Projektdokumentation

Inhaltsverzeichnis

[Projektplan - 2 -](#_Toc59372414)

[Projektauftrag - 2 -](#_Toc59372415)

[Auftraggeber - 2 -](#_Toc59372416)

[Aufgabenstellung - 2 -](#_Toc59372417)

[Projektorganigramm - 3 -](#_Toc59372418)

[Projektstrukturplan - 4 -](#_Toc59372419)

[Aufgaben: Projekt Agi’s Eisstockbahnen - 4 -](#_Toc59372420)

[Projektarbeitspaketspezifikation - 5 -](#_Toc59372421)

[Teilaufgabe: Projektdokumentation - 5 -](#_Toc59372422)

[Teilaufgabe: Webseite entwickeln - 6 -](#_Toc59372423)

[Teilaufgabe: Bilder Bearbeitung - 8 -](#_Toc59372424)

[Teilaufgabe: Webseite veröffentlichen - 8 -](#_Toc59372425)

[Projektmeilensteine - 9 -](#_Toc59372426)

[Projektdokumentation - 10 -](#_Toc59372427)

[Visual Studio - 10 -](#_Toc59372428)

[Installation - 10 -](#_Toc59372429)

[Entwicklungsumfeld - 10 -](#_Toc59372430)

[Version Control System - 10 -](#_Toc59372431)

[Github - 10 -](#_Toc59372432)

[Azure Pipelines - 10 -](#_Toc59372433)

[Azure Cloud - 13 -](#_Toc59372434)

[MSSQL Datenbank - 13 -](#_Toc59372435)

[Entwicklung - 14 -](#_Toc59372436)

[Datenbank Anbindung - 14 -](#_Toc59372437)

[Backend - Rest API - 15 -](#_Toc59372438)

[BlazorWASM Frontend - 16 -](#_Toc59372439)

[CSS-Framework - 16 -](#_Toc59372440)

[Verlinkungen - 16 -](#_Toc59372441)

[Landing Page - 17 -](#_Toc59372442)

[Geschichtsseite - 18 -](#_Toc59372443)

[Preislistenseite - 19 -](#_Toc59372444)

[Authentifizierungsseite - 20 -](#_Toc59372445)

[Berechnungsseite - 22 -](#_Toc59372446)

[Kontaktseite - 25 -](#_Toc59372447)

[Bilderbearbeitung - 27 -](#_Toc59372448)

[Verwendetes Programm - 27 -](#_Toc59372449)

[Verwendete Funktionen - 27 -](#_Toc59372450)

[Verwendete Funktionen bei Bild - 28 -](#_Toc59372451)

[Vorher – Nachher Vergleich - 28 -](#_Toc59372452)

# Projektplan

## Projektauftrag

### Auftraggeber

Aga Ratzenböck

Au 9, 4733 Heiligenberg

### Aufgabenstellung

Die Webseite beinhaltet eine Landing Page auf der alle relevanten Informationen zu den Eisstockbahnen wie zum Beispiel.:

• Preise  
• Öffnungszeiten

• Bahnregeln

Auf einer weiteren Seite kann man die Historie der Eisbahnen nachlesen und findet aus den vergangenen Jahren Bildern. Hier wird es mehrere thematisch zusammenpassende Bildergalerien geben damit eine interessierte Person sich einen Überblick über die Eisbahnen machen kann.

Auf der Impressums Seite gibt es alle Kontaktmöglichkeiten sowie auch einen Google Maps Link zur vereinfachten Anfahrt zu den Eisbahnen.

Sowohl der Header als auch der Footer werden zur Navigation der Webseite verwendet.

Der Footer wird zusätzlich mit einer rudimentären Art eines Impressums bestückt das diese Information immer vorhanden ist und schnellst möglich vom Interessenten gefunden wird.

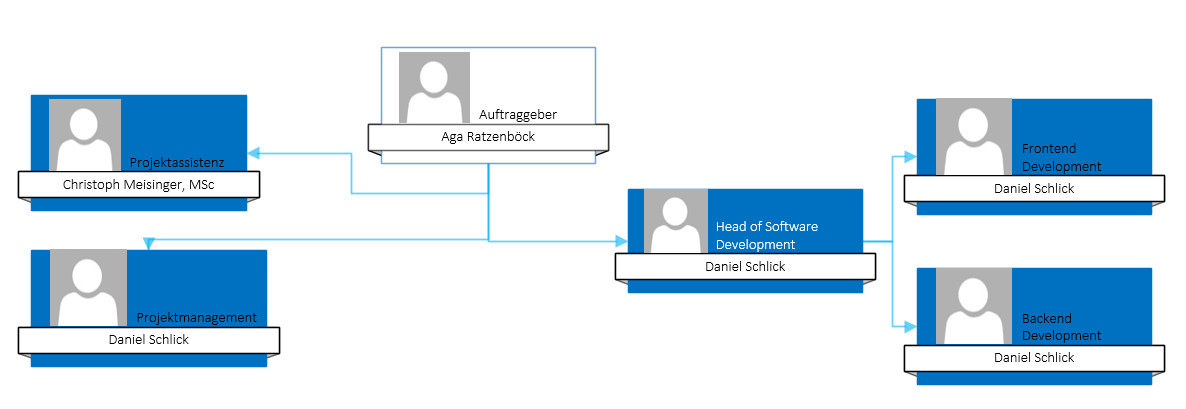
Des Weiteren gibt es eine Seite auf der man als Webseiten Inhaber eine formlose Bilanzrechnung erstellen kann um eine Vereinfachung für den Auftraggeber zu erzielen. Diese werden für das Finanzamt benötigt und stellen somit eine Aufwandsminderung dar.

#### Verwendete Technologien

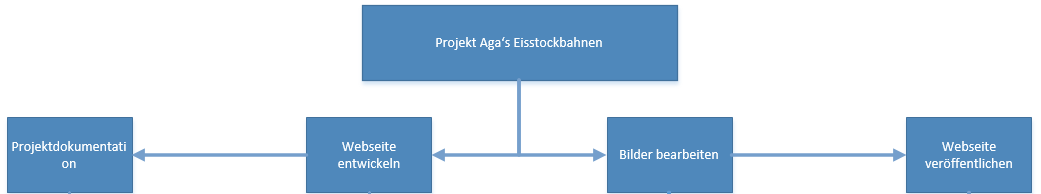
* Blazor WebAssembly
* HTML
* CSS
* .Net Core
* MSSQL

## Projektorganigramm

Für das Projekt „Agi’s Eisstockbahnen“ wurde ein Organigramm ausgearbeitet. In diesem Organigramm wurden die Hierarchien sowie alle gültigen Kommunikationswege im Projekt definiert.



## Projektstrukturplan



### Aufgaben: Projekt Agi’s Eisstockbahnen

1. Teilaufgabe: Projektdokumentation
   1. Inhaltverzeichnis anlegen
   2. Projektmanagement beschreiben
   3. Arbeitspakete definieren
   4. Dokumentation der Entwicklungsumgebung
   5. Dokumentation der Arbeitsschritte
   6. Finalisierung Dokumentation
2. Teilaufgabe: Webseite entwickeln
   1. Entwicklungsumgebung einrichten
   2. Projekt erstellen
   3. Datenbank aufsetzten
   4. Datenbank Model erstellen
   5. ASP NET Core Web API bauen
   6. Startseite erstellen
   7. Authentifizierung einbauen
   8. Kontaktseite erstellen
   9. Preisübersichtsseite erstellen
   10. Berechnungstool Entwickeln
3. Teilaufgabe: Bilder bearbeiten
   1. Installation Gimp
   2. Bilder bearbeiten
   3. Format anpassen
4. Teilaufgabe: Webseite veröffentlichen
   1. Webserver erstellen
   2. Webseite auf Webserver bereitstellen

