human errors.

## 5.5 Project Results

Result visualisations were done with Microsoft's Power BI, a tool for unifying data and creating interactive, immersive dashboards and reports. With Power BI it is possible to connect straight to the data source, or in this case, the MariaDB database. The collected performance data and JavaScript Performance tool provided by Mozilla Firefox was studied. The Performance tool allows to create a recording of a site. This recording shows an overview of the tasks the browser did. [84; 85]

The most important results to inspect are the ones of sorting. It was observed that the filtering and filtering by bounds functions' performance is difficult to measure. For example, the filtering function was called each time a key was pressed on the keyboard, meaning that the key given to filter could be only one letter, or a whole word. Same with filtering by bounds, the more zoomed the map was, the less cities were to be found. These might affect the results since the usage of the application was not done with automation but by a person, giving room for unequal filtering. During the measurement, the user actions were attempted to perform as identically as possible. The sorters and short descriptions of them can be found in Table 3.

Figure 23 below shows the total times taken to finish the functions. Observing this, it would seem that WebAssembly does not perform better than JavaScript in web applications, since for example sorting is four times slower than with plain JavaScript.

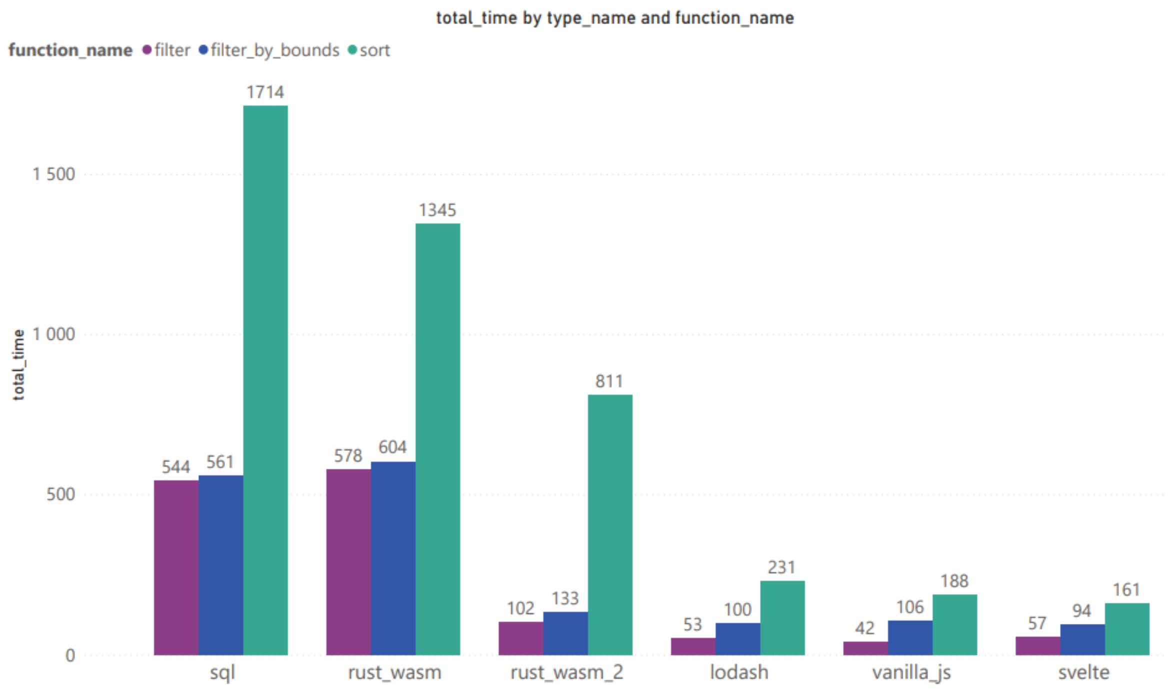


Figure 23. Averages of total times for all functions.

Further examining the results, differentiating computation time, and pre-processing time, the results are quite different. Computation time refers to the time it takes to finish the sorting and pre-processing time is all the extra work happening before or after this. Figure 24 below shows only the computation times and here one can see that the rust_wasm sorters did in fact perform better in computing. For the “filter_by_bounds” function of rust_wasm_2, it was noticed that the cloning function was accidentally considered in measuring of the time, therefore it is 20 milliseconds or so slower. This does give information on the speed of the cloning function which should be considered in other findings also.

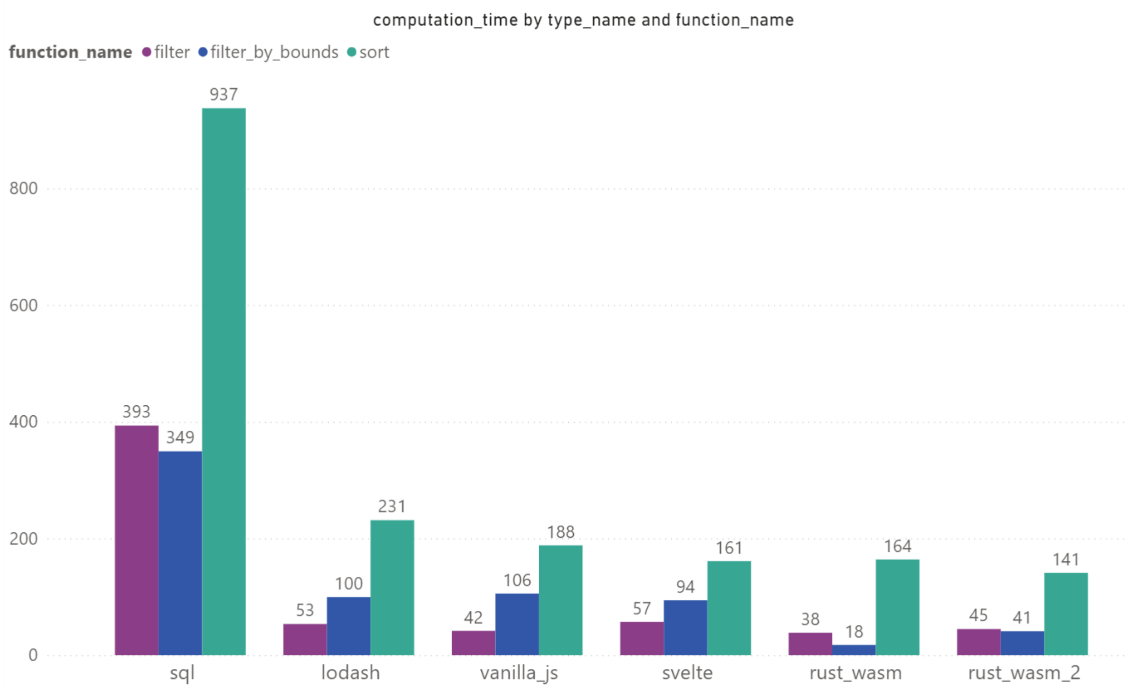


Figure 24. Average of computation time with all sorters and functions.

