

Aalto University

School of Science

Master's Programme in Computer, Communication and Information Sciences

Interactive Web Application Development with .NET and WebAssembly

Master's Thesis

December 15, 2022

Boris Hyttinen

Tekijä:	Boris Hyttinen
Työn nimi:	Interaktiivisten web-sovellusten kehittäminen .NET:lla ja WebAssembly:llä
Päiväys:	15. joulukuuta 2022
Sivumäärä:	62
Pääaine:	Computer Science
Koodi:	SCI3042
Vastuupettaja:	Tekniikan tohtori Lassi Haaranen
Työn ohjaaja(t):	Diplomi-insinööri Janne Kivilaakso
<p>Selaimessa renderöitävien web-applikaatioiden kehitys on perinteisesti täytynyt tehdä JavaScript:lla, sillä se on ollut ainoa selainten tukema ohjelmointikieli. Tilanne muuttui vuonna 2019, kun W3C julkaisi WebAssembly:n internetin neljänneksi viralliseksi kieleksi. Suurin osa selaimista tukee WebAssembly:n binäärikoodia, jota voidaan kääntää useista ohjelmointikielistä, kuten C# ja C++. Laajalla kielivalikoimalla pystytään vähentämään selaimessa suoritettavien sovellusten JavaScript-riippuvuutta.</p> <p>Diplomityö tarkasteli useita olemassa olevia tutkimuksia WebAssembly:n liittyen. Suurin osa tutkimuksista käsitteli WebAssembly:n suorituskykyä, turvallisuutta ja käyttökohteita. Tarkastellut tutkimukset eivät kuitenkaan juurikaan esittäneet tuloksia tai keskustelua siitä, miten WebAssembly vaikuttaa itse kehitysprosessiin. Tämä diplomityö käsittelee juuri kehitysprosessia WebAssembly:n kanssa.</p> <p>Tutkimuksen tuloksissa on suunniteltu WebAssembly:n pohjautuva kehitysprosessi, joka mahdollistaa selain-sovellusten kehittämisen .NET-arkkitehtuurilla ja C#:lla. Kehitysprosessi mahdollistaa myös muiden ohjelmointikielien hyödyntämisen, esimerkiksi C ja C++. Kehitysprosessi todennettiin toimivaksi toteuttamalla toimiva selain-sovellus. Tulokset osoittavat, että WebAssembly vähentää web-kehityksen kieliriippuvaisuutta, sekä mahdollistaa uusien ominaisuuksien lisäämisen .NET-applikaatioon.</p>	
Avainsanat:	WebAssembly, Blazor, .NET, ASP.NET, Frontend, Selaintietokanta, Kieliriippumattomuus, Interaktiivisuus
Kieli:	Englanti

Author:	Boris Hyttinen
Title of thesis:	Interactive Web Application Development with .NET and WebAssembly
Date:	December 15, 2022
Pages:	62
Major:	Computer Science
Code:	SCI3042
Supervisor:	D.Sc (Tech) Lassi Haaranen
Instructor:	M.Sc (Tech) Janne Kivilaakso
<p>Traditionally, browsers have supported one programming language: JavaScript. Client-side rendered web application development has thus been limited to JavaScript-only. This changed in 2019, when W3C announced WebAssembly as the fourth language of web. WebAssembly is a portable binary code format supported by majority of browsers. Multiple programming languages can be compiled to WebAssembly, such as C# and C++, increasing language independency of web development.</p> <p>Most of existing research focused on performance and security aspects of WebAssembly. Some results among examined research suggested use scenarios for wasm, but no clear best use cases could be interpreted. Interestingly, the existing research covered in this thesis had minimal results and discussion regarding the software engineering aspect of WebAssembly. This thesis aims to fill the gap in existing research.</p> <p>The results of this research proposed a development process based on WebAssembly, that allows developing client-side web applications with C# and .NET. The designed process also allowed development in other languages, such as C and C++. The process was verified by a proof-of-concept client-side application. The results show, that WebAssembly increases language independency of client-side development, and allows new features to be included in .NET web applications.</p>	
Keywords:	WebAssembly, Blazor, .NET, ASP.NET, Client-side rendering, In-browser database, Language Independency, Interactivity
Language:	English

Contents

1	Introduction	6
2	WebAssembly: Background and Related Work	8
2.1	Motivation	8
2.2	Background	9
2.3	Performance	10
2.4	Security	12
2.5	Existing Use Scenarios	15
2.6	Previous and Similar Solutions	17
2.7	Language Dependency of Web Development	18
2.8	.NET and WebAssembly	21
2.9	Summary	22
3	Methodology	24
3.1	Problem Interpretation & Research Context	25
3.2	Requirements Definition	27
3.3	Evaluation	32
4	Results	34
4.1	Development Process with WebAssembly	34
4.2	Created Software Artefact and Functionality	39
4.3	Evaluation Based on Defined Requirements	43
5	Discussion	52
5.1	RQ1: What are the focus areas of existing research and literature written about WebAssembly?	52

5.2	RQ2: Does WebAssembly increase language independency of .NET web application development?	53
5.3	RQ3: What features and capabilities does WebAssembly enable in the case study?	54
5.4	Threats to Validity	55
5.5	Future Work	55
6	Conclusion	57
	References	58

1 Introduction

Modern browsers offer very limited native support for programming languages, as the only built-in language so far has been JavaScript. Unfortunately, especially the performance and security of JavaScript as a programming language or a compilation target are not sufficient for many modern use scenarios, such as video games (Haas et al., 2017, p.185).

In addition to the technical limitations of JavaScript, also the language restrictions of web development affect developers. According to Breaux and Moritz (2021), employees value the possibility to work with modern technologies. With a very limited language selection, in this case only JavaScript, the attractiveness of a task or a workplace could decrease.

Meanwhile, the shortage of software developers has been present in Finland for years and will probably continue increasing, as in 2025 there could be over 25000 vacancies not filled (Ahopelto, 2018, p.1). The same issue has been identified in the US, where thousands of software developer positions could become hard to fill, as the demand is estimated to increase, while the amount of Computer and Information Science graduates decreases (Breaux and Moritz, 2021, p.40).

This thesis examines WebAssembly, a relatively new language to be utilized in web applications, alongside JavaScript. WebAssembly offers a portable low-level bytecode, which is supported by modern web browsers. Even though WebAssembly bytecode can be executed also in other environments, this thesis focuses on the features and capabilities of WebAssembly in web application development, especially .NET applications. This does not only include performance and security aspects, but also how the capabilities of WebAssembly can be utilized in the development process.

WebAssembly is not the first attempt to overcome the restrictions of JavaScript. Many different approaches have been implemented to obtain native-code performance benefits in a browser, such as Adobe Flash and Microsoft ActiveX (Musch et al., 2019, p.23). However, most of the previously presented solutions introduce platform-dependency or security risks, that have significantly limited or decreased their popularity. Additionally, these technologies require an additional plugin to be installed, as they are not natively supported.

Compared to the previous attempts presented above, WebAssembly aims to provide a safe, fast and portable solution to execute native code in a browser, as stated by Haas et al. (2017). Portability provided by WebAssembly includes not only platform and hardware independency, but also not being restricted to a single programming language.

The goal of this thesis is to evaluate the capabilities and possibilities of WebAssembly in the development of an existing .NET application. This application is provided by a Finnish SaaS company, offering invoicing automation as a service. The research is conducted by extending the application with WebAssembly code. To achieve the research goal, this thesis answers the following three research questions:

- RQ1** What are the focus areas for existing research and literature written about WebAssembly?
- RQ2** Does WebAssembly increase language independency of .NET web application development?
- RQ3** What features and capabilities does WebAssembly enable in the case study?

The structure of the thesis is the following. Chapter 2 presents the background and related work regarding WebAssembly, necessary to understand and answer the research questions. Chapter 3 presents the Design Science based research methodology, including the evaluation criteria utilized in the following chapters. Chapter 4 contains the results of the research. Finally, Chapter 5 discusses the results and their limitations, followed by conclusions in Chapter 6.

2 WebAssembly: Background and Related Work

This chapter examines existing research and literature regarding WebAssembly, focusing initially on performance, security and known use scenarios. Based on the initial specification in Haas et al. (2017), performance benefits could be considered to be the motivation behind WebAssembly. However, the possibilities of WebAssembly in web software development go beyond performance, and the existing research of these features will also be examined later in this chapter.

Additionally, this chapter reviews previous implementations for overcoming limitations of JavaScript and/or running native code in a browser, to allow comparison with WebAssembly.

2.1 Motivation

Before discussing the capabilities of WebAssembly that reason the existence and usage of it, understanding the overall motivation for such language is necessary.

Since the early years of modern Web, JavaScript has been the dominating programming language for implementing logic on the client-side of web applications (Yan et al., 2019, p.522). Among other possible reasons, the dominance of JavaScript is first of all a consequence of being the only language with built-in support in the majority of browsers. According to CanIUse (2022a), ECMAScript 2015, better known as the ES6 version of JavaScript, is supported by the browsers of more than 98% of web users.

JavaScript offers performance good enough for most use scenarios, but has its limitations. According to De Macedo et al. (2021), JavaScript ranked 15th out of 27 programming languages in tests comparing performance in 10 different software problems. The best-performing language in the particular test was C, which was more than 6 times faster and 4 times more energy efficient than JavaScript (De Macedo et al., 2021, p.225). Even though these comparisons are not directly applicable to performance in browsers, the results suggest that JavaScript might not be optimal for performance-critical code. On the other hand, WebAssembly can be compiled from C and other better-performing languages, which could lead to performance gains, especially in computationally difficult tasks.

2.2 Background

Development of WebAssembly was initiated by engineers from major browser vendors, such as Google and Mozilla (Haas et al., 2017, p.185). Having the vendors onboard from the beginning helped in getting the most popular browsers to support WebAssembly, which was a prerequisite for creating a language with native support among different users and platforms. The result can be seen in support of WebAssembly, which according to CanIUse (2022b) is over 95% among users of Web. This is a quite substantial figure, especially considering that WebAssembly is a relatively new technology, officially launched in November 2017. WebAssembly became a recommendation of World Wide Web Consortium (W3C) in December 2019, which made it officially the fourth language of Web, in addition to CSS, HTML and JavaScript (Consortium, 2019).

As the latest native language supported by browsers, WebAssembly is not built to replace JavaScript, but to complement it in areas, where JavaScript does not excel. If only specific parts of an application benefit from the features and capabilities of WebAssembly, the other parts can remain as JavaScript code.

Utilizing both WebAssembly and JavaScript in the same web application requires a possibility for these two languages to co-operate. Otherwise separate modules or functions of an application could not communicate with each other. Interoperability between JavaScript and WebAssembly is achieved using the WebAssembly JavaScript API, which for example allows importing and exporting both functions and memory between JavaScript and WebAssembly code.

As a downside, WebAssembly introduces development overhead due to the the required interoperability and bridge between WebAssembly and JavaScript code (Reiser and Bläser, 2017, p.10). If the performance or other benefit provided by WebAssembly is limited, consideration and comparison between the benefits and caused overhead is required. Such situation could arise, for example, if the amount and/or significance of performance-critical code produced with WebAssembly is limited.

WebAssembly itself is technically a binary code format, which is used as a compilation target for other programming languages, such as C++ and Rust. However, WebAssembly can also be expressed as a language with a structure

and syntax, according to Haas et al. (2017). This expression is achieved with WebAssembly Text Format, which is a human-readable representation of the binary code.

WebAssembly Text Format combined with browser development utilities, such as Chrome Developer Tools, provides a quite sufficient set of tools for the development of WebAssembly applications. Some related work has been done to provide even more transparency to WebAssembly development. One example is WasmView, which allows tracing and visualizing call graphs of function calls in both JavaScript and WebAssembly code (Romano and Wang, 2020, p.13-16). The transparency provided by call graph tracing and visualization should help develop and debug especially interactive WebAssembly applications (Romano and Wang, 2020, p.13).

