1.Mai.2022



Entwicklung einer Webanwendung Prototyp zur administrativen Verwaltung von Händlern

**Prüfung Bewerber:**

Iljasgadzhi Makhsunov

Ingersheimer Hauptstraße 57, 74564 Crailsheim

Identnummer: 530638

E-Mail: [iljasgadzhi.Makhsunov@plan-b-gmbh.com](mailto:iljasgadzhi.Makhsunov@plan-b-gmbh.com)

Telefon: +49 1577 9854880

**Ausbildungsbetrieb:**

PlanB. GmbH

Kocherstraße 15,

73460 Hüttlingen

**Projektbetreuer:**

Wendelin Niesl

E-Mail: [wendelin.niesl@gmail.com](mailto:wendelin.niesl@gmail.com)

Telefon: +49 172 4312760

Inhalt

[1. Einleitung 3](#_Toc102569981)

[1.1 Projektumfeld 3](#_Toc102569982)

[1.2 Projektziel 4](#_Toc102569983)

[1.3 Projektbegründung 4](#_Toc102569984)

[1.4 Projektschnittstellen 4](#_Toc102569985)

[1.5 Projektabgrenzung 4](#_Toc102569986)

[1.6 Ansprechpartner 4](#_Toc102569987)

[2. Projektplanung 5](#_Toc102569988)

[2.1 Ablaufplan 5](#_Toc102569989)

[2.2 Ressourcenplanung 5](#_Toc102569990)

[2.3 Pflichtenheft 5](#_Toc102569991)

[2.3.1 Ist-/Soll-Analyse 5](#_Toc102569992)

[2.3.2 Musskriterien 6](#_Toc102569993)

[2.3.3 Wunschkriterien 6](#_Toc102569994)

[2.3.4 Prozessschritte 7](#_Toc102569995)

[2.4 Vorbereitungsphase 7](#_Toc102569996)

[2.4.1 Kick-off-Meeting 7](#_Toc102569997)

[2.4.2 Auswahl des Frameworks 7](#_Toc102569998)

[2.4.3 Auswahl der Azure Services 8](#_Toc102569999)

[3. Durchführung 8](#_Toc102570000)

[3.1 Implementierungsphase 8](#_Toc102570001)

[3.1.1 Servicebereitstellung 8](#_Toc102570002)

[3.1.2 Implementierung des Backends 9](#_Toc102570003)

[3.1.3 Unit Tests 15](#_Toc102570004)

[3.1.4 Implementierung des Frontends 18](#_Toc102570005)

[3.2 Logging mit Hilfe von ILogger 24](#_Toc102570006)

[3.3 Qualitätssicherung 24](#_Toc102570007)

[4. Projektergebnis 24](#_Toc102570008)

[4.1 Zeitlicher Rahmen 24](#_Toc102570009)

[4.2 Soll- /Ist-Vergleich 24](#_Toc102570010)

[5. Benutzerhandbuch 25](#_Toc102570011)

[5.1 Vorwort 25](#_Toc102570012)

[5.2 Starten 25](#_Toc102570013)

[5.3 Navigation 25](#_Toc102570014)

[5.3.1 Startseite 25](#_Toc102570015)

[5.4 Übersichtsseite der Händler 26](#_Toc102570016)

[5.5 Händler Erstellen 26](#_Toc102570017)

[5.6 Händler Bearbeiten 27](#_Toc102570018)

[6. Tabellenverzeichnis 28](#_Toc102570019)

# Einleitung

## Projektumfeld

Die PlanB. GmbH ist ein IT-Dienstleistung Unternehmen, deren Hauptsitz in der Kocherstraße 15, 73460 Hüttlingen liegt. Das Unternehmen mit über 135 Mittarbeitern ist über drei weiteren Standorten deutschlandweit verteilt. Sie bieten komplexe Anwendungen und IT-Infrastrukturlösungen an, die auf Microsoft-Technologien basiert sind. Die Kunden von PlanB. GmbH sind meistens Unternehmen des gehobenen Mittelstands und Konzerne.

Das Projekt ESB („Enterprise Service Bus“) ist ein Beispiel Projekt, was die PlanB. GmbH seit Anfang 2017 und bis heute weiterhin entwickeln. ESB bietet die Kommunikation zwischen zwei verschiedenen Endpunkten an, so dass ein System in der Lage ist mit anderem System zu kommunizieren.

Abbildung 1: PlanB. GmbH

## Projektziel

Für ein internes Projekt soll eine Webanwendung entwickelt werden, um Händler, Benutzer und Bezahlvorgänge verwalten zu können. Ich implementiere die Verwaltung von Händlerdaten. Es werden folgende Teilaufgaben implementiert. Die Händler werden in tabellarischer Form im Frontend angezeigt. Für diese Ansicht wird eine separate View erstellt. Es besteht die Möglichkeit diese Händler zu aktualisieren. Um dies zu ermöglichen, müssen mehrere Schnittstellen im Backend entwickelt werden, welche es ermöglichen, Händler Lesen/Löschen/Schreiben/Updaten zu können. Für das Backend werden die notwendigen Azure Funktionen implementiert, diese beinhalten die entsprechende Logik für die Verwaltung der Händlerdaten, welche in einem Azure Blob Storage als Datei in JSON Format gespeichert werden.

## Projektbegründung

Die Webanwendung gehört zu einem internen Projekt der „PlanB. GmbH“. Die Anwendung

kann zukünftig als Prototyp im weiteren Kundenprojekten eingesetzt werden, um ähnliche

Anwendungsfälle abzudecken. Da die „PlanB. GmbH“ Dienstleister für digitale Produkte ist,

wird dieser Prototyp für die Kundenakquise verwendet werden.

## Projektschnittstellen

Die Webanwendung zur Verwaltung von Händlern verwenden eine Verbindung zum „Azure Blob Storage“, um die Daten von Händlern abzurufen, aktualisieren, erstellen und zu löschen. Diese Aktionen finden in der Azure Funktionen statt. Die einzelnen Schnittstellen werden über den sogenannten „HTTP-Trigger“ ausgelöst, die uns Azure Funktion anbietet.

Die Entwicklungsumgebungen sind Visual Studio 2022 und Visual Studio Code. Zusätzlich werden noch einige Pakete benötigt, wie die „Microsoft.Azure.WebJobs.Extensions.Storage“ Version 5.0.0, der für zugreifen von Azure Storage ermöglicht und die „Microsoft.NET.Sdk.Functions“ Version 4.1.0, um Azure Funktionen nutzen zu können.

## Projektabgrenzung

Die Webanwendung zur Verwaltung von Händlern kann wie im Pflichtenheft beschrieben umgesetzt werden.

## Ansprechpartner

Der Ansprechpartner für das Managementportal von Händlern ist mein Projektleiter Herr Wendelin Niesl. Er ist Solution Expert und ist in ESB Projekt tätig.

