Endress+Hauser

OST
Ostschweizer
Fachhochschule

Migration einer bestehenden C#-Codebasis in eine Webanwendung: Ein Technologievergleich

Studienarbeit

Studiengang Informatik
OST - Ostschweizer Fachhochschule
Campus Rapperswil-Jona

Herbstsemester 2021

Autoren:	Reto Ehrensperger Christian Rutzer
Betreuer:	Prof. Dr.-Ing. Frieder Loch
Projektpartner:	Endress + Hauser AG

Version: 23. Dezember 2021

Abstract

Heutzutage setzen Firmen zunehmend auf Webtechnologien für die Erstellung ihrer Benutzerschnittstellen. Oftmals haben aber grössere Unternehmen ihre bestehende Codebasis mit Technologien umgesetzt, die nicht unmittelbar mit solchen Webtechnologien kompatibel sind. In der vorliegenden Arbeit wird untersucht, wie eine bestehende .NET-Codebasis in eine HTML5 Umgebung integriert werden kann und wie sich dies auf die Performance auswirkt. Um diese Forschungsfrage zu beantworten, wurde eine bestehende Applikation von Endress + Hauser AG als Single-Page-Applikation in HTML5 nachprogrammiert. Insbesondere wurde das Frontend einmal mit dem React-Framework und einmal mit dem Blazor-Framework entwickelt. Um möglichst unabhängig von Endress + Hauser AG arbeiten zu können, wurde als bestehende Codebasis ein selbst geschriebenes .NET-Projekt aufgesetzt. Zudem wurde die Performance von unterschiedlichen Technologien für die Erstellung von HTML5-Applikationen verglichen. Die Arbeit kommt zum Schluss, dass eine bestehende .NET-Codebasis mit Hilfe des Blazor-Frameworks in die HTML5 Umgebung integriert werden kann. Dies ermöglicht grundsätzlich auch die Integration in eine Applikation die unter Verwendung des React-Frameworks erstellt wird. Allerdings wird momentan von der produktiven Verwendung der vorgestellten Varianten abgeraten, da alle Varianten auf aktuell inoffiziellen Konzepten basieren.

Lay-Summary

Moderne Webtechnologien können für Firmen eine Chance sein, um plattformübergreifende Systeme zu entwickeln. Viele Firmen haben ihre Applikationen mit Technologien entwickelt, die jedoch nicht direkt mit modernen Webtechnologien verbunden werden können. Oftmals ist diese Codebasis sehr kompliziert und unübersichtlich aufgebaut. Eine Reimplementiertung der Applikation in eine neue Technologie ist deshalb in der Praxis sehr aufwendig oder nicht möglich.

In dieser Arbeit wurde untersucht, wie eine bereits existierende .NET-Codebasis in die Webumgebungen integriert und deren Code wiederverwendet werden kann.

Als Codebasis, die in die Webumgebung integriert werden soll, wurde ein selbst erstelltes .NET-Projekt verwendet. Mit systematischem Vorgehen wurde untersucht, wie dieses in ein React-Projekt integriert werden kann. Schlussendlich wurde die Performance der unterschiedlichen Technologien verglichen.

Unter Verwendung des Blazor-Framework ist es möglich, eine bestehende .NET-Codebasis mit gewissen Anpassungen an der Codebasis in ein React-Projekt zu integrieren. Allerdings wird von diesen Varianten im produktiven Einsatz abgeraten, da es sich dabei um aktuell inoffizielle Konzepte handelt.

Inhaltsverzeichnis

1 Einleitung

Grosse Teile des Programmcodes von Firmen sind in Technologien geschrieben, die nicht unmittelbar mit modernen Webtechnologien kompatibel sind. Somit ergeben sich Herausforderungen, wenn neue Anforderungen adressiert, aber bestehende Codebasen weiterverwendet werden sollen. Ein Beispiel stellt die Verwendung von C#-Code in React-Anwendungen dar. Es stellt sich deshalb die Herausforderung, wie sich diese Technologien in moderne Frameworks übertragen lassen.

1.1 Problemstellung

In der vorliegenden Arbeit wird die Konfigurationssoftware eins Durchflussmessgeräts von Endress + Hauser AG als Beispiel betrachtet. Diese Geräte können die Konzentration verschiedener Stoffen in Flüssigkeiten bestimmen. Für eine genaue Messung sind Parameter notwendig, die mit einer bestehenden Windows-Anwendung bestimmt werden können.

Anerkannte Industriestandards entwickeln sich laufend weiter und das Web wird mit seinem hohen Grad an Plattformunabhängigkeit immer beliebter. Für viele Firmen stellt sich die Frage, ob bestehender C#-Code in einer Webanwenung weiterverwendet werden kann.

Für Endress + Hauser AG stellt sich die Frage, ob der C#-Code, welcher die genauen Parameter für die Bestimmung der Konzentration einer Flüssigkeit berechnet, in einer Webanwendung wiederverwendet werden kann.

1.2 Ziel

Ziel der vorliegenden Arbeit ist die Erarbeitung und Erprobung von Konzepten, um eine bestehende .NET-Codebasis in eine HTML5-Umgebung zu übertragen.

Die Ergebnisse der Arbeit (schriftlicher Projektbericht, Code, Dokumentation) werden am Ende den Projektbeteiligten zur Verfügung gestellt.

2 Technologische Grundlagen

2.1 Eine kurze Geschichte des Webs

Der folgende Absatz wurde sinngemäss von (Zalewski, 2011) übernommen.

Das Web ist relativ jung und die heutige Version sehr ähnlich zur ursprünglich entwickelten Form. Die Vorgeschichte dazu ist dennoch einen genaueren Blick wert.

1945 bis 1994
Computerhistoriker beschreiben häufig ein hypothetisches Gerät, welches etwa die Grösse eines Schreibtisches hatte und von Vannevar Bush 1945 vorgestellt wurde. Mit diesem Gerät, was den Namen Memex trug, sollte es möglich sein, dokumentübergreifende Links auf Mikrofilm zu erstellen, zu kommentieren und zu verfolgen. Diese Technik erinnert an die heutigen Lesezeichen und Hyperlinks, welche von einem Browser verarbeitet werden. Memex konnte nie umgesetzt werden und blieb deshalb immer eine Vision, bis Transistor-basierte Computer immer populärer wurden.

Der nächste Meilenstein war die "Generalized Markup Language" (GML), welche von IBM entwickelt wurde. Mit dieser war es möglich, Dokumente mit maschinenlesbaren Anweisungen zu versehen. Somit gilt GML oder die Nachfolgeversion "Standard Generalized Markup Language" (SGML) als Vorgänger von HTML.

1989 stellten Tim Berners-Lee und Dan Connolly, welche am CERN in Genf forschten, die erste Version von HTML vor. Später veröffentlichten sie zusätzlich das HyperText Transfer Protocol (HTTP), welches in verbesserten Versionen heute noch ein weit verbreitetes Protokoll im Internet ist. HTTP war eine einfache Lösung um HTML Ressourcen mit der zu dieser Zeit bereits existierenden Technologien wie Internet Protokoll (IP) und Domain Namen zu übertragen. Diese HTML Dateien wurden im ebenfalls von Tim Berners-Lee entwickelten Browser mit dem Namen "World Wide Web" dargestellt.

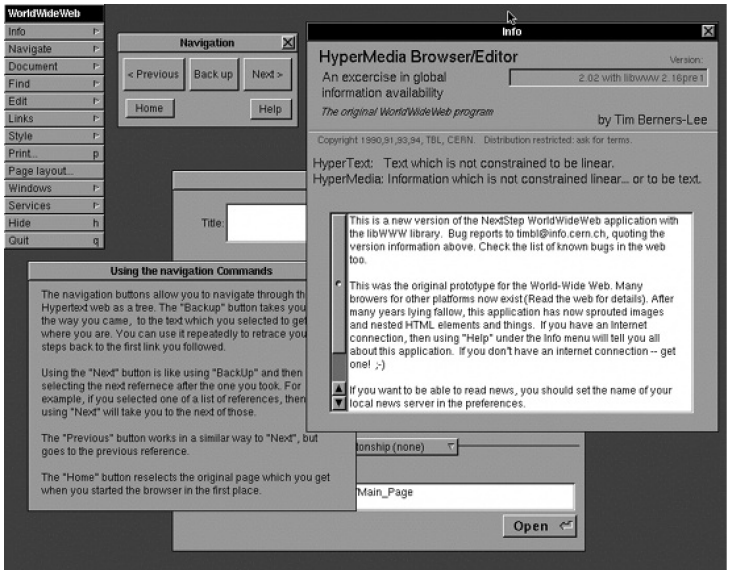

Abbildung 1: Tim Bendners-Lee's World Wide Web

1995 bis 1999
Als das Web populärer wurde, steckten viele Firmen enorme Ressourcen in die Entwicklung von verschiedenen Browsern. Das daraus folgende Wettrüsten mit neuen Funktionen führte dazu, dass kaum eine der Entwicklungen standardisiert wurde und jeder Hersteller seine eigenen Funktionen und Standards verwendet hat.

Um dieser Entwicklung entgegenzuwirken gründete Tim Berners-Lee in Zusammenarbeit mit verschiedenen Firmen das "World Wide Web Consortium" (W3C). Der Einfluss dieser Organisation war am Anfang aber klein, da die definierten Standards bei ihrer Veröffentlichung bereits wieder veraltet waren.

2000 bis 2009
Mit den gescheiterten Versuchen das Web zu vereinheitlichen, wuchs, dank der zunehmenden Popularität von Windows als Betriebssystem, der Marktanteil des von Microsoft entwickelten "Internet Explorers". Zu Spitzenzeiten hielt der "Internet Explorer" einen Marktanteil von über 80%. Dies hatte zur Folge, dass die meisten Webseiten für die Verwendung mit dem Browser von Microsoft optimiert wurden und die Entwicklungsgeschwindigkeit im Vergleich zu den Anfangszeiten des Webs stark abgenommen hatte.

Eine der grössten Neuerungen im Web war die Veröffentlichung der "XMLHttpRequest" API, welche ursprünglich für die Lösung eines Problems in der Webversion von Microsoft Outlook gedacht war. Mit dieser Funktion ist es möglich, per JavaScript asynchrone Anfragen an einen Webserver zu machen. Diese Neuerung war eine der Hauptfunktionen, welche später zum sogenannten Web 2.0 führte.

2010 - 2021
Mit der Möglichkeit asynchron Daten in den Browser zu laden, war der Startschuss für die heutzutage weit verbreiteten Single-Page-Applikationen (SPA) gefallen. SPAs bestehen aus einem einzigen HTML-Dokument und alle Inhalte werden dynamisch per JavaScript nachgeladen. Google veröffentlichte im Jahr 2010 das erste Framework, mit welchem SPAs in JavaScript entwickelt werden konnten. In den nachfolgenden Jahren wurden SPAs immer beliebter und es wurden verschiedene dieser Frameworks veröffentlicht. Zu den bekanntesten gehören React, welches hauptsächlich von Facbeook entwickelt wird und Angular (Weiterentwicklung von AngularJS), welches von Google als Hauptentwickler unterhalten wird.

Nach 17 Jahren mit HTML 4 wurde 2016 die Version 5 veröffentlicht. Dabei wurden verschiedenste neue Tags eingeführt, mit welchen multimediale Inhalte wie Audio, Video oder sonstige Objekte in eine Webseite integriert werden können (W3Schools, o. J.).

2.2 Überblick über WebAssembly

WebAssembly (Wasm) ist ein offener Standard, der es möglich macht, ein low-level Code Format effizient und plattformunabhängig auszuführen. Das Hauptziel besteht darin, die maximale Ausführungsgeschwindigkeit zu erreichen (Latham & olprod, 2021). Wasm enthält keine webspezifischen Funktionen und kann deshalb auch ausserhalb des Browsers ausgeführt werden.

Der WebAssembly Standard wird von der "W3C Community Group", bei welcher auch alle grossen Browserhersteller dabei sind, mit den folgenden Hauptzielen entwickelt:

- **Schnelligkeit:** Annähernd native Code Performance
- **Sicherheit:** Validierung und Ausführung von Codes in einer speichersicheren Umgebung
- **Hardware-Unabhängigkeit:** Kann auf allen modernen Architekturen kompiliert werden
- **Sprach-Unabhängigkeit:** Privilegiert keine bestimmte Sprache
- **Plattform-Unabhängigkeit:** Kann in Browser integriert werden, läuft aber auch in anderen Umgebungen

2.2.1 Aufbau

WebAssembly Programme sind in Module aufgeteilt, welche einzeln bereitgestellt werden können. In einem Modul werden Typen, Funktionen, und verschiedene Deklarations-Elemente definiert. Ebenfalls können Funktionen für andere Module exportiert oder importiert werden.

Bytecode
Module werden jeweils anhand einer abstrakten Syntax in eine Bytefolge codiert. Zur Laufzeit wird der Bytecode wieder decodiert, validiert und ausgeführt.

Für die Ausführung sind keine zusätzlichen Browser-Plugins notwendig und alle modernen Browser unterstützen WebAssembly. Der Bytecode wird in der gleichen virtuellen Maschine ausgeführt, in welcher auch JavaScript interpretiert wird. Die Kommunikation mit dem Browser und dem Betriebssystem funktioniert über die gleichen Schnittstellen, welche auch für JavaScipt zur Verfügung stehen (MDN-Contributors, 2021a).

```
1    (module
2        (func (param $lhs i32) (param $rhs i32) (result i32)
3            local.get $lhs
4            local.get $rhs
5            i32.add))
```

Listing 1: WebAssembly Bytecode Beispiel - Funktion zum Addieren von zwei Zahlen

2.2.2 Kompatibilität

Da WebAssembly bereits vor der Ausführung kompiliert wird, gibt es zwei Möglichkeiten dies auszuführen. Einerseits kann dies durch direktes Kompilieren zu WebAssembly oder andererseits durch Implementierung der jeweiligen virtuellen Maschine erreicht werden.

Aktuell ist es nur mit den Programmiersprachen C und C++ möglich, direkt zu WebAssembly zu kompilieren. In zukünftigen Versionen soll Multithreading und Garbage Collection unterstützt werden, womit WebAssembly ein Kompilationsziel für Technologien wie .NET oder Java werden kann (Smith, 2018).