2.3 Performance

A comparison between the performance of WebAssembly and JavaScript can be conducted by compiling a C program to both WebAssembly and JavaScript, preferably using a compiler that can do both compilations, such as Cheerp¹. Cheerp is a commercial, open-source compiler, that can use both JavaScript and WebAssembly as a compilation target, for C and C++ code. Cheerp is based on the LLVM/clang² compiler and compilation strategy. Another compiler utilized in the presented performance tests is Emscripten³, also based on the LLVM/clang architecture.

Extensive research conducted by Yan et al. (2019) revealed, that WebAssembly execution times significantly outperformed JavaScript with small input sizes, but larger inputs caused some programs to be faster with JavaScript. The memory usage of WebAssembly was 2,3x higher with small input sizes, but very large input caused WebAssembly to consume over 100 times more memory compared to JS (Yan et al., 2019, p.541). Memory-efficiency of JavaScript is better due to garbage collection, while WebAssembly utilizes linear memory which has to be extended with large input sizes.

¹<https://docs.leaningtech.com/cheerp/WebAssembly-output>

²<https://clang.llvm.org/>

³<https://emscripten.org/docs/compiling/WebAssembly.html#webassembly>

Similar results have been reported by other studies, regarding execution times with different input sizes. A study by De Macedo et al. (2021) also confirmed that WebAssembly was faster than JS, but the difference between the two diminishes with larger input sizes.

The relative performance of WebAssembly compared to JavaScript is not straightforward to measure, as it depends on several variables, most importantly on the task being executed. Compiler and runtime environment selection also affects the performance. As an example, Yan et al. (2019) showed, that C programs compiled with Emscripten are 2,7x faster compared to Cheerp, and Firefox desktop browser executes WebAssembly 1,6x faster compared to Chrome. The tests between WebAssembly and JavaScript were done with Chrome, meaning the performance advantage of WebAssembly could have been more significant with Firefox. This causes a generalizability issue regarding the results of the presented research, as well as other performance tests conducted with WebAssembly.

Performance of WebAssembly has also been compared to native code performance in several existing research papers. As it is possible to compile WebAssembly binary code from C/C++, comparing the performance of the same source code in different environments can provide interesting information. Sandhu et al. (2018) compared the performance of native C, WebAssembly, and JavaScript in sparse matrix-vector multiplication. As a result, WebAssembly quite surprisingly had similar or better performance compared to native C, and outperformed JavaScript by 2-3 times faster execution, depending on the used platform (Sandhu et al., 2018, p.7).

The previously mentioned research evaluated performance based on a minimal scientific computation. According to Jangda et al. (2019), these types of tests do not reveal the performance of WebAssembly in more extended use scenarios, such as image recognition and simulation tasks. However, evaluating performance with extensive applications is more challenging, as WebAssembly compilers, such as Emscripten, might not provide the necessary support for system calls or memory scalability (Jangda et al., 2019, p.108). The research conducted by Jangda et al. (2019) proposed an in-browser kernel, BROWSIX-WASM, that allows WebAssembly to utilize system services, with only 0.2% added overhead. The results for SPEC CPU2006 and SPEC CPU2017 test suites showed, that WebAssembly was 1.45x slower compared to native code (Jangda et al., 2019, p.111). The results can be

considered quite promising, both the nearly insignificant overhead, as well as the relative performance.

One notable area without proper existing research is the code size of WebAssembly compared to pure JavaScript. According to Haas et al. (2017), JavaScript source code is less compact than WebAssembly binary code, introducing more overhead in bandwidth and loading times. Unfortunately, no existing research results were found that would prove this claim. WebAssembly binary code size can be optimized with compilations settings, but the variance between different optimization levels was less than 2% (Yan et al., 2019, p.539). Unfortunately, this paper did not provide comparisons between JS and WebAssembly code size.

2.4 Security

Safety considerations regarding code executed directly in the browser are extremely relevant, since the code is often downloaded from untrusted sources. Haas et al. (2017) stated, that safety is an important goal for WebAssembly, but not easy to achieve due to the memory-unsafe nature of compiled C and C++ code. Disselkoe et al. (2019) divided the memory safety concerns into three categories: spatial safety, temporal safety, and pointer integrity.

As an example, a pointer referencing memory that already has been deallocated causes undefined behavior, threatening the temporal safety of the memory. Preventing such situations would require the compiler to confirm if the referent of a pointer has been deallocated, which causes significant overhead (Disselkoe et al., 2019, p.2).

Spatial safety concerns are a result of the linear memory utilized by WebAssembly. Linear memory is continuous without any unmapped pages, which causes every pointer between the 0 and maximum address to be valid. As a consequence, buffer and stack overflows are powerful against WebAssembly applications (Lehmann et al., 2020, p.221).

The study conducted by Lehmann et al. (2020) was able to utilize the security vulnerabilities of C/C++ code compiled to WebAssembly to enable Cross-Site Scripting and server-side Remote-Code Execution. These are severe vulnerabilities,

that were allowed mainly due to unmapped pages, as well as missing implementations of read-only memory and ASLR (Address space layout randomization) (Lehmann et al., 2020, p.221). Even though these issues are partly a consequence of WebAssembly design choices, especially linear memory, they are nevertheless very exploitable. Interestingly, the safety concerns are in conflict with the initial publication of WebAssembly by Haas et al. (2017), which as mentioned earlier, states safety as an important goal of WebAssembly.

Temporal and spatial safety could be enforced, but the need to do so depends on the use scenarios. The question is, whether the performance lost is more significant than the security gained. As an example, an authentication service would probably value increased safety over performance. For such needs, Disselkoe et al. (2019) proposed MS-Wasm to provide a balance between overhead and memory safety, by enforcing both software and hardware mechanisms to detect and prevent vulnerabilities. Hardware mechanisms, such as ARM pointer authentication could perform better compared to software techniques, but since WebAssembly is designed to be portable, it cannot rely on specific hardware only.

WebAssembly code and memory are fortunately executed in a sandbox, meaning vulnerabilities and safety issues by default do not compromise other parts of the runtime environment. Execution stack, data structures, and code are separated from the linear memory controlled by WebAssembly (Lehmann et al., 2020, p.217). However, according to Hilbig et al. (2021), the vulnerabilities of WebAssembly code could affect the host environment through security-critical functions imported to WebAssembly modules, such as JavaScript *eval*. The research by Hilbig et al. (2021) examined an extensive dataset of WebAssembly binaries, of which 21% imported a security-critical JavaScript function, that would leave the underlying environment vulnerable (Hilbig et al., 2021, p.2704).

Again, the security concerns presented above are related to the compilers and source languages used, even though they are finally allowed by the design choices of WebAssembly. A few proposals have been made to prevent attacks caused by the vulnerabilities, such as MS-Wasm presented earlier in this section. MS-Wasm was only suggested on paper, but other implemented proposals also exist. Aimed to be used in cryptographic web applications, Protzenko et al. (2019) proposed the

following, formally verified translation:

$$\lambda ow^* \rightarrow Cb \rightarrow WebAssembly$$

This translation is somewhat similar to an existing translation first from Low^* to C, and then from C to WebAssembly with Emscripten. However, the latter approach suffers from the vulnerabilities discussed earlier in this section, which considering the cryptographic functionality is a severe issue.

The proposed new translation was utilized to build secure WebAssembly versions of $HACL^*$ and especially Libsignal. $HACL^*$ is a cryptographic library originally written in F^* , that implements encryption algorithms and message authentication protocols (Zinzindohoué et al., 2017, p.1789). Libsignal implements the Signal protocol, which provides end-to-end encryption in messaging and voice calls. For example, both Libsignal and Signal are utilized by WhatsApp (Protzenko et al., 2019, p.1256)

The original Libsignal JavaScript implementation has side-channel attack vulnerabilities. However, the WebAssembly implementation, $Libsignal^*$, is quite promising, as stated by Protzenko et al. (2019, p.1266): "Our Signal implementation is likely the first cryptographic protocol implementation to be compiled to WebAssembly, and is certainly the first to be verified for correctness, memory safety, and side-channel resistance."

Interestingly, Protzenko et al. (2019) also focused on building a defensive API between WebAssembly and JavaScript code. Even though WebAssembly code itself runs in a sandbox, JavaScript code can read and write to WebAssembly memory. This could allow bugs in JS code, such as in the Libsignal implementation, to depreciate the security guarantees of WebAssembly code.

As another option for preventing Spectre side-channel attacks, Narayan et al. (2021) presented Swivel as an extension to Lutec-compiler for WebAssembly. Spectre-attacks are a set of microarchitectural attacks, that cause speculative execution of instruction sequences not originally included in the program (Kocher et al., 2019, p.1). Standard Lutec compilation was found vulnerable for three proof of concept attacks, but the Swivel extension was not vulnerable (Narayan et al., 2021, 1444). Swivel introduced 47% overhead for the software-only version, and 96% for the hardware-oriented version utilizing Intel CET and MPK, but promisingly both of

these solutions outperform modern fence-based technologies (Narayan et al., 2021, p.1434).

2.5 Existing Use Scenarios

Existing research and literature about WebAssembly is often focused on different use cases. Since the language was published quite recently, the most suitable use cases could still remain to be found. This section examines and presents some of the use scenarios that have been researched.

As suggested in the previous section, WebAssembly has potential use cases in the field of cryptography, that will not be discussed again in this section.

IoT

Even though the scope of this thesis is limited to web applications, it is relevant to discuss also other intensively researched areas of WebAssembly usage. Based on the amount of existing research, one of these is clearly the Internet of Things (IoT).

Mäkitalo et al. (2021) proposed replacing native technologies in IoT devices with WebAssembly, to achieve isomorphic software architecture, where hardware-related constraints can be removed from the application code itself. WebAssembly runtime would be used to build lightweight containers for IoT devices, with additional implementation of execution time dynamic linking that allows devices to load different modules based on self-diagnosing their role (Mäkitalo et al., 2021, p.7).

Similarly, Li et al. (2022) suggested a lightweight WebAssembly runtime WAIT to solve the structural limitations of IoT devices. The constraints and challenges include limited memory, especially RAM, an unsafe execution environment due to device-cloud integration, and limited energy resources (Li et al., 2022, p.261-262). As mentioned earlier, WebAssembly is not memory efficient at least compared to JavaScript, which combined with very limited RAM available in IoT devices is problematic. On the other hand, promising results were published in Hasselt et al. (2022) regarding the energy efficiency of WebAssembly, as the research resulted in JavaScript consuming 2.6x more energy than WebAssembly.

Games

Modern video games are typically executed as desktop applications, but WebAssembly could provide a growing platform for browser-based video games in the near future. Many of the popular game engines, such as Unity and Unreal Engine, are written at least partially in C++, suggesting that compiling video games to WebAssembly could be reasonable. According to both Musch et al. (2019) and Hilbig et al. (2021), games appear to be the most popular application domain for unique WebAssembly binaries.

Security

In addition to legitimate use cases presented above, WebAssembly has been used in malicious ways. The research conducted in Musch et al. (2019) analyzed the utilization of WebAssembly among Alexa top 1 million sites. The results revealed, that at the time of writing in 2019, 56% of websites loading WebAssembly modules were utilizing it for malicious purposes (Musch et al., 2019, p.11). The malicious categories included cryptojacking and code obfuscation. However, it is notable, that only 1 of every 600 sites loaded WebAssembly modules, meaning the sample size was quite limited.

Cryptojacking means utilizing computing resources for cryptocurrency mining without proper authorization. As WebAssembly can offer near-native performance while executed in a browser, it offers an effective platform for cryptojacking. Fortunately, the popularity of cryptojacking has led to the development of software, which can detect cryptojacking features from a Wasm module.