# Projektplanung

## 2.1 Ablaufplan

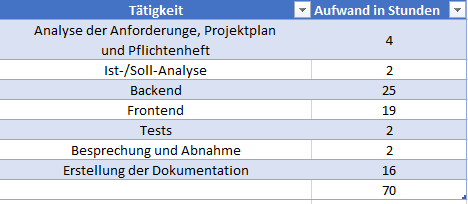
Anbei die Projektphasen inklusive Zeitaufwand für jeden Arbeitsschritt:

Tabelle 1: Projektphasen

## 2.2 Ressourcenplanung

Für die Entwicklung von der Webanwendung für Verwaltung von Händlern, wurden folgenden Ressourcen benötigt.

Arbeitsmittel:



Tabelle 2: Arbeitsmittel

Personalkosten:

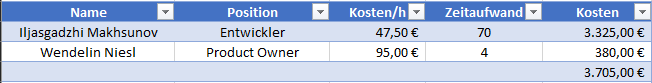


Tabelle 3: Personalkosten

## 2.3 Pflichtenheft

### 2.3.1 Ist-/Soll-Analyse

**Ist-Analyse:**

Mit dem aktuellen Stand des Managementportals besteht es die Möglichkeit, alle Händler abzurufen, dann einzelnen Händler zu bearbeiten/aktualisieren/löschen und einen neuen zu erstellen. Für die Händlern werden derzeit nur Testdaten verwenden, die ebenfalls lokal abgespeichert werden.

**Sollkriterien:**

Zunächst sollte ein Managementportal für die Händlern erstellt werden, zu dem müssen einige Aspekte im Vorfeld überlegt werden. Zum einen welches Framework für die Umsetzung verwendet wird. Wie wird die Anwendung in Azure Cloud gehostet und einen Lösungsvorschlag machen, der die ganzen Anforderungen abdeckt.

Es sollten auf einer Seite alle bereits existierenden Händlern in einer Tabellarischer Form angezeigt werden. Die sollten durch eine „Id“ identifizierbar sein. Dann sollte es eine Option geben einzelne Händler auszuwählen und zu bearbeiten. Zum Beispiel durch einen „Edit“ Button. Als nächstes sollte man die Möglichkeit haben einzelne Infos von dem Händler abzuändern und diese abzuspeichern. Unveränderten sollte es möglich sein einen auswählten Händler zu löschen. Ein wichtiger Punkt wäre, dass die einzelne Händlern irgendwo ablegt und gepflegt werden müssen.

Die einzelne Funktionalität sollten irgendwie testbar sein und dazu sollten auch eine Möglichkeit geben, Fehlern zu sehen, falls beim einen Vorgang auftreten sollten. Die Gedanken über die Schnittstellen müssen ebenfalls gemacht werden. Wie soll die Kommunikation zwischen den Frontend und Backend stattfinden und es sollten kosten beachtet verwenden, deshalb wird das genüg, wenn man den Prozess Lokal vollkommen zum Laufen kriegt.

### 2.3.2 Musskriterien

Die Web Anwendung wird komplett von neu kreiert, deshalb müssen nur folgende Anforderungen erledigt werden, die als notwendig definiert sind. Es müsste eine Startseite und eine Navigation vorhanden sein, die auf die Händler Einträge navigierbar ist. Die zeigt die komplette Händlerliste an, die man verwaltet kann. Zunächst sieht man nur die eindeutige ID des Händlers und einen Button fürs bearbeiten. Wenn man auf den Klick, dann sollten die einzelnen Infos zu den Händlern angezeigt werden, die dann entweder aktualisiert und gelöscht werden können. Ganz oben finden man noch einen weiteren Button, um einen neuen Händler einzutragen. Unterwanderten muss beim erfolgreichen/fehlgeschlagenen bearbeiten oder löschen von Händlern, visuell einen Popup mit der Fehler Meldung auftauchen.

Die Web Anwendung sollte in beliebigem Framework geschrieben werden, aber es wird aus den „Single Page Application“ Hintergrunde, eine davon bevorzugt, weil die Vorteile von „Single Page Application“ wären, dass die leicht auf Azure gehostet werden kann und wegen den kürzere Ladezeiten. Zukünftig sollte die Webanwendung auf Azure Cloud gehostet werden. Die Azure Funktionen sollten unter der aktualer .NET Version geschrieben werden, wie zum Beispiel mit .NET 6.0 Framework.

### 2.3.3 Wunschkriterien

Wünschenswert wäre es, dass das Managementportal auch als Response design verfügbar ist. Da viele Kunden auf sowas schauen und es beim Kunden einen guten Eindruck hinterlassen, wenn Sie unter Smartphone kurz auf die Web Anwendung zugreifen und es bei Ihnen immer noch so gut aussieht wie auf dem Desktop.

Unteranderem beim Aufrufen von Händlern Seite friert die erste Seite für wenige Sekunden und es wäre ebenfalls wünschenswert, wenn man statt eingefrorene Seite anstarrt, einen Lade Balken hätte, wodurch man visuell den Eindruck kriegt, dass es in Hintergrund die Daten geladen werden und es einige Sekunden dauern kann.

Anschließen wenn es noch Puffer gibt, wäre eine CSV-Protokoll sinnvoll zu implementieren, weil dann könnte ich die einzelne Fehlern unter den CSV-Protokoll nachsehen und filtern.

### 2.3.4 Prozessschritte

Tabelle 4: Prozessschritte

## 2.4 Vorbereitungsphase

### 2.4.1 Kick-off-Meeting

In einem Meeting mit dem Product Owner Herr Wendelin Niels, haben wir klare Anforderungen und Ziele des Projektes geklärt. Auch verschiedene Lösungsansätze, sowie Ideenvorschläge wurden ebenfalls besprochen.

### 2.4.2 Auswahl des Frameworks

Es wurde entschieden, dass für Logik also für das Backend werden die Azure Funktionen verwendet, die mit .NET 6 geschrieben werden. Grund dafür war, dass man die problemlos in Azure Cloud bereitstellen kann und die Kosten, die man dadurch spart. Man zahlt nur so viel, wie viel auch die Azure Funktion ausgeführt wird.

Für die Erstellung der Webanwendung wird die Angular Framework verwendet, um den Vorteil von „Single Page Applications“ zu nutzen. Sie beinhalten einen optimierten Entwicklungsprozess und verbrauchen weniger Serverressourcen. Ein weiterer Grund hier für war, dass ich bereits einige Erfahrungen mit Angular gesammelt habe. Für die Gestaltung wird Bootstrap als Framework verwendet, weil der einfach zu bedienbar ist und die Gestaltung der Vorstellung entspricht.