## Projektarbeitspaketspezifikation

### Teilaufgabe: Projektdokumentation

|  |  |  |
| --- | --- | --- |
| Arbeitspaketbeschreibung | | |
| PSP-Code:  1.1 | AP Bezeichnung:  Inhaltsverzeichnis anlegen | AP Verantwortlichkeit: Daniel Schlick |
| Inhalte: | * Word Dokument erstellen * Projektschritte erarbeiten * Inhaltsverzeichnis erstellen | |
| Ziele: | * Erstellung Projektdokumentation * Definiertes Inhaltsverzeichnis | |

|  |  |  |
| --- | --- | --- |
| Arbeitspaketbeschreibung | | |
| PSP-Code:  1.2 | AP Bezeichnung:  Projektmanagement erstellen | AP Verantwortlichkeit: Daniel Schlick |
| Inhalte: | * Projektauftrag beschreiben * Projektorganigramm erstellen * Projektstrukturplan ausarbeiten * Meilensteine definieren | |
| Ziele: | * Projektmanagement fertiggestellt | |

|  |  |  |
| --- | --- | --- |
| Arbeitspaketbeschreibung | | |
| PSP-Code:  1.3 | AP Bezeichnung:  Arbeitspakete definieren | AP Verantwortlichkeit: Daniel Schlick |
| Inhalte: | * Arbeitspakete aus Projektstrukturplan erstellen * Inhalte und Ziele definieren | |
| Ziele: | * Arbeitspakete fertig erstellt | |

|  |  |  |
| --- | --- | --- |
| Arbeitspaketbeschreibung | | |
| PSP-Code:  1.4 | AP Bezeichnung:  Dokumentation der Entwicklungsumgebung | AP Verantwortlichkeit: Daniel Schlick |
| Inhalte: | * Schritt für Schritt Anleitung zum Aufsetzten der Entwicklungsumgebung | |
| Ziele: | * Dokumentation für die Einrichtung der Entwicklungsumgebung | |

|  |  |  |
| --- | --- | --- |
| Arbeitspaketbeschreibung | | |
| PSP-Code:  1.5 | AP Bezeichnung:  Dokumentation der Arbeitsschritte | AP Verantwortlichkeit: Daniel Schlick |
| Inhalte: | * Detaillierte Dokumentation aller getätigten Arbeitsschritte | |
| Ziele: | * Übersicht aller geleisteten Arbeitsschritte für das Projekt | |

|  |  |  |
| --- | --- | --- |
| Arbeitspaketbeschreibung | | |
| PSP-Code:  1.6 | AP Bezeichnung:  Finalisierung der Dokumentation | AP Verantwortlichkeit: Daniel Schlick |
| Inhalte: | * Korrektur lesen der Projektdokumentation * Finale Formatierung der Projektdokumentation | |
| Ziele: | * Übersichtliche und saubere Projektdokumentation | |

### Teilaufgabe: Webseite entwickeln

|  |  |  |
| --- | --- | --- |
| Arbeitspaketbeschreibung | | |
| PSP-Code:  2.1 | AP Bezeichnung:  Entwicklungsumgebung einrichten | AP Verantwortlichkeit: Daniel Schlick |
| Inhalte: | * Arbeitsgerät aufsetzen * Entwicklungsumgebung installieren | |
| Ziele: | * Vollständige Entwicklungsumgebung | |

|  |  |  |
| --- | --- | --- |
| Arbeitspaketbeschreibung | | |
| PSP-Code:  2.2 | AP Bezeichnung:  Projekt erstellen | AP Verantwortlichkeit: Daniel Schlick |
| Inhalte: | * WebAssembly Frontent Projekt erstellen * Asp Net Core Web API erstellen | |
| Ziele: | * Basic Frontent und API Projekt für Eisstockbahnen | |

|  |  |  |
| --- | --- | --- |
| Arbeitspaketbeschreibung | | |
| PSP-Code:  2.3 | AP Bezeichnung:  Datenbank aufsetzten | AP Verantwortlichkeit: Daniel Schlick |
| Inhalte: | * Erstellung MS SQL Datenbank in der Azure Cloud | |
| Ziele: | * Lauffähige MS SQL Datenbank gehostet in der Azure Cloud | |
| Arbeitspaketbeschreibung | | |
| PSP-Code:  2.4 | AP Bezeichnung:  Datenbank Model erstellen | AP Verantwortlichkeit: Daniel Schlick |
| Inhalte: | * Erstellung der C# Klassen die Datenbank Tabellen repräsentieren * Konfiguration der Entitäten im DatabaseContext | |
| Ziele: | * Fertiges Datenbank Model mittels Entity Framework Core erstellen | |

|  |  |  |
| --- | --- | --- |
| Arbeitspaketbeschreibung | | |
| PSP-Code:  2.5 | AP Bezeichnung:  ASP NET Core Web API bauen | AP Verantwortlichkeit: Daniel Schlick |
| Inhalte: | * Implementierung Service Ebene für die Kommunikation zwischen Datenbank Schicht und API Schicht * CRUD Operationen auf der API Schicht für jede Datenbank Entität | |
| Ziele: | * Fertige ASP NET Core Web API die mit der Datenbank kommunizieren kann und Requests entgegennehmen kann | |

|  |  |  |
| --- | --- | --- |
| Arbeitspaketbeschreibung | | |
| PSP-Code:  2.6 | AP Bezeichnung:  Startseite erstellen | AP Verantwortlichkeit: Daniel Schlick |
| Inhalte: | * Blazor WebAssembly Projekt vollständige Landing Page bauen | |
| Ziele: | * Finalisierung der Landing Page | |

|  |  |  |
| --- | --- | --- |
| Arbeitspaketbeschreibung | | |
| PSP-Code:  2.7 | AP Bezeichnung:  Authentifizierungsseite erstellen | AP Verantwortlichkeit: Daniel Schlick |
| Inhalte: | * Authentifizierung bei Sever abfragen * Wenn Webinhaber, dann Berechnungsseite laden * Wenn nicht Webinhaber, Fehlermeldung anzeigen | |
| Ziele: | * Finalisierung der Authentifizierungsseite | |

|  |  |  |
| --- | --- | --- |
| Arbeitspaketbeschreibung | | |
| PSP-Code:  2.8 | AP Bezeichnung:  Kontaktseite erstellen | AP Verantwortlichkeit: Daniel Schlick |
| Inhalte: | * Kontaktdaten der Auftraggeberin einfügen * Googel Maps Karte mit Adresse der Eisbahnen einfügen | |
| Ziele: | * Fertige gut übersichtliche Kontaktseite | |

|  |  |  |
| --- | --- | --- |
| Arbeitspaketbeschreibung | | |
| PSP-Code:  2.9 | AP Bezeichnung:  Preisübersichtsseite erstellen | AP Verantwortlichkeit: Daniel Schlick |
| Inhalte: | * Erstellen einer Seite die alle Speißen, Getränke und Bahn Geld auflistet | |
| Ziele: | * Fertige Preisübersichtsseite | |

|  |  |  |
| --- | --- | --- |
| Arbeitspaketbeschreibung | | |
| PSP-Code:  2.10 | AP Bezeichnung:  Berechnungstool entwickeln | AP Verantwortlichkeit: Daniel Schlick |
| Inhalte: | * API Abfrage aller Produkte in der Datenbank * Implementierung des dynamischen Aufbaus der Berechnungsseite * C# Algorithmus zur Berechnung der Gewinne * Dynamisches Updaten der Webpage zum Anzeigen des Berechnungsergebnisses | |
| Ziele: | * Fertiges Berechnungstool zum Ermitteln der erzielten Gewinne/Verluste der Eisstockbahnen | |