The total time of execution is measured with a JavaScript wrapper function and the computation time for rust_wasm sorters is acquired with a function implemented in the Rust project. With these two variables, it is possible to calculate the pre-processing time. These functions are shown in Listing 35 below.

```
// In React application
const timeWrapper = async(func) => {
      const t1 = performance.now();
      const result = await func();
      const time = performance.now() - t1;
      return [result, time];
}

// Usage of timeWrapper
const sortCities = async () => { … }
const [sortedCities, time] = await timeWrapper(sortCities);

// This is a function in the Wasm module
const computationTime = wasm.get_rust_performance();

// The time used for pre-processing
const preprocessingTime = time - computationTime
```

Listing 35. Time measuring wrapper and its usage.

Lodash performance was to be expected. It uses the vanilla JavaScript sorting function underneath the hood, why it is a tad slower than the vanilla_js sorter. It was considered in this project because it is highly popular and it was interesting to see, how big the decline in performance is compared to vanilla JavaScript. It is also important to point out beneficial tools and libraries to ease the development.

The worst performance is that of SQL functions. Its total time is immense compared to many of the other sorters. The computation time was a bit surprising. SQL is largely used for this kind of applications but perhaps not with this large a data set. The ordering of the entries can be optimised by retrieving smaller chunks of data at a time. The average total time was 1714 milliseconds as seen in Figure 25.

With vanilla_js, lodash and svelte, the computation and pre-processing times are not differentiated since the point of measuring pre-processing time was to know if it substantially affects the external functions' measured times. The pre-processing measurements of SQL are mainly the time it takes to make HTTP requests and the high consumption of time was to be expected and as with ordering, also the time taken by HTTP requests can be optimised with smaller data chunks. Figure 25 below shows the averages of computation and pre-processing times. This shows that the pre-processing times cause the slowness of the rust_wasm sorters.

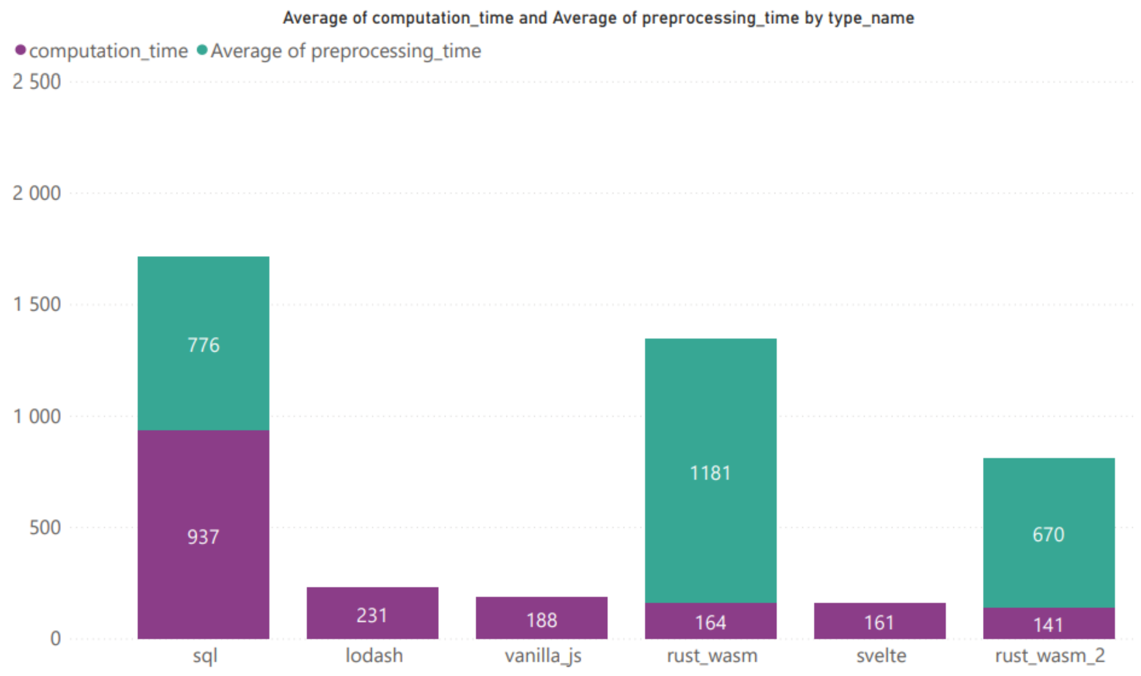


Figure 25. Averages of computation and pre-processing times on sort function.

The pre-processing times of both rust_wasm sorters came from the functions in JavaScript glue code, such as serialisation of the object, and even passing a string to WebAssembly requires multiple lines of code. Figure 26 below shows a snapshot taken from Mozilla's JavaScript Performance tool. Here can be detected several WebAssembly functions that are called when sorting. The functions marked with a red box are the JavaScript glue code's functions. For example, the "wbindgen_json_serialize" function takes about 200 milliseconds to finish.