### 2.4.3 Auswahl der Azure Services

Damit die Händlern Information gespeichert, aktualisiert, gelöscht und abgerufen werden. Wird der Azure Blob Storage verwendet. Diese Entscheidung wurde ausfolgenden Gründen gewählt, um Kosten zu sparen, weil als vergleich von einer SQL Datenbank, fallen die Server Kosten weg und man zahlt nur noch für den Datenspeicher. Der zwei Grund wäre, dass die Händlern Daten als JSON Format gespeichert werden, somit müssen wir beim Blob Storage keine Gedanken um Struktur der Daten machen, weil Blob Storage in der Lage ist Objekte zu speichern.

# Durchführung

## 3.1 Implementierungsphase

### 3.1.1 Servicebereitstellung

Die Webanwendung läuft momentan Lokal, aber um lokal die Webanwendung nutzen zu können, sollte trotzdem eine Verbindung zum Azure Blob Storage aufgebaut werden. Dafür muss zunächst der „Azure Storage Explorer“ installiert werden und in der Azure Funktionen unter den „localhost.json“ sollte „UseDevelopmentStorage“ auf „true“ gesetzt werden. Somit wird die Verbindung lokal zum Azure Blob Storage aufgebaut.

Damit der Managementportal zukünftig in einer Azure Cloud gehostet werden kann, werden folgende Azure Ressourcen benötigt.

1. App Services -> Web Anwendung hosten
2. Storage Account -> Händler Liste abzuspeichern
3. Function App -> Auslagerung von Logik mit Azure Funktionen
4. Application Insights -> Überwachung von der Web Anwendung

### 3.1.2 Implementierung des Backends

#### 3.1.2.1 Implementierung der Klasse „MerchantAccountModel“

Um die Händler Daten zu verarbeiten wird die Klasse „MerchantAccountModel“ implementiert, die wird für Azure Funktionen verwendet, um ankommenden Händlern später zu mappen. Die Klasse beschreibt, wie die Händler später auszusehen haben. Sie besteht aus mehreren Eigenschaften und der Syntax „JsonProperty“ und dient für den Zweck, dass die Informationen über den Händler klein geschriebene Eigenschaften erkennen und dementsprechend mappen. Laut der C# Best Practice, werden Eigenschaften großgeschrieben.

Abbildung 2: MerchantAccountModel in Form einer Klasse

Für die nutzen der generische Liste wird die Using-Anweisung „System.Collections.Generic“ verwendet und um „Jsonproperty“ zu benutzen wird folgender Using-Anweisung verwendet „Newtonsoft.Json“.

#### 3.1.2.2 Implementierung der Klasse „ResponseModel“

Die Klasse „ResponseModel“ wurde implementiert aus dem Grund, dass man eine einheitliche Antwortnachricht zurückbekommt. Die wird bei den Fehlern, Erfolgen und weiteren Aktionen benutzt, um wie gesagt den Nutzer Bescheid zu geben, ob die Aktion erfolgreich durchgeführt wurde.

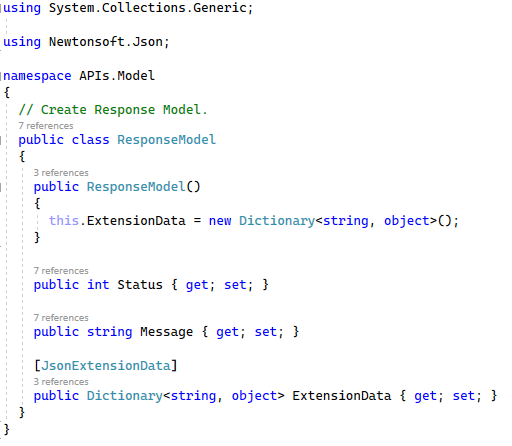


Abbildung 3: ResponseModel in Form einer Klasse

#### 3.1.2.3 Implementierung der Klassen „Create/Read/Update/Delete“

**Create:**

Damit die Händler erstellt, abgerufen, aktualisiert und gelöscht werden können, wurden dafür die Azure Funktionen erstellt.

Für erstellen einen neuen Händler wurde die Klasse „Create“ zusammengeschrieben. Sie wird wie eine Methode deklariert und hat die Klasse „HttpResponseMessage“ als Rückgabe Wert. Darauf folg immer ein Trigger, in unserem Fall ist das der „HttpTrigger“ der nur die „Post“ Methode akzeptiert. Beim Auslösen der Funktion wird dann eine „HttpRequestMessage“ gespeichert, die dann die notwendige Information enthält, wie welcher „RequestBody, headers etc.“ geschickt wurden. Dann folgt der „Blob Output Binder“, mit dem wir auch die Verbindung zwischen unserem Azure Funktion und der Azure Blob Storage herstellen. Die nächsten zwei Parametern „ILogger, ExecutionContext“ sind nur für Erstellen von Logs notwendig.



Abbildung 4: Azure Funktion, um Händlern zu erstellen

Dann werden alle Parameter geprüft, weil Sie beim ausführen von der Azure Funktion nicht „null“ sein dürfen.

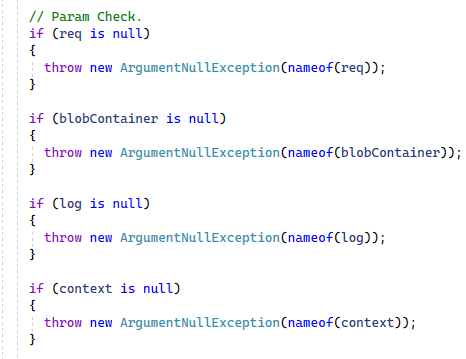


Abbildung 5: Parametern nach Null Prüfungen

Hier werden die wichtigen Objekte/Attribute initialisiert.

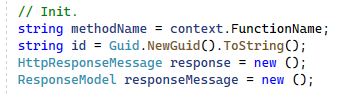


Abbildung 6: Initialisierung von Attributen und Objekten

Die Zeile enthält drei Funktionen, einmal liest sie den „RequestBody“ der von außen gekommen ist und dann Mappt sie gleich den „RequestBody“ mit unserer „MerchantAccountModel“ klasse, die wir oben bereits implementiert haben. Anschließend mit „Result“ geben wir den Rückgabewert synchron zurück.



Abbildung 7: Einlesen von Inhalt der Anfrage

Dann wird eine angemessene Antwort zurückgegeben falls der Benutzer einen leeren „RequestBody“ schickt.



Abbildung 8: Prüfen ob der Inhalt etwas enthält

Als nächstes wird aus dem „Request Body“ die ID (Identifikator) übergeben und unter Blob Storage ein neuer Container namens „merchants“ erstellt. Der ebenfalls, eine Rückmeldung zurückgibt, falls etwas schiefgelaufen ist.