### Teilaufgabe: Bilder Bearbeitung

|  |  |  |
| --- | --- | --- |
| Arbeitspaketbeschreibung | | |
| PSP-Code:  3.1 | AP Bezeichnung:  Installation GIMP | AP Verantwortlichkeit: Daniel Schlick |
| Inhalte: | * Installation von GIMP auf Entwicklungsmaschine | |
| Ziele: | * Einsatzbereite Version von GIMP zur Bildbearbeitung | |

|  |  |  |
| --- | --- | --- |
| Arbeitspaketbeschreibung | | |
| PSP-Code:  3.2 | AP Bezeichnung:  Bilder bearbeiten | AP Verantwortlichkeit: Daniel Schlick |
| Inhalte: | * Alle verwendeten Bilder bearbeiten sofern nötig | |
| Ziele: | * Passende Bilder für die Webseite erstellen | |

### Teilaufgabe: Webseite veröffentlichen

|  |  |  |
| --- | --- | --- |
| Arbeitspaketbeschreibung | | |
| PSP-Code:  4.1 | AP Bezeichnung:  Webserver erstellen | AP Verantwortlichkeit: Daniel Schlick |
| Inhalte: | * Windows Webserver erstellen in der Azure Cloud | |
| Ziele: | * Laufenden Produktiven Webserver in der Azure Cloud | |

|  |  |  |
| --- | --- | --- |
| Arbeitspaketbeschreibung | | |
| PSP-Code:  4.2 | AP Bezeichnung:  Webseite auf Webserver bereitstellen | AP Verantwortlichkeit: Daniel Schlick |
| Inhalte: | * Windows Webserver erstellen in der Azure Cloud | |
| Ziele: | * Laufenden Produktiven Webserver in der Azure Cloud | |

## Projektmeilensteine

|  |  |  |  |
| --- | --- | --- | --- |
| Projektmeilensteine | | | |
| PSP Code | **Meilenstein** | **Geplanter Termin** | **Ist Termin** |
| - | Projektstart | 02.11.2020 | 02.11.2020 |
| 1.1, 1.2, 1.3 | Projektdokumentationsstart | 08.11.2020 | 08.11.2020 |
| 1.4, 2.1 - 2.3 | Projektaufsetzung fertigstellen | 14.11.2020 | 14.11.2020 |
| 2.4, 2.5 | Web API fertigstellen | 02.12.2020 | 02.12.2020 |
| 2.6 – 2.9 | Blazor WebAssembly Frontend fertigstellen | 16.12.2020 |  |
| 3.1, 3.2 | Bilder Bearbeitung fertigstellen | 19.12.2020 |  |
| 4.1, 4.2 | Webseite Produktiv stellen | 22.12.2020 |  |
| - | Projektabschluss | 08.02.2020 | 08.02.2020 |

# Projektdokumentation

## Visual Studio

Für das Projekt wurde als Entwicklungsumgeben Visual Studio 2019 in der Community Edition ausgewählt, da diese Version des Visual Studio kostenlos ist, solange man nur nicht kommerzielle Projekt damit umsetzt. Ebenfalls wird für eine Blazor WebAssembly Webanwendung auch die .Net Core Runtime benötigt, welche automatisch bei der Installation von Visual Studio mit installiert wird.

Ein weiterer Punkt warum Visual Studio als Entwicklungsumgebung ausgewählt wurde ist, da in diesem Projekt das Version Control System GIT ist und jenes nativ von Visual Studio nach der Installation unterstützt wird.

### Installation

Bei der Installation ist es wichtig die Komponente „ASP.NET und Webentwicklung“ auszuwählen. Durch diese wird das neueste .Net Core Framework mit installiert welches wir für Blazor WebAssembly benötigen. Nach diesem Schritt muss nichts mehr bei der Installation beachtet werden und man kann sich durch den Installer klicke bis alles fertig installiert wurde.

## Entwicklungsumfeld

### Version Control System

In diesem Projekt wurde als Version Control System GIT verwendet. Der Vorteil von diesem System besteht darin das jede Änderung in logische Blöcke zusammengefasst werden kann. Solche Blöcke nennt man Commits. Diese Commits kann man sich mit Software wie zum Beispiel Github Desktop veranschaulichen. Dadurch besteht die Möglichkeit auf jeden logischen Block von vorher zurück zu setzten und somit von einem sauberen früheren Stand wieder weiter zu Arbeiten. Dies ist umso hilfreicher sollte man mit mehreren Personen an einem Projekt arbeiten.

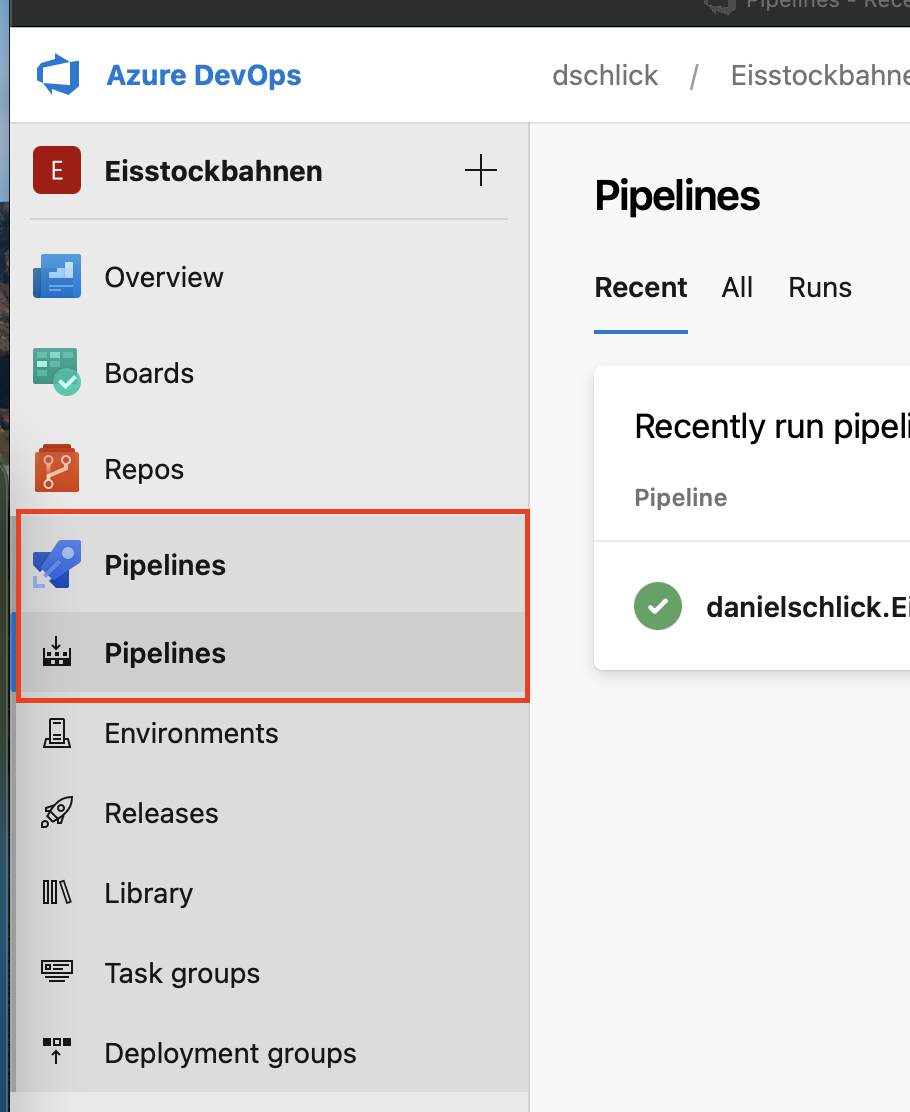
### Github

Bei diesem Projekt wird Github als Version Control Provider verwendet. Einen Version Control Provider verwende ich aus dem Grund das somit mein Projekt gesichert auf einem Server gespeichert ist. So kann im Fall eines technischen versagen das ganze Projekt auf einem anderen System weiter Entwickelt werden.