Das Blazor Framework, welches hauptsächlich von Microsoft entwickelt wird, macht es möglich die von .NET verwendete Common Intermediate Language (CIL) Code in einer in WebAssembly implementierten virtuellen Maschine auszuführen. Neben C# werden noch etwa 40 weitere Sprachen untersützt, welche mit dem gleichen Verfahren im Browser ausgeführt werden können.

3 Recherche verschiedener Varianten

Um das geforderte Ziel, eine C#-Codebasis in einer HTML 5 Umgebung ausführen zu können, wurden zu Beginn verschiedene Ansätze recherchiert. Da WebAssembly eine neue Technologie ist und C# meist nur auf der Serverseite eingesetzt wird, mussten auch unkonventionelle und nicht offiziell vorgesehene Varianten in Betracht gezogen werden.

Neben der klassischen Internetrecherche wurde ebenfalls auf der Frage-und-Antwort-Plattform "Stack Overflow" eine Frage zu der beschriebenen Problemstellung erstellt. Die Antwort hat darauf hingewiesen, eine Client-Server Architektur, also mit Front- und Backend, zu verwenden (samanime, 2021). Das Frontend soll mit React und das Backend mit ASP.NET implementiert werden. Die gewünschten Daten können dann mittels .NET-Technologie aufbereitet und per REST-Schnittstelle darauf zugegriffen werden.
Diese Variante wurde schon zu Beginn der Arbeit als "worst-case" Lösung von Endress + Hauser AG deklariert.

Folgende Ansätze wurden während der Recherche ausfindig gemacht und genauer analysiert:

1. C# zu WebAssembly kompilieren und per JavaScript ausführen
2. Client-Side Blazor Framework
3. Blazor Framework in Kombination mit React
4. .NET 6 Blazor Komponenten für JavaScript Frameworks

3.1 Variante 1 - C# zu WebAssembly und per JavaScript ausführen

Ziel dieser Variante ist es, C#-Code über einen Compiler zu WebAssembly zu kompilieren und in den Browser zu laden. Im Browser stehen die WebAssembly-Methoden zur Verfügung und können per JavaScript ausgeführt werden.

Diese Variante kann in zwei Teile aufgeteilt werden:
- Kompilation von C#-Code zu WebAssembly
- Von JavaScript aus WebAssembly-Methoden ausführen

3.1.1 Kompilation von C#-Code zu WebAssembly

Emscripten
Emscripten (Emscripten-Contributors, 2021) ist eine beliebte Compiler-Toolchain für WebAssembly, welche in der aktuellen Version 3.0.1 nur die Kompilation von C- und C++-Code unterstützt. C#-Code müsste zuerst in eine dieser Sprachen übersetzt werden, um dann in einem weiteren Schritt nach WebAssembly kompiliert werden zu können. Das Kompilieren von C++-Code nach WebAssembly funktioniert mit der von Emscripten zur Verfügung gestellten Dokumentation sehr gut. Diese Variante wurde aufgrund der erforderlichen Convertierung von C#- zu C++-Code nicht mehr weiterverfolgt.

Mono-Projekt
Das Mono-Projekt (Mono-Project, 2021) ist ein von Microsoft gesponsertes Projekt, welches sich der Aufgabe stellt, Cross-Plattform Applikationen als Teil von .NET erstellen zu können. In einem Artikel (Sansonetti, 2018) des Mono-Projekts wird erklärt, wie mit dem mono-wasm (de Icaza, 2018) Command-line Tool, C#-Code zu WebAssembly kompiliert werden kann. Das mono-wasm-Projekt auf Github wurde im Jahr 2018 zum letzten Mal bearbeitet, worauf auf Basis der fehlenden Weiterentwicklung entschieden wurde, diese Variante nicht weiter zu verfolgen.

Uno.Wasm.Bootstrap
Uno.Wasm.Bootstrap ist ein "unabhängiger .NET WebAssembly SDK Bootstrapper" in der Form eines NuGet-Pakets. Laut eigenen Aussuagen ist es "eine einfache Möglichkeit, C#-Code zu verpacken und ihn in einer kompatiblen Browserumgebung auszuführen" (unoplatform, 2021). Aufgrund von fehlender Dokumentation unbekannter Fehlermeldungen war es nicht möglich, den C#-Code im Browser auszuführen. Dies hat dazu geführt, auch diese Variante nicht weiter zu vertiefen.

3.1.2 Von JavaScript aus WebAssembly-Methoden ausführen

Für diese Arbeit wurde das Ausführen von WebAssembly-Methoden in JavaScript nicht ausführlich untersucht, da keine befriedigende Lösung für das das Kompilieren von C#-Code zu WebAssembly gefunden wurde. Von Mozilla existiert eine Anleitung im Web zu diesem Thema (MDN-Contributors, 2021c).

3.2 Variante 2 - Client-Side Blazor Framework

Blazor WebAssembly (Microsoft, 2021b) ist ein Framework für die Entwicklung von Single-Page-Applikationen. Es ist ein Feature des ASP.NET Web Frameworks und Teil der .NET-Plattform. Wie in Abbildung 2 ersichtlich ist, bestehen die damit entwickelten Applikationen aus .NET- und WebAssembly Code. Als Teil der .NET-Plattform erlaubt Blazor WebAssembly die Wiederverwendung von C# Code und den .NET Standard Libraries. Die Benutzeroberfläche wird mit Razor-Komponenten entwickelt. Schlussendlich wird die Applikation unter Verwendung von WebAssembly im Browser ausgeführt. Damit mit dem DOM interagiert werden kann, muss über JavaScript darauf zugegriffen werden (Latham & olprod, 2021).

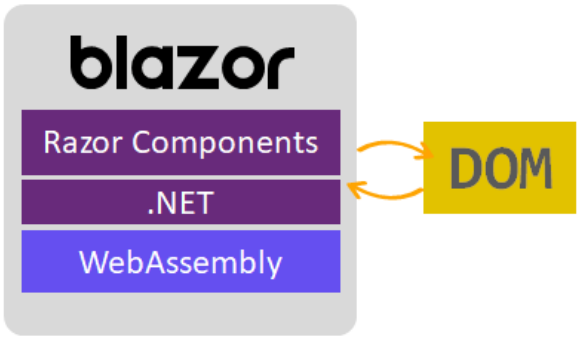

Abbildung 2: Blazor WebAssembly
(Latham & olprod, 2021)

3.2.1 Aufbau

In dieser Variante wird ein normales Blazor WebAssemly-Projekt, wie in Abbildung 3 vereinfacht dargestellt, verwendet. Untenstehend werden die wichtigsten Komponenten dieses Projekts genauer beschrieben.

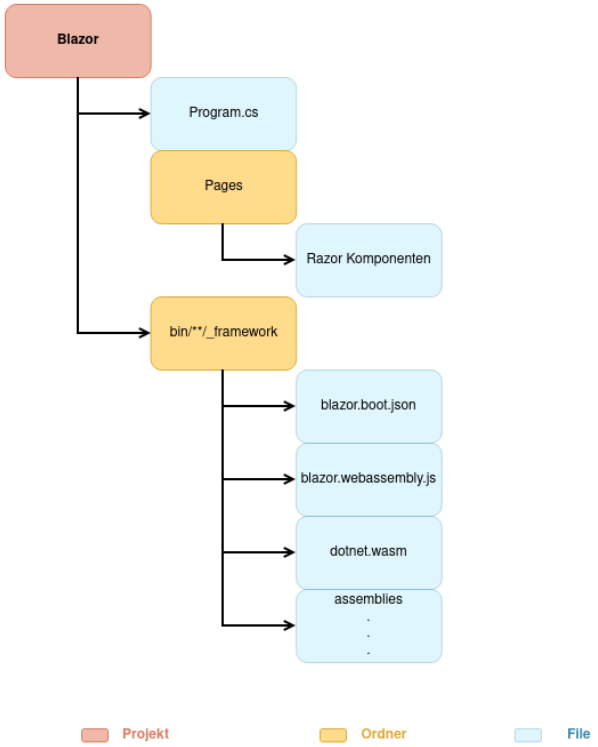

Abbildung 3: Übersicht der Blazor Projektstruktur

Razor-Komponenten

Razor ist eine Programmiersyntax, die in den Razor-Komponenten verwendet wird. Sie erlaubt das einbetten von C#- in HTML-Code.

Razor-Komponenten werden mit dem Programmiermodell von Razor-Pages entworfen, dieses kann für die Entwicklung von Web Applikationen verwendet werden (Abuhakmeh, 2020). Es unterscheidet sich von dem ASP.NET Programmiermodell insbesondere darin, dass jede Razor-Datei einen eigenen Endpoint oder eine Komponente darstellt und so das Implementieren von page-fokusierten Szenarien erleichtern kann (Anderson, Brock & Larkin, 2021).

Wie C# im Browser ausgeführt werden kann

Die .NET Toolchain trifft beim Ausführen eines Blazor WebAssembly-Projekts alle Vorkehrungen, damit dies im Browser ausgeführt werden kann. Dabei werden verschiedene Dateien mit unterschiedlichen Absichten erstellt. Die wichtigsten Dateien werden folgend noch erläutert und befinden sich im _framework-Ordner. Dieser wird automatisch durch die Toolchain erstellt.

Assemblies (.dll-Dateien)

In .NET werden Assemblies als Dynamic Link Libraries (DLL) implementiert. Dies ermöglicht, dass Assemblies erst in den Speicher geladen werden, wenn diese benötigt werden. Sie sind somit eine effiziente Lösung, um Ressourcen in einem grossen Projekt zu verwalten (Pine et al., 2021). Die Assemblies bestehen aus Intermediate Language (IL) Code und werden beim kompilieren von .NET-Programmiersprachen erstellt. Der IL-Code wird von der .NET Runtime interpretiert und in einer virtuellen Maschine ausgeführt.

dotnet.wasm

Die dotnet.wasm-Datei ist die in WebAssembly implementierte .NET Runtime. Damit diese im Browser ausgeführt werden kann, wurde sie von Microsoft im WebAssembly Standard implementiert. Sie ermöglicht, dass der IL-Coe im Kontext von WebAssembly interpretiert und ausgeführt werden kann.

blazor.webassembly.js

Diese JavaScript-Datei ist verantwortlich, dass alle Dateien die nötig sind, damit die Applikation im Browser ausgeführt werden kann, in den Browser geladen werden (Collins, 2020). Zusätzlich werden im Script die nötigen globalen Properties gesetzt, damit von JavaScript Funktionen C# Methoden, oder vice versa, aufgerufen werden können (Smacchia, 2021). Die Verbindung mit JavaScript ist vor allem wichtig, damit eine Blazor Applikation mit dem DOM interagieren kann, da WebAssembly zurzeit nicht selbst auf den DOM zugreifen kann (MDN-Contributors, 2021a).

blazor.boot.json

Wenn Blazor die Startup-Dateien in den Browser lädt, werden im Browser Integritätstests ausgeführt (Latham, Anderson, Addie, K & Sanderson, 2021). Dabei werden SHA-256 Hashes der Dateien, welche zusätzlich in den Browser geladen werden müssen, mit den im *blazor.boot.json*-Datei gespeicherten Hashes verglichen. Damit wird verhindert, dass

- inkonsistente Dateien geladen werden, was zu undefiniertem Verhalten führen kann
- inkonsistente Dateien im Browser zwischengespeichert werden, welche den Start der Applikation verhindern können.

3.2.2 Umgang mit rechenintensiven Aktionen im Browser

Wenn im Browser rechenintensive Aktionen durchgeführt werden sollen, wird die Verwendung von BlazorWorkers empfohlen. Der Umgang mit BlazorWorkers ist unter 5.2.2 genauer ersichtlich.

3.3 Variante 3 - Blazor Framework in Kombination mit React

Ein Blazor WebAssembly-Projekt erstellt alle notwendigen Dateien, damit dieses im Browser ausgeführt werden kann. Zusätzlich werden dabei alle nötigen globalen Properties gesetzt, damit von JavaScript aus, C#-Methoden aufgerufen werden können. Im diesem Abschnitt wird erklärt, wie in einer mit dem React-Framework geschriebenen Applikation, C#-Methoden aufgerufen werden können.

3.3.1 Aufbau

Damit von einem React-Projekt C#-Methoden aufgerufen werden können, müssen zwei separate Projekte, wie in Abbildung 4 ersichtlich, erstellt werden. Dabei handelt es sich um Standard React- und Blazor-Projekte.

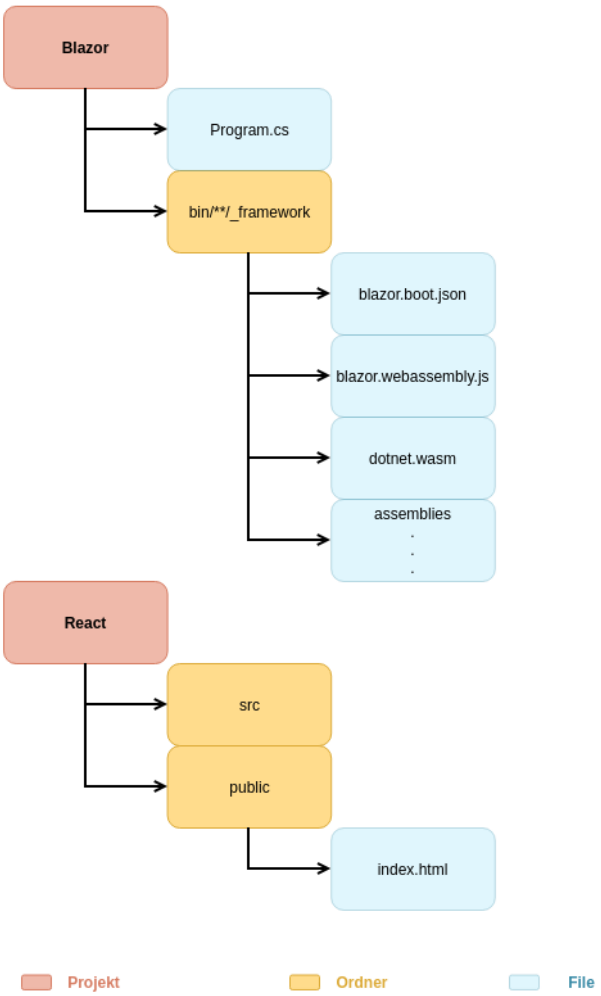

Abbildung 4: Übersicht der React und Blazor Projektstruktur

3.3.2 Blazor Projekt

Ein Blazor WebAssembly-Projekt erstellt bereits viele notwendigen Abhängigkeiten. Zusätzlich müssen die Methoden, welche von dem React-Projekt aus aufgerufen werden sollen, eine bestimmte Signatur haben. Weitere Informationen über das Blazor WebAssembly-Projekt sind in Kapitel 3.2 ersichtlich.