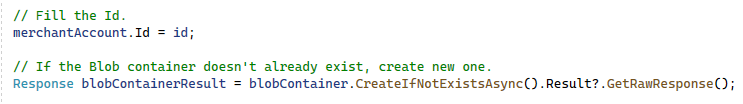


Abbildung 9: Einen Blob Container erstellen

Nach Erstellen eines Containers muss als nächstes die Händler Daten in Container abgelegt werden. Dazu wird zuerst ein neues Objekt von „MemoryStream“ erstellt werden, dann wird ein „BlobClient“ mit jeweiligen Namen „ID“ erstellt. Dann wird der „MemoryStream“ mit dem Inhalt von „RequestBody“ geschrieben und auf die Position „0“ gesetzt, weil nach reinschreiben, befindet sich



Abbildung 10: Händler in Blob Storage ablegen

der „MemoryStream“ bei der letzten Stelle. Anschließend wird über „UploadAsync“ Methode in den Blob Storage ein neuer Eintrag eingetragen.

Beim erfolgreichen abspeichern von Händler Information wird eine Antwortnachricht definiert, aber auch beim fehlgeschlagenen abspeichern (Exception).

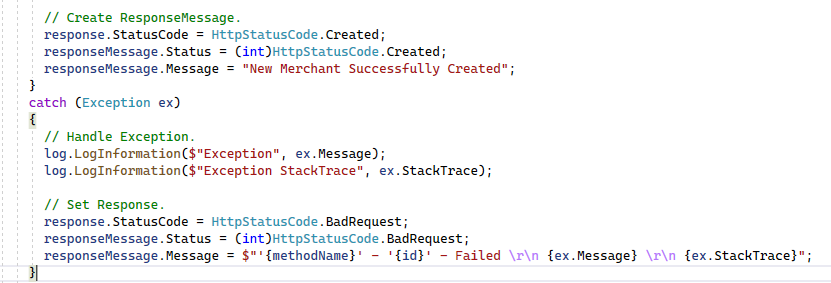


Abbildung 11: Antwortnachrichten definieren

**Read:**

Der Aufbau von der „Read“ Azure Funktion ist ähnlich wie beim „Create“. Der Wesentliche Teil der sich ändern ist, dass wir keinen „Blob Output Binder“ mehr haben, sondern einen „Blob Input Binder“. Mit dem lesen wir alle gespeicherten Blobs die in „merchants“ Container liegen. In dem Fall erwarten wir einen „IEnumerable“ von Datentyp „string“.



Abbildung 12: Azure Funktion, um Händlern einzulesen

Bevor wir die Händler einlesen, werden noch einige Objekte/Attribute initialisiert.

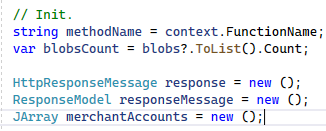


Abbildung 13: Initialisierung von Attributen und Objekten

Der „JArray“ wird später für die Rückgabe benötigt.

Wenn es welche Container gefunden sind, dann werden die alle durchgegangen und der „JArray“ Liste über die „Add“ Methode hinzugefügt, davor werden die noch als „JObject“ umgewandelt, damit später korrekter JSON Format ausgegeben werden kann.

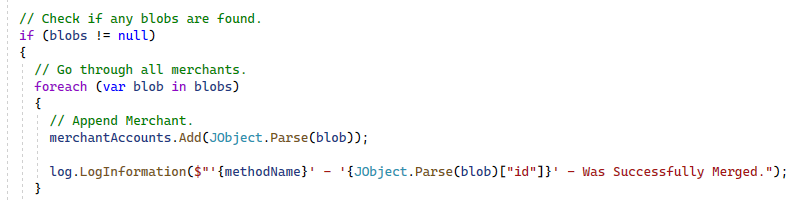


Abbildung 14: Einen JSON von gefunden Händlern abspeichern

**Update:**

Beim „Update“ Klasse, wird beim „Blob Output Binder“ die „id“ sowohl beim Pfad von dem „Http Request“ als auch Param definiert. Das hat den Hintergrund, dass der Benutzer beim Auslösen der Funktion, eine „id“ mitgeben sollen. Somit kann man später mit der „id“ den jeweiligen Händler aus dem Blob Storage rausziehen und zurückgeben.

****

Abbildung 15: Azure Funktion, um Händler aktualisieren

Sowie beim „Create“ wird zuerst auch der Inhalt von „RequestBody“ eingelesen. Dann beim Aktualisieren, haben wir bereits beim „Blob Output Binder“ den Pfad mit der „id“ übergeben „merchants/{id}“. Deshalb beim Einlesen können wir den gleichen Schritten folgen, wie beim „Read“.

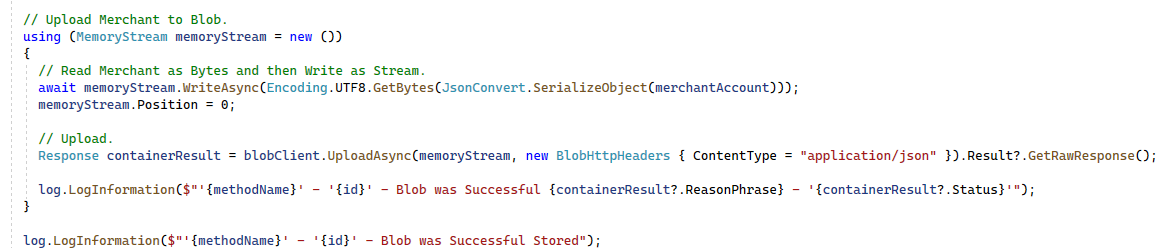


Abbildung 16: Aktualisierten Händler auf Blob Storage hochladen

**Delete:**

Die „Delete“ Funktion sieht genau gleich wie die „Update“ Funktion, weil wir den „BlobClient“ mit der richtiger „id“ benötigen und schon kann der Container mit dem Händler Inhalt gelöscht werden.



Abbildung 17: Azure Funktion, um Händler zu löschen

Mit der „DeleteIfExistAsync“ Methode löschen wir unseren benötigten Container, denn wir beim „id“ mitgeben, welcher gelöscht werden soll.

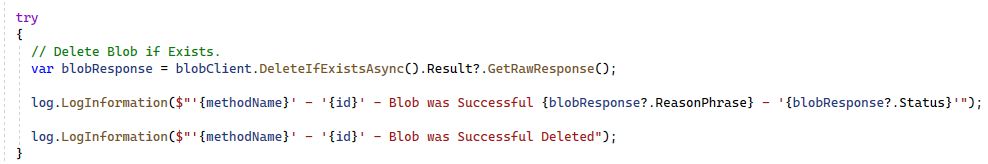


Abbildung 18: Händler aus dem Blob Storage entfernen

### 3.1.3 Unit Tests

Um sicher zu stellen, dass die Azure Funktionen, auch so funktionieren, wie es erwartet wird, wurden es einige Unit Tests erstellen. Die Unit Tests müssen so aussehen und aufgebaut werden, wie die Azure Funktionen, die wir selbst erstellt haben. Alle Parameter müssen entweder gemockt (Simuliert) oder initialisiert werden. Zum Beispiel beim Erstellen von einem neuen Händler sieht der Unit Test folgen maßen aus:

Zuerst werden folgende Attribute deklariert, die beim später fürs testen verwendet werden.