Github ist seit Jahren mit Marktführer und gehört dem Microsoft Konzern an. Dies vereinfacht die Einbindung in Azure Pipelines.

### Azure Pipelines

Azure Pipelines dienen hier als Rückgrat für den Continuous Integration/Continuous Deployment Prozess. Auf der Seite „<https://dev.azure.com/>“ kann man mit Hilfe eines Microsoft oder Github Kontos sich einloggen und ein Projekt erstellen. Bei diesem Projekt verwenden wir nur den Punkt „Pipelines“.



Mithilfe dieser Pipelines können wir nach jedem Check In in unser Github Repository automatisiert einen Build Vorgang anstoßen. Der Ablauf dieses Build Vorganges wird in sogenannten YAML (Yet Another Markup Language) beschrieben und liegt ebenfalls im Github Repository so das wir auch hier einen Versionsverlauf haben. In diesem Projekt habe ich folgendes YAML geschrieben.

trigger:

branches:

include:

- main

paths:

exclude:

- azure-pipelines.yml

# Erstellen von globalen Variablen

variables:

buildConfiguration: 'Release'

vmImageName: 'ubuntu-latest'

stages:

- stage: Build

displayName: Build stage

jobs:

- job: Build

displayName: Build

pool:

vmImage: $(vmImageName)

# Projekt bauen um Richtigkeit des source codes zu überprüfen

steps:

- task: DotNetCoreCLI@2

displayName: Build Client

inputs:

command: 'build'

projects: '\*\*/\*.csproj'

arguments: '--configuration $(buildConfiguration)'

- stage: Release

displayName: Release stage

jobs:

- job: Release

displayName: Release to Zip

pool:

vmImage: $(vmImageName)

# Release fähige Version von dem Projekt bauen (leichtgewichtiger und optimiert für produktiv Umgebung)

steps:

- task: DotNetCoreCLI@2

displayName: Publish the project

inputs:

command: 'publish'

publishWebProjects: true

arguments: '--output $(Build.ArtifactStagingDirectory)/blazor --configuration $(buildConfiguration)'

# zip archive mit gebauten dll's erstellen

- task: PublishBuildArtifacts@1

inputs:

ArtifactName: 'blazordist'

### Azure Cloud

In diesem Projekt wird die Azure Cloud verwendet um sowohl das Testsystem als auch das Produktiv System zu hosten. Die Azure Cloud wurde dafür ausgewählt da man hier sowohl die API als auch das BlazorWASM Frontend Projekt. Des Weiteren wird auch die Datenbank in der Cloud gehostet.

#### Vorteile

Einer der größten Vorteile der Azure Cloud liegt darin das einige Service für Testsysteme beziehungsweiße unter gewissen Voraussetzungen auch bei Produktiv Systemen kostenlos zur Verfügung stehen.

Ein weiterer Vorteil den die Azure Cloud in diesem Projekt Setup hat, ist das durch das automatische bauen der Software mit Azure Pipelines man hier einen automatischen Release Prozess erstellen kann. Jener deployed automatisch nach einem Validen Build die Software auf einem Testsystem. Wurde dieses Testsystem dann auf Funktion und Qualität geprüft kann man diese Version freigeben und sie wird direkt auf das Produktiv System deployed. Dieser Prozess funktioniert somit fast Vollautomatisch und man muss sich um keine Händischen Deployement Prozesse kümmern.

### MSSQL Datenbank

Bei diesem Projekt wird auf eine MSSQL Datenbank gesetzt. Diese wird in der Azure Cloud gehostet. Es wurde sich für eine MSSQL Datenbank entschieden da diese einerseits kostengünstig in der Azure betrieben werden kann, als auch mitunter am besten unterstützt wird.

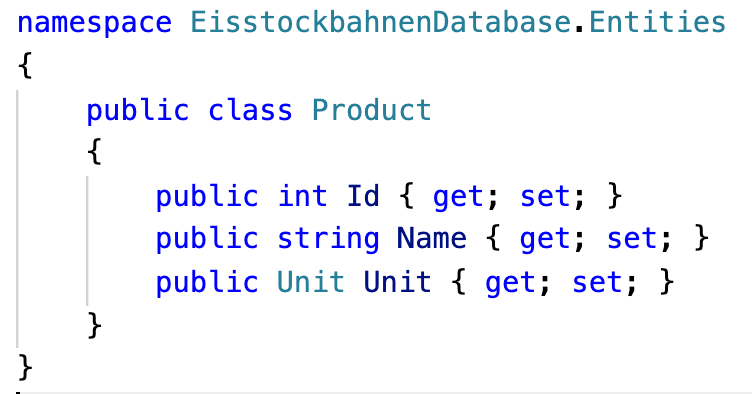
Ein weiterer Grund liegt darin das auf das Entity Framework Core in diesem Projekt gesetzt wird, welches ebenso von Microsoft bereitgestellt wird. Somit bewegt sich der ganze technische Bereich des Projektes in dem Microsoft Kosmos. Dies fördert die Agilität und reduziert zugleich die Fehleranfälligkeit.

## Entwicklung

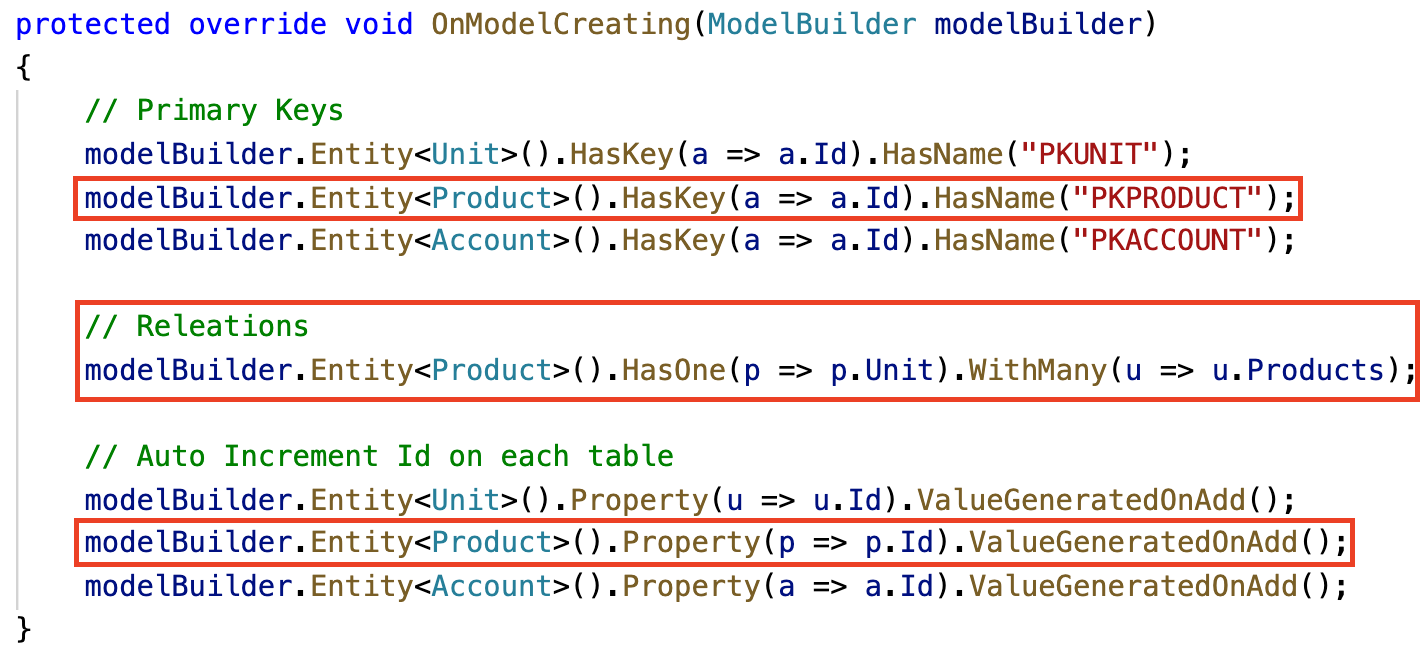
### Datenbank Anbindung

Die Datenbank Anbindung wird mit dem Entity Framework Core umgesetzt. Dieses ermöglicht eine vollständige Abstrahierung der Datenbankseite. Das bedeutet obwohl in diesem Projekt eine MSSQL Datenbank eingesetzt wurde ist man von dieser nicht abhängig.