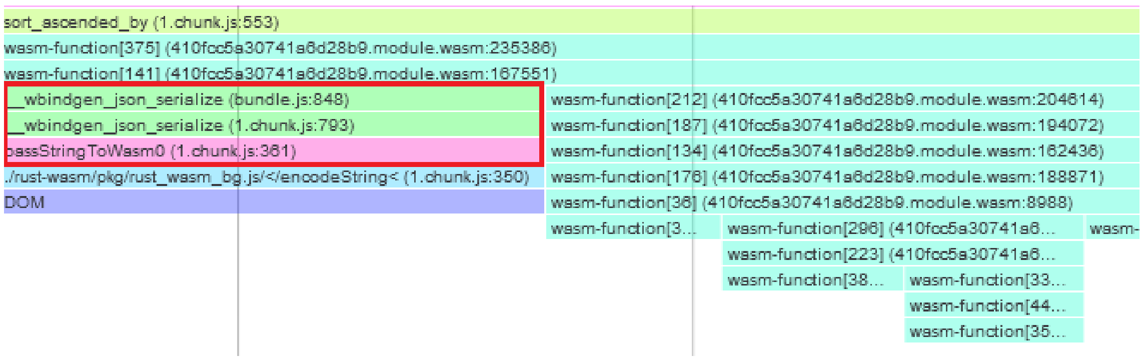

sort_ascended_by (1.chunk.js:553)
wasm-function[375] (410fcc5a30741a6d28b9.module.wasm:235386)
wasm-function[141] (410fcc5a30741a6d28b9.module.wasm:167551)
__wbindgen_json_serialize (bundle.js:848)
__wbindgen_json_serialize (1.chunk.js:793)
passStringToWasm0 (1.chunk.js:361)
./rust-wasm/pkg/rust_wasm_bg.js/</encodeString< (1.chunk.js:350)
DOM
wasm-function[212] (410fcc5a30741a6d28b9.module.wasm:204614)
wasm-function[187] (410fcc5a30741a6d28b9.module.wasm:194072)
wasm-function[134] (410fcc5a30741a6d28b9.module.wasm:162436)
wasm-function[176] (410fcc5a30741a6d28b9.module.wasm:188871)
wasm-function[36] (410fcc5a30741a6d28b9.module.wasm:8988)
wasm-function[3...
wasm-function[296] (410fcc5a30741a6...
wasm-
wasm-function[223] (410fcc5a30741a6...
wasm-function[38...
wasm-function[33...
wasm-function[44...
wasm-function[35...

Figure 26. WebAssembly function showed in JavaScript Performance tool.

It is interesting that the pre-processing time of rust_wasm is over 500 milliseconds more than that of rust_wasm_2 as can be seen in Figure 25. This means that the functions dealing with serialisation and deserialization of data takes this much time. Since the functions of the rust_wasm sorter, take the list of cities as a parameter, serialisation and deserialization happen more often. For example, when the function receives the data as a “JsValue” it has to be deserialized into a vector of city structs with the “into_serde” function. The function invokes “JSON.stringify” function and then parses the resulting JSON into the given struct [86]. The functions of rust_wasm_2 clone the list of cities, which takes noticeably less time (about 20 milliseconds).

Now, to compare this with the sort function implemented with JavaScript, one can see that there is significantly less functions to be called (Figure 27). No wonder the total times for rust_wasm and rust_wasm_2 sorters are greater than those implemented with JavaScript.

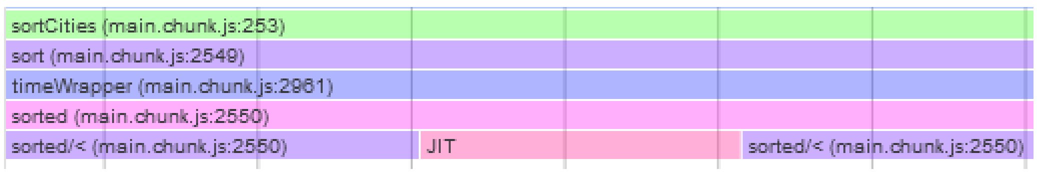

sortCities (main.chunk.js:253)
sort (main.chunk.js:2549)
timeWrapper (main.chunk.js:2961)
sorted (main.chunk.js:2550)
sorted/< (main.chunk.js:2550)
JIT
sorted/< (main.chunk.js:2550)

Figure 27. JavaScript function showed in JavaScript Performance tool.

The effect of WebAssembly was decided to be investigated a bit further by creating standalone C++, Rust and JavaScript applications sorting an array of floats, latitudes and longitudes parsed from the list of cities, and cities by name.

This revealed that when sorting floats Rust beats JavaScript for speed. C++ was even faster. On the other hand, Rust was faster than C++ when sorting structs, JavaScript was much slower. These results can be found in Figure 28 below.

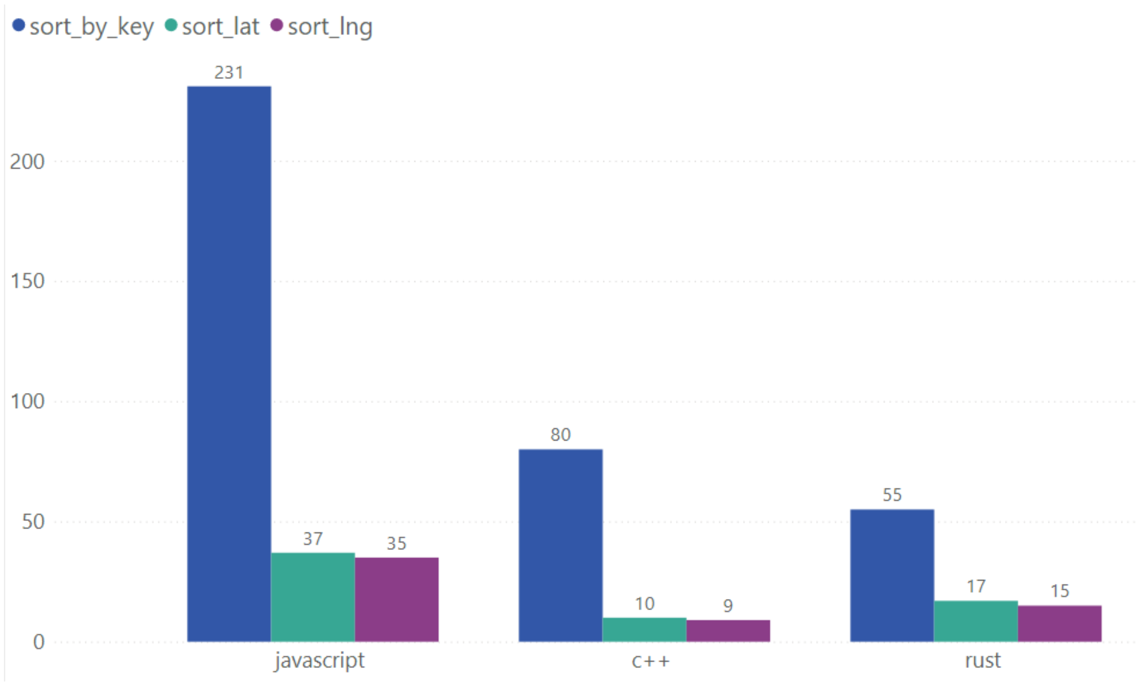


Figure 28. Speed in milliseconds when sorting with JavaScript, C++, and Rust.