Circumventing such detection software can be tried with code obfuscation. There are several different obfuscation techniques, with the same goal of hiding what actions the given code is performing. Obfuscating techniques occurring at the source code level proved to be very efficient, as 93% of obfuscated cryptojacker samples were not detected by a state-of-the-art cryptojacking detection software MINOS (Bhansali et al., 2022, p.145). Code obfuscation can also be used to hide JavaScript and HTML code into wasm-modules, where they are not detected or understood by analysis tools, such as adblockers.

Nevertheless, possibly as a result of the research and development done with malicious WebAssembly code detection, the relevance of cryptojacking as a problem has decreased: A recent study conducted on use cases of WebAssembly in 2021 showed, that only 1% of binaries in the dataset identified as possible crypto mining software (Hilbig et al., 2021, p.2705). Compared to Musch et al. (2019), the relative portion of cryptojacking WebAssembly code has vastly decreased.

2.6 Previous and Similar Solutions

The predecessors of WebAssembly for running low-level native code on the Web were Native Client, asm.js and ActiveX (Haas et al., 2017, p.198).

Asm.js is a proper subset of JavaScript, which can be used as a compilation target for statically typed languages, such as C. This is allowed by removing dynamic features of JavaScript, which for example enables manual memory management featuring in C. As the remaining language features can be compiled and optimized ahead of time, asm.js achieves a performance benefit compared to regular JavaScript (Es et al., 2016, p.1944).

Since asm.js is pure JavaScript, it can be executed in all browsers that support JavaScript, making asm.js very hardware and platform independent. This does not come without a price, since introducing new features to asm.js would first require extending JavaScript with the same features (Haas et al., 2017, p.186). This can be seen as a clear disadvantage compared to WebAssembly, which has anyway achieved nearly the same support among browsers, without the JavaScript dependency. Performance-wise, according to the original specification, WebAssembly was 1.3x faster in performance tests compared to asm.js (Haas et al., 2017, p.197). Jangda et al. (2019) confirmed these results, as the research also showed a similar 1.3x speedup from asm.js to WebAssembly.

Native Client, also known as NaCl, was developed and published by Google in 2011. Similar to WebAssembly, the goal was to bring native code performance to browsers without compromising safety (Yee et al., 2009, p.79). Native Client supports running x86 and ARM code in a sandboxed environment, which provides more isolation and security. Unfortunately, NaCl and the more portable version of it called PNaCl, were only supported by Google Chrome, which severely decreased their portability (Haas

et al., 2017, 186). Native Client was finally deprecated by Google in 2020.

Lack of portability has clearly been a limiting factor in the popularity of previous solutions to execute native code in a browser. Microsoft ActiveX relied on a browser extension to operate, inherently reducing portability similar to other predecessors. Compared to previous attempts, WebAssembly has quickly gained extensive support among browsers.

2.7 Language Dependency of Web Development

In the early stages of Web, application logic was executed mainly on the server-side, but adopting JavaScript allowed building modern web applications, with more logic and interactivity added to client-side (Møller, 2018, p.106). Until the release of WebAssembly, JavaScript has also remained as the only natively supported language in browsers, alongside CSS and HTML.

As a result, the language selection available for front-end web development has been very limited. In client-side web applications, JavaScript was the only compilation target that can be used to build logic into the application. Partly to circumvent this issue, numerous web frameworks and development platforms have been created to allow writing web apps using other languages. As an example, Flask allows the creation of web pages with Python, whereas .NET web applications are written in C#.

Even with the utilization of such frameworks, the original problem could not be solved: No matter what framework is used, the logic executed in the client browser must have been written in JavaScript. Especially with modern single-page applications (SPA), the role of client-side logic written in JavaScript has increased, as has the relevance of its limitations.

First of the above-mentioned limitations is obviously performance, which was covered in Section 2.3. This section focuses on the other limitation: If client-side logic must be written in JavaScript only, web developers have no other choice but to use JS. This issue was identified by Microsoft documentation as well: "SPAs use AJAX and HTML5 to create fluid and responsive Web apps, without constant page reloads. However, this means much of the work happens on the client side, in

JavaScript. For the traditional ASP.NET developer, it can be difficult to make the leap.” (Wasson, 2013)

The problem can be presented in greater detail using an example. Consider a traditional ASP.NET developer working as a software developer: If the developer has no freedom in language selection, both the particular task and consequently the employment could become significantly less attractive, in case the developer does not favor JavaScript. On the other hand, the employer is forced to either find or train developers who can write the exact language, which limits the potential candidates.

WebAssembly could provide a solution to the described situation, as it can be executed on the client-side. Theoretically, any language which can be compiled to WebAssembly, could be used to write client-side logic for a web application. Regarding the available languages, the official WebAssembly Core Specification does not state any official or initial languages to be compiled to WebAssembly (W3C, 2019). According to the specification, WebAssembly ”does not privilege any particular language, programming model, or object model.” (W3C, 2019)

Nevertheless, the WebAssembly team initially focused on building binary code from C and C++ (Møller, 2018, p.106). As a result, C/C++ have remained as the most popular source code language, since 64% of WebAssembly binaries are compiled from C and/or C++ (Hilbig et al., 2021, p.2701). Besides C and C++, only Rust had a notable market share with 15%, followed by AssemblyScript and Go with less than 5% combined share (Hilbig et al., 2021, p.2702).

Below are presented some of the languages, that have existing compiler implementations to produce WebAssembly code. It is worth noting, that this is not a complete list of the languages or compilers. C# could as well be part of the list, but due to the close relation with .NET framework, it is discussed in the next section.

C/C++

C and C++ are the most popular source languages to be compiled to WebAssembly, and have been supported from the early steps of WebAssembly. This has also affected the existing research of Wasm, which at least based on the sample of this chapter, is often conducted using WebAssembly code built from C or C++.

Some of the compilers have been mentioned earlier, including Emscripten and Cheerp. Both of these utilize LLVM in the background, which is a quite dominant toolchain: According to Hilbig et al. (2021), every C, C++ and Rust compiler targeting WebAssembly is based on the LLVM toolchain. Since these languages are also the most popular source code languages for WebAssembly, nearly 80% of Wasm binaries are built with compilers utilizing LLVM (Hilbig et al., 2021, p.2702). As mentioned by Hilbig et al. (2021), such popularity introduces notable side effects, since a security bug in LLVM could compromise the majority of existing WebAssembly binaries.

Rust

Rust⁴ is a relatively new programming language compared to C and C++, as the first stable version was released in 2014. Rust is designed to provide a tool set that allows low-level control and features, for example regarding memory management, similar to C. However, Rust helps avoiding some of the pitfalls in memory management: As an example, the compilers do not allow dangling pointers. Somewhat similar development can be seen in recent versions of C++, with the introduction of smart pointers.

Rust can be compiled to WebAssembly using *wasm - pack*⁵, which is also the recommended compiler (MDN Web Docs, n.d.). *wasm - pack* provides additional features via an API to help interoperability between WebAssembly and JavaScript code, such as calling WebAssembly functions with strings from JS code.

TypeScript

A variant of TypeScript can be compiled to WebAssembly binary format using AssemblyScript⁶. AssemblyScript compiles WebAssembly binary code using a Binaryen toolchain⁷. Even though TypeScript implements some of the necessary typing required in WebAssembly, it still lacks some strict typing checks.

⁴<https://www.rust-lang.org/>

⁵<https://github.com/rustwasm/wasm-pack>

⁶<https://www.assemblyscript.org/introduction.html>

⁷<https://github.com/WebAssembly/binaryen>

Additionally, as TypeScript is a superset of JavaScript, it also implements some dynamic features of JavaScript, that cannot be present in WebAssembly. The variant lacks these features to allow compilation to WebAssembly.

A positive feature in AssemblyScript is, that the source language is quite similar to JavaScript, allowing a JavaScript developer to learn it with less effort. As a result, WebAssembly binaries can be written by a JS developer with less training.

Go

Go, also known as Golang, is a statically typed programming language, released initially in 2009. The syntax of Go is similar to C, but it implements additional features, such as memory-safety and garbage collection. Go does not require any other compilers to be built into WebAssembly, but can be compiled to a Wasm binary as such, supporting all language features.

There is also an alternative for the standard Go compiler, which is TinyGo⁸. According to TinyGo specification, it is designed to be used especially in embedded systems, as the resulting binary size is significantly smaller compared to the standard Go compiler. WebAssembly has use cases in Internet of Things, where limited binary sizes are of great importance.

2.8 .NET and WebAssembly

The design science research conducted and presented in Chapters 3 and 4 will be done utilizing an existing web application. The application is developed with .NET Framework, so it is reasonable to also provide some background related to this framework.

.NET is a software development platform originally developed by Microsoft, which allows creating applications for several platforms, such as desktop and web applications. Two main components of the framework are Common Language Runtime (CLR) and Class Library. CLR manages the execution of applications, and for example is responsible for memory management. Class Library gives applications access to a common API and types, such as file system functions.

⁸<https://tinygo.org/docs/guides/webassembly/>

The web application-specific functionality and tools are provided in ASP.NET, which is an extension to .NET framework. ASP.NET has multiple versions, but the application utilized in this thesis is built using ASP.NET MVC⁹. This version of the web framework utilizes a Model-View-Controller design pattern. MVC pattern was described by Krasner and Pope (1988) as following: "This three-way division of an application entails separating (1) the parts that represent the model of the underlying application domain from (2) the way the model is presented to the user and from (3) the way the user interacts with it."

As suggested by ASP.NET Documentation (2022), MVC represents a traditional server-rendered approach. In addition to the more traditional MVC approach, ASP.NET also has a solution for client-side rendered web applications: Blazor WebAssembly¹⁰. Blazor allows the creation of interactive websites with C#, instead of JavaScript.

By default, Blazor WebAssembly applications are executed in a .NET runtime implemented in WebAssembly. The browser utilizes a .NET Intermediate Language Interpreter in the execution, which decreases the performance of the application, due to missing Just-in-time compilation.

The performance of Blazor WebAssembly applications can be increased by allowing Ahead-of-time (AOT) compilation. The .NET code of the application is compiled directly to WebAssembly Ahead-of-time, after which it can be natively executed by browsers. AOT compilation can introduce major performance improvements compared to Language Interpreter. However, since the AOT compiled application typically has a larger size, the initial download of the website is slower. Additionally, the compilation can require a long time with larger projects.

2.9 Summary

WebAssembly is a binary code format that can be executed by most popular browsers. The binary code can be compiled from multiple programming languages, such as C, C++ and Rust. As a result, client-side rendered web applications can

⁹<https://dotnet.microsoft.com/en-us/apps/aspnet/mvc>

¹⁰https://docs.microsoft.com/en-gb/aspnet/core/blazor/?WT.mc_id=dotnet-35129-website&view=aspnetcore-6.0

be developed with these languages, even though JavaScript is still required for importing the binaries, as well as altering the rendered Document Object Model.

Based on existing research, the key benefit of utilizing WebAssembly is near-native performance. Especially with smaller input sizes, WebAssembly had significantly better performance compared to JavaScript. The performance of WebAssembly code compiled from C also proved to be compatible with the same code executed natively.

Security was stated as an important goal of WebAssembly in the initial publication by Haas et al. (2017). However, based on existing research, there are severe security concerns with WebAssembly memory management. For example, cross-site scripting and remote-code execution attacks were performed successfully by Lehmann et al. (2020). Fortunately, the vulnerabilities of WebAssembly do not directly compromise other parts of software, as browsers execute WebAssembly binary code in a sandbox.

WebAssembly has an implementation for the Microsoft .NET web development framework, named Blazor WebAssembly. It allows developing client-side web applications with C#, instead of JavaScript. Blazor WebAssembly will be discussed more in later chapters of this thesis.