Mit dem Entity Framework Core definiert man C# Klassen welche dann als Tabellen verwendet werden. Diese kann man über mitgelieferte Methoden des Frameworks konfigurieren. Hier wird der Vorgang bei der Product Tabelle veranschaulicht.



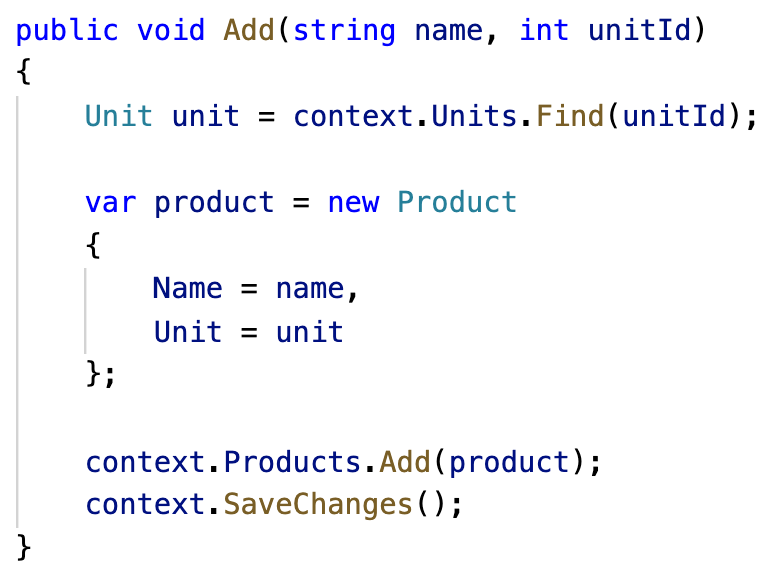
Danach wird die Klasse so konfiguriert wie gewollt. In diesem Beispiel bekommt die Product Tabelle einen Primary Key welcher automatisch erhöht wird. Sowie eine 1:N Beziehung zu der Tabelle Unit.



Um Daten in die Datenbank zu speichern oder von der Datenbank zu lesen benötigt man einen sogenannten DatabaseContext. Dieser DatabaseContext definiert jene Tabellen welche man über das Entity Framework ansprechen kann. Diese werden als DbSet<Tabellenname> definiert und können so dann angesprochen werden.

public DbSet<Product> Products { get; set; }

Mit dieser „Products“ Variable kann nun auf die Datenbank zugegriffen werden und man ist in der Lage Daten sowohl zu lesen als auch zu speichern.



Über die „SaveChanges“ Methode werden alle bisher erstellten Änderungen in die Datenbank per Transaktion gespeichert.

### Backend - Rest API

In diesem Projekt wurde die Rest API mittels eines ASP.NET Core Web API abgebildet. Dieses Projekt besteht aus 3 Ebenen. In der ersten und untersten Ebene liegt die Datenbank Ebene.

In dieser Ebene werden alle Datenbank Tabellen definiert. Des Weiteren werden auf dieser Ebene die Beziehungen der Tabellen zu einander konfiguriert. Auf dieser Ebene findet man auch den DatabaseContext wieder. Jener sorgt dafür das man Daten von der Datenbank abfragen und speichern kann. Auf dieser Ebene werden auch alle „Migration“ gespeichert. Migration werden von Entity Framework Core erstellt, wenn man sein Datenbank Schema erstellen will. Des Weiteren wird für jede Änderung die man auf seiner Datenbankebene tätigt eine neue Migration erstellt um Updatefähig zu bleiben. Ein weiterer Vorteil dieser Migration Skripte ist das jedes Skript auch mit einer Down grade Funktion kommt. So kann man bei einer unerwünschten Änderung die Datenbank sofort wieder auf die vorherige Version zurücksetzten.

Die nächste Ebene ist die sogenannte Businesslogik. In der Businesslogik befindet sich das Herz der Asp Net Core API. Hier werden die Daten von der Datenbank für die API vorbereitet. Hierzu wurde in diesem Projekt eine geteilte Klassenbibliothek für die Modelle erstellt. So kann die Businesslogik sowie die API als auch das Blazor Frontend auf dieselben Klassen zugreifen und man musste diese nicht doppelt implementieren. Die Businesslogik kümmert sich hier um die richtige Aufbereitung für das Frontend bzw. eingehende Daten werden auf die Richtigkeit Serverseitig überprüft. Diese Überprüfung dient zur Erhöhung der Sicherheit der Applikation. So können keine ungewollten Daten in die Datenbank gelangen. Hierfür wurde für jede Tabelle ein eigener Service geschrieben. Dieser Service kümmert sich um alle wichtigen Funktionen wie dem Löschen oder dem Speichern von Datensätzen.

Die letzte Ebene ist die API Ebene. Diese API Ebene bieten den Zugang der Daten von außerhalb und für das Frontend in diesem Projekt. In der API Schicht wurde das CRUD Prinzip angewandt. Dieses CRUD Prinzip steht führ Create, Read, Update oder Delete. Dies ist das gängigste Prinzip wie man eine moderne Web API aufbaut.

Das Routing der Web API wird in ASP NET Core über ein Route Attribute in der Klasse umgesetzt. Hier wurde sich in diesem Projekt dafür entschieden das jede Route mit „api“ beginnt und erst danach folgt die Schnittstelle die man Ansprechen will. Dies definiert man wie folgt in ASP NET Core.

[Route("api/[controller]")]

In einer Produktivumgebung wird dann automatisch von .Net Core Framework der Platzhalter [controller] durch den Klassennamen ersetzt und wird somit erreichbar gemacht.

[ApiController]

[Route("api/[controller]")]

public class ProductController : Controller

In den oben Angeführten Beispiel würde die API dann über folgenden Link erreichar sein

„domain-name/api/product“. Dies wäre auch zugleich ein GET Abfrage welche alle Produkte zurück gibt.

Dieser ProductController beinhaltet dann die Get Methode. Diese Methode liefert den Aufrufer eine IEnumerable von dem ProductModel zurück. Über das Attribut [HttpGet] teilen wir der Klasse mit das hier ein Aufrufer nur dann Daten bekommt, wenn die Anfrage eine Get Anfrage ist.

[HttpGet]

public IEnumerable<ProductModel> Get()

{

return new ProductService(context).Get();

}

Dieses Konzept wurde für alle Methoden fortgesetzt die benötigt werden.

### BlazorWASM Frontend

### CSS-Framework

Als CSS Framework wird in diesem Projekt materializecss verwendet. Dies ist ein sehr funktionsstarkes Framework welches global eingesetzt wird.

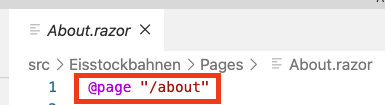
Es ist sich für materializecss entschieden worden das dieses Framework sehr leichtgewichtig im Vergleich zu anderen Markführern ist. Ein weiter Punkt war das materializecss keine Abhängigkeit zu jQuery hat. Dadurch wird die Größe der Webseite weiter komprimiert.

Diese Leichtgewichtigkeit ermöglicht es eine Webseite zu bauen welche trotz schlechtem Internet verhältnismäßig schnell lädt und wenig Datenverkehr erzeugt.

### Verlinkungen

Verlinkungen in BlazorWASM funktioniert nicht wie in HTML, hierfür wurde ein eigenes Konzept entwickelt.