Abbildung 19: Deklaration von Attributen, die für Unit Tests benötigt werden

Für den Blob Storage müssen wir einen gemockten „BlobContainerClient“ deklarieren. Dafür muss die Using-Anweisung „Moq“ eingetragen werden, die uns ermöglicht Objekte zu mocken und auf die Bibliothek zuzugreifen.

Dann initialisieren wir die standartwerte für Attribute/Objekte, die werden in der Vordefinierte Methode „Init“ ermöglicht, weil bevor die „TestMethod“ ausgeführt wird, läuft zuerst die „Init“ Methode durch, um alle notwendige Werte zu setzen.

Wie man einen Mock einstellen und konfigurieren kann findet man beispielsweise unter der Seite: „<https://softchris.github.io/pages/dotnet-moq.html#add-unit-tests>“. Dort wird ausführlich erklärt, wie man „Moq“ nutzen kann und sollte.

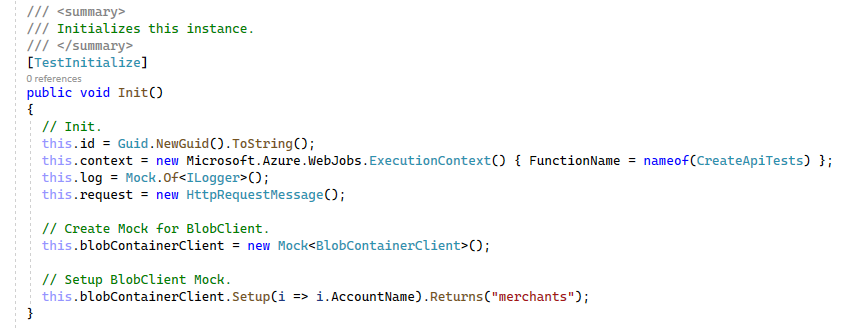


Abbildung 20: Attribute und Objekte Initialisieren

Anschließend folgt die „TestMethod“ die unsere Azure Funktion mit richtigen Parametern ausführt. Davor wird der „RequestMessage“ vorbereitet mit den Händler Beispiel Daten und andere Einstellungen. Am Ende sind dann die „Assert“ zu sehen, die sagen aus, ob der Test erfolgreich durchgelaufen ist. Einmal wird hier geprüft ob „result“ nicht „null“ ist und ob der Status code den „201 Created“ entspricht.



Abbildung 21: Unit Tests Beispiel, um Azure Funktion zu testen

### 3.1.4 Implementierung des Frontends

Der Frontend wurde wie bereits erwähnt mit Angular umgesetzt. Die Startseite Ansicht, besteht aus statischen Html, also aus einer Überschrift, Text und ein paar Icons.

Der Wesentliche Schwierigkeit bestand in der Händler Verwaltungsanwendung, die zum einen eine Tabelle dargestellt hat und mehrere Funktionen enthält, auf die später unten genauer eingegangen wird.

**Interface:**

Es wird ein Interface für die Händler verwendet, der enthält Eigenschaften, die auch aus Fronend Ansicht ins Backend geschickt werden. Unten wird noch eine Methode definiert um Standartwerte für die Händler zu erstellen.

Ein Bild, das Text enthält.

Automatisch generierte Beschreibung

Abbildung 22: Ein Interface für die Händler

Es wird ein Service erstellt, in dem die Verbindung zwischen Backend und dem Frontend aufgebaut wird. Hier findet man alle Anfragen, die an die Azure Funktionen geschickt werden.

Zuerst werden wichtige Attribute initialisiert, wie der Host, des Headers usw. Dann ist gleich eine Methode zu sehen, die setzt den Händler Objekt, der von der Ansicht später ausgewählt wird.

Ein Bild, das Text enthält.

Automatisch generierte Beschreibung

Abbildung 23: Erstellen eines Service, um die Verbindung zwischen Frontend und Backend herzustellen

Als nächsten werden die CRUD Operatoren nacheinander implementiert. Diese werden später in den Ansichten beim Klick Aktionen oder beim Laden der Seite benutzt. Die Anfragen lösen unseren backend Logik.

Ein Bild, das Text enthält.

Automatisch generierte Beschreibung

Abbildung 24: CRUD Operationen

So sind der Parent und Child Komponente aufgebaut. Der Parent Komponente „merchant-acconts“ enthält nur eine Tabelle, bei den einzelnen Händlern aufgelistet sind. Dann auf einen Klick Button wird der Child Komponente „merchant-card“ aufgerufen, diese wiederum enthält einzelne Eigenschaften des Händlers, die man manipulieren kann.

Ein Bild, das Text enthält.

Automatisch generierte Beschreibung

Abbildung 25: Komponente Struktur

**Parent Komponente:**

Hier wird zunächst die Händler initialisiert und bei Konstruktor wird der Service deklariert, damit können wir auf unseren Erstellen Service zugreifen.

Ein Bild, das Text enthält.

Automatisch generierte Beschreibung

Abbildung 26: Eltern Komponente - Initialisierung von Attributen und Service

Beim ersten Mal laden der Webseite wird sogenanntes „ngOnInit lifecycle“ ausgeführt. Somit wird beim laden der Ansicht, eine Anfrage an unserer „Read“ Azure Funktionen gemacht. Somit werden hier alle gefundene Händlern gelesen.

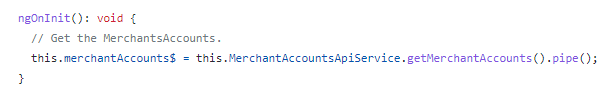


Abbildung 27: Aufruf zum Backend, um Händler Liste zu bekommen

Letzt endlich wird beim Button Klick auf „Edit“ die Methode ausgelöst, diese setzt den ausgewählten Händler, damit wir bei der anderen Komponente auf ausgewählten Händler zugreifen können.

Ein Bild, das Text enthält.

Automatisch generierte Beschreibung

Abbildung 28: Methode die einen Händler hinzufügt

In der Ansicht, wird hier die Tabelle definiert, die drei Spalten enthält und mit dem „ngFor directive“ können wir die gelesene Liste aus Backend durchgehen und einzelne Eigenschaften darstellen. In unseren Fall wird nur die „id“ dargestellt und jeder Händler bekommt noch einen „Edit“ Button, der auf unsere Child Komponente weiterleitet. Der Child Komponente übernimmt den ausgewählten Händler und die wechselt die Route des Browsers auf „/merchant/{id}“.