This suggests that Rust is faster than WebAssembly with these kinds of functions, sorting by key is almost three times slower. It is to be noted that for both Rust and C++ an optimisation level -O3 was used. A similar feature for JavaScript is possibly the JIT compilers optimisation. The result for JavaScript was collected after multiple runs. Even though the optimisation level was added to the application's rust-wasm project (Listing 36), the result did not get better.

```
[profile.dev]
lto = true
opt-level = 3
```

Listing 36. Optimising Wasm code in Cargo.toml file.

Wasm-pack uses a Binaryen toolkit which provides a “wasm-opt” tool to optimise Wasm code after it has been generated. WebAssembly code can be optimised for performance or size. [87]

An interesting find was made when working with the standalone test application. Sorting by city name was extremely slow compared to sorting by another key, since many city names contain special characters. After this, a new column was added to the performance table, differentiating the keys by which data can be sorted and the performance data was collected again. Figure 29 below shows the distribution of the computation times when sorting with different keys.

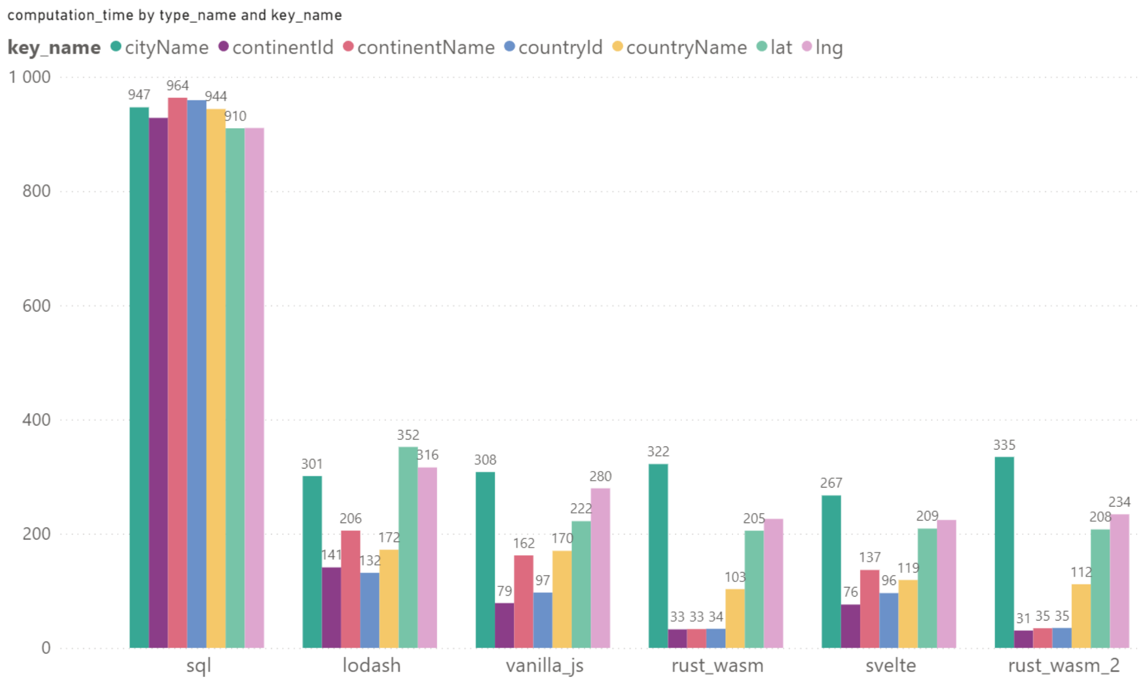


Figure 29. Computation times when sorting with different keys.

Sorting by city name or longitude were slowest in most cases. City names because of the special characters and longitude because the range of it is bigger than the range of latitude (-180 to 180 vs -90 to 90). Also, the initial order of the cities affects the sorting speed a lot.

# 6 Summary and Conclusion

This thesis gives an overview of how code is translated from high-level programming language to target code, and why this is considered in a thesis concerning WebAssembly. Also, the design, concepts, and usage of WebAssembly are briefly discussed.

WebAssembly is a machine language for a virtual machine, making it faster than JavaScript since it is closer to the target machine language than high-level JavaScript. Wasm is designed to be fast and portable with small file sizes. Wasm code has two formats, text, and binary, it provides only four basic values types and its memory is linear.

So, should some functionalities that require high performance be implemented using WebAssembly? The results of the project developed for this thesis would suggest no. Still, WebAssembly could be a real asset for the right kind of applications. This has been proved with some large desktop applications that have been ported to the web, also mentioned in this thesis. The studying of WebAssembly should have been much more extensive before writing even a line of code, making it possible to choose a more suitable application.

Using the WebAssembly module in JavaScript was relatively straightforward after the initial confusion and trial and error with the configurations. Otherwise the development of the application was simple since full-stack development and Docker usage was already familiar. WebAssembly could be an easy way for back end developers accustomed to for example C++ or Rust to take part in Web development.

The development of WebAssembly is ongoing and it will likely be safer and more versatile in the future. It is important to remember that this is just the first release of WebAssembly and it is continuously gaining new features through a standardisation process.

In conclusion, WebAssembly could be and most likely will be a lifesaver to multiple kinds of applications battling with performance problems in the web. This of course, after all the security problems are dealt with and the WebAssembly standard and the tools surrounding it has been finished to the point that it is suitable for a more variety of applications.

## References

1 Ben Sassi, Rakia. 2021. *Compiler vs. Interpreter: Know the Difference and When to Use Each of Them*. Medium. <https://betterprogramming.pub/compiler-vs-interpreter-d0a12ca1c1b6>. Accessed 10.4.2021.

2 Difference Between Compiler and Interpreter. 2018. Tech Differences <https://techdifferences.com/difference-between-compiler-and-interpreter.html>. Accessed 14.1.2021.

3 Rezvi, Mahdhi. JavaScript Engines: An Overview. 2020. Medium. <https://blog.bitsrc.io/javascript-engines-an-overview-2162bffa1187>. Accessed 14.1.2021.

4 Clark, Lin. A Crash Course in Just-In-Time (JIT) Compilers. 2017. Mozilla Hacks. <https://hacks.mozilla.org/2017/02/a-crash-course-in-just-in-time-jit-compilers/>. Accessed 10.1.2021.

5 Hinkelmann, Franziska. 2017. <https://fhinkel.rocks/2017/08/16/Understanding-V8-s-Bytecode/>. Updated 2019. Accessed 19.2.2021.

6 Assembly - Introduction. Tutorials Point. <https://www.tutorialspoint.com/assembly_programming/assembly_introduction.htm>. Updated 2021. Accessed 12.2.2021.