Auf jeder Seite die man Verlinken will benötigt man das Property „@page“. Danach unter doppelten Anführungszeichen der Pfad unter welchem die Seite erreichbar sein soll. Im Falle der About Seite sieht die Verlinkung wie folgt aus.



Will man jedoch jetzt diese Seite über eine andere Blazor Seite aufrufen muss man das Blazor HTML Property „NavLink“ verwenden. Welches grundsätzlich gleich funktioniert wie das HTML „nav“ Element, da jedoch die Blazor Seiten kompiliert werden, muss man dies hier expliziert definieren.



### Landing Page

#### Gestaltung

Die Landing Page wurde so aufgebaut, dass jeder Webseitenbesucher als erstes eine der beiden Eisstockbahnen im Winter sieht.

Nach diesem Bild findet der Webseitenbesucher einen Einleitungstext welche Ihn darüber Informiert warum die Eisbahnen gebaut wurden und welche Arten von Wintersport grundsätzlich ausgeübt werden können.

Im nächsten Bereich finden sich die Bahnregeln für das Eisstockschießen wieder.

Als Abschluss findet man eine kleine Speisekarte und Preisliste, welche einen Link zu einer ausführlichen Auflistung aller Preise und Speisen führt.

#### Implementierung

Auf der Startseite wurde das eingebaute Grid System vom materializecss verwendet. Dies sorgt dafür das die Webseite auf allen Geräten leserlich und übersichtlich aussieht.

Dies wird in materializecss so implementiert das man zuerst eine div als Zeile erstellen muss und danach kann man die Spalten definieren. Hierzu muss man wissen ein materializecss Grid aus 12 Spalten besteht die man frei belegen kann.

Der Banner auf der Webseite wird als normales Bild eingebunden. Dies funktioniert über eine eigene materializecss Klasse namens „row“. Danach wurde das Bild über das „img“ Tag eingebunden.

Danach folgt in einer weiteren Zeile der H1 Header und ein Paragraph in welchen die einleitenden Worte stehen.

Die Bahnregeln werden in einer Liste dargestellt welche die Klasse „collection“ von materializecss bekommen haben. Dies sorgt für einen moderne Variante einer einfachen Liste und Wertet optisch die Webseite auf.

Der letzte Bereich dieser ist eine sogenannte „Card“. Eine materializecss Card besteht aus 3 Bereichen. Der erste Bereich ist ein Bild, danach kommt ein Content Bereich in den man seinen Text einfügen kann. Als drittes und zugleich optional kann man unter dem Content Beriech auch noch einen Link einfügen. Dieser verweist in diesem Projekt auf die detaillierte Preisübersicht.

<div class="titelLines col s12 ">

<h4>Speisekarte und Preiseliste</h4>

</div>

<div class="col s6 offset-s3">

<div class="card horizontal">

<div class="card-image">

<img src="img/preisliste.jpg">

</div>

<div class="card-stacked">

<div class="card-content">

<p>Unter den angegebene Link finden Sie alle Informationen zu unseren Preisen und Speisen</p>

</div>

<div class="card-action">

<NavLink class="nav-link" href="pricelist">Speisekarte und Preiseliste</NavLink>

</div>

</div>

</div>

</div>

### Geschichtsseite

#### Gestaltung

Auf der Geschichtsseite sieht der Webseitenbesucher als erstes die Entstehungsgeschichte der Eisstockbahnen und kann sich jene durchlesen.   
  
Danach folgen mehrere Bildergalerien der diversen Events die auf den Bahnen durchgeführt worden sind, wie zum Beispiel Ortsmeisterschafft oder Musikvereinsschießen.

#### Implementierung

Für die Entstehungsgeschichte wurde wieder ein Paragraph herangezogen.

Für die Bildergalerie wurde ein materializecss Karussell verwendet.

Dies ist ein fertiges vorgefertigtes Karussell welche starken Ähnlichkeiten zu einer normalen HTML Liste aufweist, jedoch mit materializecss Klassen ausgestattet wird um ein schönes Erscheinungsbild ab zu geben.

Die Implementierung in diesem Projekt sieht wie folgt aus.

<div class="carousel carousel-slider center">

<div class="carousel-item ">

<img src="img/eisstockbahnen\_musik6.jpg">

</div>

<div class="carousel-item white-text">

<img src="img/eisstockbahnen\_musik7.jpg">

</div>

<div class="carousel-item white-text">

<img src="img/eisstockbahnen\_musik3.jpg">

</div>

<div class="carousel-item white-text">

<img src="img/eisstockbahnen\_musik4.jpg">

</div>

</div>

Auch auf dieser Seite wurde auf das Gridsystem von materializecss gesetzt damit eine Seite erstellt wird welche auf allen Endgeräten übersichtlich ist.

### Preislistenseite

#### Gestaltung

Die Preislistenseite besteht aus 3 Tabellen. In der ersten Tabelle sieht der Webseitenbesucher das Bahn Geld. Darauffolgend wird eine Tabelle angezeigt in der es eine Übersicht aller Getränke gibt. Die letzte Tabelle zeigt alle Speisen die ausgegeben werden an.

#### Implementierung

Für die Implementierung dieser Seite wurde auf eine herkömmliche HTML Tabelle zurückgegriffen. Diese Tabellen wurden dann mittels materializecss Klassen noch adaptiert und verschönert.

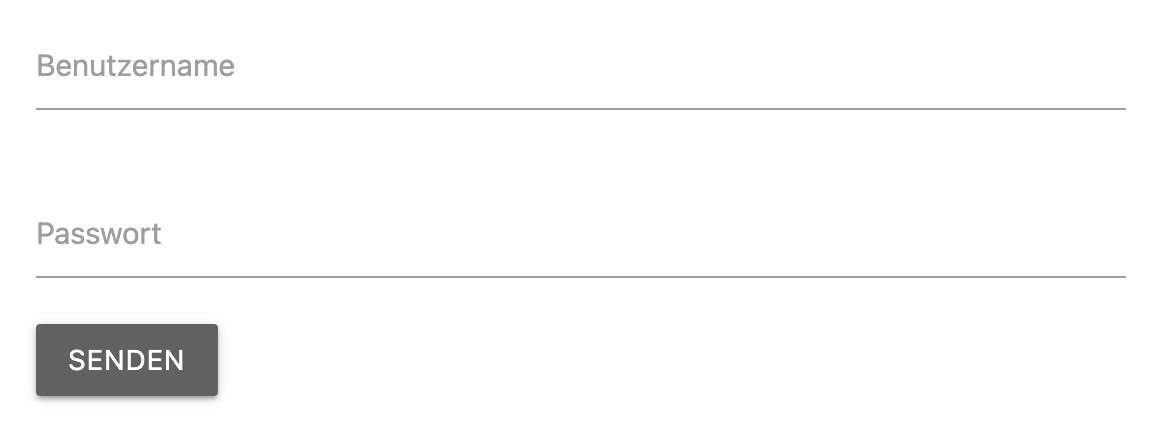
### Authentifizierungsseite

#### Gestaltung

Wenn der User auf den Punkt „Kalkulation“ in dem Menü drückt wird er zuerst auf die Authentifizierungsseite weitergeleitet. Diese Seite ist notwendig, um feststellen, ob es sich bei dem Webseitenbesucher wirklich um den Webseiteninhaber handeln oder nur um einen interessierten Webseitenbesucher.

Um dies verlässlich herauszufinden zu können wurde auf dieser Seite eine einfache Art des Logins eingebaut. Der Webseitenbesucher findet auf dieser Seite ein Eingabefeld für den Benutzernamen, ein Eingabefeld für das Passwort und einen Senden Button.

Wenn der Webseitenbenutzer falsche Daten eingibt wird unterhalb des Senden Button eine Fehlermeldung ausgegeben. War die Eingabe erfolgreich wird der Webseiteninhaber auf die Berechnungsseite weitergeleitet.