C# Methodensignatur

Die C#-Methoden, welche aus dem React-Projekt aufgerufen werden sollen, können in C#-Klassen (.cs Files), oder in Razor-Komponenten (.razor File) definiert werden. Diese Methoden müssen das *JSInvokable*-Attribut haben, sowie *public* und *static* sein. Das *JSInvokable*-Attribut befindet sich im Microsoft.JSInterop-Namespace.

```
1     [JSInvokable]
2     public static int MethodName(int a, int b)
```

Listing 2: Notwendige Signatur für C# Methoden

Als Rückgabetypen können alle primitiven Datentypen, Strings und Enums verwendet werden. Um selbst definierte Objekte zurück zu geben, müssen diese zuerst in ein JSON-Objekt konvertiert und dann als String zurückgegeben werden.

3.3.3 React-Projekt

Das React-Projekt kann ein normales React-Projekt sein. Um C#-Methoden des Blazor-Projekts aufrufen zu können, müssen allerdings zusätzlich noch die folgenden Anpassungen vorgenommen werden.

Blazor Projekt in den Browser laden

Die index.html-Datei ist nach dem Standard die erste Datei, welche beim Ausführen der React Applikation in den Browser geladen wird. Damit zusätzlich alle notwendigen Dateien des Blazor Projekts beim Start der Applikation in den Browser geladen werden, muss die blazor.webassembly.js-Datei von der index.html- Datei aufgerufen werden.

```
1     <script src="_framework/blazor.webassembly.js"></script>
```

Listing 3: index.html

Aufruf von C# Methoden in JavaScript

Von JavaScript können die in C# definierten Methoden wie folgt, synchron oder asynchron, über das globale window-Objekt aufgerufen werden. Das DotNet-Property des window-Objekts wird im blazor.webassembly.js-Script definiert.

```
1     window.DotNet.invokeMethod('{assemblyName}', 'MethodName')
2     window.DotNet.invokeMethodAsync('{assemblyName}', 'MethodName')
```

Listing 4: C# Methoden aufruf aus JavaScript

Anstelle des *assemblyName*-Platzhalters muss der tatsächliche Name der Assembly-Datei des Blazor-Projekts angegeben werden. Dieses trägt meistens den Namen des Projekts.

Ablauf um C#-Methoden in React auszuführen

Damit von dem React-Projekt die im Blazor-Projekt definierten C#-Methoden aufgerufen werden können, müssen diese wie unter 3.3.2, definiert werden. Danach muss der _framework-Ordner des Blazor-Projekts in den public-Ordner des React-Projekts kopiert werden. Der _framework-Ordner wird beim kompilieren des Blazor-Projekts automatisch erstellt. Die Dateien, welche darin enthalten sind, werden unter 3.2 genauer erläutert. Damit alle notwendigen Verknüpfungen zum Blazor-Projekt erstellt werden, muss beim Start der Applikation die blazor.webassembly.js-Datei ausgeführt werden. Dazu muss dies in der index.html-Datei eingebunden werden. Nach diesen Vorkehrungen kann von JavaScript aus, wie unter Kapitel 3.3.3 beschrieben, die C#-Methoden aufgerufen werden.

3.4 Variante 4 - .NET 6 Blazor Komponenten für JavaScript Frameworks

Mit der .NET Version 6 werden Updates für ASP.NET und dessen Feature Blazor eingeführt (Roth, 2021a). Eines der neuen Features ermöglicht es, dass Blazor Komponenten von JavaScript aus dargestellt werden können. Somit können Blazor Komponenten zum Beispiel für React oder Angular Anwendungen zur Verfügung gestellt werden. Damit die Komponenten korrekt generiert werden, müssen einige wichtige Punkte, welche in diesem Abschnitt beschriebene werden, beachtet werden.

Als Test-Projekt wurde das von Microsoft zur Verfügung gestellte Beispielprojekt (Roth, 2021b) mit selbst erstellten Komponenten erweitert und ausgeführt. Zum Zeitpunkt, als verschiedene Lösungen zur Umsetzung der Anforderungen analysiert wurden, war .NET 6 noch nicht offiziell veröffentlicht. Microsoft stellt jedoch "Release Candidate" Versionen zur Verfügung, bei welchen es meist keine grundlegenden Veränderungen zum offiziellen Release gibt.

3.4.1 Aufbau

Damit die Komponenten erfolgreich generiert werden, sind fünf Teilprojekte notwendig. Die genaue Funktion der wichtigsten drei Teilprojekten und deren Dateien werden im folgenden Abschnitt genauer beschrieben.

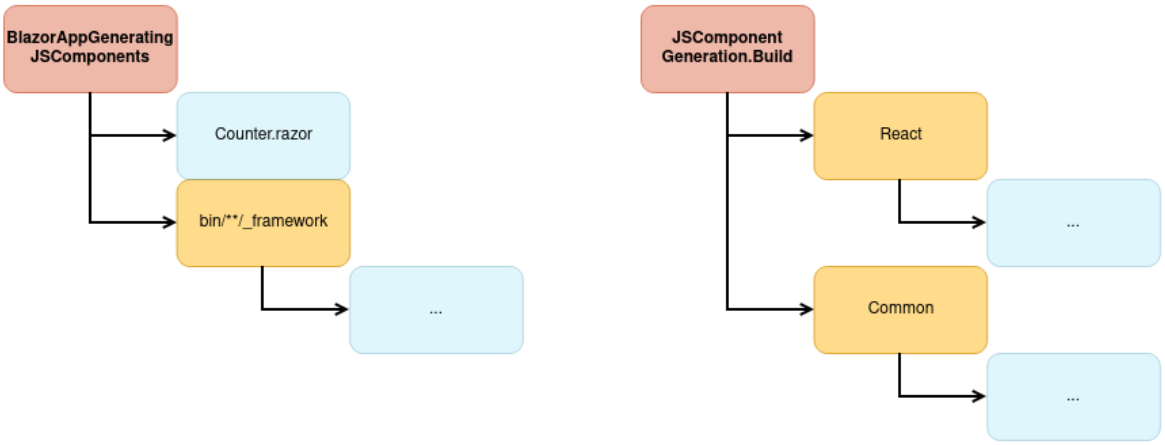

Abbildung 5: Übersicht der .NET 6 Projektstruktur

BlazorAppGeneratingJSComponents

- **Counter.razor:** Definition Komponente in C#
- **bin/**/_framework:** WebAssembly Build (wird in React *public* Ordner kopiert)

JSComponentGeneration.Build

- C# Dateien um Komponenten zu generieren

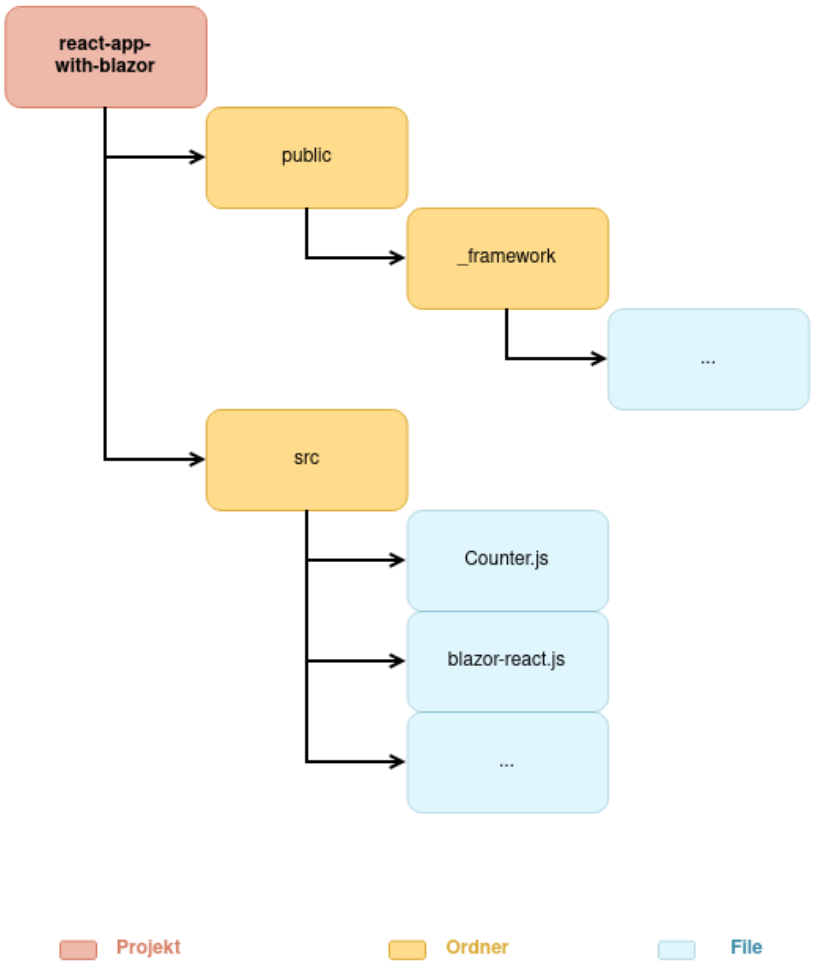

Abbildung 6: Übersicht Projektstruktur .NET 6 Lösung - React

react-app-with-blazor (React Projekt)

- **public:** WebAssembly Build aus *BlazorAppGeneratingJSComponents*
- **src/Counter.js:** Generierte React Komponente
- **src/blazor-react.js:** Laden des WebAssembly Code aus *_public*

3.4.2 Konfiguration

MyComponent.razor
Damit für eine Komponente im Build Prozess ein React Wrapper generiert wird, muss dies am Anfang jeder Komponente deklariert werden.

```
1     @attribute [GenerateReact]
```

Listing 5: GenerateReact Attribut

Program.cs
Im Program.Main des Blazor Projekts müssen alle Komponenten angegeben werden, welche aus React verwendet werden sollen.

```
1     var builder = WebAssemblyHostBuilder.CreateDefault(args);
2     builder.RootComponents.RegisterForReact<MyComponent>();
```

Listing 6: Deklaration von Komponenten in Program.Main

blazor-react.js
In dieser JavaScript Datei wird die Funktion *useBlazor* definiert, welche aus der generierten React Komponente aufgerufen wird. Die *useBlazor* Funktion ist zuständig, um eine Komponente in den Komponenten-Baum einzufügen und wenn gewünscht wieder zu entfernen.
Über das *Blazor* Element, welches beim Laden der Webseite instanziert wird, kann eine Komponente und deren Eltern-Element angefügt werden.

```
1     Blazor.rootComponents.add(parentElement, `${identifier}-react`, props);
```

Listing 7: Blazor Objekt

3.4.3 Verwendung

.NET mit C#

Einer generierten JavaScript Komponente können Parameter als Built-in Typen wie (bool, int, string usw.) oder Komplexe Objekte, welche JSON-Serialisierbar sind, übergeben werden. Ebenfalls besteht die Möglichkeit als *EventCallback<T>* Callback Funktionen zu übergeben.

Das folgende Beispiel zeigt eine komplette Komponente, welcher ein Parameter CounterValue (Typ: int) und eine Callback Funktion *CustomCallback* (Paramter: int) übergeben werden kann.

```
1  @attribute [GenerateReact]
2
3  <p>Current count: @CounterValue</p>
4  <button @onclick="CustomCallback">Invoke callback</button>
5
6  @code {
7      [Parameter] public int? CounterValue { get; set; }
8      [Parameter] public EventCallback<int?> CustomCallback { get; set; }
9  }
```

Listing 8: MyComponent.razor

React

In der React Applikation kann die generierte Komponente wie eine andere JavaScript React Komponenten verwendet werden.

```
1  <Counter
2      counterValue={counter}
3      customCallback={customCallbackFunction}
4  ></Counter>
```

Listing 9: Verwendung MyComponent in React JS

4 Vergleich der Varianten

Um eine Übersicht zu den Vor- und Nachteile der verschiedenen Lösungsansätze zu bekommen, wurden verschiedene Kriterien definiert. Anhand dieser Punkte wurde jede Möglichkeit bewertet.

4.1 Kriterien

Eine Kombination aus Aufgabenstellung, Anforderungen von Endress + Hauser AG und eine Sammlung von Kriterien für Open Source Software (Semeteys, Pilot, Baudrillard, Lebouder & Pinkhardt, 2006) führten zum nachfolgenden Kriterienkatalog. Zwei Punkte wurden als unumgänglich und somit als "Muss-Kriterium" markiert.