7 A Fundamental Introduction to x86 Assembly Programming. 2016. Project Nayuki. <https://www.nayuki.io/page/a-fundamental-introduction-to-x86-assembly-programming>. Updated 7.1.2016. Accessed 12.2.2021.

8 Clark, Lin. A Crash Course in Assembly. 2017. Mozilla Hacks. <https://hacks.mozilla.org/2017/02/a-crash-course-in-assembly/>. Accessed 12.2.2021.

9 Clark, Lin. Creating and Working with WebAssembly Modules. 2017. Mozilla Hacks. <https://hacks.mozilla.org/2017/02/creating-and-working-with-webassembly-modules/>. Accessed 13.2.2021.

10 Mehrabani, Afshin. Understanding asm.js. 2014. Sitepoint. <https://www.sitepoint.com/understanding-asm-js/>. Accessed 15.02.2021.

11 Zakai, Alon. Why WebAssembly Is Faster Than asm.js. 2017. Mozilla Hacks. <https://hacks.mozilla.org/2017/03/why-webassembly-is-faster-than-asm-js/>. Accessed 16.02.2021.

12 World Wide Web Consortium (W3C) brings a new language to the Web as WebAssembly becomes a W3C Recommendation. 2019. W3C.

<https://www.w3.org/2019/12/pressrelease-wasm-rec.html.en>. Accessed 19.03.2021.

13 Dr. Parthasarathi, Ranjani. Instruction Set Architecture. Computer Architecture. 2018. <https://www.cs.umd.edu/~meesh/411/CA-online/chapter/instruction-set-architecture/index.html>. Accessed 20.02.2021.

14 Rossberg, Andreas. WebAssembly Specification. 2021. WebAssembly GitHub. <https://webassembly.github.io/spec/core/_download/WebAssembly.pdf>. Accessed 02.03.2021.

15 CoinEx Chain. Wasm Introduction (Part 6): Table & Indirect Call. 2020. Medium. <https://coinexchain.medium.com/wasm-introduction-part-6-table-indirect-call-65ad0404b003>. Accessed 15.04.2021.

16 Security. WebAssembly Documentation. <https://webassembly.org/docs/security/>. Accessed 04.03.2021.

17 POSIX.1-2008 Standard Details. 2016. Institute of Electrical and Electronics Engineers Standards Association (IEEE SA). <https://standards.ieee.org/standard/1003_1,2016Edition.html>. Accessed 19.04.2021.

18 Clark, Lin. Memory in WebAssembly (and why it's safer than you think). 2017. Mozilla Hacks. <https://hacks.mozilla.org/2017/07/memory-in-webassembly-and-why-its-safer-than-you-think/>. Accessed 12.03.2021.

19 Address Space Layout Randomization. 2021. Microsoft Documentation. <https://docs.microsoft.com/en-us/windows/security/threat-protection/overview-of-threat-mitigations-in-windows-10#address-space-layout-randomization>. Accessed 06.03.2021.

20 Lehmann, Daniel; Kinder, Johannes; Pradel, Michael. Everything Old is New Again: Binary Security of WebAssembly. 2020. USENIX. <https://www.usenix.org/system/files/sec20-lehmann.pdf>. Accessed 04.03.2021.

21 Specifications. WebAssembly. <https://webassembly.org/specs/>. Accessed 27.02.2021.

22 WebAssembly Web API. 2021. WebAssembly GitHub. <https://webassembly.github.io/spec/web-api/index.html>. Accessed 27.02.2021.

23 WebAssembly.instantiateStreaming(). 2021. MDN Web Docs. <https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/WebAssembly/instantiateStreaming>. Accessed 27.02.2021.

24 WebAssembly.instantiate(). 2021. MDN Web Docs. <https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/WebAssembly/instantiate>. Accessed 27.02.2021.

25 WebAssembly.Memory() constructor. 2021. MDN Web Docs. <https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/WebAssembly/Memory/Memory>. Accessed 27.02.2021.

26 WebAssembly.Table() constructor. 2021. MDN Web Docs. <https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/WebAssembly/Table/Table>. Accessed 28.02.2021.

27 WebAssembly.Global() constructor. 2021. MDN Web Docs. <https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/WebAssembly/Global/Global>. Accessed 28.02.2021.

28 WASI: WebAssembly System Interface. 2019. WASI GitHub. <https://github.com/WebAssembly/WASI/blob/main/docs/WASI-overview.md>. Accessed 01.03.2021.

29 WASI Libc. 2021. Wasi Libc GitHub <https://github.com/WebAssembly/wasi-libc>. Accessed 01.04.2021.

30 Wasmer Runtime. 2021. Wasmer Documentation. <https://docs.wasmer.io/ecosystem/wasmer>. Accessed 02.04.2021.

31 Hello World example. 2021. Wasmer Documentation. <https://docs.wasmer.io/integrations/js/wasi/server/examples/hello-world>. Accessed 02.04.2021.

32 Wasmer Features. 2021. Wasmer Documentation. <https://docs.wasmer.io/ecosystem/wasmer/wasmer-features>. Accessed 02.04.2021.

33 Introduction. 2021. Yew Docs. <https://yew.rs/docs/en/>. Accessed 10.03.2021

34 Wasm Build Tools. 2021. Yew Docs. <https://yew.rs/docs/en/getting-started/project-setup#wasm-build-tools>. Accessed 10.03.2021.

35 Introduction to ASP.NET Core Blazor. 2020. Microsoft Documentation. <https://docs.microsoft.com/en-us/aspnet/core/blazor/?view=aspnetcore-5.0>. Accessed 10.03.2021.

36 ASP.NET Core Blazor Hosting Models. 2020. Microsoft Documentation. <https://docs.microsoft.com/en-us/aspnet/core/blazor/hosting-models?view=aspnetcore-5.0>. Accessed 10.03.2021.

37 About Qt. 2019. Qt Wiki. <https://wiki.qt.io/About_Qt>. Accessed 10.03.2021.

38 Qt on the Web, WebAssembly. 2020. Qt. <https://www.qt.io/qt-examples-for-webassembly>. Accessed 10.03.2021.

39 User Interfaces. 2021. Qt Documentation. <https://doc.qt.io/qt-5/topics-ui.html>. Accessed 10.03.2021.

40 Qt Quick. 2021. Qt Documentation. <https://doc.qt.io/qt-5/qtquick-index.html>. Accessed 10.03.2021.

41 Qt for WebAssembly. 2021. Qt Documentation <https://doc.qt.io/qt-5/wasm.html>. Accessed 10.03.2021.

42 What is Vugu? 2020. Vugu Documentation. <https://www.vugu.org/doc>. Accessed 10.03.2021.

43 Vugu Files: Overview. 2020. Vugu Documentation <https://www.vugu.org/doc/files>. Accessed 10.03.2021.

44 Nattestad, Thomas. WebAssembly brings Google Earth to more browsers. 2019. Chromium Blog. <https://blog.chromium.org/2019/06/webassembly-brings-google-earth-to-more.html>. Accessed 13.03.2021.