#### Implementierung

Wie bei jeder Seite wurde als erster Schritt ein div Tag mit der Klasse „Container“ verwendet und gleich danach ein div Tag mit der Klasse „row“.

Für die Login Form wurde 2 Eingabefelder und 1 Button implementiert.

Als erster wurde ein Inputtext für den Benutzernamen eingebaut. Damit das sich die Eingabe des Benutzernamens mittig befindet wurde um das Eingabefeld rundherum ein div Tag gelegt, welche die Klasse „input-field col s6 offset3“ hat. Die Klasse input-field ist von materializecss und bewirkt, dass das Input Tag und das Label Tag als eine Einheit benutzt werden. Die Klasse col s6 offset-s3 bedeutet, dass das Eingabegeld 3 Columns später anfängt und 6 Colums von dem Grid benutzen. In diesem div wurde dann 2 tag genutzt. Einmal das Input Tag und einmal das Label Tag.

Das Input Tag ist für die Eingabe zuständig. Das Label-Tag zeigt eine Überschrift oberhalb des Eingabefeldes an, damit der Webseitenbesucher weiß welche Eingabe von ihm gefragt ist.

Als nächstes wurde das Eingabefeld für das Passwort eingebaut. Der Aufbau dieses div ist der gleich wie bei dem von dem Benutzernamen div. Der einzige Unterschied liegt bei dem Input Tag, den hier wurde nicht der Type „Text“ verwendet, sondern der Type „Password“. Durch diesen kleinen Unterschied werden im Passworteingabefeld nur Pünktchen angezeigt und kein Text.

Das nächste div-Tag beinhaltet bei der Klasse, nur noch das col 6 offset 3. Die Klasse input-field wurde hier nicht benötigt, dafür hat dieses einen style bekommen hat. Der Style beinhaltet die Eigentlich „display“. Diese Eigenschaft wurde mit einer Bindungsvariable verbunden. Durch diese Bindungsvariable ist es möglich diesen Block anzuzeigen oder ihn zu verstecken. Innerhalb dieses div-Tags befindet sich nur ein p-Tag. Diese p-Tag beinhaltet die Fehlermeldung die angezeigt wird, wenn der User die falschen Daten angibt.

Das letzte div Tag hat denselben Aufbau wie das div Tag davor.

Dieses div-Tab beinhaltet nun den Senden Button. Dieser Button wurde mit einem button-Tag erstellt und mit den Klassen: „btn waves-efftect waves-light grey darken-2“ ausgestattete. Des Weiteren hat dieser Button ein onclick-Event, auf welches die Methode „CheckUser“ registriert ist.

Wenn der Knopf „Senden“ gedrückt wird, wird die Methode „CheckUser“ ausgeführt. In dieser Methode holt sich die Blazor Seite alle Benutzerkonten von der API. Es werden alle erhaltenen Accounts auf Benutzername und Passwort überprüft. Wenn es eine Übereinstimmung gibt, wird das Benutzerkonto in eine separate Variable gespeichert, ansonsten wird die Variable auf null gesetzt.

Falls es einen Benutzer mit diesem Passwort und Benutzernamen gibt, wird man zur Berechnungsseite weitergeleitet, ansonsten wird die Fehlermeldung angezeigt.

Um die allgemeine Sicherheit des Log Ins zu erhöhen wurde mittels SHA256 verschlüsselt.

Somit steht in der Datenbank das verschlüsselte Passwort und wird auch nur so übertragen. In der CheckUser Methode wird dann das eingegebene Passwort ebenso in einen SHA256 Hash umgewandelt um dann verglichen zu werden.

Danach wird ein Schlüssel in einen Hash umgewandelt und mit auf die Kalkulationsseite geschickt. Ohne diesen Schlüssel in Hash Format kann die Kalkulationsseiten nicht aufgerufen werden. Dies sichert das ganze System weiter ab.

private async void CheckUser(MouseEventArgs e)

{

using (SHA256 sha256Hash = SHA256.Create())

{

hash = GetHash(sha256Hash, password);

}

var accountModels = await Http.GetFromJsonAsync<AccountModel[]>("https://localhost:44361/api/account");

var account = accountModels.FirstOrDefault(x => x.Username == userName && x.Password == hash);

if (account != null)

{

using (SHA256 sha256Hash = SHA256.Create())

{

hash = GetHash(sha256Hash, "authentified");

}

NavigationManager.NavigateTo($"calculation/{hash}");

}

else

{

visibility = "block";

}

}

### Berechnungsseite

#### Gestaltung

Die Berechnungsseite ist ausschließlich für den Webseiteninhaber erreichbar. Sie dient dazu den Webseiteninhaber zu ermöglichen eine Einnahme/Ausgabenbilanz seiner Produkte zu erstellen.

Die Berechnungsseite wurde so aufgebaut das oben immer der Titel des Produktes angezeigt wird. Darunter befinden sich mehrere Eingabefehler. Jedes Eingabefeld wird für die Einnahmen/Ausgabebilanz benötigt. Auf der linken Seite befinden sich die Eingabefelder für die Anzahl der jeweils gekauft und verkauften Produkte.

Auf der rechten Seite sind die Preise für die gekauften und verkauften Produkte.

Ganz unten auf der Webseite befinden sich der Knopf um die Kalkulation auszuführen.

Wenn die Kalkulation fertig berechnet wurde, wird unterhalb der aufgelisteten Produkte eine dynamisch generierte Tabelle mit den berechneten Daten angezeigt. In diesem Daten sieht der Webseiteninhaber, wieviel er für das Produkt ausgegeben hat, wieviel er durch das Produkt eingenommen hat und was der Profit von dem Produkt war.

Ganz unten befindet sich natürlich auch auf dieser Seite der Footer der Webseite.

#### Implementierung

Die Berechnungsseite ist unter „calculation/{hash}“ erreichbar. Ist dieser Hash nicht Valide wird man automatisch zu der Log In Seite umgeleitet.

Hierzu wurde die OnParameterSet() Methode überschrieben. In dieser Methode wird überprüft ob übergebene Hash Wert mit dem erwarteten Hash Wert übereinstimmt. Sollte dies nicht der Fall sein dann wird der Webseitenbesucher zurück auf die Log In Seite geleitet. Sollte der Hash stimmen wird die API um die Produktdaten angefragt.

Um die Werte die der Benutzer eingegeben hat zu bekommen wird das in Blazor Integrierte Binding Konzept verwendet. Dabei wird eine C# Variabel auf ein html input Element gebindet. Dabei entsteht ein two way binding welches uns ermöglicht die Eingaben eines Benutzers automatisch mit zu bekommen bzw. können wir die Variabel im Programmcode verändern und die Webseite updated sich automatisch. Das Ganze sieht in BlazorWASM wie folgt aus.

@foreach (var calculationItem in calculationItems)

{

<div class="row">

<h5>@calculationItem.Product.Name:</h5>

<div class="input-field col s12 l6">

<input id="b\_@calculationItem.Product.Id" type="number" min="0" @bind="calculationItem.Bought">

<label for="b\_@calculationItem.Product.Id">Anzahl gekauft</label>

</div>

In diesem Beispiel sieht man wie dieses Binding Konzept funktioniert.

Hier wird für jedes Produkt der gleiche HTML Code erzeugt und jedes Mal wird von dem Produkt die „Bought“ Variabel dem input Element zugewiesen.

Wenn nun der Berechnen Knopf gedrückt wird, löst dieser ein Event aus auf welches wir uns mit einem C# Eventhandel verbinden.