3 Methodology

WebAssembly could be researched in many different fields of science. For example, the performance, language semantics and underlying implementation could fall into the field of formal science. However, this thesis is more interested in identifying and solving a more specific problem with WebAssembly. Even though the solution and proof of concept will be presented in the context of an existing application, both the problem and solution are of general interest. These qualities place this research in the field of design science, aiming to solve a practical issue that can be generalized (Johannesson and Perjons, 2014, p.8).

The research in this thesis is conducted using a method framework for design science research, proposed by Johannesson and Perjons (2014). The framework consists of the following activities, which will be further explained and presented in Chapters 3 and 4 (Johannesson and Perjons, 2014, p.78):

- Problem interpretation
- Requirements definition
- Artefact design and development
- Artefact demonstration
- Artefact evaluation

The research is not done linearly from phase to phase, but more in a cyclic fashion. Especially the three first phases are iterated over and over until requirements and the artefact satisfy all stakeholders. The group of stakeholders consists of developers and a product manager.

After the artefact design and development is complete, the designed functionality and structure is demonstrated in a proof of concept application. Proof of concept will be implemented in relation to an existing web application, described more in depth below.

3.1 Problem Interpretation & Research Context

The first step of a design science research project is to understand and define the addressed problem, as well as the context in which the problem exists. A strict interpretation of the problem is needed to allow further steps in the research. Most importantly, without accurate problem interpretation, defining requirements for the actual solution would become impossible.

The research is conducted for a Finnish SaaS company, later referred to as "Company" in this thesis. The Company offers invoicing automation services to business customers. The customers and internal consultants control the invoicing automation process with a web application.

The web application was originally developed with ASP.NET MVC, and this design pattern is still in use in the current version. In applications built with ASP.NET MVC, the rendering happens mainly on the server side, meaning the HTML files are created on the server and sent to the client. Additionally, developers still can include JavaScript code and libraries in the HTML files. JavaScript code can be executed in the browser, allowing for example the utilization of jQuery¹¹ to fetch new data from the server to be shown in the UI.

As a result, the utilization of ASP.NET MVC requires all client-side rendered code to be written in JavaScript. A few contradictions resulting from this limitation can easily be identified. First, one of the key features of a web framework, such as ASP.NET, is to provide tools that make web development and deployment easier. In comparison, a developer working without any framework could as well create a web application by writing HTML files by hand and adding as much code as necessary inside the `<script>` tags. Still, for larger applications, this approach would be completely infeasible and unattractive to developers.

In the context of the ASP.NET MVC application provided by the Company, similar characteristics arise: JavaScript code has to be included in the `<script>` section, losing many capabilities of the framework. For example, the interoperability with the actual framework code, such as classes defined in C#, is not sufficient. Based on the input from the stakeholders, JavaScript and C# code in a .NET project could almost be considered as two separate entities, even if they are both used to build

¹¹<https://api.jquery.com/>

a single page. According to the stakeholders, this becomes problematic especially when hiring and training new developers, as understanding and especially writing functionality with mixed JS and C# code is very challenging.

Additionally, .NET developers might not have the required skills or interest to write JavaScript code, even though it is one of the most popular programming languages. Hiring new developers or training the existing ones to write JavaScript also introduces more R&D cost. These issues were presented in Section 2.7, and were confirmed by the stakeholders during the research.

Concluding the problem interpretation so far, the technical problem has been specified as client-side logic and functionality being strictly limited to JavaScript. This interpretation is quite general and not dependent on the context, and was the starting point-of-view for the thesis process, found through limitations of the existing application and development process used in the Company.

In order to conduct the research, the problem and the severity of it has to be specified in a more precise fashion, related to the context where it exists. Understandably, client-side executed code being restricted to only JavaScript would not be an issue, if the amount of client-side code required in the (ASP.NET) application is limited. For example, if the functionality of JS code is limited to updating data of a server-rendered grid, writing JavaScript inside the script tags could be sufficient. In such situations, JavaScript does change the contents of the DOM (Document Object Model) displayed in the browser, but the structure is left mostly untouched. JavaScript of course does have all the necessary functionality to do any necessary changes also to the structure of a website, but it introduces a vast amount of complexity.

Rendering the HTML structure can be done on server-side, if the structure of the website is known or can be evaluated beforehand. However, deciding the structure of the web page before first render introduces a clear limitation: It removes to some extent the possibility of user interaction, since the actions of the user and the resulting views cannot be known at beforehand. This would leave two options: Request a new server-rendered HTML file based on user actions, or write client-side JavaScript code that changes the HTML file structure and contents. The first option introduces delays and bad user experience due to loading times. The latter

JavaScript-oriented approach introduces the issues related to JavaScript usage.

In the context of this thesis and the application of the Company, interactive features have existed so far. But as described, they have been written in JavaScript, as there have been no other options. Interactivity is required in the application to allow development of features, that would otherwise not be feasible. One example of such feature is allowing the end user to drill into provided data to see more details. Additionally, interactive pages can often be considered more attractive to the users.

After multiple iterations and discussions with the stakeholders, the resulting problem definition in the given context can be expressed as following: ASP.NET MVC applications require writing JavaScript code to produce interactive web pages, introducing complexity in the development process and the application itself.

As required by the method framework for design science research by Johannesson and Perjons (2014), the given problem has to be of general interest. The described problem can be generalized to affect all interactive web applications developed with server-side ASP.NET. The question remains, whether ASP.NET is still a popularly utilized framework for web development: According to a questionnaire conducted among professional developers, 20% of developers used or planned to use ASP.NET Core in 2021 (Stack Overflow, 2021). High popularity of the framework combined to a generalizable problem is sufficient to make the problem of general interest.

3.2 Requirements Definition

After the problem has been interpreted, specifying strict requirements for the artefact is necessary. Requirement definition in the given method framework should express the problem in the form of demands and requirements (Johannesson and Perjons, 2014, p.78). According to the framework proposed by Johannesson and Perjons (2014), "The requirements will be defined not only for functionality but also for structure and environment." The defined requirements are utilized both in the design of the artefact, as well as in the evaluation.

The requirements provide a frame that the proposed artefact should fit into. Importantly, an artefact that fulfills the requirements should also solve the interpreted problem. The artefact in the context of this thesis refers to a piece

of software developed with a specifically defined process. Both the software and the development process, including the utilized tools and technologies, are interesting in this context, and requirements are thus set for both of them.

To summarize the frame set by the requirements, the research results should propose a development process, that allows developing client-side rendered web applications with .NET. The development process should not require writing JavaScript code, to mitigate complexity and overhead. The research results should then show, how the designed development process can be utilized to produce a software artefact. The software artefact should demonstrate the capabilities of client-side rendering, such as interactivity.

The exact requirements are presented in Table 1. This section presents the requirements in more detail, to support evaluation of them later in the thesis.

Defining required parameters for the artefact was an iterative process, that also required input from stakeholders. Even though the parameters could mostly be inferred from the problem interpretation, discussions with the representatives from the Company ensured, that the parameters are inline with expectations. Additionally, well defined and aligned requirements allowed the evaluation to be done more independently from the Company and stakeholders.

The defined requirements can be divided into two categories, as also suggested in the Method Framework for Design Science Research (Johannesson and Perjons, 2014, p.85):

1. **Functional requirements** are defined for the exact capabilities and functionality of the designed artefact.
2. **Non-functional requirements** define the artefact more broadly

Without requirements definition, the evaluation of the proposed solution would rely mainly on answering the question "Did this artefact solve the problem". Understandably, also this particular question has to be answered by the end, as it is built into the research questions. However, answering the question becomes significantly more feasible, if the requirements and parameters for an accepted artefact or solution are predefined.

	Artefact (software)	Development Process
1	Artefact includes interactive features	Components are rendered client-side
2	Existing parts of the application remain functional	Mix of server and client-side rendered components supported
3	-	Client-side rendered code can be written in several other languages than JS
4	Share authentication state with existing application	-
5	Minimal data requests to server	Support technologies to store large amounts of requested data in the session
6	Support filtering and sorting data locally	-
7	Sufficient performance from user perspective with large amounts of data	Utilized technologies should have similar performance to JS in browser
8	Function properly in modern browsers	Portable and well supported technologies used, no 3rd party plugins
9	-	High compatibility with .NET development framework
10	Similar UI components with existing application	Support most popular UI libraries

Table 1: Requirements defined for the development process and resulting software artefact

The approach used to define the requirements was to initially design the required features of the software artefact. Then, the corresponding requirement for the development process could be derived, if applicable. Additionally, requirements 3 and 9 presented in Table 1 are strictly related to the development process.

The division between functional and non-functional requirements becomes more vague due to the nature of the artefact. One example would be portability: Considering the artefact, portability is a non-functional requirement, as it cannot be defined as a functionality of the resulting software. However, from the process perspective, the requirement defines that used technologies are portable among

browsers, which is a functional requirement. The following categorization between functional and non-functional requirements is done firstly from the perspective of the artefact.

Functional Requirements

The key demand for the proposed process is to allow the addition of interactive features into an existing ASP.NET Core web application. Interactive features in this context are defined as changes to the rendered HTML structure and content based on user actions. Implementation of such features should require requests to the server only to fetch data, in other words requesting a new server-side rendered HTML file is not allowed.

The possibility to render interactive features on the client side should be possible even if other parts of application are rendered on server-side. This requirement is very relevant due to the existing application provided by the Company, as it is not feasible or realistic to redesign the complete application to be rendered client-side. Thus, other and especially existing parts of the application should not be required to be rendered client-side, in order to the proposed artefact to be rendered in the browser.

The first two requirements could be fulfilled by adding JavaScript to the server-rendered HTML pages. However, development of interactive features is required not to be limited to a single programming language. This inherently means, that JavaScript cannot be the only language utilized to develop interactive features to the software artefact. Essentially, this requirement also implies, that the languages available to develop client-side code cannot be limited to some other programming language either. Instead, the development process should allow developing the interactive features with a variety of programming languages.

Regarding authentication, the software artefact should share the same authentication state with the existing parts of the application. In other words, the software artefact should not require the user to login separately.

The use scenario of the existing application is related to invoicing automation, which requires it to manage and visualize very large amounts of data. Traditionally, a server-side rendered application would be required to request and reload data

from the server after most user actions, even if the data was previously fetched a minute ago. Additionally, most of data filtering would be done on the server. Both of these characteristics would cause server load, resulting in possibly decreased performance for all users, as well as increased costs of running the server. The proposed development process should provide a solution to both problems, as defined below.

The proposed process should provide a solution that minimizes the repetitive data requests to the server. The optimal performance would be to request data only if the particular data set has not been requested earlier in the session. Additionally, only relevant data should be requested when needed, not the complete data set. Understandably, the latter requirement could be restricted by server/API capabilities, if the endpoints do not have a sufficient parameter selection to only request relevant data.

The resulting software artefact should allow filtering data on the client-side, even if the amount of data is large, approximately 10000 rows. The software artefact should additionally be capable visualizing large amounts of data to the user, without significant performance drops.

Non-functional Requirements

Performance-wise, the development process should allow creating interactive features that result in seamless user interaction. The performance from the user perspective should at minimum be similar to a solution built with JavaScript. Strict performance metering is not part of the scope of this thesis, but would include loading times and delays after user input. As they are not metered, no strict performance requirements are set. Still, the end result is required to be usable from user perspective. Specifically, the artefact should be capable of visualizing large amounts of data to the user without significant performance drops. 10000 rows is considered to be enough data for evaluation, in the context of this research.

As mentioned, the resulting software artefact should be portable. In addition to language independency, the development process should produce software that does not require a specific execution platform or hardware to function properly. In practice, the software artefact should be supported by the majority of modern web

browsers. Utilization of third-party plugins would reduce the portability, and thus is forbidden.