45 Cheung, Kevin. AutoCAD & WebAssembly: Moving a 30 Year Old Code Base to the Web. 2019.QCon. <https://qconnewyork.com/ny2018/presentation/autocad-webassembly-moving-30-year-code-base-web>. Accessed 19.04.2021.

46 Soundation is the first online music production software to implement WebAssembly Threads – gains over 300% performance improvement. 2018. Soundation Station. <https://qconnewyork.com/ny2018/presentation/autocad-webassembly-moving-30-year-code-base-web>. Accessed 13.03.2021.

47 TensorFlow. 2020. Made with WebAssembly. <https://madewithwebassembly.com/showcase/tensorflow/>. Accessed 14.03.2021.

48 Introducing the WebAssembly backend for TensorFlow.js. 2020. TensorFlow Blog. <https://blog.tensorflow.org/2020/03/introducing-webassembly-backend-for-tensorflow-js.html>. Accessed 14.03.2021.

49 Bebenita, Michael. Sneak Peek at WebAssembly Studio. 2018. Mozilla Hacks. <https://hacks.mozilla.org/2018/04/sneak-peek-at-webassembly-studio/>. Accessed 21.04.2021.

50 Made with WebAssembly. 2021. <https://madewithwebassembly.com/>. Accessed 21.04.2021.

51 Getting Started. 2021. React Documentation. <https://reactjs.org/docs/getting-started.html>. Accessed 04.02.2021.

52 Leaflet. 2021. <https://leafletjs.com/>. Accessed 19.04.2021.

53 Leaflet Pixi Overlay. 2020. Leaflet.PixiOverlay GitHub. <https://github.com/manubb/Leaflet.PixiOverlay>. Accessed 19.04.2021.

54 PixiJS – The HTML5 Creation Engine. 2021. PixiJS GitHub. <https://github.com/pixijs/pixi.js#pixijs--the-html5-creation-engine>. Accessed 19.04.2021.

55 Meet Material-UI – your new favorite user interface library. 2018. Free Code Camp. <https://www.freecodecamp.org/news/meet-your-material-ui-your-new-favorite-user-interface-library-6349a1c88a8c/>. Accessed 27.04.2021.

56 Node.js. 2021. <https://nodejs.org/en/>. Accessed 04.02.2021.

57 Express.js. 2019. <https://expressjs.com/>. Accessed 04.02.2021.

58 MariaDB Docker image. 2021. Docker Hub. <https://hub.docker.com/_/mariadb>. Accessed 27.04.2021.

59 Docker Overview. 2021. Docker Docs. <https://docs.docker.com/get-started/overview/>. Accessed 04.02.2021.

60 Why Rust? Rust. <https://www.rust-lang.org/>. Accessed 10.02.2021.

61 Introduction. 2021. The `wasm-bindgen` Guide. <https://rustwasm.github.io/docs/wasm-bindgen/#introduction>. Accessed 15.03.2021.

62 Introduction. 2020.  Wasm-pack documentation. <https://rustwasm.github.io/docs/wasm-pack/introduction.html>. Accessed 15.03.2021.

63 Commands. 2020. Wasm-pack documentation. <https://rustwasm.github.io/docs/wasm-pack/commands/index.html>. Accessed 15.03.2021.

64 Dom diffing. 2019. Go Make Things. <https://gomakethings.com/dom-diffing-with-vanilla-js-part-1/#what-is-dom-diffing>. Accessed 28.04.2021.

65 Harris, Rich. Svelte 3: Rethinking Reactivity. 2019. Svelte Blog. <https://svelte.dev/blog/svelte-3-rethinking-reactivity>. Accessed 28.04.2021.

66 Gupta, Ayush. Which algorithm does the JavaScript Array#sort() function use? 2019. Tutorials Point. <https://www.tutorialspoint.com/which-

algorithm-does-the-javascript-arrayhashsort-function-use >. Accessed 21.04.2021.

67 Getting things sorted in V8. 2018, V8 dev blog. <https://v8.dev/blog/array-sort>. Updated 2019. Accessed 21.04.2021.

68 Array.prototype.sort(). 2021. MDN Web Docs. <https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/sort>. Accessed 17.02.2021.

69 Why Lodash? Lodash Documentation. <https://lodash.com/>. Accessed 17.02.2021

70 Current Implementation of sort. Rust Documentation. <https://doc.rust-lang.org/std/vec/struct.Vec.html#current-implementation-1>. Accessed 20.03.2021.

71 Sort_unstable_by. Rust Documentation. <https://doc.rust-lang.org/std/vec/struct.Vec.html#method.sort_unstable_by>. 20.03.2021.

72 Struct wasm_bindgen::JsValue. 2021. Wasm-bindgen API. <https://rustwasm.github.io/wasm-bindgen/api/wasm_bindgen/struct.JsValue.html>. Accessed 22.04.2021.

73 Static Items. 2021. The Rust Reference. <https://doc.rust-lang.org/reference/items/static-items.html#mutable-statics>. Accessed 22.04.2021.

74 SQL ODRDER BY Keyword. 2021. W3Schools. <https://www.w3schools.com/sql/sql_orderby.asp>. Accessed 22.03.2021.

75 Template literals (Template strings). 2021. MDN Web Docs. <https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Template_literals>. Accessed 25.04.2021.

76 Array.prototype.filter(). 2021. MDN Web Docs. <https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/filter>. Accessed 25.03.2021.

77 String.prototype.includes(). 2021. MDN Web Docs. <https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/String/includes>. Accessed 25.03.2021.

78 Trait std::iter::Iterator.filter(). 2021. Rust Documentation. <https://doc.rust-lang.org/std/iter/trait.Iterator.html#method.filter>. Accessed 26.03.2021.

79 Trait std::iter::Iterator.collect(). 2021. Rust Documentation. <https://doc.rust-lang.org/std/iter/trait.Iterator.html#method.collect>. Accessed 26.03.2021.

