1.Mai.2022



Entwicklung einer Prototyp -Webanwendung 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](#_Toc102674184)

[1.1 Projektumfeld 3](#_Toc102674185)

[1.2 Projektziel 4](#_Toc102674186)

[1.3 Projektbegründung 4](#_Toc102674187)

[1.4 Projektschnittstellen 4](#_Toc102674188)

[1.5 Projektabgrenzung 4](#_Toc102674189)

[1.6 Ansprechpartner 4](#_Toc102674190)

[2. Projektplanung 5](#_Toc102674191)

[2.1 Projektphasen und Ablaufplan 5](#_Toc102674192)

[2.2 Ressourcenplanung 5](#_Toc102674193)

[2.3 Pflichtenheft 6](#_Toc102674194)

[2.3.1 Ist-/Soll-Analyse 6](#_Toc102674195)

[2.3.2 Musskriterien 6](#_Toc102674196)

[2.3.3 Wunschkriterien 7](#_Toc102674197)

[2.3.4 Prozessschritte 7](#_Toc102674198)

[2.4 Vorbereitungsphase 8](#_Toc102674199)

[2.4.1 Kick-off-Meeting 8](#_Toc102674200)

[2.4.2 Auswahl des Frameworks 8](#_Toc102674201)

[2.4.3 Auswahl der Azure Services 8](#_Toc102674202)

[3. Durchführung 8](#_Toc102674203)

[3.1 Implementierungsphase 8](#_Toc102674204)

[3.1.1 Servicebereitstellung 8](#_Toc102674205)

[3.1.2 Implementierung des Backends 9](#_Toc102674206)

[3.1.3 Unit Tests 17](#_Toc102674207)

[3.1.4 Implementierung des Frontends 19](#_Toc102674208)

[3.2 Logging mit Hilfe von ILogger 25](#_Toc102674209)

[3.3 Qualitätssicherung 26](#_Toc102674210)

[4. Projektergebnis 28](#_Toc102674211)

[4.1 Zeitlicher Rahmen 28](#_Toc102674212)

[4.1 Soll- /Ist-Vergleich 28](#_Toc102674213)

[5. Benutzerhandbuch 28](#_Toc102674214)

[5.1 Vorwort 28](#_Toc102674215)

[5.2 Starten 29](#_Toc102674216)

[5.3 Navigation 30](#_Toc102674217)

[5.3.1 Startseite 30](#_Toc102674218)

[5.4 Übersichtsseite der Händler 30](#_Toc102674219)

[5.5 Händler Erstellen 31](#_Toc102674220)

[5.6 Händler Bearbeiten 32](#_Toc102674221)

[6. Tabellenverzeichnis 33](#_Toc102674222)

[7. Abbildungsverzeichnis 33](#_Toc102674223)

[8. Glossar 35](#_Toc102674224)

[9. Anlagen 37](#_Toc102674225)

# 1. Einleitung

## 1.1 Projektumfeld

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

Das Projekt ESB („Enterprise Service Bus“) ist ein Beispielprojekt, das die PlanB. GmbH seit Anfang 2017 bis heute fortlaufend betreut und weiterentwickelt. Der ESB ermöglicht die Kommunikation zwischen verschiedenen Endpunkten an, so dass Systeme in der Lage sind, mit anderen Systemen zu kommunizieren.

Abbildung : PlanB. GmbH

## 1.2 Projektziel

Für ein internes Projekt soll eine Webanwendung entwickelt werden, um Händler, Benutzer und Bezahlvorgänge verwalten zu können. Ich implementiere die Verwaltung von Händlerdaten. Ein Händler hat typischerweise folgende Eigenschaften: Zahlungsmethoden, Anbieter eines Zahlungsprozesses, Standort, Währung und Firmennamen. 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, erstellen sowie 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 verwendet 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, das Zugriffe auf einen „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.

# 2. Projektplanung

## 2.1 Projektphasen und Ablaufplan

Das Managementportal, mit dem man den Händlern verwalten kann, sollte innerhalb 70 Stunden umgesetzt werden. Die Umsetzung des Projektes erfolgte mit der Absprache dem Product Owner.