As the research is aimed to solve a problem regarding ASP.NET applications, the proposed development process is naturally required to be compatible with the framework. More precisely, the process is required to utilize tools and technologies that are natively supported by the ASP.NET development platform. Additionally, the process should not introduce unjustified complexity and overhead to the development.

Modern web applications often utilize a third-party UI library. Especially with existing web applications, it is very important to be able to utilize the same UI library in new components also. Otherwise, the user interface would easily become inconsistent, as different components are used on different pages.

Giving a functional requirement for the development process in the form of a list of supported UI libraries is not practical. Still, the proposed process should support using UI components from some of the most popular UI libraries, such as Ant Design¹² or MUI¹³.

3.3 Evaluation

When evaluating the artefact, the overall goal is to find out, whether the artefact provides a solution to the explicated problem, and to what extent is the solution valid (Johannesson and Perjons, 2014, p.141).

Evaluation of the artefact must be based on a predefined strategy. Both Johannesson and Perjons (2014) and Venable et al. (2012) discuss a two-dimensional evaluation strategy presented in Figure 1. A brief description of the different strategies is given below.

The selection between *Ex Ante* and *Ex Post* depends on the state of the artefact, when it is evaluated. *Ex Ante* evaluation strategy would allow evaluation even if the artefact is not used or developed, where as in *Ex Post* the artefact must be employed when evaluated (Johannesson and Perjons, 2014, p.143). As the goal of this research

¹²<https://ant.design/components/overview/>

¹³<https://mui.com/>

	Ex Ante	Ex Post
Artificial		X
Naturalistic		

Figure 1: Evaluation strategy selection for this research

is to find a functional solution to the given problem, *Ex Post* is the more relevant strategy.

Artificial and naturalistic evaluation strategy represent the second dimension. When executing the naturalistic strategy, the artefact must be evaluated in a real-world situation. In this thesis, naturalistic evaluation would require deploying the software artefact to customer environments and collecting feedback from the customers. Such evaluation is not relevant in this thesis, as the software artefact itself is not a production-ready feature. Thus, the artificial evaluation strategy is selected, as shown in Figure 1. More information and details regarding the strategies can be found in Johannesson and Perjons (2014).

The selected evaluation strategy is utilized to evaluate the artefact based on the defined requirements, and to what extend does the proposed artefact fulfill them. To allow the evaluation to be done consistently, the method for evaluating a single requirement must be predefined.

Both the interpreted problem and the defined requirements are technical, which allows the analysis and evaluation also to focus on technical feasibility, instead of a qualitative analysis. Additionally, the requirements were defined quite specifically with enough details, leaving less room for differentiating interpretations. As a result, the analysis and evaluation of a single feature and requirement can be done very objectively, reducing the input needed from stakeholders or other experts.

4 Results

This chapter presents the results of the research. The results are structured based on the design science research method framework, starting from Artefact design and development.

Section 4.1 designs and proposes a development process. The process should fulfill the requirements defined in the previous chapter. If the requirements are satisfied, the proposed process should allow developing a software artefact, that also complies with the defined requirements. This artefact is demonstrated in Section 4.2.

Section 4.3 evaluates the research results, especially regarding how well the defined requirements were fulfilled, and to what extent.

4.1 Development Process with WebAssembly

Interactivity

One of the main requirements was, that the software artefact should include interactive features. As a result, the key starting point for process design was that the resulting HTML must be rendered client-side, in the browser. Otherwise, the performance requirements could not have been satisfied. For example, rendering the components or full page on server-side after user actions would have caused unjustified delays considering a good user experience.

Rendering the application in the browser excluded many technologies from consideration. For example, the requirements could not be met with the more traditional .NET design patterns, such as ASP.NET MVC, which so far had been used to develop the existing application provided by the Company.

A few years ago, the only available design choice for client-side rendering would have been to write JavaScript code, either standalone or within some framework, such as React. However, as stated in the requirements, being limited to the use of a single programming language, especially JavaScript, was not allowed in this research. To satisfy this requirement, the initial design choice when starting the research was to utilize WebAssembly, as it would allow the execution of other than JavaScript code in the browser.

Compatibility

The resulting software artefact utilized WebAssembly binary code executed in the browser. The corresponding design choice regarding the development process was to determine, how the WebAssembly binary code is produced. As presented in Chapter 2, WebAssembly can be compiled from several different languages, such as C or Rust. The resulting wasm binary compiled from these languages could then be imported and utilized for example via the JavaScript code embedded into a HTML file.

However, one of the functional requirements was, that the development process should include tools and technologies that are compatible with the existing framework, in this case ASP.NET. Especially, the technologies should not cause any significant overhead to development process. Initial tests revealed, that the interoperability between JavaScript code and WebAssembly binary was not sufficient to satisfy the compatibility requirement, when wasm binaries were imported to the application via JavaScript code inside `<script>` tags. For example, importing and exporting functions and memory became very challenging even in a very simple application. As a result, the development overhead would have been unbearable in a more complex use case.

To fulfill the requirements regarding compatibility when integrating WebAssembly code to the application, the final design choice was to utilize .NET Blazor WebAssembly. The reasoning behind this choice, as well as the results, are presented below.

When developing client-side rendered applications with Blazor WebAssembly, the framework and runtime handled the WebAssembly compilation and execution independently in the background. The amount of required configuration work was very minimal, and there was no significant complexity added from the perspective of a developer. The key feature was, that all of the code could still be written with C#, without the need to consider underlying technical implementation done with WebAssembly. As a result, the development process did not differ much from more traditional ASP.NET server-side application development. Indeed, Blazor allowed executing same C# code both on the server and in the browser. These characteristics heavily diminished the complexity of utilizing WebAssembly in the web application.

Also, because the code and architecture was very similar compared to the code in the original application, existing developers could have start developing apps with Blazor WebAssembly with very limited training required. As a consequence, the resulting overhead remained limited, as was required.

Another reason for selecting Blazor WebAssembly was, that the code can be hosted from an ASP.NET Core server, according to Microsoft documentation¹⁴. The original application was built with ASP.NET MVC, which in newer versions is part of ASP.NET Core, meaning it should be able to host the Blazor WebAssembly pages and components without problems. Unfortunately, the ASP.NET MVC version of the original application was not yet sufficient to support ASP.NET Core hosted deployment. As a result, hosting the Blazor WebAssembly code directly from the existing application was not possible, so this possibility remained to be confirmed in future work.

The alternative option for hosting a software artefact was to utilize standalone deployment, in which the published static files could be served from an arbitrary static file server. With this approach, the pages built with Blazor WebAssembly could not be included in the main application directly, as the applications were served from separate servers. As a result, it was not possible to utilize ASP.NET built-in routing to direct the user from the main application to the software artefact. Instead, the software artefact would have to be accessed by an URL, directing the user to the static file server serving the Blazor WebAssembly application.

Visualizing large data sets

The software artefact was required to include functionality that would reduce server load, especially regarding data requests and filtering. The design philosophy was, that browser does not need to request data repeatedly, if there is a reasonable way to store existing data in the browser session. The proposed solution in this artefact was to utilize an in-browser database, where the data could be stored after receiving it from the server.

Interestingly, WebAssembly was an asset also in this regard. A popular SQL

¹⁴<https://learn.microsoft.com/en-us/aspnet/core/host-and-deploy/?view=aspnetcore-6.0>

database engine, SQLite¹⁵, is implemented in C. Because WebAssembly could be compiled from C, also the SQLite source code could be compiled to WebAssembly. This possibility was exploited in the development process, which allowed the utilization of SQLite directly in the browser. Initially, SQLite source code compiled to a native code format was added to the project as a separate file. However, the NuGet package manager used in .NET applications provided a ready-made SQLite package¹⁶, which did not require manual compilation before utilization, and was thus the final choice. Together with the EF Core SQLite package¹⁷, writing logic for managing the in-browser SQLite database was quite convenient.

There were some bugs and issues encountered, when developing an application with WebAssembly. In the utilized Blazor WebAssembly version 6.0.10¹⁸, there was a bug regarding the support of float and double number representations, which caused a runtime error. The remaining option for representing decimal numbers was to use the decimal type. Unfortunately, SQLite did not support sorting between values that are defined as decimal, limiting some capabilities of the artefact.

Utilizing an in-browser database allowed visualizing large amount of data to the user with great performance: The only significant delay occurred, when the particular set of data was initially requested from the server. The initial request consisted of waiting for server response, as well as inserting the data in to a database table. The standard insertion functionality provided by EF Core proved to lack performance in this use scenario, as it inserted rows to the database one by one. To improve the insertion performance, a custom implementation of bulk insertion was utilized, where multiple rows were inserted to the database at once. This approach drastically improved the data insertion performance.

After initial fetch, the data resided in the in-browser database, from where it was fetched for rendering as needed. As there was very little delay when fetching records from the database, pagination or virtualization could be utilized to prevent performance decreases with large data sets. Neither of these techniques would have functioned well if the data was requested every time from the server, as the delay

¹⁵<https://www.sqlite.org/index.html>

¹⁶https://www.nuget.org/packages/SQLitePCLRaw.bundle_e_sqlite3/2.1.2

¹⁷<https://www.nuget.org/packages/Microsoft.EntityFrameworkCore.Sqlite.Core/6.0.10>

¹⁸<https://www.nuget.org/packages/Microsoft.AspNetCore.Components.WebAssembly/6.0.10>

in server response in general is significantly higher compared to local in-browser database query.

Language independency

As stated in the requirements, the development process was required to allow writing client-side rendered applications with a variety of languages. With the features already designed, the artefact would have supported two languages, C# and JavaScript.

The design choice to utilize WebAssembly consequently allowed writing client-side rendered code with a variety of supported languages. More precisely, Blazor WebAssembly supported Native Dependencies¹⁹, allowing direct utilization of different portable native code formats, such as wasm-binaries and Object files.

To limit the scope, the process was designed to support at least C and C++. This design choice allowed two different approaches for importing native code into the application via Native File Reference:

- Compile the native code with a compatible external compiler, and add the compiled code as a Native File Reference. According to the documentation, the Native File Reference and .NET WebAssembly runtime should be built with the same version of Emscripten.
- Write C/C++ code directly to a source code file in the project, and add the *.c* or *.cpp* as a Native File Reference directly. The Emscripten-compiler used to build the .NET WebAssembly runtime, will also compile the Native File Reference.

After importing native code as a Native File Reference, other parts of the software artefact could utilize the functions implemented for example in C++. Unfortunately, the same bug regarding decimal number representation was causing errors also in this case, and passing float or double representation of numbers as a parameter to the C++ function was not possible.

¹⁹<https://learn.microsoft.com/en-us/aspnet/core/blazor/webassembly-native-dependencies?view=aspnetcore-6.0>

4.2 Created Software Artefact and Functionality

After the development process has been designed, it is beneficial to test it in realistic circumstances. The test case should reveal, if the development process can be utilized to solve one instance of the problem. Understandably, the results for one test case cannot be generalized, but suggest that the process possibly could solve other instances of the problem also.

The test scenario was designed together with stakeholders. The objective was to select a test case, which includes new functionality that could provide value to the customers. With this approach, the results of the demonstration would directly be beneficial for the Company, as well as encourage to utilize the designed process in other use cases also.

The designed test case provided the user a view, where the user could interactively examine contents of an invoice on multiple levels. The levels ranged from invoice headers to more atomic invoice tickets, which form the content of the invoice. This view gave the user visibility on which elements of source data are included on which invoice and invoice line. The view was considered to be an extension or additional feature to the existing application provided by the Company.

Understandably, a good test scenario also has to include features, that demonstrate how the artefact satisfies the given requirements. The selected test case first of all included interactivity, as the content and structure of the HTML file varied significantly based on user input. Also, the interactive features had to remain well performing, when the amount of visualized data increased, as an invoice line could consist of thousands of tickets. Additionally, because the demonstrated view presented private data, the view had to share the authentication state with the main application, to have access to the API where data is fetched.

