Projektdokumentation

Inhaltsverzeichnis

[Projektplan - 2 -](#_Toc58057027)

[Projektauftrag - 2 -](#_Toc58057028)

[Projektstrukturplan - 2 -](#_Toc58057029)

[Projektarbeitspaketspezifikation - 2 -](#_Toc58057030)

[Projektzeitplan - 2 -](#_Toc58057031)

[Projektdokumentation - 2 -](#_Toc58057032)

[Visual Studio - 2 -](#_Toc58057033)

[Installation - 2 -](#_Toc58057034)

[Konfiguration - 2 -](#_Toc58057035)

[Entwicklungsumfeld - 2 -](#_Toc58057036)

[Version Control System - 2 -](#_Toc58057037)

[Github - 2 -](#_Toc58057038)

[Azure Pipelines - 3 -](#_Toc58057039)

[Azure Cloud - 5 -](#_Toc58057040)

[MSSQL Datenbank - 5 -](#_Toc58057041)

[Entwicklung - 5 -](#_Toc58057042)

[Projekterstellung - 5 -](#_Toc58057043)

[CSS-Framework - 5 -](#_Toc58057044)

[Verlinkungen - 5 -](#_Toc58057045)

[Login Mechanismus - 6 -](#_Toc58057046)

[Bilanzberechnung in C# - 6 -](#_Toc58057047)

[Google Maps Integration - 6 -](#_Toc58057048)

[Bilderbearbeitung - 6 -](#_Toc58057049)

[Verwendete Programme - 6 -](#_Toc58057050)

[Vorschau Bildbearbeitung - 6 -](#_Toc58057051)

[Vorher – Nachher Vergleich - 6 -](#_Toc58057052)