80 SQL LIKE Operator. 2021. W3Schools. <https://www.w3schools.com/sql/sql_like.asp>. Accessed 27.03.2021.

81 _.inRange. 2021. Lodash Documentation. <https://lodash.com/docs/4.17.15#inRange>. Accessed 05.04.2021.

82 Struct std::ops::Range. Rust Documentation. <https://doc.rust-lang.org/std/ops/struct.Range.html>. Accessed 05.04.2021.

83 SQL BETWEEN Operator. 2021. W3Schools. <https://www.w3schools.com/sql/sql_between.asp>. Accessed 05.04.2021.

84 What is Power BI? 2021. Microsoft. <https://powerbi.microsoft.com/en-us/what-is-power-bi/>. Accessed 10.02.2021.

85 Performance. 2021. MDN Web Docs. <https://developer.mozilla.org/en-US/docs/Tools/Performance>. Accessed 26.04.2021.

86 Struct wasm_bindgen::JsValue.into_serde(). 2021. Wasm-bindgen API. <https://rustwasm.github.io/wasm-bindgen/api/wasm_bindgen/struct.JsValue.html#method.into_serde>. Accessed 20.04.2021.

87 Shrinking .wasm code size. 2020. The Rust and WebAssembly Book. <https://rustwasm.github.io/docs/book/reference/code-size.html?highlight=optim#use-the-wasm-opt-tool>. Accessed 21.4.2021.

## React application configurations for WebAssembly modules

These changes are made to a project created with “Create React App” tool. To use WebAssembly modules in a React application, some Webpack configurations must be changed. This can be done with a tool called “React App Rewired”. It must be made sure the npm package manager and cargo package manager are installed. For loading the Wasm binaries, this project uses wasm-loader, which imports the Wasm modules directly into the application bundle.

1. Create React application with create-react-app tool

2. Install react-app-rewired and wasm-loader with npm.

3. Install wasm-pack with npm, the -g flag installs it as a global package.

4. Create new Rust project with “wasm-pack new” command.

```
npx create-react-app <application-name>
cd <application-name>
npm install react-app-rewired --save-dev
npm install --save-dev wasm-loader
sudo npm install -g wasm-pack --unsafe-perm=true
wasm-pack new <wasm-module-name>
```

5. Rewrite React application’s package.json “scripts”.

```
"scripts": {
     "start": "react-app-rewired start",
     "build": "react-app-rewired build",
     "test": "react-app-rewired test",
}
```

6. Add config-overrides.js to the project root. First a rule is made so .wasm files are ignored by the default file loader and after that, a dedicated loader, wasm-loader, for .wasm files is added.

```
const path = require('path');


module.exports = function override(config, env) {
      const wasmExtension = /\.wasm$/;

      config.resolve.extensions.push('.wasm');

       // make file-loader ignore .wasm files
      config.module.rules.forEach(rule => {
            (rule.oneOf || []).forEach(oneOf => {
            if (oneOf.loader && oneOf.loader.indexOf('file-loader') >= 0) {
                  oneOf.exclude.push(wasmExtension);
            }
      });
  });

      // add a dedicated loader for Wasm
      config.module.rules.push({
            test: wasmExtension,
            include: path.resolve(__dirname, 'src'),
            use: [{ loader: require.resolve('wasm-loader'), options: {} }]
      });

      return config;
};
```

7. Add the created Wasm module to package.json, pkg folder is created with “wasm-pack build” command. Remember the relative path of the folder.

```
“dependencies”: {
      …
      “rust-wasm”: “file:./<wasm-module-name>/pkg”
}
```

8. Import Wasm module to JavaScript. The name given to “import” must match the one in package.json.

```
const wasmModule = await import(“rust-wasm”);
```

A Wasm tool called “wasm-pack-plugin” is a Webpack plugin for Rust and could automate some of this. Could not get it to work.

## Running MariaDB server inside a Docker container with Docker-compose

Example for running a MariaDB server with Docker-compose.

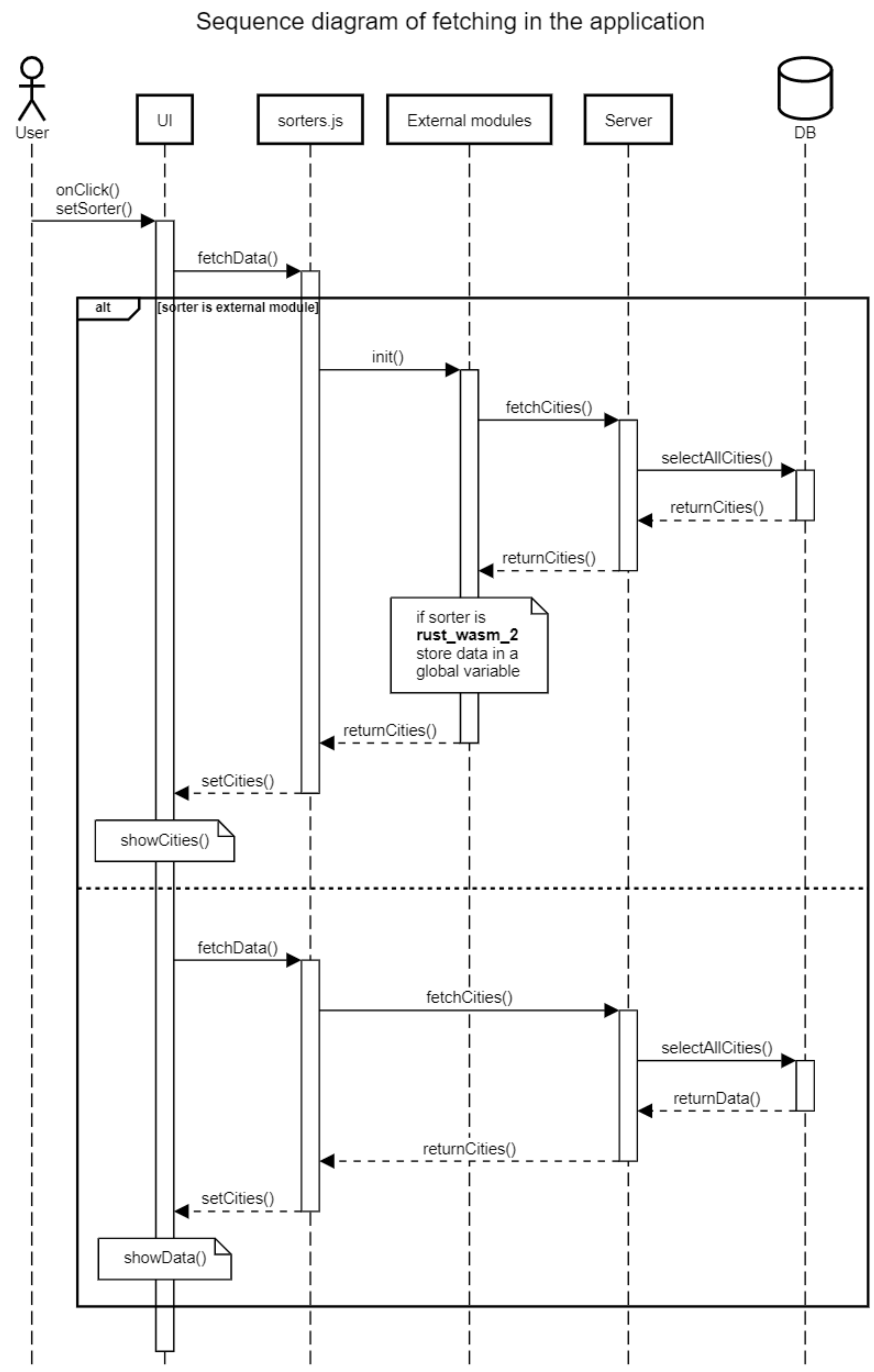
Create docker-compose.yaml file in the root of the project, with the contents shown below. The dump file resides in “./db/init” directory.

```
version: "2.2"

services:
  db:
    user: ${DB_USER}
    container_name: ${CONTAINER_NAME}
    image: mariadb
    volumes:
      - ./db/init/:/docker-entrypoint-initdb.d/
    environment:
      - MYSQL_ROOT_PASSWORD=${DB_PASSWORD}
      - MYSQL_DATABASE=${DB}
      - MYSQL_USER=user
      - MYSQL_PASSWORD=${DB_PASSWORD}
    ports:
      - "${DB_PORT}:${DB_PORT}"
```