4.1.1 Übersicht

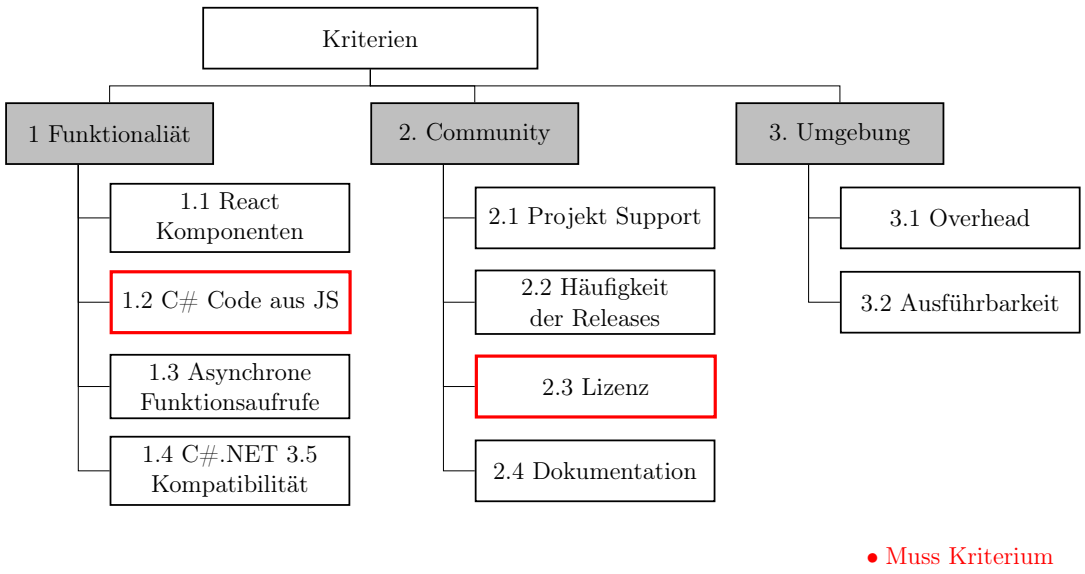

Abbildung 7: Übersicht Kriterien

4.1.2 Details

Tabelle 1: Kriterienkatalog

1	**Funktionalität**			
1.1	**React Komponenten**	Um das geforderte Ziel (C# Code aus einer modernen HTML5 Umgebung zu verwenden, ist es zwingend nötig, dass mit dem React Framework gearbeitet werden kann.		
	+ React kann mit allen Features inklusive JSX verwendet werden.			- React kann nicht verwendet werden.
1.2	**C# Code aus JS**	Damit der bestehende C# Code verwendet werden kann, ist es zwingend nötig, dass C# Funktionen aus JavaScript aufgerufen und Parameter übergeben werden können.		
	+ C# Funktionen können inklusive Parameter aus JavaScript aufgerufen werden.		**0** C# Funktionen können nicht direkt, sondern nur über Umwege aus JavaScript aufgerufen werden.	- Es ist nicht möglich C# Funktionen aus JavaScript aufzurufen.
1.3	**Asynchrone Funktionsaufrufe**	Damit es im Browser zu einer optimalen User Experience kommt und die Benutzeroberfläche beim Ausführen einer Berechnung nicht einfriert, muss es möglich sein, Funktionen asynchron aufzurufen.		
	+ Asynchrone Funktionsaufrufe möglich			- Asynchrone Funktionsaufrufe nicht möglich
1.4	**C#.NET 3.5 Kompatibilität**	Der aktuell vorhandene Code, welcher wiederverwendet werden soll, ist in .NET 3.5 geschrieben. Im besten Fall sollte der Lösungsansatz damit kompatibel sein.		
	+ C#.NET Code kann zu 100% verwendet werden.		**0** Mindestens 80% des vorhandenen C#.NET 3.5 Code können verwendet werden.	- Es kann unter 80% des C#.NET 3.5 Code verwendet werden.

2	**Community**		
2.1	**Projekt Support**	Namhafte Sponsoren mit entsprechenden Budgets vergrössern die Wahrscheinlichkeit, dass eine Software / Framework in mehreren Jahren immer noch aktiv weiterentwickelt wird.	
	+	**0**	**-**
	Das Framework wird aktiv von grossen Softwareunternehmen unterstützt und verwendet.	Es ist klar ersichtlich, dass mehrere Firmen oder Person das Projekt unterstützen.	Das Projekt wird von einzelnen Personen unterhalten.
2.2	**Häufigkeit der Releases**	Die Anzahl regelässiger Releases, welche von einer Software veröffentlich werden, sind ein guter Indikator dazu, wie aktiv ein Projekt weiterentwickelt wird.	
	+	**0**	**-**
	Es wird mindestens monatlich eine neue Version der Software veröffentlicht.	Mindestens halbjährlich wird eine verbesserte Version der Software veröffentlicht.	Das Projekt wurde schon mehrere Jahre nicht mehr aktiv verbessert.
2.3	**Lizenz**	Um eine entsprechende Software kommerziell zu verkaufen, müssen alle verwendeten Libraries und Frameworks eine entsprechende Lizenz haben.	
	+		**-**
	Freizügige Open-Source-Lizenz		keine Freizügige Open-Source-Lizenz
2.4	**Dokumentation**	Um allfällige Fehler und den Aufwand für die Implementierung möglichst klein zu halten ist eine gute Dokumentation essenziell.	
	+	**0**	**-**
	Das Framework ist sehr ausführlich inklusive Anwendungsbeispiele dokumentiert.	Das Projekt ist in einer minimalen Version dokumentiert.	Es ist keine Dokumentation vorhanden.

3	**Umgebung**			
3.1	**Overhead**	Wenn der Anteil an Code, welcher für eine funktionierende Anwendung geschrieben werden muss zu gross wird, kann die Entwicklung sehr mühsam und aufwendig werden.		
		+	**0**	**-**
		Es muss kein zusätzlicher Code geschrieben werden.	Der ganze Overhead Code muss nur einmalig geschrieben werden.	Jede neue Funktion beinhaltet Overhead Code.
3.2	**Ausführbarkeit**	Damit das Integrieren der Software in eine bestehende Lösung möglichst einfach funktioniert, sollte der Aufwand für die Ausführung der Software möglichst klein sein.		
		+	**0**	**-**
		Software kann ohne zusätzliche Frameworks statisch im Browser betrieben werden.	Software kann nur über zusätzlichen Server im Browser ausgeführt werden.	Es ist ein Framework (z.B. .NET) nötig, um die Software auszuführen.

4.2 Bewertung der Varianten anhand der Kriterien

Tabelle 2: Bewertung der verschiedenen Varianten

		Variante 2 Client-Side Blazor	**Variante 3** Blazor + React	**Variante 4** .NET 6 Komponenten
1.1	React Komponenten	-	+	+
1.2	C# Code aus JS	+	+	+
1.3	Asynchrone Funktionsaufrufe	+	-	-
1.4	C#.NET 3.5 Kompatibilität	+	+	+
2.1	Projekt Support	+	-	+
2.2	Häufigkeit der Releases	+	+	+
2.3	Lizenz	+	+	+
2.4	Dokumentation	+	-	0
3.1	Overhead	0	0	-
3.2	Ausführbarkeit	0	+	+

4.2.1 Variante 1 - C# zu WebAssembly kompilieren und per JavaScript ausführen

- Aktuell war es nicht möglich C#-Code nach WebAssembly zu kompilieren.
- Dennoch ist Emscripten ein sehr vielversprechendes Werkzeug, welches in Zukunft die mit dieser Arbeit gesuchte Lösung werden könnte.
- Aus den oben genannten Gründen ist diese Variante nicht in der Bewertungstabelle aufgeführt, aber dennoch im Dokument zu finden.

4.2.2 Variante 2 - Client-Side Blazor Framework

- Um die Probleme, welche in Variante 3 und 4 aufgetreten sind zu vermeiden, wurde eine Demoapplikation mit dem kompletten Blazor-Framework und all seinen Funktionen erstellt. Bei dieser Variante wurde auf die Verwendung von React verzichtet.
- Dank des "BlazorWebWorkerHelper" NuGet-Packages ist es möglich asynchrone Funktionsaufrufe zu tätigen.
- Es wurde keine Lösung gefunden um das Blazor-Framework ohne Server an den Browser auszuliefern.

4.2.3 Variante 3 - Blazor Framework in Kombination mit React

- Da es sich bei dieser Variante um keine offiziell von Microsoft veröffentlichte Lösung handelt, kann im Internet kaum auf Unterstützung gezählt werden. Somit sind die zwei grossen Minuspunkte der nicht vorhandene Projekt Support sowie die fehlende Dokumentation.
- Wie bei .NET 6 besteht das Problem, dass Funktionen nicht asynchron aufgerufen werden können.

4.2.4 Variante 4 - .NET 6 Blazor Komponenten für JavaScript Frameworks

- .NET 6 wurde erst Anfang November 2021 veröffentlicht. Deshalb gab es zur Zeit der Variantenanalyse, bis auf wenige Blogartikel, kaum Informationen zu den neu hinzugefügten Funktionen der .NET Version sechs.
- Wie bei "Variante 2 - Client-Side Blazor Framework", ist es nicht möglich asynchrone Funktionsaufrufe zu tätigen, ohne dass der Browser einfriert.

4.3 Diskussion

Leider konnte der geforderte Prototyp nicht mit allen Varianten umgesetzt werden. Dennoch sind die beschriebenen Varianten für die Zukunft sehr vielversprechend und eine bevorstehende Version der benutzten Tools und Frameworks könnte eine erfolgreiche Umsetzung möglich machen.

Für den weiteren Verlauf der Arbeit wurde entschieden, je ein Prototyp für die Variante 2 "Client-Side Blazor Framework" und Variante 3 "Blazor Framework in Kombination mit React" zu erstellen. Diese Entscheidung wurde zum einen Anhand der oben beschriebenen Kriterien getroffen. Zum anderen aber auch, weil die Variante 2 unserer Ansicht nach am ehesten in der Produktion verwendet werden kann und die Variante 3 die Ziel-Anforderungen der Arbeit am besten abdeckt.

4.4 Umsetzung Prototyp

Um zu prüfen wie praktisch sich eine geforderte Applikation umsetzen lässt und allfällig versteckte Probleme zu finden, wurde ein Prototyp mit den ausgewählten Variante umgesetzt. Das entsprechende User Interface wurde anhand von Screenshots und Informationen, welche von Endress + Hauser AG zur Verfügung gestellt wurden, implementiert.

4.4.1 Anforderungen

Folgende Anforderungen wurden an die Protoypen gestellt:

- Anhand von eingegebenen Zahlen soll eine aufwendige Berechnung in C# durchgeführt werden und das entsprechende Resultat in der Applikation dargestellt werden
- Auswahlfelder sollen die Werte vordefinierter Enums anzeigen
- Es muss nur die Ansicht der Base Settings implementiert werden

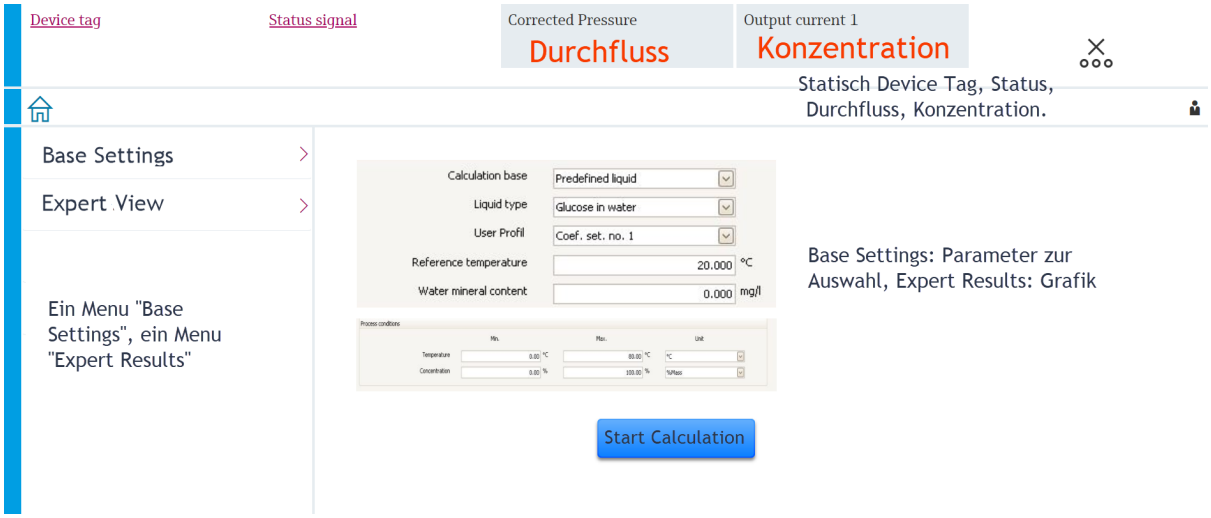

Abbildung 8: UI Mockup Prototyp

4.4.2 Umsetzung Frontend

Um doppelte Arbeit zu vermeiden, wurde im ersten Schritt die React Version mit JavaScript umgesetzt. Die Struktur der Komponenten konnte ohne grossen Aufwand in die Razor Syntax des Blazor Frameworks übernommen werden.

Daraus entstanden zwei, vom Aussehen identische, Versionen des Frontends:

- Version 1: JavaScript und React
- Version 2: Razor Syntax und Blazor Framework

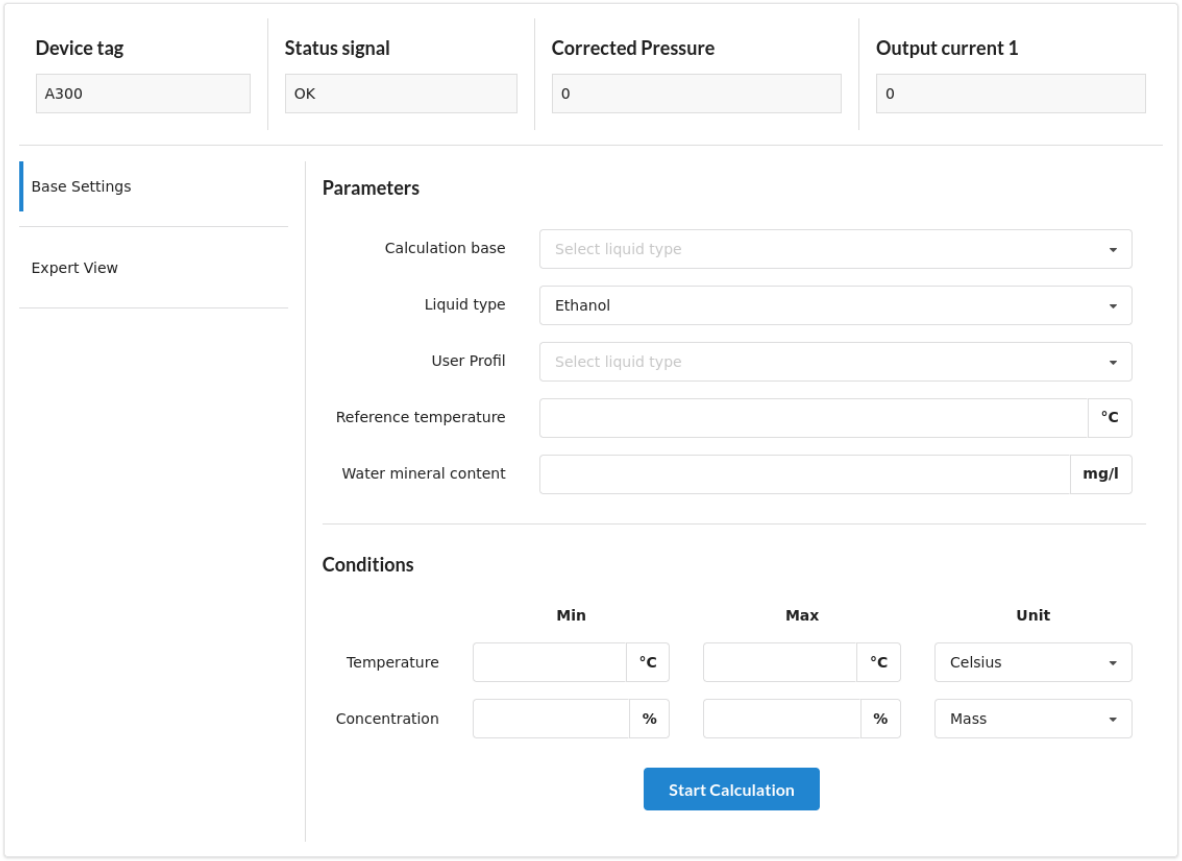

Abbildung 9: Screenshot Prototyp

4.4.3 Umsetzung C# Berechung

In der Client-Side Blazor Applikation kann ohne weiteren Aufwand auf C# Methoden des gleichen Projekt zugegriffen werden.