# Projektplan

## Projektauftrag

## Projektstrukturplan

## Projektarbeitspaketspezifikation

## Projektzeitplan

# 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.

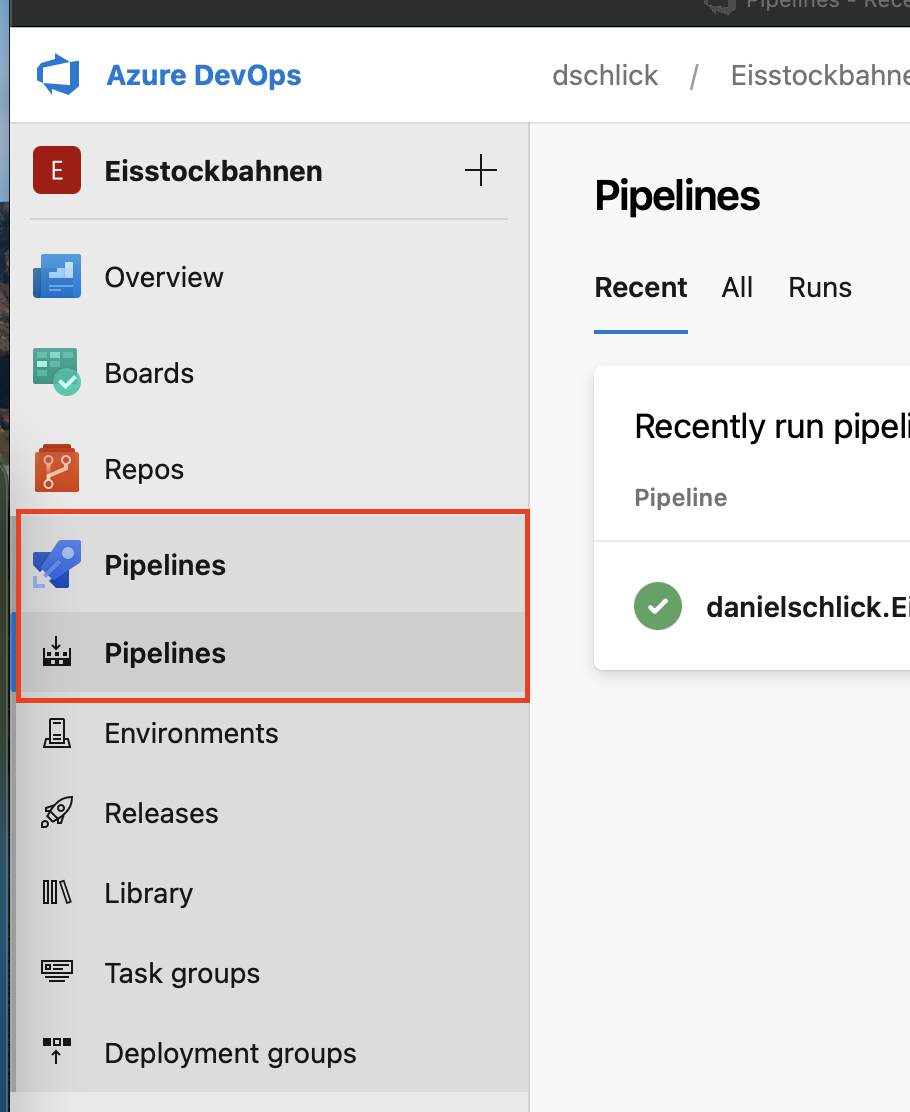
### Github

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.