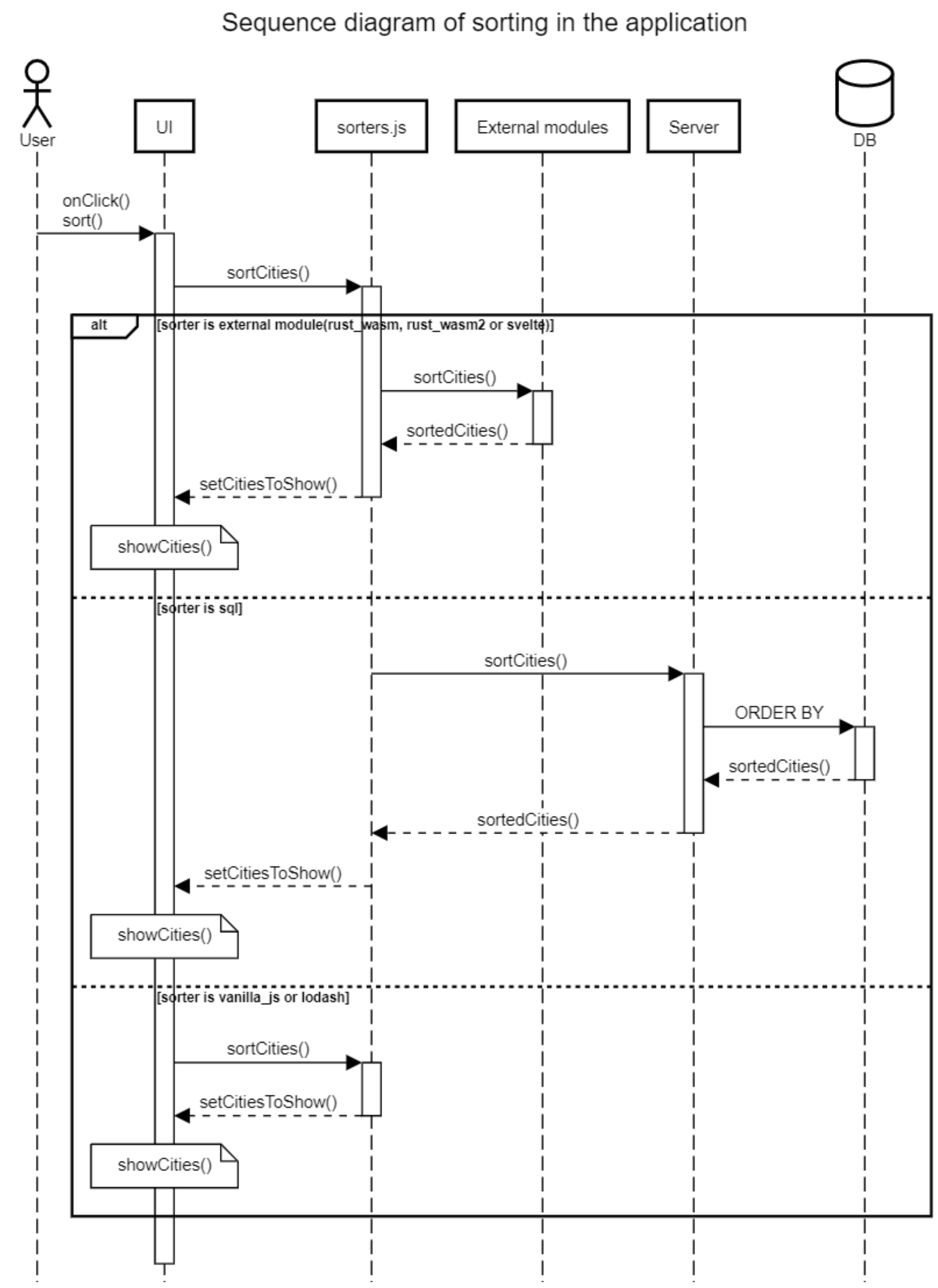
Run command.

docker-compose

## Sequence Diagram of Fetching in the Application



## Sequence Diagram of Sorting in the application


Sequence diagram of sorting in the application
User
UI
sorters.js
External modules
Server
DB
onClick()
sort()
sortCities()
alt
[sorter is external module(rust_wasm, rust_wasm2 or svelte)]
sortCities()
sortedCities()
setCitiesToShow()
showCities()
[sorter is sql]
sortCities()
ORDER BY
sortedCities()
sortedCities()
setCitiesToShow()
showCities()
[sorter is vanilla_js or lodash]
sortCities()
setCitiesToShow()
showCities()