For convenience, the software artefact created for the test scenario is referred to as the application. The core logic of this application is necessary to understand the value of the demonstration, and is presented in Figure 2.

The application initially visualized a list of invoices, based on the organization ID specified in the application URL. The visualization was done in a grid format, which the user could filter and sort. The user could then select an invoice from the list,

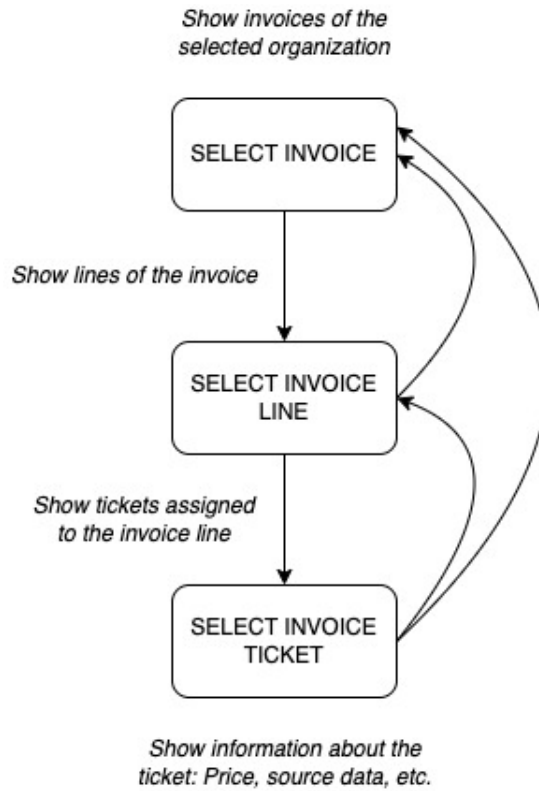



Figure 2: Core logic of the developed software artefact

after which the grid was changed from invoices to invoice lines, that existed on the selected invoice. Additionally, the selected invoice and most relevant information of it was shown to the user above the grid, in a card format, as presented in Figure 3. This card also included a revert-button, by which the user could return to select the invoice again. Similar functionality was implemented for invoice lines and invoice tickets.

As described, the resulting application clearly included interactive features as required, since both content and structure of the UI were dictated by user input. The artefact enabled such functionality, as all of the changes to the UI were rendered directly in the browser. Requests to the server were sent only on initial load, as well as when new data was needed. The client-side rendering was implemented purely based on C# and C code, thus no JavaScript was required.

Invoice Header ID: 186						
<div>Price</div> <div>Org</div> <div>Date</div>						
5011.8 €						
						

Invoice Lines (7)				Refresh Data	Clear Filters
-------------------	--	--	--	--------------	---------------

Select	Product	Description	Amount	Price Per Unit	Price	VAT %
<input type="button" value="Trace"/>	HS0062	Content Management	1.0		1250.0	19
<input type="button" value="Trace"/>	HS0063	Web Hosting	2.0		196.0	19
<input type="button" value="Trace"/>	IS0022	SFTP Service	5.0		30.0	19
<input type="button" value="Trace"/>	NS0030	Firewall Security	1.0		20.0	19
<input type="button" value="Trace"/>	NS0031	VPN Token	12.0		45.6	19
<input type="button" value="Trace"/>	PS0070	Printing Service Base Fee	1.0		120.0	19
<input type="button" value="Trace"/>	WORKSTATIONSUPPORT	Workstation Support	102.0		2550.0	19

Figure 3: Software artefact UI after invoice selection

To reduce server load, the application only requested data from the server in two situations:

- There are zero records of relevant data in the in-browser database. For example, no invoice lines of the invoice are found when querying the database
- User clicks the *Refresh Data* button. Relevant data based on the current grid, such as the lines of the selected invoice, are deleted from the database and reloaded from the server

As per designed artefact, the requested data was always stored in the in-browser database. After insertion, UI elements such as cards and grids directly queried the database for the data required to be visualized. As the grid data was fetched from the database, both filtering and sorting were also technically implemented as database queries, without the need to write any filters with C#. Especially the QuickGrid²⁰ component utilized as the UI grid had great compatibility with

²⁰<https://aspnet.github.io/quickgridsamples/>

database operations. The QuickGrid component took an IQueryable-object as a parameter, allowing shown data to be filtered based on SQL Queries evaluated for the IQueryable-object.

The performance of both filtering and sorting even large amounts of visualized data proved to be excellent, especially when built-in virtualization features of QuickGrid were utilized. Additionally, the loading and rendering times remained competitive with large amounts of data, as the data often was already available in the database, removing the delays of client-server communication.

The discussed bugs regarding decimal number representation with WebAssembly and SQLite can be observed from Figure 3, where the Amount and Price fields cannot be filtered or sorted. When sorting a decimal field, SQLite produced an error due to the incompatible data type. Rendering decimal numbers on the grid was fortunately still possible.

The card element above the grid in Figure 3, as well as all buttons in the UI, were implemented using Ant Design library. As mentioned in the non-functional requirements, the possibility to utilize at least some popular UI libraries was necessary. Ant Design had an implementation available for Blazor applications²¹, which allowed the elements familiar from AntD Javascript library to be utilized natively within .NET C# components.

As discussed in the previous section, the designed process supported writing client-side rendered code in C and C++. To demonstrate this capability, the application included functionality that visualizes the amount of rows shown on the grid. More importantly, this functionality was implemented with C code. The .c-file was added to the project as a Native File Reference, after which it was compiled by built-in Emscripten compiler, as in the designed process. All three grid components then imported functions defined in the .c-file, and utilized them to calculate the amount of rows. For example, the invoice line grid passed the invoice header ID as a parameter to a C function, which then returned the amount of invoice lines on the invoice.

To be more precise, the functions written in C did not explicitly return the amount of rows shown on the grid, but instead the count of relevant rows in the in-browser database. This was possible, as the SQLite source code is available in C, and could

²¹<https://antblazor.com/en-US/docs/introduce>

thus be utilized directly in the functions implemented also in C. Accessing the database created in the .NET C# code was quite straightforward, as it could be found simply with the database name.

Regarding user authentication in the software artefact, the requirement was, that the artefact would share the authentication state with the existing application. In other words, the software artefact should not require the user to login separately, as the user has already logged in to the existing application. Because the software artefact and existing application were served from separate hosts, the authentication state, more precisely API token, had to be transferred to the software artefact. The API token was passed to the software artefact as a query parameter, after which it was stored into the session storage of the browser. This allowed the authentication to persist after refreshing the page, but not after reopening the page.

4.3 Evaluation Based on Defined Requirements

The technical evaluation of the artefact was conducted as described in Section 3.3. The evaluation was strictly based on the defined requirements:

1. Does the artefact satisfy the requirement?
2. To what extent does the artefact satisfies the requirement?

The evaluation results for each requirement are presented below, one by one.

Artefact includes interactive features

Process Requirement: Components are rendered client-side

Interactive features were defined earlier in this thesis as changes to the structure and/or content of the rendered HTML, based on user actions. The software artefact clearly included such features. For example, the user could change the grid contents to represent a different data class, from invoice headers to invoice lines. This action also added a new card-element to the HTML structure.

These features were possible to implement, because the components were rendered on the client-side. The logic written into the components as WebAssembly binary

code was executed in the browser, allowing changes to the rendered HTML based on user input.

The requirement additionally stated, that the artefact should not request pre-rendered HTML pages from the server. This requirement was fulfilled by designing the software artefact to be a single page application (SPA). As a single page application, all pages and components of the software artefact were rendered in the browser, instead of fetching the pre-rendered HTML pages from the server. The development process supported creating a single-page-application, because Blazor WebAssembly is a SPA framework.

After a discussion and review with stakeholders and experts, one limitation regarding this requirement was found: Blazor WebAssembly does not allow accessing and altering the Document Object Model (DOM) contents directly. JavaScript provides such functionality for example with jQuery, where a specific element in the DOM can be appended or altered. Such action could be focusing to a specific HTML element after user clicks a button.

With Blazor WebAssembly, the limitation was caused by wasm-binaries not being able to access the DOM at all, as specified by MDN Web Docs²². The changes to DOM had to be done via the JavaScript API.

The inability to alter DOM directly did not limit the interactive features in the software artefact. However, according to the stakeholders, the existing application included quite many scenarios, where DOM elements were manipulated directly. A workaround would be required to implement similar features with Blazor WebAssembly.

Existing parts of the application remain functional

Process Requirement: Mix of server and client-side rendered components supported

As described in research results, the software artefact was hosted from a separate server, and was not directly integrated to the original application. Thus, the software artefact had no effect on the existing parts of the application. As a result, the requirement set for the software artefact was not relevant in this context.

²²<https://developer.mozilla.org/en-US/docs/WebAssembly/Concepts>

The development process was designed so, that both the software artefact and existing application could have been hosted from the same ASP.NET Core server. Because the existing application was not yet updated to .NET version 6, integrating the two applications to the same server was not possible in this research. However, based on Microsoft documentation, there were no considerable limitations for integrating the applications. Confirming this functionality will remain to be done in future work.

Client-side rendered code can be written in several other languages than JS

The software artefact was mainly developed with C#. Additionally, one feature of the software artefact was written in C. Both C and C# code were compiled to WebAssembly with the built-in Emscripten compiler in Visual Studio 2022. As the resulting WebAssembly binary code could then be executed and rendered in the browser, this requirement was fulfilled by the proposed development process and tools.

The ability to compile several programming languages to WebAssembly enabled some additional capabilities. As presented in the software artefact demonstration, data needed by the application was stored into an SQLite database running in the browser itself. This functionality was enabled by compiling the SQLite C-source code into WebAssembly.

The Blazor WebAssembly development process did not introduce any major limitations regarding what type of code could be written in C#. For example, writing HTML components and HTTP requests to the server did not introduce any additional complexity compared to JavaScript frameworks, such as React.

The minimal amount of introduced complexity was quite surprising, as initial tests of importing WebAssembly functions and memory manually from JavaScript code did not provide promising results. For example, interoperability between JavaScript code and imported WebAssembly binaries was limited, because WebAssembly as such only supported floats and integers as data types. Even though other data types, such as strings, could be represented as integers, it introduced vast development overhead and complexity.

Blazor WebAssembly did not include the limitations described above, however

one issue was encountered in the development process: The execution crashed at runtime, if float or double data types were utilized in the C# code compiled into WebAssembly. The remaining option, decimal representation, had limited support in the SQLite database, which forced some sorting capabilities to be removed from the software artefact. This limited the capability to write client-side rendered code with C# to some extent, but was not a major issue. The issue has been identified by .NET developers, and should be fixed in version 7.

Share authentication state with existing application

The authentication state was transferred from the existing application to the software artefact as a query parameter, after which the API requests could be authenticated with the transferred API token. The user was thus not required to log into the software artefact separately, meaning the requirement was fulfilled.

In production and public use, the transfer and storing mechanisms of the API token would require further consideration to ensure security. However, authentication in this extent was not in the scope of this research, and possible limitations and security risks will remain to be studied in future work.

Minimal data requests to the server

Process Requirement: Support technologies to store large amounts of requested data in the session

The visualization of large data sets in the context of this thesis consisted of three components listed below. Minimizing the data requests to the server was related to the first step, and the latter two components are related to the requirements covered later in this section.

1. Fetch data from source
2. Apply sorting and filtering based on user input
3. Render the data, i.e. add necessary structures and content to the rendered HTML

Reducing the amount of data requests concerned both the count of requests, as well as the amount of data requested. As mentioned in the requirements, the optimal solution would minimize both the count and size of data requests.