Für die Prototypen, bei welchen das Frontend mit React umgesetzt wurde, braucht es speziell deklarierte Methoden, auf welche per JSInterop zugegriffen werden kann. Für die Simulation einer grösseren Berechnung wurde analog zu den Berechnungen im Abschnitt Performance Vergleich mit Hilfe der Leibniz-Reihe die Zahl Pi angenähert.

```
    [JSInvokable]
    public static double CalculatingConcentration(
        Fluid fluid,
        double MinConcentration,
        double MaxConcentration,
        ConcentrationUnits unitConcentration,
        double MinTemperature,
        double MaxTemperature,
        TemperatureUnits unitTemperature,
        double currentTemperature,
        double mineralContentWater){

            int numberOfSummands = 10_000_000;
            double sum = 0;

            // Berechnung
            for (int k = 0; k < numberOfSummands; k++) {
                sum += Math.Pow(-1, k) / (2 * k + 1);
            }
            double pi = sum * 4;


            return MinConcentration + MaxConcentration;

    }
```

Listing 10: JavaScript Invoakable C# Funktion

5 Probleme und mögliche Lösungen

5.1 Unmögliche asynchrone Berechnung

Durch das Implementieren einer Funktion, die eine mehrere Sekunden dauernde Berechnungen ausführt, wurde ein schwerwiegendes Problem entdeckt. Sobald eine Berechnung gestartet wird, friert die gesamte Benutzeroberfläche im Browser ein und es können bis zum Abschluss der Funktion keine Interaktionen mit der Benutzeroberfläche getätigt werden.

5.1.1 Grund des Problems

JavaScript verwendet das Modell der Nebenläufigkeit mit einem Event Loop. Dieser verfolgt das sogenannte "run to completion" Scheduling Modell. Dies bedeutet, dass eine Funktion immer bis zum Ende laufen muss, bevor weiterer Code ausgeführt werden kann (Brickett, 2020).

Der grosse Nachteil dieses Modells ist, dass Funktionen welche lange Laufzeiten aufweisen, die ganze Benutzeroberfläche zum Einfrieren bringen. Da JavaScript in nur einem Thread ausgeführt wird, ist die Applikation unfähig Benutzerinteraktionen zu verarbeiten, während sie mit dem Bearbeiten lange andauernden Funktionen beschäftigt ist.
Die meisten Browser besitzen für jeden Tab einen eigenen Event Loop. Somit blockiert, ein unendlich andauernder Loop, nicht den ganzen Browser, sondern nur dessen Tab.

Die folgende Grafik stellt einen Event Loop dar. Der rote Pfeil zeigt die aktuelle Stelle im Programm, auf der linken Seite sind verschiedene Tasks dargestellt, welche jeweils nachrücken, wenn einer davon vom Programm abgearbeitet wurde. Die rechte Seite stellt verschiedene Schritte zum rendern der aktuellen Ansicht dar (Archibald, 2018).

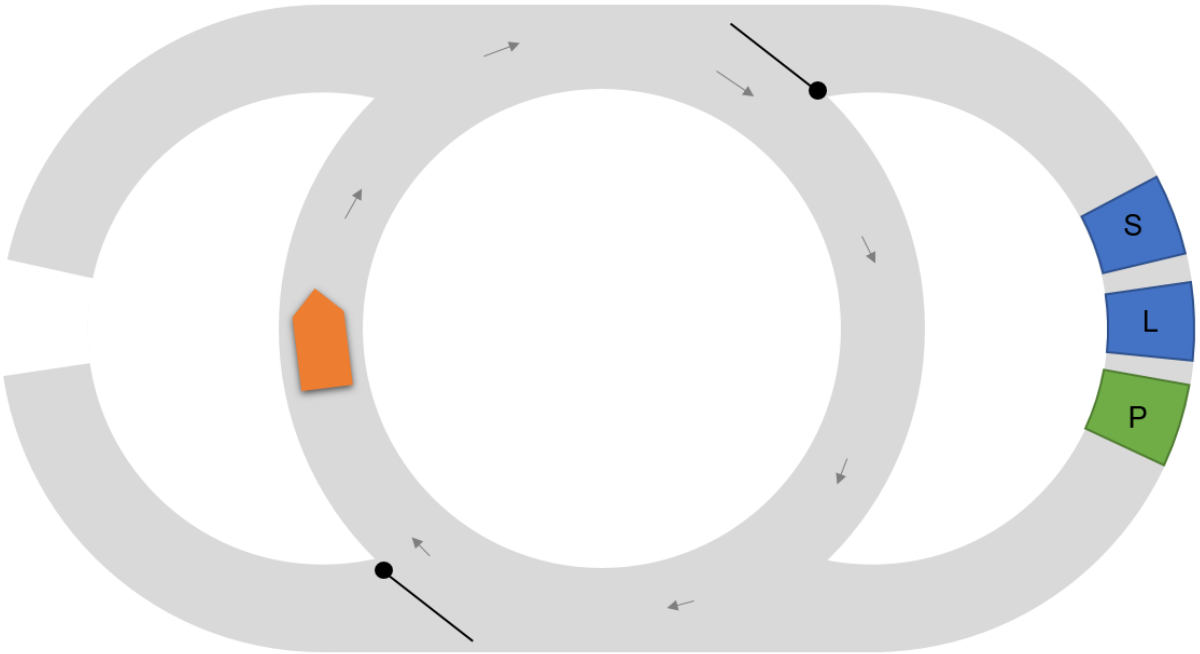

Abbildung 10: JavaScript Event Loop

Wenn in einer Applikation nichts passiert und somit im Leerlauf ist, dreht der Pfeil im inneren Kreis. Sobald ein Task nachrückt, welcher abgearbeitet werden soll, wird dieser vom Programm komplett verarbeitet. In der Darstellung würde sich der rote Pfeil somit beim Task befinden. Wie vorangehend beschrieben werden diese Funktionen immer komplett verarbeitet. Dies führt dazu, dass das Layout nicht neu gerendert wird und zu diesem Zeitpunkt eingefroren ist.

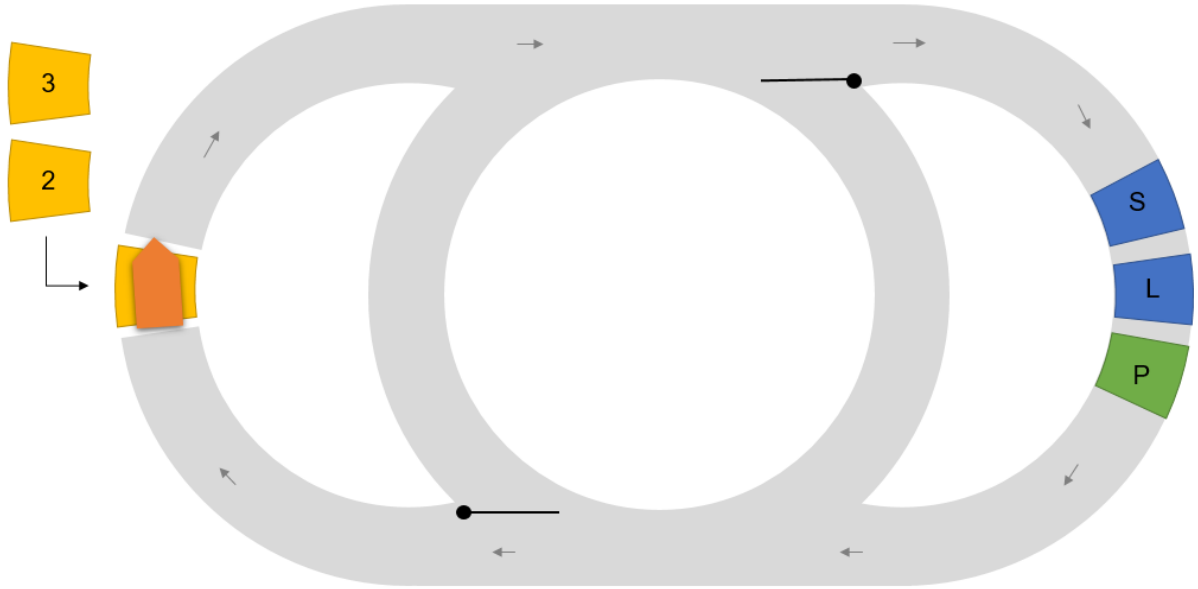

Abbildung 11: Event Loop mit blockiertem UI

Während die Funktion verarbeitet wird, wird der rechte Teil des Event Loops, welcher für das Rendern der Benutzeroberfläche zuständig ist, nie durchlaufen. Somit werden auch keine grafischen Anpassungen sichtbar und die Applikation fühlt sich fühlt sich für den Benutzer wie blockiert an.

5.1.2 Lösung des Problems

Sogenannte Web Workers sind ein einfacher Weg, um Scripts im Hintergrund, abseits des UI-Threads auszuführen. Ein Worker kann beliebige Nachrichten an den Event Handler des Erstellers und umgekehrt senden.

Einen neuen Worker zu erstellen, ist simpel. Alles was dazu nötig ist, ist das entsprechende Script als Parameter dem Konstruktor zu übergeben.

```
1    let myWorker = new Worker('worker.js');
```

Listing 11: Instanziierung Web Worker

Über die *onmessage*- und *postMessage*-Methoden werden mit dem Worker Messages versendet.

```
1    onmessage = function(e){
2        console.log('Nachricht vom main script erhalten!');
3        postMessage("");
4    }
```

Listing 12: Übermitteln von Informationen

5.2 Fehlender Zugriff auf globale DotNet Variable

Im vorherigen Abschnitt wurden JavaScript Web Worker als Lösung für asynchrone Methodenaufrufe erwähnt. Diese funktionieren mit JavaScript einwandfrei und aufwendige Berechnungen können darüber in einen anderen Thread ausgelagert werden.

In Kombination mit WebAssembly und C# gibt es jedoch ein entscheidendes Problem. Aus JavaScript wird über das *window.DotNet* Objekt, welches beim Starten der Anwendung, vom blazor.webassembly.js Script automatisch generiert wird, auf C# Methoden zugegriffen.

```
1    window.DotNet.invokeMethod('{assemblyName}', 'MethodName')
```

Listing 13: C#-Methoden aufruf aus JavaScript

Web Workers arbeiten in einem globalen Kontext, der sich von dem *window* unterscheidet (MDN-Contributors, 2021d). Dies bedeutet, dass ein Fehler retourniert wird, wenn im Web Worker auf das globale window Objekt zugegriffen wird. Folgend eine Liste von Objekten, auf die innerhalb eines Web Workers nicht zugegriffen werden kann:

- *window* Objekt
- *document* Objekt
- *parent* Objekt

5.2.1 Mögliche Erklärung

Eine mögliche Erklärung dazu ist, dass über die aufgelisteten Objekte auch das DOM manipuliert werden könnte und dies damit unterdrückt wird. Ebenfalls ist es nicht möglich das *window* Objekt per Nachricht an einen Worker zu übergeben. Wäre dieser Zugriff möglich, könnte es zu Fehler im Rendering einer Webseite kommen.

5.2.2 Lösung des Problems

Leider wurde für die Variante mit React Frontend keine funktionierende Lösung gefunden, um das einfrieren der Benutzeroberfläche für lang andauernde Berechnungen zu verhindern.

Für die Variante Client-Side Blazor wurde eine externe Library gefunden, welche es möglich macht Web Worker aus einem Blazor Projekt zu verwenden.

In diesem GitHub Repository (https://github.com/Tewr/BlazorWorker) wird eine Lösung vorgestellt, wie Web Workers innerhalb einer Blazor Applikation verwendet werden können. Dafür müssen die Pakete Tewr.BlazorWorker.Core und Tewr.BlazorWorker.BackgroundService über den Package Manager NuGet dem Projekt hinzugefügt werden.

Die BlazorWorker funktionieren zur Zeit des Verfassens dieser Arbeit nicht mit .NET 6. Die Entwickler des BlazorWorker Library haben aber bereits einen Request erstellt um diese in .NET 6 aufzunehmen.

6 Performance-Vergleich

Um einen Vergleich der Performance zwischen einer konventionellen C# und einer Blazor Anwendung zu testen, wurden verschiedene Messungen mit unterschiedlich aufwendigen Berechnungen gemacht.

6.1 Leibniz-Reihe

Als rechenintensive Berechnung wurde die Leibniz-Reihe gewählt, welche zur Näherung von Pi verwendet werden kann. Die Folge konvergiert nur sehr langsam und ist deshalb zur effizienten Berechnung von Pi nicht geeignet. Da für die durchgeführten Performance Tests nur eine rechenintensive Aufgabe gesucht wurde, spielt dies für die Messergebnisse keine Rolle.