Ein Bild, das Text enthält.

Automatisch generierte Beschreibung

Abbildung 29: Ansicht von der Tabelle mit Händlern

**Child Komponente:**

Die enthält zu nächst die „Form“ Eigenschaft, damit wir die Inputs und Labels nutzen können, die „merchantAccountForm“ wird unten beim TypeScript definiert.

Es folgen ein paar „ng-containers“ die einfach eine Bedingung enthalten, sobald die erfühlt ist, wird auch der Input angezeigt, dass hat den Hintergrund, dass ich mir gespart habe zwei Mal Formular fürs Erstellen und Aktualisieren von Händlern zu implementieren. Somit kann ich prüfen, wenn die „id“ bei der Route von Browser gefunden wird, dann kann es nur bedeutet, dass der Händler ausgewählt wurde und man Ihn bearbeiten möchte.

Ein Bild, das Text enthält.

Automatisch generierte Beschreibung

Abbildung 30: Formular aus Daten von Händlern

Mit Hilfe von Framework Bootstrap, erstelle ich eine Reihe aus jeweils zwei Spalten mit Label und vier Spalten mit Input. Damit erreiche, dass der Label und der Input nebeneinanderstehen. Für jeden Input gibt es eine „inputChanded“ Methode, die wird mithilfe Angular Binder „ngModelChange“ immer ausgeführt, wenn sich der Input Feld geändert hat. So kann ich den Input in Typescript prüfen und den Benutzer Bescheid geben, ob das was er schreibt auch richtigen Format enthält.

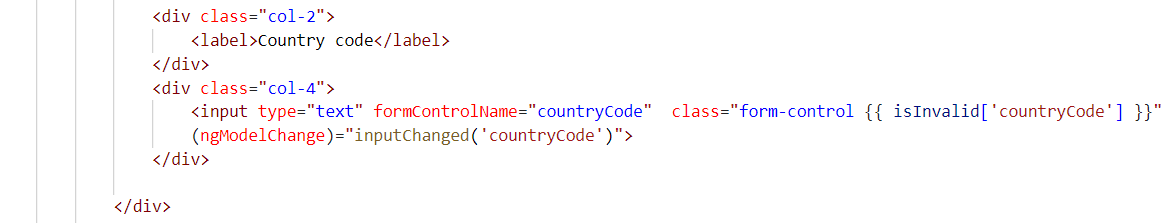


Abbildung 31: Hinzufügen von "countryCode" Eigenschaft

Beim „ngOnInit“ werden einige Dinge geprüft und initialisiert. Das war der Check mit der „id“, bei dem ich sicherstellen kann, ob der Benutzer bei der Ansicht, auf den „Edit“ oder „Create“ Button geklickt hat. Unterandrem wird der Händler über den Service geholt, falls die „id“ existiert.

Ein Bild, das Text enthält.

Automatisch generierte Beschreibung

Abbildung 32: Typescript - Id nach Existenz prüfen

Wie vorhin schon erwähnt es wird eine Form erstellt und auf den „merchantAccountForm“ Referenziert, somit können wir die Form in der Ansicht verwenden. Es wird eine „Form Group“ erstellt, in der wie sagen können wie der einzelnen Inputs sich zu verhalten haben.

Ein Bild, das Text enthält.

Automatisch generierte Beschreibung

Abbildung 33: React Formular - Validierung von einzelnen Feldern

Die Methode liefert ob der Input den „Validators.pattern oder Validators.required“ entspricht, wenn ja dann werden nur zwei Zeichenketten benötigt, die mithilfe Bootstrap visuell ausmachen, ob das Input Feld rot oder grün leuchten soll.

Anschießend wird durch die „createButton“ über den Service ein Händler erstellt. Hier wird ebenfalls eine Anfrage an den Backend gesendet, das gleiche gilt für „deleteButton und updateButton“

Ein Bild, das Text enthält.

Automatisch generierte Beschreibung

Abbildung 34: Methoden für CRUD Operationen

## 3.2 Logging mit Hilfe von ILogger

Die Logs die beim Backend erstellt wurde, dienen zu rausfiltern von Fehlern und zur Prüfung, ob die Aktion Erfolgreich ausgeführt wurde. Unten sehen Sie ein paar Aktionen, wie das ganze momentan aussieht:

Hier kommen Sceenschots von mein ILogger….

## 3.3 Qualitätssicherung

Um die Qualität zu garantieren und sicherzustellen, dass zwischen den Prozessen alles sauber läuft, wurde viele Prozesse Manuel mit Hilfe von Visual Code (Mit Erweiterung von „http“), die können ebenfalls über bekannten „Postman“ gemacht werden. Für die Einzelne Funktionen in Backend wurden Unit Tests geschrieben. Es wurden keine automatisierten Tests geschrieben, die von Anfang des Prozesses und bis Ende des Prozesses testen, aus den Zeitlichen Einschränkungen. Deswegen musste ich Manuel an die Buttons klicken und davon ausgehen, dass bei nächster Änderung nichts kaputt geht.

Kommt noch ein Screen von VS mit http Request!!!!!!!!!!!!

# Projektergebnis

## Zeitlicher Rahmen

Es konnten alle Muss-Kriterien umgesetzt werden, bis auf einen. Die Händler Liste ist bearbeitbar und man kann einen Händler löschen und aktualisieren. Wegen den Zeitlichen Rahmen, konnte der Popup mit den Fehler Meldungen nicht umgesetzt werden, aber diese Fehlermeldungen, sind unter den backend Logs sichtbar und kann nach Bedarf jeder Zeit erweitert werden.

## Soll- /Ist-Vergleich

Die Soll-Kriterien konnten ebenfalls alle umgesetzt werden. Die Webanwendung läuft lokal einwandfrei, die Kommunikation zwischen den Ansichten von Webanwendung und die backend Logik funktioniert ebenfalls. Es werden einzelnen Händlern in Blob Storage abgelegt und gepflegt, so dass die jeder Zeit abgerufen und bearbeitet werden können.

Bei dem Wunsch-Kriterien sieht das etwas anders aus, es konnte nur teilweise umgesetzt werden, die Mobile Design ist übriggeblieben und in alle anderen Größen sieht die Webanwendung gut aus. Was die Ladezeit angeht konnte leider Icon nicht implementiert werden und was die CSV-Protokolle angeht, ebenfalls zu wenig Zeit dafür gehabt.

# Benutzerhandbuch

## Vorwort

Um den Managementportal bestmöglich und fehlerfrei nutzen können, wäre folgende Schritte wichtig zu prüfen.

Prüfen Sie ob folgende Frameworks/ Laufzeitumgebungen installiert sind.