The software artefact optimized the count of data requests by utilizing an in-browser SQLite database. All received data was stored in the database, and components always fetched data directly from the in-browser database, instead of working with HTTP response contents directly. The artefact included logic, which ensured that the necessary data did not already reside in the database, before it was requested from the server. This limited the amount of unnecessary requests.

The size of the requests, or more precisely the size of the responses, was optimized by only requesting the exact data set needed. For example, to visualize invoice lines of a specific invoice, only the relevant set of invoice lines was requested.

As a result, the software artefact had optimal performance regarding minimal data requests, per definition given in the requirements. Good performance and the usage of an in-browser database was enabled by the tools used in the development process, meaning both requirements were fulfilled.

Support filtering and sorting data locally

The software artefact utilized QuickGrid custom component to visualize data. The component included great capabilities to filter and sort data, mainly due to direct connection to the IQueryable API, that allowed communicating with the database.

QuickGrid also implemented virtualization using the Blazor built-in component virtualization. Without virtualization, filtering and sorting was still quite fast, but the perceived performance from user perspective was significantly worse: The filtered or sorted results were rendered only after the operation was finished, until which point the previous grid and data was shown. Even if the time until updates are rendered would be relatively short considering the amount of data, it would still be perceived as bad performance by the user. The waiting period was concealed by enabling virtualization, as the browser almost immediately rendered the rows visible in the viewport, instead of rendering the full data set. Virtualization also added several visual effects to the grid, improving user experience when loading and scrolling the grid.

The artefact thus fulfilled this requirement, as data could be filtered and sorted locally. Only encountered limitation was caused by the discussed problem regarding float and double data types, which prevented sorting decimal number columns.

Sufficient performance from user perspective with large amounts of data

Process Requirement: Utilized technologies should have similar performance to JS in browser

The scalability tests were done with the Invoice Ticket grid, where nearly 9000 rows were rendered to the user at once. Even though in this test case the data was not real, an invoice line could in reality have such amount of tickets.

The test set included 12.55 megabytes of data. Fetching the data set from the API did not get noticeably slower, as the response time was typically less than 100ms.

After the data was fetched, it had to be inserted to the in-browser database. This was the most time-consuming operation with large amount of data: Inserting data into the database table took approximately 9 seconds, which could be considered to be a significant delay. However, in the context and use scenario of the application, the delay was still feasible. Especially, since the database insertion was done only during the initial load, after which the data already resides in the database for future needs.

The final step of visualizing data to the user was to render it. Rendering times for 9000 rows were less than a second, which is quite well considering the amount of data visualized. The grid functioned very well, as filtering and sorting was performed nearly in real-time, without any noticeable delays.

Grid virtualization had a tremendous effect on the perceived performance with large amount of data. With virtualization turned off, sorting the grid based on one column took multiple seconds, compared to instant response with virtualization turned on. Rendering times also increased without virtualization.

The requirements were fulfilled to the extent that was required for the context of this research and application. However, the delay occurring when inserting data to the in-browser database could become an issue in other applications, where such delays are not acceptable from user perspective.

Function properly in modern browsers

Process Requirement: Portable and well supported technologies used, no 3rd party plugins

The key design choice to fulfill this requirement was to utilize WebAssembly in the development process. Different alternatives were discussed earlier in Chapter 2, but none of them provided portability competitive with WebAssembly. As discussed in Chapter 2, the support for WebAssembly among browsers was nearly perfect. As a result, the software artefact created with Blazor WebAssembly did function properly in modern browsers. The application was tested on newest versions of Google Chrome, Mozilla Firefox and Microsoft Edge.

As portability also includes device-independency, mobile usage had to be considered. The software artefact itself was not designed to be responsive, and would not function well on a mobile device. This was not required considering the nature of the application. However, the development process itself did not set any limitations regarding mobile device hardware or browsers, meaning that the requirements were satisfied.

High compatibility with .NET development framework

The utilization of Blazor WebAssembly in the development process ensured great compatibility with .NET framework development tools. Most importantly, the software artefact could be developed with pure C# code, where the code and solution structure was similar to an ASP.NET MVC solution. The development work in both applications could also be done with the same tool, Visual Studio. As a result, the developers previously working with other ASP.NET applications could easily adopt the new development process. This result was confirmed by the input and feedback received from stakeholders.

The development process thus did not introduce any significant overhead or complexity to creating web applications. The defined requirement was fulfilled, and no limitations were discovered during the research.

Similar UI components with existing application

Process Requirement: Support most popular UI libraries

Support for most popular UI libraries required, that components of the UI libraries could be utilized in a Blazor WebAssembly application natively. More precisely, it should have been possible to add these components to the application similar to any standard HTML elements, using C# code.

The software artefact utilized Ant Design as the UI library for buttons and cards. There was a specific Blazor WebAssembly implementation of Ant Design available, and it contained a sufficient amount of elements at least for the needs of the software artefact.

The existing application used Telerik Kendo UI²³ as the UI library. Kendo UI was also available to Blazor applications, but was not utilized in this research due to pricing. Nevertheless, according to the documentation, utilizing Kendo UI elements in the software artefact would have been possible.

Many other popular UI libraries also had implementations for Blazor WebAssembly, as shown in the following list. Semantic UI implementation was still at a beta-stage, but all other mentioned libraries already had production versions.

- Bootstrap - BlazorStrap²⁴
- Material Design - MatBlazor²⁵
- Semantic UI - SemanticBlazor²⁶

Ant Design, as well as the other mentioned UI libraries based on documentation, were quite effortless to take into use. All of them followed the same pattern: Download the package in NuGet package manager, add the CSS and JS links to *index.html*, and register the service in *Program.cs*. After these steps, the elements were usable in the razor components of the application. Based on the definition given to native usage of UI libraries, the requirement was fulfilled.

²³<https://www.telerik.com/aspnet-mvc>

²⁴<https://blazorstrap.io/V5/>

²⁵<https://www.matblazor.com/>

²⁶<http://semblazor.azurewebsites.net/Components/>

Additionally, the requirement defined that popular UI libraries should be supported. As mentioned already in the requirement definition, it would be very challenging to gather an exhaustive list of UI libraries supported in Blazor WebAssembly. For this reason, the requirement was not defined as functional. Based on the broad support for major UI libraries discussed above, the requirement was satisfied.

5 Discussion

Three research questions were defined to be answered by the research conducted in this thesis. Sections 5.1, 5.2 and 5.3 discuss the research questions, by analyzing the results presented earlier in Chapters 2-4.

Section 5.4 addresses the identified threats to the validity of the research results. Finally, Section 5.5 discusses possible topics and questions for Future Work.

5.1 RQ1: What are the focus areas of existing research and literature written about WebAssembly?

Most of the existing research focused on the performance and safety aspects of WebAssembly, as well as on how WebAssembly could be utilized in different scenarios. However, there was a very limited amount of literature about the effects of WebAssembly on the development process. The gap in existing research acts as a motivation for the research conducted in this thesis, which examines the impact and capabilities of WebAssembly in the Web software development process.

The majority of literature regarding the security of WebAssembly was related to code compiled from C, C++ and Rust. The compilation was often handled by Emscripten, an LLVM-based (low-level virtual machine) compiler, mainly due to its popularity. However, the number of compilers that can compile WebAssembly is increasing, and their security would be beneficial to be reviewed more as well.

Many of written articles focused on the performance of WebAssembly. The results had significant variance depending on compilers, browsers and input sizes used. Nevertheless, existing research showed that there is a performance advantage in using WebAssembly instead of JavaScript. The exact performance benefit of course depends on the use scenario in question.

In general, the most suitable use scenarios for WebAssembly could not be interpreted from existing research, as the research topics and areas were quite scattered. To facilitate the adoption of WebAssembly in software companies, characteristics of use scenarios in which WebAssembly excels, should be better understood.

Additionally, especially in research related to security and use cases of WebAssembly,

most of the reviewed papers proposed some type of extension or modification of WebAssembly, such as WAIT or MS-Wasm. The extensibility of WebAssembly is clearly an upside, but the various extensions could also be seen as a deficit of the original implementation.

5.2 RQ2: Does WebAssembly increase language independency of .NET web application development?

Language selection for developing interactive, client-side rendered .NET web applications was previously strictly limited to JavaScript. The results of this research indicate, that the introduction of Blazor WebAssembly enabled writing such applications with pure C#.

The research identified some bugs and limitations regarding how the C# code functioned in the browser, such as the float/double representation issue, but these problems were quite minor. No significant limitations or challenges regarding data types or functions were found, that would have been caused by WebAssembly. The expectation was, that working for example with strings would have been more challenging, due to missing data type in WebAssembly specification. This expectation was proven to be wrong.

To truly increase language independency, it is not sufficient to introduce C# as the only language to replace JavaScript. The utilization of WebAssembly hypothetically allowed developing features into the application with any language, that could be compiled to a WebAssembly binary with Emscripten. The research results confirmed this hypothesis to some extent, as functions written in C and C++ were utilized successfully in the web application.

Importantly, Blazor WebAssembly also allowed importing JavaScript functions and libraries. Key part of the designed development process was the possibility to utilize JavaScript UI libraries, which increased language independency, as well as the attractiveness of the framework.

A limiting factor regarding language independency was, that WebAssembly applications still require JavaScript code in the background, as WebAssembly cannot alter the rendered Document Object Model directly. However, as results suggest,

from developer's point of view, it was possible to develop an interactive web application with .NET and pure C#, as the JavaScript interoperability was handled by the framework.

As presented in Section 2.7, the JavaScript limitation of web development was not only a technical limitation regarding performance, but also introduced constraints for hiring and training developers. The results of this thesis imply, that WebAssembly does remove these constraints at least to some extent.

Existing literature covered in the beginning of this thesis mainly focused on performance, security and use cases of WebAssembly. Consequently, existing literature did not have many research results regarding the impact of WebAssembly on software engineering. This makes the results of this research more significant, especially regarding the language independency aspect of WebAssembly.

5.3 RQ3: What features and capabilities does WebAssembly enable in the case study?

As the research results showed, utilizing WebAssembly had two key impacts on the artefact and its development, that are presented in the list below. The significance and effect of language selection was already discussed with previous research questions, so the focus is on the allowed features and design choices.

1. Client-side rendering allowed features and design choices otherwise not possible
2. Client-side logic could be written using multiple programming languages

The key feature of the artefact allowed by WebAssembly and client-side rendering was the possibility to create an in-browser database. It allowed reducing requests made to the server, while improving user experience simultaneously, as data was quickly available for visualization. Additionally, the backend used by the existing application of the Company is very dependent on an SQL database. With the in-browser database, the same data model could be copied and used both in frontend and backend, improving co-operation and reducing development costs.

Developing the case study application with WebAssembly enabled interactive features, with less development overhead compared to a server-side rendered .NET

application. The interactive features importantly had good performance from user perspective, as updates to the rendered DOM were quite instant.

5.4 Threats to Validity

This research designed and proposed a development process, which was then utilized in single proof-of-concept application, as suggested in the design science research method framework. Even though the development process proved to be suitable in the case study, the results cannot be generalized, especially for other companies and industries. This threat to validity was known beforehand and was described in Chapter 3, as it is a characteristic of the chosen research method.

Additionally, the results of this research suggest, that WebAssembly does increase language independency of .NET web development. The capabilities of C# in the development process are clear based on the research results, but the requirement was to allow developing features with more than one language. The results showed, that it is possible to develop client-side executed functions with C and C++, which fulfills the requirement from a technical standpoint. However, the results did not show, to what extent to which C and C++ can be utilized in .NET web application development. For example, the capability to develop an entire HTML component with these languages was not examined.

It must also be noted, that the method framework used in this research was iterative. The key consequence was, that the requirements for the artefact were not strictly set before starting the design phase. As a result, artefact design and requirement definition were done somewhat simultaneously, meaning that the observations from design and development phase also affected the final requirement list.