6.1.1 Formel

$$\sum_{k=0}^{\infty} \frac{(-1)^k}{2k+1} = 1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \frac{1}{9} - ... = \frac{\pi}{4}$$

6.1.2 Berechnung als Code

```
1     int numberOfSummands = 10_000_000;
2     double sum = 0;
3
4     // Berechnung
5     for (int k = 0; k < numberOfSummands; k++) {
6         sum += Math.Pow(-1, k) / (2 * k + 1);
7     }
8
9     double pi = sum * 4;
```

Listing 14: Berechnung Leibniz Reihe

Durch den Parameter *numberOfSummands* kann definiert werden, wie viele Summanden zum Berechnen des Resultats verwendet werden sollen.

6.2 Durchführung

Damit die Anwendungen jeweils eine angemessen aufwändige Berechnung durchführen mussten, wurde jeweils eine Leibniz-Reihe mit **10'000'000 Summanden** berechnet. Damit einzelne Ausreisser nicht ins Gewicht fallen,wurden jeweils **50 Messungen** durchgeführt und als Ergebnis der Median dieser Resultate bestimmt.

6.2.1 Testumgebung

Die Tests wurden auf verschiedenen Computern mit unterschiedlichem Betriebssystem und Browser durchgeführt. Dabei wurden grundsätzlich immer sehr ähnliche Resultate erreicht. Die unten aufgeführte Statistik wurde mit folgendem Setup gemessen:

- Lenovo ThinkPad T480
- i7-8550U 1.8 GHz
- 16 GB RAM
- Linux Fedora 34

- Chromium Version 94.0
- Firefox 94
- .NET Framework 5.0.206

6.2.2 Testvarianten

- **JavaScript Synchron:** Berechnung im UI-Tread
- **JavaScript mit Web Worker:** Berechnung im Web Worker
- **Konsolenanwendung:** Berechnung im Main-Thread
- **Emscripten C++:** C++ Berechnung als WebAssembly im Browser
- **Blazor mit Web Worker:** Blazor mit "BlazorWebWorker" Library
- **Blazor Synchron:** Blazor im UI-Thread

6.3 Ergebnisse

50 Messungen à 10'000'000 Summanden

X-Achse: getestete Variante
Y-Achse: Zeit in Millisekunden

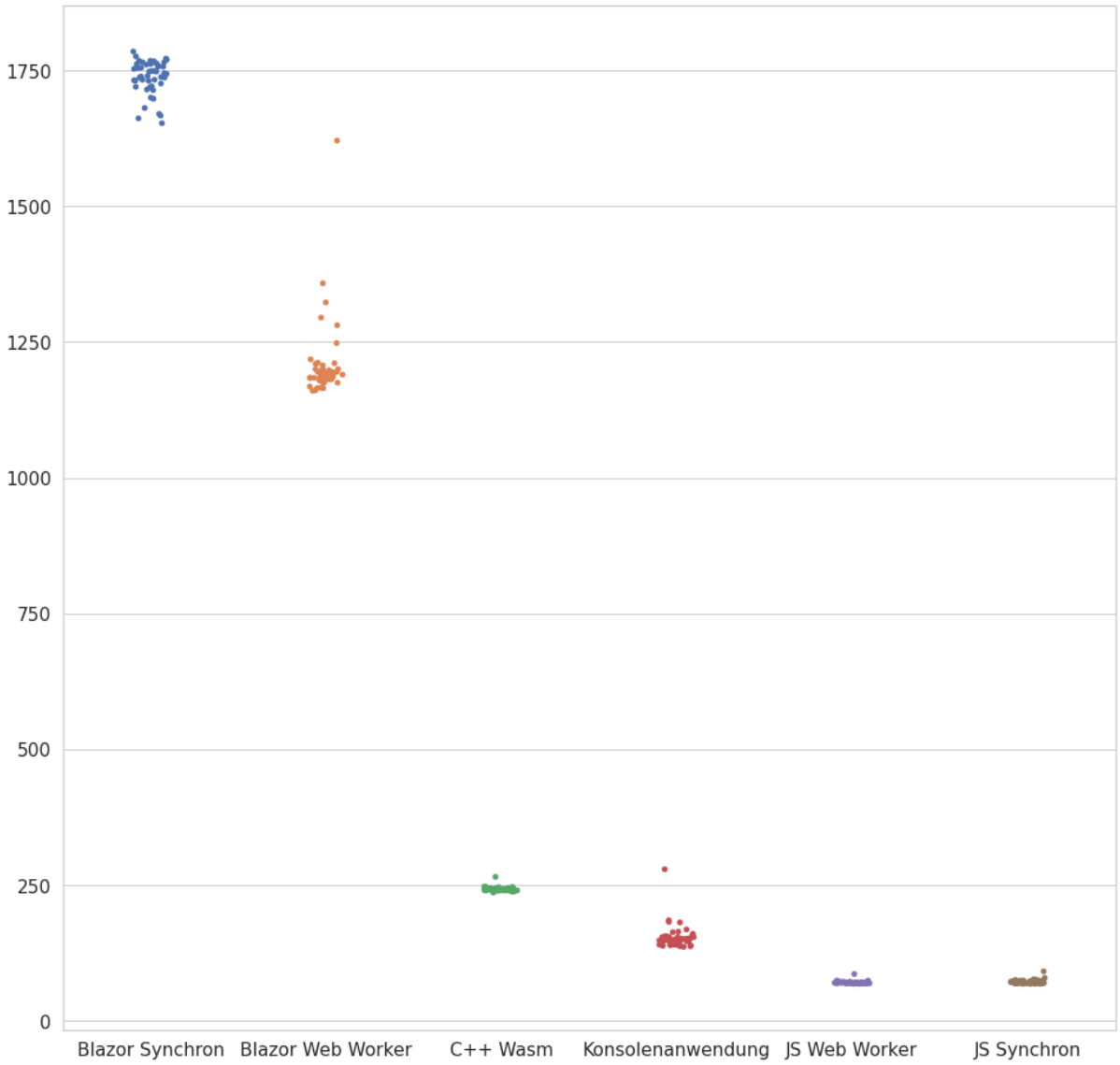

Abbildung 12: Resultat Performance Test

In der Grafik ist zu sehen, dass die durchgeführten 50 Berechnungen pro Variante immer etwa gleich lang dauerten und es kaum zu Ausreissern kam.

Tabelle 3: Ergebnisse Performance Test

Variante	Medianwert in Millisekunden
JavaScript Synchron	70
JavaScript mit Web Worker	70
Konsolenanwendung	137
C++ Wasm	242
Blazor mit Web Worker	1200
Blazor Synchron	1746

6.3.1 Interpretation

Wie in der oben dargestellten Grafik deutlich ersichtlich ist, gibt es einen grossen Performance Unterschied zwischen den verschiedenen Testvarianten. Verglichen mit der .NET Konsolenanwendung, brauchte die Berechnung in Blazor ohne Web Worker etwa **12x länger** und mit dem Einsatz von Web Worker ungefähr **8x länger**.
Überraschenderweise dauerte eine Berechnung in reinem JavaScript weniger lang als in der C# Konsolenanwendung.

Varianten mit Blazor Framework
Eine mögliche Erklärung für dieses Resultat ist der Umstand, dass C# in Blazor nicht direkt, sondern über den in WebAssembly geschriebenden IL-Interpreter ausgeführt wird. Dies führt offensichtlich zu einer erheblichen Performance-Einbusse.

Variente C++ zu WebAssembly
Die Geschwindigkeit des direkt ausgeführten WebAssembly-Bytecode ist sehr nahe am Wert aus der reinen Konsolenanwendung.Das Versprechen der WebAssembly-Entwickler, welche “near native speed” versprechen, kann eingehalten werden.

7 Fazit

In dieser Arbeit wurden Konzepte erarbeitet, wie "native C# in HTML5" integriert werden kann. Dabei wurde in erster Linie verfolgt, wie C# Methoden in einer, mit dem React Framework implementierten, Applikation ausgeführt werden können.

Anhand der vorgestellten Lösungen ist ersichtlich, dass die Möglichkeit besteht, bestehende C# Codebasen in ein React Applikation zu integrieren. Allerdings gibt es Einschränkungen. Zum einen müssen die C# Methoden mit einer bestimmten Signatur definiert werden. Dies führt dazu, dass die Vorteile der objektorientierten Programmierung nicht mehr ausgenützt werden können. Zum anderen werden bei der Programmierung mit JavaScript üblicherweise Web Workers für rechenintensiven Code verwendet. Dabei ist es nicht möglich den C# Code in diese Web Workers auszulagern. Zudem liegen zum Zeitpunkt der Verfassung dieser Arbeit keine offiziellen Lösungen für unser Anliegen vor.

Zusätzlich wird ein Konzept ohne das React Frameworks vorgestellt, in welchem die ganze Applikation mithilfe des Blazor Frameworks implementiert wird. Dieses hat den Vorteil, dass dank den Razor-Pages, die Benutzeroberfläche auch mit C# implementiert werden kann und somit dieselbe Technologie für die Business Logik und die Benutzeroberfläche verwendet werden kann. Zudem ist es dank dem BlazorWorker NuGet Paket möglich, rechenintensiver Code in Web Workers auszulagern. Die Popularität des Blazor Frameworks für die Entwicklung von Benutzeroberflächen, lässt sich von den Schreibern dieser Arbeit nicht eindeutig einordnen, auch da in den letzten Jahren viele andere Frameworks verfügbar wurden und sich etabliert haben.

Obwohl es möglich ist, C# Methoden in JavaScript auszuführen, raten wir davon ab, die in dieser Arbeit vorgestellten Ergebnisse in der Praxis zu verwenden. Der Grund dafür ist, dass diese Konzepte keine offiziellen Lösungen sind und die Möglichkeit besteht, dass unbekannte Probleme auftreten, die im Rahmen dieser Arbeit nicht aufgedeckt worden sind. Allerdings scheint das Verlangen, bestehender C# Codebasen im React Framework zu integrieren, gross zu sein. Aus diesem Grund wird empfohlen, abzuwarten und künftige .NET und React Versionen für die Implementation zu verwenden. Falls die Implementierung der Benutzeroberfläche nicht mehr länger hinausgezögert werden kann, wird empfohlen, dass Blazor Framework zu verwenden. Zusätzlich sollte in Betracht gezogen werden, ob HTML5 erforderlich ist, oder auch andere Technologien wie zum Beispiel Uno Platform oder .NET MAUI verwendet werden können.

In Zukunft erwarten wir viel Entwicklung in diesem Bereich. Zumal mit .NET 6 bereits Schritte in Richtung Integration mit dem React Framework getätigt wurden. Aber auch, weil es von Seiten des World Wide Web Consortium (W3C) Pläne geben soll, dass WebAssembly Multithreading und Garbage Collection unterstützen wird, womit C# Code direkt zu WebAssembly kompiliert werden könnte. Dies würde die Türe für viele verschiedene Ansätze für die Verwendung von C# in HTML5 öffnen.

8 Glossar

Begriff	Beschreibung
.NET	Eine kostenlose und open-source Plattform für die Entwicklung von verschieden Arten von Applikationen. Diese wird von der .NET Foundation entwickelt und unterhalten, welche mehrheitlich aus Microsoft-Mitarbeitern besteht (Foundation & Contributors., 2021).
Angular	Single Page Application Framework, dass hauptsächlich von Google entwickelt und unterhalten wird.
ASP.NET	Framework für die Erstellung von server-seitigen Web Applikationen. Es ist ein Feature der .NET Plattform (Microsoft, 2021a).
Blazor	Ein Framework das C# verwendet um Benutzeroberflächen zu entwickeln. Ist ein Feature von .NET (Microsoft, 2021b).
Compiler	Erstellt aus dem Source-Code, Bytecode, der auf einem Computer ausgeführt werden kann.
DLL (Dynamic Link Libraries)	Sind Bibliotheken die dynamisch in ein Programm eingebunden werden können. Dadurch wird verhindert, das beim Start einer Applikationen alle Bibliotheken geladen werden und vor allem auch das keine unnötigen Bibliotheken eingebunden werden (Rugerio, Chen, Xu & Liang, 2021).
Domain Object Model	Eine Schnittstelle, die ein HTML-Dokument als eine Baumstruktur behandelt. Es wird hauptsächlich verwendet, um dynamisch mit den HTML Elementen interagieren zu können (MDN-Contributors, 2021b).
Framework	Stellen eine Grundstruktur bereit. In der Software Programmierung werden Frameworks oft verwendet, um die Arbeit zu erleichtern.
HTML	HyperText Markup Language ist die Standard Markup Sprache für Webseiten (w3schools, o. J.).
Hash	Eine Hash-Funktion ist eine Funktion, die für unterschiedlichen Input niemals denselben Output ergeben sollte. Diese werden vor allem im Bereich der Cyber Security verwendet.
Intermediate Language (IL)	Eine Zwischensprache die von .NET verwendet wird. Dadurch kann Code in unterschiedlichen, von .NET unterstütze, Sprachen geschrieben werden und auf der selber Runtime ausgeführt werden.
JSInterop	So wird die Interoperabilität von C# und JavaScript im .NET Kontext genannt (Latham, Parikh, Berry & Roth, 2021).

Nebenläufig	Gleichzeitig oder verzahnt ausführbare Abläufe, welche auf gemeinsame Ressourcen zugreifen.
NuGet	Der Package Manager von .NET.
Razor Syntax	Eine für das Blazor Framework erstellte Syntax, dank der im selben File C#- und HTML-Code geschrieben werden kann (Anderson, Pine, Latham & Addie, 2021).
React	Ein Framework das JavaScript verwendet, um Benutzeroberflächen zu entwickeln. Wird von Facebook entwickelt und unterhalten.
Single-Page Applikation	Eine Web Applikation bei der grundsätzlich immer die gleiche Seite dargestellt wird. Dies unterscheidet sich von herkömlichen Web Applikationen darin, dass nicht ganze Seiten vom Server angefragt werden müssen (Lawson, o. J.).
Web Worker	Werden verwendet um Scripts in einem Hintergrund Thread auszuführen (MDN-Contributors, 2021d).