1. Die Version von Node.js sollte entweder höher oder 16 sein. Um zu prüfen welche Version installiert ist, geben Sie bitte in CMD/Powershell **node -v** ein. Falls Sie einen installieren müssen, finden sie unter den Link: <https://nodejs.org/en/download/> einen Node.js Reiter.
2. Git installieren unter den Link: <https://git-scm.com/downloads>
3. VS CODE? VS? Zum Ausführen von Azure Functions -> muss ich mir noch anschauen

## Starten

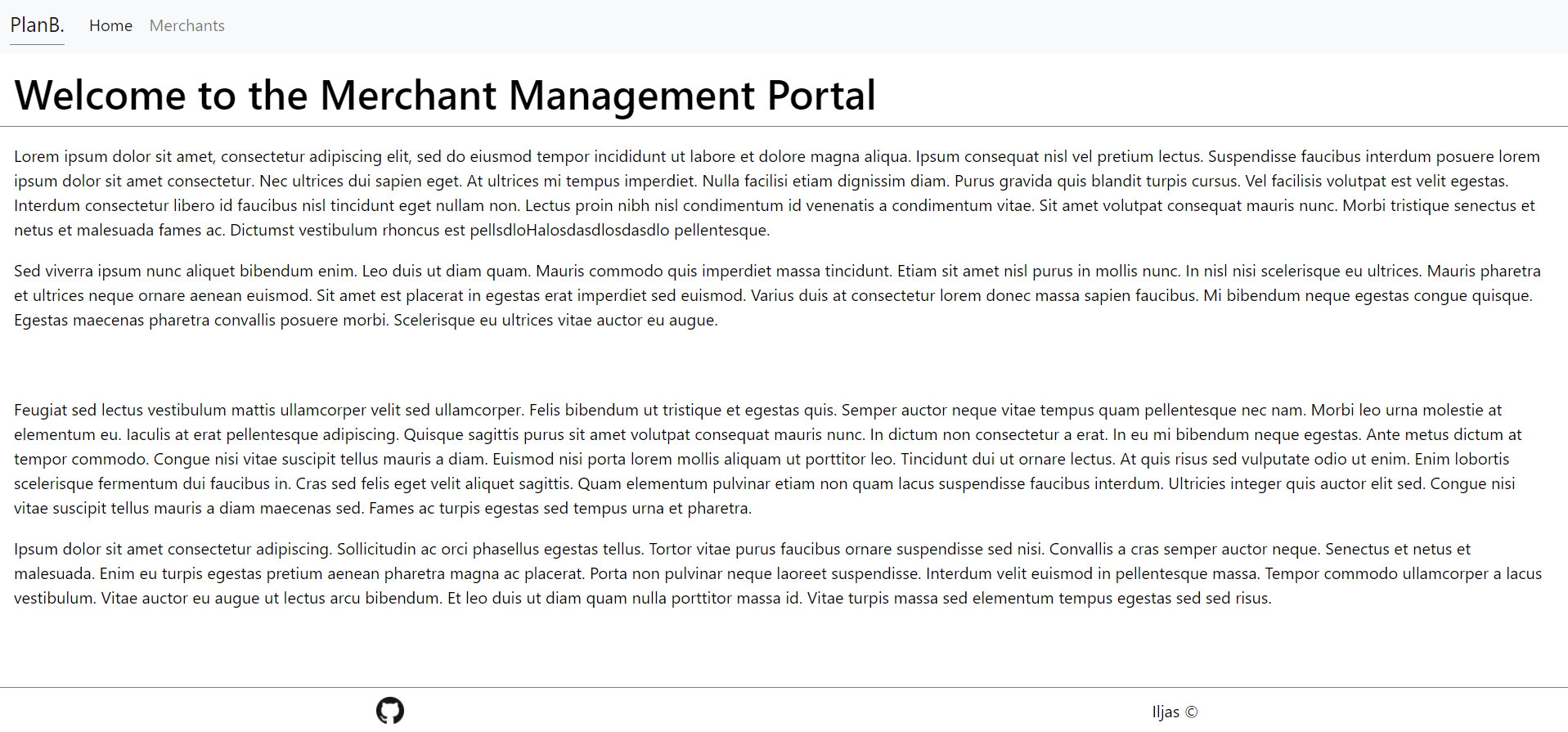
serve --o

func start

Muss ich mir überlegen, wie man mit wenig Aufwand alles starten kann…

## Navigation

### Startseite

Nach dem Sie die Webanwendung und die Azure Funktionen gestartet haben. Werden Sie in Ihren Standartbrowser die Startseite sehen, die wir bei dem letzten Schritt ausgeführt haben.

Navigation, zwischen der Sie frei navigieren könnt

Bei der Navigation stehen Ihnen zwei Reitern zu verfügen. Der „Home“ Reiter leitet Sie wieder auf die Startseite zurück und der „Merchant“ Reiter leitet Sie zu der Verwaltungsliste von Händlern.

## 5.4 Übersichtsseite der Händler

Wenn Sie zu den Händlern navigieren, dann können Sie folgenden Ereignissen sehen.

Ein Bild, das Tisch enthält.

Automatisch generierte Beschreibung

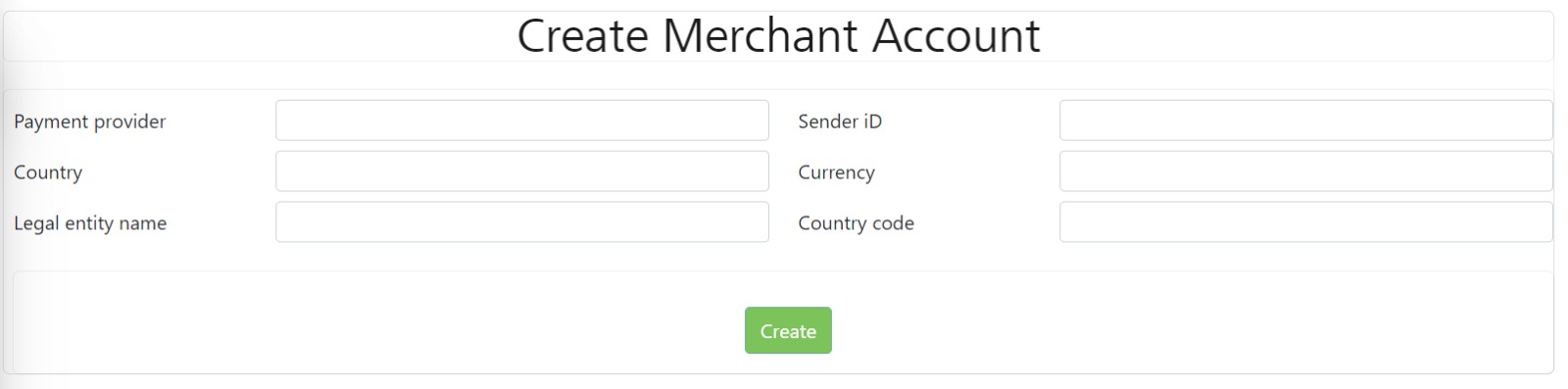
3

2

1

Auf der Linkte Seite (Markierung 1) können Sie den Händlern aufgelistet sehen, die werden von aufsteigend nummeriert und Ihr könnt auch Ihre Id (Merchant Id) sehen. Auf der Rechte Seite sehen sie mehrere Buttons zu der Auswahl. Der Button „Create“ (Markierung 2) leitet Sie auf einem leeren Formular weiter, wo sie dann einen neuen Händler erstellen können. Die Buttons mit „Edit“ leiten Sie ebenfalls auf einem Formular der bereits mit Daten ausgefühlt ist. Sie können damit einzelnen Händlern verwalten.