HTML:

<button class="btn waves-effect waves-light grey darken-2" @onclick="Calculate">Berechnen</button>

C#:

private void Calculate(MouseEventArgs e)

{

CalculationService service = new CalculationService();

calculationItems = service.CalculateItems(calculationItems);

visibility = "block";

}

In diesem CalculationService sehen wir dann wie das two way binding funktioniert hat. Obwohl nirgendwo explizit die Werte, die der Benutzer eingegeben hat, abgefragt werden kann der CaculationService mit diesen Werten Arbeiten.

public List<CalculationItem> CalculateItems(List<CalculationItem> calculationItems)

{

foreach (var item in calculationItems)

{

item.Profit = item.SellPrice \* item.Sold - item.BuyPrice \* item.Bought;

}

return calculationItems;

}

Durch das zurückgeben der calculationItems werden nun automatisch die Ergebnisse richtig angezeigt da hier wieder das two way binding greift und die Werte updated.

Wenn diese Berechnung abgeschlossen ist wird der Ergebnisbereich eingeblendet. Dies funktioniert über die CSS Property „display“ welche auf „visible“ gesetzt wird. Auch hier wird nur eine Änderung in der C# Variabel vorgenommen.

HTML:

<div style="display: @visibility">

C#:

string visibility = "none";

In der Calculate Methode:

visibility = "block";

### Kontaktseite

#### Gestaltung

Die Kontaktseite besteht aus zwei Hauptbereichen. In dem ersten Bereich der Webseite wird eine materializeecss Card eingefügt. In dieser Card sieht man ein Bild von der Auftraggeberin und dem Man. Des Weiteren findet man die Kontaktdaten der Auftraggeberin sowie einen Link zur Webseite des Orts.

Im zweiten Bereich der Webseite wird eine Google Maps Karte um eine Anfahrt zu erleichtern. Diese Karte benötigt eine Kombination von HTML, JavaScript und C#.

#### Implementierung

Der erste Bereich dieser Seite wird mit einer materializecss Card dargestellt. Diese Card ermöglicht es einerseits ein Bild der Auftraggeberin einzubinden als auch alle Kontakt Infos. Des Weiteren wird hier auch noch ein Link zur offiziellen Gemeindeseite eingebunden.

<div class="card horizontal">

<div class="card-image">

<img class="materialboxed" src="img/1419529931945.jpg">

</div>

<div class="card-stacked">

<div class="card-content">

<p>Aga Ratzenböck</p>

<p>Au 9</p>

<p>4733 Heiligenberg</p>

<p>Österreich</p>

<br />

<p>Telefon: 06608363475</p>

<p>E-Mail: agaratzenboeck@gmail.com</p>

</div>

<div class="card-action">

<a href="http://www.heiligenberg.at/">Gemeinde Heiligenberg</a>

</div>

</div>

</div>

Im zweiten Bereich der Webseite wurde eine Google Maps Karte eingebunden um die Anfahrt zu erleichtern. Dies wurde über die Google Maps JavaScript Bibliothek umgesetzt.

Diese JavaScript Bibliothek hat zudem noch eine Abhängigkeit auf „Polyfill“. Polyfill wird dafür verwendet das die Google Maps JavaScript Bibliothek auch in älteren Browsern funktioniert. Dies heißt die Google Maps Bibliothek kann wie in neuen Browser Versionen arbeiten und Polyfill schließt die Funktionslücke bei älteren Browsern.

<script src="https://maps.googleapis.com/maps/api/js?key=<YOUR\_KEY>&v=3"></script>

<script src="https://polyfill.io/v3/polyfill.min.js?features=default"></script>

Wie man im ersten Script Tag sieht benötigt man einen API Key um mit der Google Maps API kommunizieren zu können. Diesen API Key bekommt man nur mit einen Google Benutzerkonto. Wenn man einen Google Benutzerkonto hat kann man unter <https://cloud.google.com/> ein neues Projekt anlegen. Nachdem dies erledigt muss man alle Bibliotheken auf der Webseite von Google aktivieren. Diese Bibliotheken findet man unter „APIs & Dienste“ > „Bibliothek“. Folgende Bibliotheken müssen aktiviert werden.

* Maps JavaScript API
* Geolocation API
* Places API
* Roads API

Die Reihenfolge in welcher diese Bibliotheken aktiviert werden ist in diesem Fall nicht von Relevanz. Jedoch müssen sie auf jeden Fall vor dem nächsten Schritt aktiv sein.

Nun können wir unter „APIs & Dienste“ > „Anmeldedaten“ einen neuen API Schlüssel erstellen indem wir auf „Anmeldedaten erstellen“ > „API Schlüssel“ klicken. Wir könnten nun im Nachgang den Schlüssel noch einschränken. Dies ist besonders wichtig das jeder Schlüssel nur eine begrenzte Anzahl an Kostenlosen aufrufe hat. Hier könnte man zum Beispiel die Domain der eigenen Webseite eintragen damit werden nur Aufrufe von dieser Domain akzeptiert. Wichtig hier ist noch zu erwähnen das man eine Zahlungsmethode in Form von einer Kreditkarte hinterlegen muss damit der API Schlüssel auch funktioniert. Hier fallen per se keine Kosten an. Dies dient lediglich nur zur Legitimierung des Google Accounts. Natürlich steht einen hier auch die Option offen das man die Kreditkarte automatisch belasten lässt damit der API Schlüssel immer funktioniert und man einen Reibungslosen Betrieb der Webseite gewährleisten kann.

Nachdem wir den generierten API Schlüssel in dem Script Tag eingefügt haben kann man mit der Implementierung starten.

HTML:

<div class="row">

<div id="map"></div>

</div>

JavaScript:

function initialize() {

var latlng = new google.maps.LatLng(48.369060, 13.820479);

new google.maps.Marker({

position: latlng,

map,

title: "Eisstockbahnen Ratzenböck!",

});

var options = {

zoom: 14,

center: latlng,

mapTypeId: google.maps.MapTypeId.ROADMAP

};

var map = new google.maps.Map(document.getElementById("map"), options);

}

C#:

protected override async Task OnAfterRenderAsync(bool firstRender)

{

if (firstRender)

{

await JSRuntime.InvokeVoidAsync("initialize", null);

StateHasChanged();

}

}

Hier wurde ein HTML div angelegt und mit der id map versehen. Wird die Seite geladen wird die „OnAfterRenderAsync“ Methode aufgerufen. Bei dem ersten Rendern der Seite wird die Google Maps Karte initialisiert indem eine JavaScript Funktion aufgerufen wird.

In dieser JavaScript Funktion wird als erstes die Position der Eisbahnen bestimmt. Danach wird in der Variable options die Karte konfiguriert. In der letzten Zeile der JavaScript Funktion wird dann die Karte erstellet und auch dadurch sichtbar.

## Bilderbearbeitung

### Verwendetes Programm

In diesem Projekt wurde bei der Bildbearbeitung auf GIMP in der Version 2.10 gesetzt. GIMP ist eine, erstmals 1996 veröffentlichte, kostenlose Software zur Bearbeitung von Bildern. GIMP wurde als kostenlose alternative zu Adobe Photoshop entwickelt.

In diesem Projekt kam GIMP zum Einsatz da es eines der weitverbreitetsten Bildbearbeitungstools ist. Zu GIMP gibt es eine sehr umfangreiche und detaillierte Dokumentation, sowie die ein oder anderen Videos zur Bildlichen Veranschaulichung der Funktionen.

### Verwendete Funktionen

Folgende Funktionen aus GIMP wurden in diesem Projekt zur Bearbeitung der Bilder verwendet.

* Zuschneiden
* Drehen
* Klonen
* Heilen
* Veränderung der Helligkeit

### Verwendete Funktionen bei Bild

* Titelbild -> wurde auf eine passende Form zugeschnitten
* Kontaktbild -> wurde zuerst zugeschnitten und im Anschluss wurde ein Bereich geklont.
* Preisliste -> wurde zuerst in die Waagerechte gedreht, danach wurde die Preisliste zugeschnitten. Im Anschluss wurde das Bild mit der Heilen Funktion bearbeitet und zuletzt noch aufgehellt.

### Vorher – Nachher Vergleich