9 Verzeichnisse

9.1 Abbildungsverzeichnis

9.2 Tabellenverzeichnis

9.3 Listing

9.4 Literaturverzeichnis

Abuhakmeh, K. (2020). *Basics of Razor Pages.* https://www.jetbrains.com/dotnet/guide/tutorials/basics/razor-pages. (Abgerufen am 16. Dezember 2021)

Anderson, R., Brock, D. & Larkin, K. (2021). *Introduction to Razor Pages in ASP.NET Core.* https://docs.microsoft.com/en-us/aspnet/core/razor-pages/?view=aspnetcore-6.0&tabs=visual-studio. (Abgerufen am 21. Dezember 2021)

Anderson, R., Pine, D., Latham, L. & Addie, S. (2021). *Razor syntax reference for ASP.NET Core.* https://docs.microsoft.com/en-us/aspnet/core/mvc/views/razor?view=aspnetcore-6.0. (Abgerufen am 21. Dezember 2021)

Archibald, J. (2018). *In the loop - jsconf.asia.* https://www.youtube.com/watch?v=cCOL7MC4Pl0. (Abgerufen am 10. November 2021)

Brickett, N. (2020). *The JavaScript Event Loop Explained.* https://medium.com/swlh/the-javascript-event-loop-explained-98a59c96b285. (Abgerufen am 23. November 2021)

Collins, B. (2020). *Blazor Internals you need to know.* https://blazorguy.net/how-blazor-webassembly-works/. (Abgerufen am 05. November 2021)

de Icaza, M. (2018). *mono-wasm.* https://github.com/migueldeicaza/mono-wasm. (Abgerufen am 04. November 2021)

Emscripten-Contributors. (2021). *Emscripten.* https://emscripten.org/. (Abgerufen am 04. November 2021)

Foundation, N. & Contributors. (2021). *.NET Foundation Homepage.* https://dotnetfoundation.org/. (Abgerufen am 21. Dezember 2021)

Latham, L., Anderson, R., Addie, S., K, P. & Sanderson, S. (2021). *Host and deploy ASP.NET Core Blazor WebAssembly.* https://docs.microsoft.com/en-us/aspnet/core/blazor/host-and-deploy/webassembly?view=aspnetcore-5.0#resolve-integrity-check-failures-1. (Abgerufen am 04. November 2021)

Latham, L. & olprod. (2021). *ASP.NET Core Blazor hosting models.* https://docs.microsoft.com/de-de/aspnet/core/blazor/?view=aspnetcore-5.0#blazor-webassembly-1. (Abgerufen am 04. November 2021)

Latham, L., Parikh, T., Berry, D. & Roth, D. (2021). *Call JavaScript functions from .NET methods in ASP.NET Core Blazor.* https://docs.microsoft.com/en-us/aspnet/core/blazor/javascript-interoperability/call-javascript-from-dotnet?view=aspnetcore-5.0. (Abgerufen am 21. Dezember 2021)

Lawson, K. (o. J.). *What Is a Single Page Application and Why Do People Llike Them so Much?* https://www.bloomreach.com/en/blog/2018/07/what-is-a-single-page-application.html. (Abgerufen am 16. Dezember 2021)

MDN-Contributors. (2021a). *How do I use WebAssembly in my app?* https://developer.mozilla.org/en-US/docs/WebAssembly/Concepts. (Abgerufen am 16. Dezember 2021)

MDN-Contributors. (2021b). *Introduction to the DOM.* https://developer.mozilla.org/en-US/docs/Web/API/Document_Object_Model/Introduction. (Abgerufen am 21. Dezember 2021)

MDN-Contributors. (2021c). *Using the WebAssembly JavaScript API.* https://developer.mozilla.org/en-US/docs/WebAssembly/Using_the_JavaScript_API. (Abgerufen am 04. November 2021)

MDN-Contributors. (2021d). *Using Web Workers.* https://developer.mozilla.org/en-US/docs/Web/API/Web_Workers_API/Using_web_workers. (Abgerufen am 17. Dezember 2021)

Microsoft. (2021a). *ASP.NET.* https://dotnet.microsoft.com/en-us/apps/aspnet. (Abgerufen am 21. Dezember 2021)

Microsoft. (2021b). *Blazor.* https://dotnet.microsoft.com/apps/aspnet/web-apps/blazor. (Abgerufen am 04. November 2021)

Mono-Project. (2021). *Mono.* https://www.mono-project.com/. (Abgerufen am 04. November 2021)

Pine, D., Warren, G., Dykstra, T., Wagner, B., Turn, N., Sherer, T., ... Vorlicek, J. (2021). *Assemblies in .NET.* https://docs.microsoft.com/en-us/dotnet/standard/assembly. (Abgerufen am 16. Dezember 2021)

Roth, D. (2021a). *ASP.NET Core updates in .NET 6 Release Candidate 1.* https://devblogs.microsoft.com/dotnet/asp-net-core-updates-in-net-6-rc-1/. (Abgerufen am 04. November 2021)

Roth, D. (2021b). *JavaScript Component Generation.* https://github.com/aspnet/samples/tree/main/samples/aspnetcore/blazor/JSComponentGeneration. (Abgerufen am 04. November 2021)

Rugerio, D., Chen, L., Xu, S. & Liang, H. (2021). *What is a DLL.* https://docs.microsoft.com/en-us/troubleshoot/windows-client/deployment/dynamic-link-library. (Abgerufen am 21. Dezember 2021)

samanime. (2021). *Call c# method in javascript via webassembly.* https://stackoverflow.com/questions/69711043/call-c-sharp-method-in-javascript-via-webassembly/69711174#69711174. (Abgerufen am 25. Oktober 2021)

Sansonetti, L. (2018). *Mono and WebAssembly - Updates on Static Compilation.* https://www.mono-project.com/news/2018/01/16/mono-static-webassembly-compilation. (Abgerufen am 04. November 2021)

Semeteys, R., Pilot, O., Baudrillard, L., Lebouder, G. & Pinkhardt, W. (2006). *Method for Qualification and Selection of Open Source software (QSOS).* http://http://www.qsos.org/.

Smacchia, P. (2021). *Blazor Internals you need to know.* https://blog.ndepend.com/blazor-internals-you-need-to-know/#:~:text=wasm.-,dotnet.,it%20within%20the%20runtime%20boundaries. (Abgerufen am 05. November 2021)

Smith, B. (2018). *Garbage collection.* https://github.com/WebAssembly/proposals/issues/16. (Abgerufen am 04. November 2021)

unoplatform. (2021). *Uno.Wasm.Bootstrap.* https://github.com/unoplatform/Uno.Wasm.Bootstrap. (Abgerufen am 04. November 2021)

W3Schools. (o. J.). *HTML History.* https://www.w3schools.in/html-tutorial/history. (Abgerufen am 18. Dezember 2021)

w3schools. (o. J.). *HTML Tutorial.* https://www.w3schools.com/html/. (Abgerufen am 16. Dezember 2021)

Zalewski, M. (2011). *The tangled web* (. Auflage, Hrsg.). No Starch Press, US.

10 Anhang

10.1 Aufgabenstellung

10.1.1 Problembeschreibung

Grosse Teile des Programmcodes grosser Firmen sind in Technologien geschrieben, die nicht unmittelbar mit modernen Webtechnologien kompatibel sind. Somit ergeben sich Herausforderungen, wenn neue Anforderungen adressiert aber bestehende Codebasen weiterverwendet werden sollen. Ein Beispiel stellt die Verwendung von C# .NET-Code in React-Anwendungen dar. Es stellt sich deshalb die Herausforderung, wie sich diese Technologien in moderne Frameworks übertragen lassen.

Als beispielhafte Anwendung soll ein Durchflussmessgerät betrachtet werden. Mit Durchflussmessgeräten von Endress + Hauser AG kann die Konzentration von Stoffen in Flüssigkeiten bestimmt werden. Für eine genaue Messung sind Parameter notwendig, die mit einer bestehenden Windows-Anwendung bestimmt werden können.

10.1.2 Konkreter Auftrag

Ziel der Arbeit ist die Erarbeitung und Erprobung von Konzepten um eine bestehende C# .NET-Codebasis in eine HTML5-Umgebung zu übertragen.

Es sollen folgende Teilschritte ausgeführt werden:

1. **Recherche** von Ansätzen um C#-Code in Webanwendungen zu integrieren (z.B. WebAssembly, Blazor). Vergleich der Technologien anhand eines Kriterienkatalogs. Es werden Technologien mit permissive-Licenses (z.B. BSD, MIT) bevorzugt betrachtet.
2. **Analyse** einer Bestandsanwendung, beziehungsweise einer Vergleichscodebasis mit ähnlicher Komplexität. Falls Bestandscode von Endress + Hauser AG verwendet wird ist ein NDA notwendig.
3. **Migration** des Codes in eine C#-Webassembly/Blazor Anwendung. Bestandteil ist auch die Migration bzw. Entwicklung einer Benutzungsoberfläche um einen vollständigen Durchstich testen zu können.
4. **Bewertung** der umgesetzten Anwendung. Es werden Vergleichstests (z.B. Performance) mit der bestehenden Implementierung in C# .NET durchgeführt. Ebenfalls wird die Komplexität der Implementierung kritisch diskutiert.
5. Formulierung von **Empfehlungen** zur Migration bestehender Codebasen in HTML5- Umgebungen.

10.1.3 Umfang und Form der erwarteten Resultate

Die Ergebnisse der Arbeit (schriftlicher Projektbericht, Code, Dokumentation) werden den Projektbeteiligten zur Verfügung gestellt.

10.1.4 Zulässige Hilfsmittel und weitere Betreuung

Alle verwendeten Hilfsmittel werden in der Arbeit aufgeführt. Die Betreuung erfolgt primär durch die genannte Betreuungsperson an der Hochschule. Mit dem Partner Endress + Hauser AG werden Webkonferenzen zur Klärung allfälliger Fragen, mindestens im 14-tägigen Rhythmus, durchgeführt.

10.2 Quellcode

Variante 2 Client-Side Blazor Framework
https://gitlab.ost.ch/sa_native_csharp_in_html5/prototype/wasmblazor

Variante 3 - Blazor Framework in Kombination mit React
https://gitlab.ost.ch/sa_native_csharp_in_html5/prototype/blazor_webassembly

Variante 4 - .NET 6 Blazor Komponenten für JavaScript Frameworks
https://gitlab.ost.ch/sa_native_csharp_in_html5/demo/dotnet6

10.3 Projektplan

10.3.1 Projektorganisation

Das Projekt "Native C# in HTML5" wird im Rahmen der Studienarbeit des Studiengangs Informatik an der Fachhochschule OST durchgeführt.

10.3.2 Organisationsstruktur

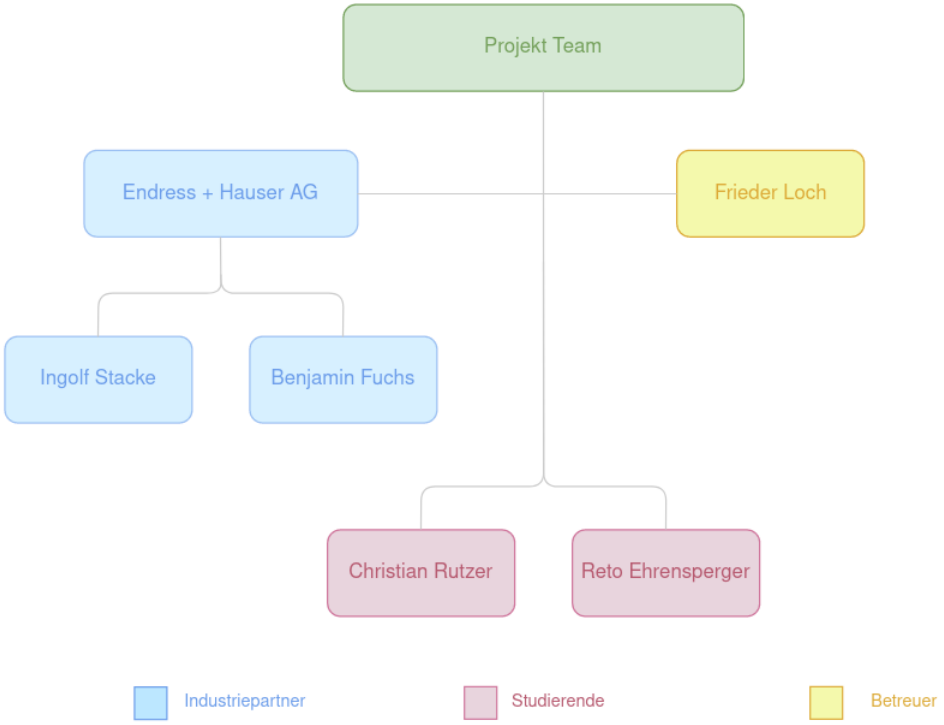

Abbildung 13: Organisationsstruktur

10.3.3 Zeitliche Planung

Das Projekt wird im Herbst 2021 vom 20. September bis 24. Dezember durchgeführt. Für das 8 ECTS-Modul sollen etwa 240h (30h pro ECTS) aufgewendet werden, womit pro Person ca.

17.2h Aufwand pro Woche gefordert sind.

10.3.4 Besprechungen

Die Besprechungen mit dem Betreuer (Frieder Loch), und wenn nötig mit dem Industriepartner (Endress + Hauser AG) finden jeweils am Dienstag um 10:00 Uhr statt und werden über Microsoft Teams gehalten.

10.3.5 Projekt Management

Das Team strebt einen Mix aus Retional Unified Process (RUP) und Kanban an. Die vier Phasen von RUP (Inception, Ellaboration, Construction und Transition) werden jeweils in verschiedene Arbeitspakete aufgeteilt, welche im Gitlab-Projekt als Issues erfasst und dem zugehörigen Meilenstein zugewiesen werden.