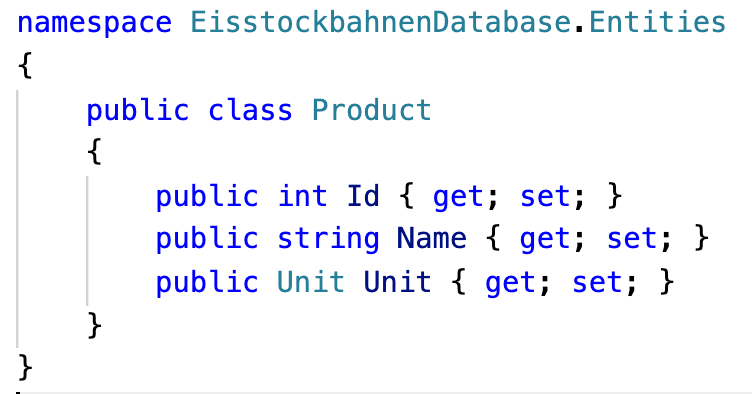
## Entwicklung

### Projekterstellung

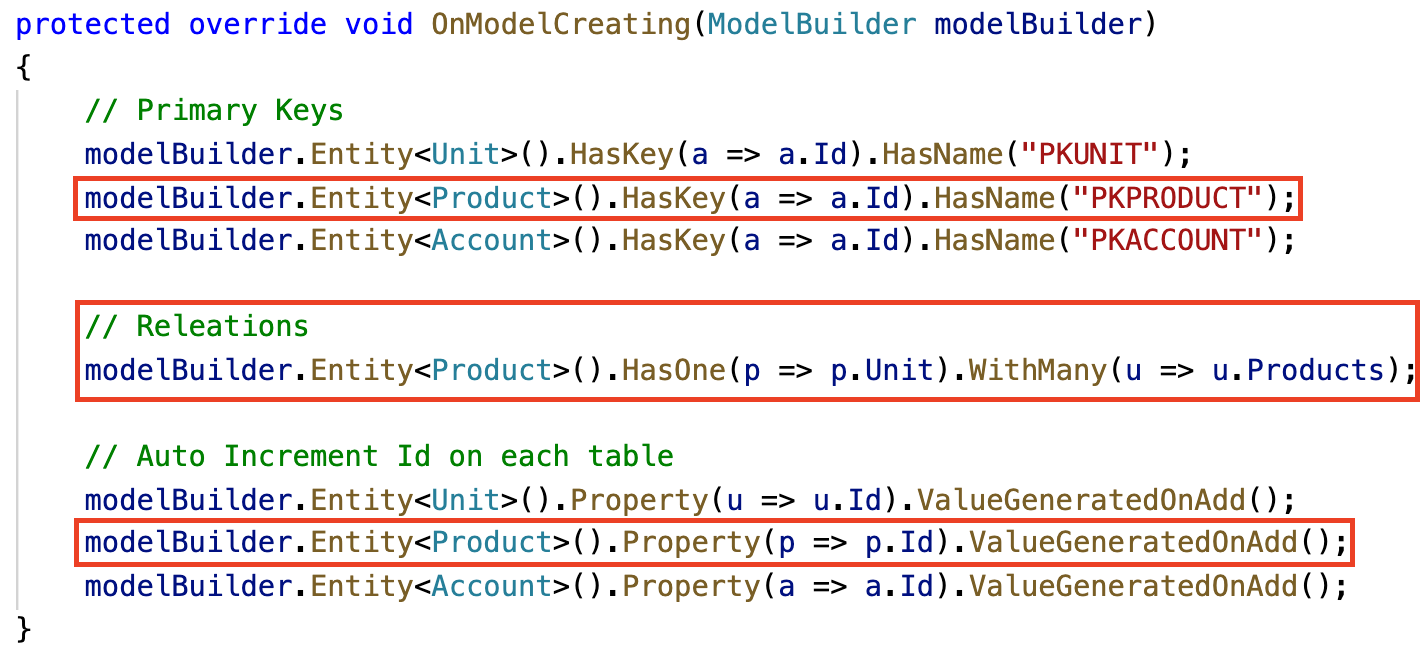
### Datenbank Anbindung

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.



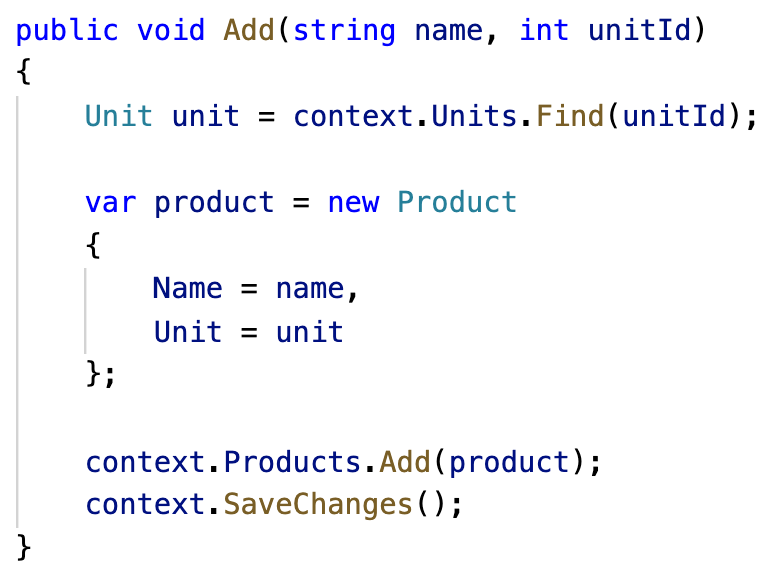
Danach wird die Klasse so konfiguriert wie gewollt. In diesem Beispiel bekommt die Product Tabelle einen Primary Key welcher automatisch erhöht wird. Sowie eine 1:N Beziehung zu der Tabelle Unit.



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.

### Rest API

In diesem Projekt wurde die Rest API mittels eines ASP.NET Core Web API abgebildet. Die API wird immer über die Route „api/…“ erreicht. Dies wird in C# über das Route Attribute definiert.

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

In einer Produktivumgebung wird dann automatisch von .Net [controller] durch den Klassennamen ersetzt und daher 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.

Dieses Konzept wurde für jede weitere Tabelle des Projektes wiederholt um sicherzustellen das alle Daten bearbeitet werden können.

### BlazorWASM Frontend

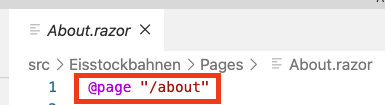
### CSS-Framework

Als CSS Framework wird in diesem Projekt materializecss verwendet. Dies ist ein sehr funktionsstarkes Framework welches schon sehr stark eingesetzt wird. Des Weiteren ist dieses Framework leichtgewichtiger als viele andere Konkurrenzprodukte wie zum Beispiel Bootstrap. Dies bringt den Vorteil das weniger Netzwerkverkehr erzeugt wird, welches ein schnelleres Laden der Webseite ermöglicht. Zudem kommt materializecss zur Gänze ohne jQuery aus welches aufgrund seiner Größe und schlechten Performance somit nicht die Geschwindigkeit der Projektseite beinträchtigen kann.

### 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.



### Login Mechanismus

### Berechnungsseite

#### Aufbau und Struktur

Die Berechnungsseite wurde so Aufgebaut das in der ersten Sektion der Webseite die Einnahmen wiederspiegelt. In der zweiten Sektion werden dann die Ausgaben definiert.

Nach dem drücken des „Berechnen“ Knopfes wird darunter die Bilanz ausgegeben.

#### Funktionalität

Die Bilanz

### Google Maps Integration

## Bilderbearbeitung

### Verwendete Programme

### Vorschau Bildbearbeitung

### Vorher – Nachher Vergleich