5.5 Future Work

The results of this research, including the encountered limitations, act as a motivation for future work. One specific theme to be researched more is serving a Blazor WebAssembly application directly from an existing ASP.NET solution, such as the existing MVC application of the Company. Understanding the possibilities and constraints of interoperability between the server-side and client-side code would

be valuable.

The software engineering impact of WebAssembly would also be one area worth examining more in the future. The real possibilities of language independency and utilizing same code both on the server and client remain to be researched.

6 Conclusion

The goal of this research was to design a development process, which would allow creating client-side rendered web applications with .NET. One of the key requirements for the process was, that it should not be limited to a single programming language, which traditionally would have been JavaScript. The designed process was verified by developing a proof-of-concept application for a Finnish SaaS company, which emphasized features enabled by client-side rendering, such as interactivity.

Software engineering aspect of WebAssembly, such as the one described above, was not a focus area in existing research examined in Chapter 2. The existing research and literature mainly discussed performance and security implications of WebAssembly, which were thus mainly left out of scope in this thesis. Based on results from existing research, WebAssembly had a performance advantage compared to JavaScript, but also introduced some security risks.

The main technology chosen for the development process was Blazor WebAssembly, a WebAssembly based .NET framework. Blazor WebAssembly allowed developing client-side applications with pure C#, which was then compiled to WebAssembly by the framework. Other programming languages, such as C and C++, were also utilized in the development process, as the Emscripten-compiler used by the framework was capable of compiling all three languages to WebAssembly binary code. The results clearly indicated, that Blazor WebAssembly increased language independency of .NET client-side web development.

The capability to compile source code from several languages, such as C, enabled the utilization of an in-browser database. SQLite source code was written in C, and thus could be utilized in the Blazor WebAssembly application, which compiled the source code to WebAssembly. An in-browser database allowed decreasing server requests, as well as managing and visualizing large amounts of data.

As a result, WebAssembly enabled developing an application, that mainly fulfilled the given requirements. Meanwhile, also the development process was verified to be functional and fulfilled the set of requirements, allowing it to be further utilized in the Company.

References

- Ahopelto, T. Nollaksi vai ykköseksi. *Elinkeinoelämän valtuuskunta EVA, Helsinki, EVA analyysi no 62*, 2018. URL https://www.eva.fi/wp-content/uploads/2018/04/eva_analyysi_no_62.pdf.
- ASP.NET Documentation. Choose an asp.net core web ui, 2022. URL <https://docs.microsoft.com/en-us/aspnet/core/tutorials/choose-web-ui?view=aspnetcore-6.0>. Accessed: 2022-10-04.
- Bhansali, S., Aris, A., Acar, A., Oz, H. and Uluagac, A. S. A first look at code obfuscation for webassembly. *Proceedings of the 15th ACM Conference on Security and Privacy in Wireless and Mobile Networks*, page 140–145. Association for Computing Machinery, 2022. doi: 10.1145/3507657.3528560.
- Breaux, T. and Moritz, J. The 2021 software developer shortage is coming. *Commun. ACM*, 64(7):39–41, 2021. ISSN 0001-0782. doi: 10.1145/3440753.
- CanIUse. EcmaScript 2015 (es6) support, 2022a. URL <https://caniuse.com/?search=ecmascript%202015>. Accessed: 2022-08-04.
- CanIUse. Wasm support, 2022b. URL <https://caniuse.com/wasm>. Accessed: 2022-08-04.
- Consortium, World Wide Web. World wide web consortium (w3c) brings a new language to the web as webassembly becomes a w3c recommendation, 2019. URL <https://www.w3.org/2019/12/pressrelease-wasm-rec.html.en>. Accessed: 2022-08-11.
- De Macedo, J., Abreu, R., Pereira, R. and Saraiva, J. On the runtime and energy performance of webassembly: Is webassembly superior to javascript yet? *2021 36th IEEE/ACM International Conference on Automated Software Engineering Workshops (ASEW)*, pages 255–262. IEEE, 2021. ISBN 1-66541-185-6. doi: 10.1109/ASEW52652.2021.00056.
- Disselkoen, C., Renner, J., Watt, C., Garfinkel, T., Levy, A. and Stefan, D. Position paper: Progressive memory safety for webassembly. *Proceedings of the 8th International Workshop on Hardware and Architectural Support for Security*

- and Privacy*, pages 1–8. Association for Computing Machinery, 2019. doi: 10.1145/3337167.3337171.
- Es, N. V., Nicolay, J., Stievenart, Q., D’Hondt, T. and Roover, C. D. A performant scheme interpreter in asm.js. *Proceedings of the 31st Annual ACM Symposium on Applied Computing*, page 1944–1951. Association for Computing Machinery, 2016. doi: 10.1145/2851613.2851748.
- Haas, A., Rossberg, A., Schuff, D. L., Titzer, B. L., Holman, M., Gohman, D., Wagner, L., Zakai, A. and Bastien, J. Bringing the web up to speed with webassembly. *SIGPLAN Not.*, 52(6):185–200, 2017. ISSN 0362-1340. doi: 10.1145/3140587.3062363.
- Hasselt, M. v., Huijzendveld, K., Noort, N., Ruijter, S. d., Islam, T. and Malavolta, I. Comparing the energy efficiency of webassembly and javascript in web applications on android mobile devices. *The International Conference on Evaluation and Assessment in Software Engineering 2022*, page 140–149. Association for Computing Machinery, 2022. doi: 10.1145/3530019.3530034.
- Hilbig, A., Lehmann, D. and Pradel, M. An empirical study of real-world webassembly binaries: Security, languages, use cases. *Proceedings of the Web Conference 2021*, page 2696–2708. Association for Computing Machinery, 2021. doi: 10.1145/3442381.3450138.
- Jangda, A., Powers, B., Berger, Emery D. and Guha, A. Not so fast: Analyzing the performance of webassembly vs. native code. *Proceedings of the 2019 USENIX Annual Technical Conference, USENIX ATC 2019*, pages 107–120. USENIX Association, 2019. ISBN 9781939133038.
- Johannesson, P. and Perjons, E. *An introduction to design science*, volume 10. Springer, 2014.
- Kocher, P., Horn, J., Fogh, A., Genkin, D., Gruss, D., Haas, W., Hamburg, M., Lipp, M., Mangard, S., Prescher, T., Schwarz, M. and Yarom, Y. Spectre attacks: Exploiting speculative execution. *2019 IEEE Symposium on Security and Privacy (SP)*, pages 1–19. IEEE, 2019. ISBN 1-5386-6661-8. doi: 10.1109/SP.2019.00002.

- Krasner, G. E. and Pope, S. T. A description of the model-view-controller user interface paradigm in the smalltalk-80 system. *Journal of object oriented programming*, 1(3):1–34, 1988.
- Lehmann, D., Kinder, J. and Pradel, Michael. Everything old is new again: Binary security of webassembly. *Proceedings of the 29th USENIX Security Symposium*, pages 217–234. USENIX Association, 2020. ISBN 9781939133175.
- Li, B., Fan, H., Gao, Y. and Dong, W. Bringing webassembly to resource-constrained iot devices for seamless device-cloud integration. *Proceedings of the 20th Annual International Conference on Mobile Systems, Applications and Services*, page 261–272. Association for Computing Machinery, 2022. doi: 10.1145/3498361.3538922.
- MDN Web Docs. Compiling from rust to webassembly, n.d. URL https://developer.mozilla.org/en-US/docs/WebAssembly/Rust_to_wasm. Accessed: 2022-09-07.
- Musch, M., Wressnegger, C., Johns, M. and Rieck, K. New kid on the web: A study on the prevalence of webassembly in the wild. *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment, Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 23–42. Springer International Publishing, 2019. ISBN 978-3-030-22038-9.
- Mäkitalo, N., Mikkonen, T., Pautasso, C., Bankowski, V., Daubaris, P., Mikkola, R. and Beletski, O. Webassembly modules as lightweight containers for liquid iot applications. *ICWE 2021*, volume 12706, pages 328–336. Springer, 2021. ISBN 978-3-030-74296-6. doi: 10.1007/978-3-030-74296-6_25.
- Møller, A. Technical perspective: Webassembly: a quiet revolution of the web. *Commun. ACM*, 61(12):106, 2018. ISSN 0001-0782. doi: 10.1145/3282508.
- Narayan, S., Disselkoen, C., Moghimi, D., Cauligi, S., Johnson, E., Gang, Z., Vahldiek-Oberwagner, A., Sahita, Ravi, Shacham, H., Tullsen, D. and Stefan, D. Swivel: Hardening webassembly against spectre. *Proceedings of the 30th USENIX Security Symposium*, pages 1433–1450. USENIX Association, 2021. ISBN 9781939133243.

- Protzenko, J., Beurdouche, B., Merigoux, D. and Bhargavan, K. Formally verified cryptographic web applications in webassembly. *2019 IEEE Symposium on Security and Privacy (SP)*, pages 1256–1274. IEEE, 2019. ISBN 1-5386-6661-8. doi: 10.1109/SP.2019.00064.
- Reiser, M. and Bläser, L. Accelerate javascript applications by cross-compiling to webassembly. *Proceedings of the 9th ACM SIGPLAN International Workshop on Virtual Machines and Intermediate Languages*, page 10–17. Association for Computing Machinery, 2017. doi: 10.1145/3141871.3141873.
- Romano, A. and Wang, W. Wasmview: visual testing for webassembly applications. *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Companion Proceedings*, page 13–16. Association for Computing Machinery, 2020. ISBN 9781450371223. doi: 10.1145/3377812.3382155.
- Sandhu, P., Herrera, D. and Hendren, L. Sparse matrices on the web: Characterizing the performance and optimal format selection of sparse matrix-vector multiplication in javascript and webassembly. *Proceedings of the 15th International Conference on Managed Languages & Runtimes*, pages 1–13. Association for Computing Machinery, 2018. doi: 10.1145/3237009.3237020.
- Stack Overflow. 2021 developer survey, 2021. URL <https://insights.stackoverflow.com/survey/2021#most-popular-technologies-webframe>. Accessed: 2022-09-22.
- Venable, J., Pries-Heje, J. and Baskerville, R. A comprehensive framework for evaluation in design science research. *International conference on design science research in information systems*, Design Science Research in Information Systems. Advances in Theory and Practice, pages 423–438. Springer Berlin Heidelberg, 2012. ISBN 978-3-642-29863-9.
- W3C. Webassembly core specification, 2019. URL <https://www.w3.org/TR/2019/REC-wasm-core-1-20191205/>. Accessed: 2022-08-26.
- Wasson, Mike. Asp.net - single-page applications: Build modern, responsive web apps with asp.net, 2013. URL <https://docs.microsoft.com/en-us/archive/msdn-magazine/2013/november/asp-net-single-page-applications->

build-modern-responsive-web-apps-with-asp-net. Accessed: 2022-08-26.

Yan, Y., Tu, T., Zhao, L., Zhou, Y. and Wang, W. Understanding the performance of webassembly applications. *Proceedings of the 21st ACM Internet Measurement Conference*, page 533–549. Association for Computing Machinery, 2019. doi: 10.1145/3487552.3487827.

Yee, B., Sehr, D., Dardyk, G., Chen, J. B., Muth, R., Ormandy, T., Okasaka, S., Narula, N. and Fullagar, N. Native client: A sandbox for portable, untrusted x86 native code. *Proceedings of the 2009 30th IEEE Symposium on Security and Privacy* :, 2009 30th IEEE Symposium on Security and Privacy (SP), pages 79–93. IEEE, 2009. ISBN 0-7695-3633-6. doi: 10.1109/SP.2009.25.

Zinzindohoué, J.-K., Bhargavan, K., Protzenko, J. and Beurdouche, B. Hacl*: A verified modern cryptographic library. *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, page 1789–1806. Association for Computing Machinery, 2017. doi: 10.1145/3133956.3134043.