10.3.6 Meilensteine Zeitplan

10.3.7 Timeline

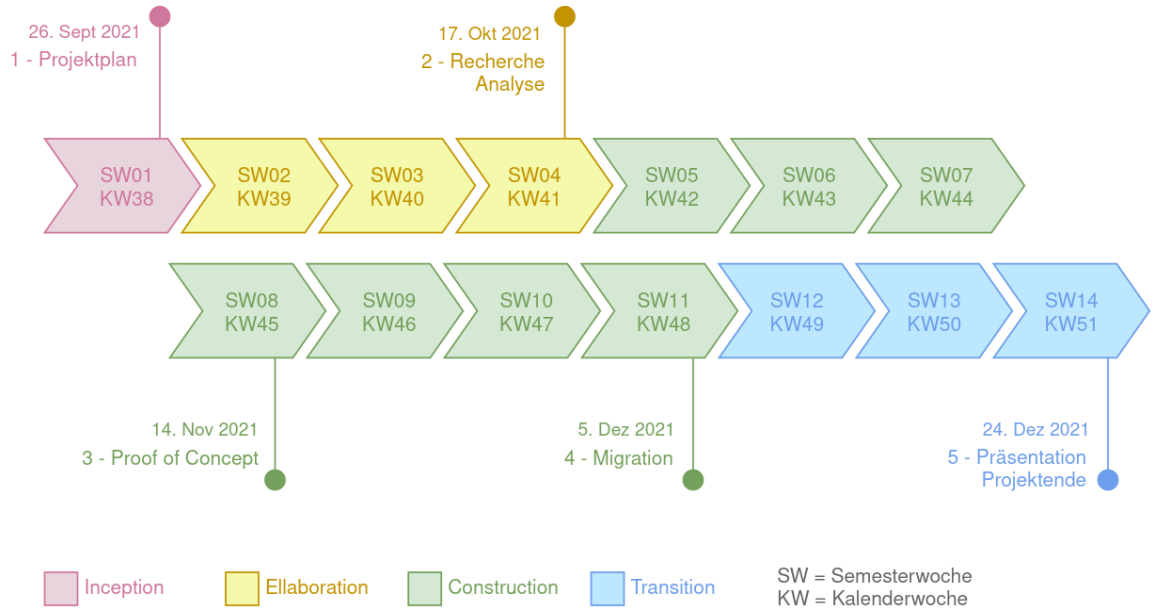

Abbildung 14: Projektplan

10.3.8 Meilenstein Details

Tabelle 6: Meilensteine Details

Nr.	Meilenstein	Datum	Produkte
1	Projektplan	26.9.2021	• Projektorganisation definiert • Meilensteine definiert • Risikoanalyse • Regeltermin mit Industriepartner planen
2	Recherche / Analyse	17.10.2021	• Kriterienkatalog erstellen • Varianten mit Kriterienkatalog abgleichen • Recherche von Emscripten • Rercherche des Blazor Frameworks • Recherche von .NET 6 • Recherche von ReactJS.NET • Analyse des Blazor Frameworks • Analyse eines React Projekts mit Integration des Blazor Frameworks • Analyse von .NET 6
3	Prototype	14.11.2021	• Umsetzung der C# Codebasis • Implementierung des Frontends mit React • Implementierung des Frontends mit Blazor
4	Migration	5.12.2021	• Async Problem weiter untersuchen • Verwendung von Web Worker implementieren • Verwendung von BlazorWorker implementieren • Besuch bei Endress + Hauser AG planen • Performance Tests erstellen • Performance Tests auswerten
5	Projektende	24.12.2021	• Schlussbericht erstellen • Abstract schreiben • Plakat erstellen

10.3.9 Qualitätsmassnahmen

Tabelle 7: Qualitätsmassnahmen

Massnahme	**Zeitraum**	**Ziel**
Sitzung mit Betreuer und Industriepartnern	Wöchentlich	Austausch über Projektstand, Identifikation möglicher Abweichungen
Meilensteine	Fixer Tag	Sicherstellen, dass das Projekt uf dem richtigen Weg bleibt

10.3.10 Zeitrapport

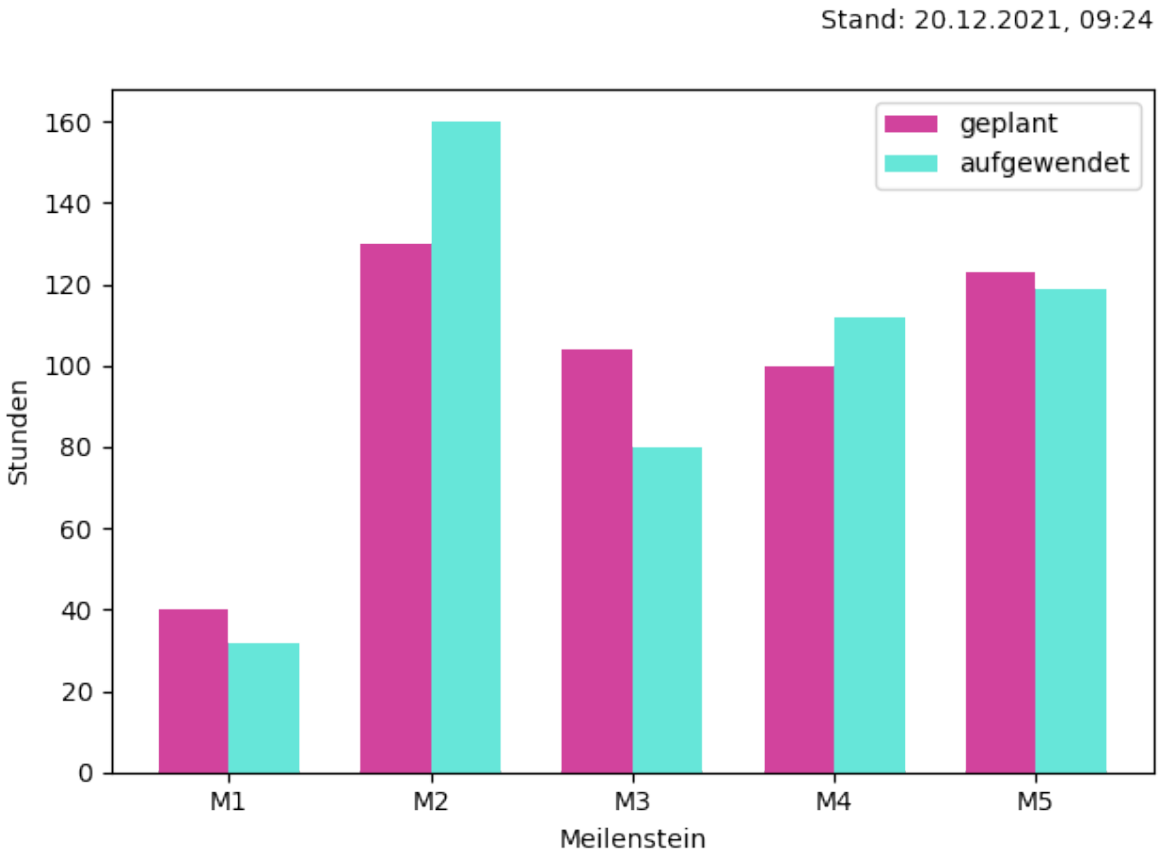

Abbildung 15: Zeitrapport - Zeit geplant / aufgewendet pro Meilenstein

- geplant: ≈ 480 Stunden
- aufgewendet: ≈ 503 Stunden

10.4 Risikomanagement

Um das Schadenspotential und die Eintrittswahrscheinlichkeit verschiedener Risiken einzuschätzen wird am Anfang und in der Mitte des Projekts eine Risikoanalyse durchgeführt. Je nach Einschätzung sind folgende Massnahmen notwendig:

- akzeptabel: keine Aktion notwendig
- ALARP (as low as reasonably practicable): Risiko muss laufend beobachtet werden
- inakzeptabel: Risiko muss durch Gegenmassnahmen verkleinert werden

Was passiert, wenn ein Risiko eintrifft?
Sollte ein Risiko eintreffen, welches den weiteren Verlauf der Semesterarbeit stark beeinflusst, wird schnellstmöglich in Absprache mit dem Betreuer / Industriepartner das weitere Vorgehen und mögliche Lösungen bestimmt.

10.4.1 Risiken Stand 27.09.2021

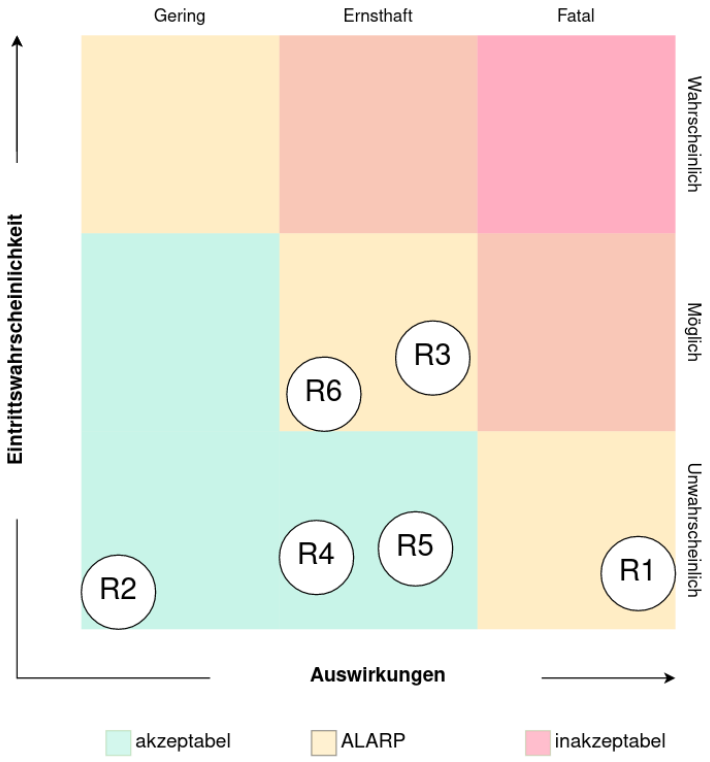

Abbildung 16: Risikomanagement

10.4.2 Beschreibungen

Tabelle 8: Risiken im Detail

Beschreibung	**Eintrittswahrscheinlichkeit** (unwahrscheinlich, möglich, wahrscheinlich)	**Auswirkungen** (gering, ernsthaft, fatal)	**Gegenmassnahmen**
(R1) Rechtliche Risiken, z.B. nicht einhalten des NDA	unwahrscheinlich	fatal	GitLab-Repositories müssen privat sein.
(R2) Anforderungsänderungen des Auftraggebers	unwahrscheinlich	gering	Es werden wöchentliche Sitzungen mit dem Betreuer durchgeführt.
(R3) Annahmen nicht zutreffend, z.B. unzutreffende Recherche / Analyse	möglich	ernsthaft	Die Recherchen und Analyse werden am Anfang des Projektes gemacht und jeweils wöchentlich mit dem Betreuer besprochen.
(R4) Nicht realisierbar	unwahrscheinlich	ernsthaft	Am Anfang des Projektes werden Recherchen und Analysen gemacht, wobei die Realisierbarkeit des Projektes sichergestellt wird.
(R5) Zu grosse Abweichungen gegenüber den Anforderungen	unwahrscheinlich	ernsthaft	Es werden wöchtentlich Sitzungen mit dem Betreuer durchgeführt. Zudem wird bei Halbzeit des Projektes den aktuellen Stand präsentiert.
(R6) Zeitplan kann nicht eingehlaten werden	möglich	ernsthaft	Es wurden Meilensteine definiert um sicherzustellen, dass die Zeit eingehalten wird.

10.4.3 Risiken Stand 01.11.2021

Am Montag, 1. November 2021 wurden die vom Projektstart ermittelten Risiken neu bewertet. Bis auf Risiko Nr. 4 - “Nicht realisierbar”, wurden alle Risiken in einem ähnlichen Rahmen bewertet.
Da eine Lösung gefunden wurde, um C# Code aus einer JavaScript Anwendung auszuführen, konnte Risiko 4 gestrichen werden.

10.5 Persönliche Berichte

Reto

Die Studienarbeit war für mich eine interessante Erfahrung. Es war die erste Arbeit die ich geschrieben habe, bei der verschiedene Konzepte ausprobiert und miteinander verglichen werden mussten. Für mich war eine wichtige persönliche Erkenntnis, dass es zu einem Problem nicht immer eine Lösung geben muss und man ein Problem manchmal von unterschiedlichen Blickwinkel betrachten soll, um dieses richtig einzuordnen. Die Zusammenarbeit mit den Betreuern und meinem Kommilitonen hat mir grossen Spass gemacht und mich angespornt um (hoffentlich) eine gute Arbeit zu schreiben.

Christian

Bewusst haben Reto und ich uns auf ein Thema mit Industriepartner beworben und ich kann nach Abschluss der Studienarbeit bestätigen, dass dies eine gute Entscheidung war. Unsere Aufgabe war eine real existierende Herausforderung, für welche wir im Auftrag der Endress + Hauser AG ein Ergebnis gesucht haben. Durch diesen Praxisbezug konnten wir unsere Motivation trotz einiger Rückschläge stets hochhalten.
Da während der Suche nach einer Lösung ständig neue, unbekannte Probleme zum Vorschein kommen, mussten wir feststellen, dass es nicht immer Sinn macht, stur dem vorhandenen Projektplan zu folgen. Diese Arbeit war eine wertvolle Erfahrung, um für mich bis anhin unbekannte Technologien kennenzulernen.

10.6 Danksagung

An dieser Stelle möchten wir uns bei Prof. Dr.-Ing. Frieder Loch, der unsere Studienarbeit betreut und begutachtet hat, für die hilfreichen Anregungen und stets angenehme Zusammenarbeit bedanken.
Unser Dank geht ebenfalls an Benjamin Fuchs und Ingolf Stacke als Vertreter der Endress + Hauser AG, für die aufgewendete Zeit in den regelmässigen Besprechungen und für den Tag vor Ort in Rheinach.

10.7 Eigenständigkeitserklärung

Ich erkläre hiermit,

- dass ich die vorliegende Arbeit selber und ohne fremde Hilfe durchgeführt habe, ausser derjenigen, welche explizit in der Aufgabenstellung erwähnt ist oder mit dem Betreuer schriftlich vereinbart wurde,
- dass ich sämtliche verwendeten Quellen erwähnt und gemäss gängigen wissenschaftlichen Zitierregeln korrekt angegeben habe.
- dass ich keine durch Copyright geschützten Materialien (z.B. Bilder) in dieser Arbeit in unerlaubter Weise genutzt habe.

Rapperswil, 23.12.2021 Reto Ehrensperger

Rapperswil, 23.12.2021 Christian Rutzer