## 5.5 Händler Erstellen

Sobald Sie auf „Create“ geklickt haben, werden Sie ebenfalls weitergeleitet und Sie können ein leeres Formular sehen. Der besteht wie herkömmlicher Formulare aus Labels/Inputs und einen Button.

D:\downloads\WhatsApp Image 2022-05-04 at 15.05.54.jpegBei manchen Feldern wurden auch geprüft, ob die null sind oder ob die bestimmte Länge haben sollten. Beispiel an dem Feld „Currency“, der sollte minderst und höchstens aus drei Zeichen bestehen.

Sobald der richtig ausgefüllt ist, wird auch dementsprechend die Ansicht grün leuchten.

Sobald man alle Felder ausgefühlt hat, wird beim Klicken von „Create“ ein neuer Händler erstellt.

## 5.6 Händler Bearbeiten

Die Händler können bearbeitet werden, also einzelne Felder können aktualisiert werden. Nachdem gewünschte Felder bearbeitet wurden, müssen sie mit dem Klick auf Button „Update“ bestätigen. Wenn Sie vorhaben, einen Händler zu löschen, die erfolgt durch den „Delete“ Button.

Ein Bild, das Text enthält.

Automatisch generierte Beschreibung

# Tabellenverzeichnis

[Tabelle 1: Projektphasen 5](file:///C:\Users\49157\Desktop\ProjektArbeit\MerchantManagement\Doku\Dokumentation.docx#_Toc102573701)

[Tabelle 2: Arbeitsmittel 5](#_Toc102573702)

[Tabelle 3: Personalkosten 5](#_Toc102573703)

[Tabelle 4: Prozessschritte 7](file:///C:\Users\49157\Desktop\ProjektArbeit\MerchantManagement\Doku\Dokumentation.docx#_Toc102573704)

# Abbildungsverzeichnis

[Abbildung 1: PlanB. GmbH 3](file:///C:\Users\49157\Desktop\ProjektArbeit\MerchantManagement\Doku\Dokumentation.docx#_Toc102573667)

[Abbildung 2: MerchantAccountModel in Form einer Klasse 9](file:///C:\Users\49157\Desktop\ProjektArbeit\MerchantManagement\Doku\Dokumentation.docx#_Toc102573668)

[Abbildung 3: ResponseModel in Form einer Klasse 10](file:///C:\Users\49157\Desktop\ProjektArbeit\MerchantManagement\Doku\Dokumentation.docx#_Toc102573669)

[Abbildung 4: Azure Funktion, um Händlern zu erstellen 11](#_Toc102573670)

[Abbildung 5: Parametern nach Null Prüfungen 11](#_Toc102573671)

[Abbildung 6: Initialisierung von Attributen und Objekten 12](#_Toc102573672)

[Abbildung 7: Einlesen von Inhalt der Anfrage 12](#_Toc102573673)

[Abbildung 8: Prüfen ob der Inhalt etwas enthält 12](#_Toc102573674)

[Abbildung 9: Einen Blob Container erstellen 12](#_Toc102573675)

[Abbildung 10: Händler in Blob Storage ablegen 13](file:///C:\Users\49157\Desktop\ProjektArbeit\MerchantManagement\Doku\Dokumentation.docx#_Toc102573676)

[Abbildung 11: Antwortnachrichten definieren 13](#_Toc102573677)

[Abbildung 12: Azure Funktion, um Händlern einzulesen 14](#_Toc102573678)

[Abbildung 13: Initialisierung von Attributen und Objekten 14](#_Toc102573679)

[Abbildung 14: Einen JSON von gefunden Händlern abspeichern 14](#_Toc102573680)

[Abbildung 15: Azure Funktion, um Händler aktualisieren 15](#_Toc102573681)

[Abbildung 16: Aktualisierten Händler auf Blob Storage hochladen 15](file:///C:\Users\49157\Desktop\ProjektArbeit\MerchantManagement\Doku\Dokumentation.docx#_Toc102573682)

[Abbildung 17: Azure Funktion, um Händler zu löschen 16](#_Toc102573683)

[Abbildung 18: Händler aus dem Blob Storage entfernen 16](#_Toc102573684)

[Abbildung 19: Deklaration von Attributen, die für Unit Tests benötigt werden 17](#_Toc102573685)

[Abbildung 20: Attribute und Objekte Initialisieren 18](#_Toc102573686)

[Abbildung 21: Unit Tests Beispiel, um Azure Funktion zu testen 18](file:///C:\Users\49157\Desktop\ProjektArbeit\MerchantManagement\Doku\Dokumentation.docx#_Toc102573687)

[Abbildung 22: Ein Interface für die Händler 19](#_Toc102573688)

[Abbildung 23: Erstellen eines Service, um die Verbindung zwischen Frontend und Backend herzustellen 20](#_Toc102573689)

[Abbildung 24: CRUD Operationen 20](#_Toc102573690)

[Abbildung 25: Komponente Struktur 21](#_Toc102573691)

[Abbildung 26: Eltern Komponente - Initialisierung von Attributen und Service 21](#_Toc102573692)

[Abbildung 27: Aufruf zum Backend, um Händler Liste zu bekommen 21](#_Toc102573693)

[Abbildung 28: Methode die einen Händler hinzufügt 22](#_Toc102573694)

[Abbildung 29: Ansicht von der Tabelle mit Händlern 22](#_Toc102573695)

[Abbildung 30: Formular aus Daten von Händlern 23](#_Toc102573696)

[Abbildung 31: Hinzufügen von "countryCode" Eigenschaft 23](#_Toc102573697)

[Abbildung 32: Typescript - Id nach Existenz prüfen 24](#_Toc102573698)

[Abbildung 33: React Formular - Validierung von einzelnen Feldern 24](#_Toc102573699)

[Abbildung 34: Methoden für CRUD Operationen 25](#_Toc102573700)

# Glossar

Die einzelnen Begriffe müssen noch erklärt werden.