Anbei sind die Projektphasen inklusive Zeitaufwand für jeden Arbeitsschritt aufgelistet:

|  |  |
| --- | --- |
| **Tätigkeit** | **Aufwand in Stunden** |
| Analyse der Anforderungen, Projektplan  und Pflichtenheft | 4 |
| Ist-/Soll-Analyse | 2 |
| Backend | 25 |
| Frontend | 19 |
| Tests | 2 |
| Besprechung und Abnahme | 2 |
| Erstellung der Dokumentation | 16 |
| **Summe** | **70** |

Tabelle : Projektphasen

## 2.2 Ressourcenplanung

Für die Entwicklung der Webanwendung zur Verwaltung von Händlern, wurden folgende Ressourcen benötigt.

Arbeitsmittel:

|  |  |
| --- | --- |
| **Arbeitsmittel** | **Anzahl** |
| Laptop | 1 |
| Arbeitsplatz | 1 |
| Internetanschluss | 1 |
| Visual Studio Community | 1 |
| Visual Studio Code | 1 |
| **Summe** | **5** |

Tabelle : Arbeitsmittel

Personalkosten:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Name** | **Position** | **Kosten/h** | **Zeitaufwand** | **Kosten** |
| Iljasgadzhi Makhsunov | Entwickler | 47,50 € | 70 | 3.325,00 € |
| Wendelin Niesl | Product Owner | 95,00 € | 4 | 380,00 € |
| **Summe** |  |  |  | **3.705,00 €** |

Tabelle : 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 einzelne Händler zu bearbeiten, aktualisieren und zu löschen und einen neuen zu erstellen. Für die Händlern werden derzeit nur Testdaten verwenden, die ebenfalls lokal abgespeichert werden.

**Sollkriterien:**

Zunächst soll ein Managementportal für die Händler erstellt werden, zu dem müssen einige Aspekte im Vorfeld überlegt werden. Zum einen, welches Frontend Framework für die Umsetzung verwendet wird, wie die Anwendung in der Azure Cloud gehostet werden soll und der Entwurf eines Lösungsvorschlags, der folgende Anforderungen abdeckt: die Verbindungen zwischen den Diensten sowie die Lese- und Schreiboperationen berücksichtigt.

Front End Anforderung:

* Händler müssen in einer eigenen View in tabellarischer Form angezeigt werden.
* Ein Händler muss auswählbar und über einen Prozess bearbeitbar und löschbar sein.
* Ein neuer Händler muss über ein eigenes Oberflächenelement erstellbar sein.

Back End Anforderung:

* Die CRUD Operationen für das Händler Objekt in der Datenbank müssen implementiert werden.
* Beim Erstellen eines Händlers muss diesem eine GUID als ID hinzugefügt werden.
* Der Datenspeicher muss ein Azure Storage Account mit Blob Support sein.

Die einzelnen Funktionalitäten sollten testbar sein und dazu sollte es auch eine Möglichkeit geben, Fehler nachzuverfolgen.

### 2.3.2 Musskriterien

Die Webanwendung wird von Grund auf neu erstellt, deshalb müssen folgende Anforderungen erledigt werden: Es muss eine Startseite mit einer Navigation vorhanden sein, welche auf die jeweiligen Views verweist. Eine View zeigt die komplette Händlerliste an, die man dementsprechend verwalten kann. Zunächst sieht man nur die eindeutige „ID“ des Händlers und einen Button fürs Bearbeiten. Wenn man auf diesen Button klickt, dann sollten die einzelnen Infos zu den Händlern angezeigt werden, die dann entweder aktualisiert und auch gelöscht werden können. Ganz oben finden man noch einen weiteren Button, um einen neuen Händler einzutragen. Außerdem muss beim erfolgreichen/fehlgeschlagenen Bearbeiten oder Löschen von Händlern, visuell ein Popup mit der Fehlermeldung dargestellt werden.

Die Webanwendung wird als „Single Page Application“ im Angular Framework geschrieben. Die Azure Function App sollten mit der aktuellsten .NET Version geschrieben werden, gegenwärtig das .NET 6.0 Framework.

### 2.3.3 Wunschkriterien

Wünschenswert wäre es, dass das Managementportal über ein „responsive layout“ verfügt. Da viele Kunden mobil im Internet surfen, hinterlässt es einen guten Eindruck, wenn Sie anhand des Smartphones auf die Webanwendung zugreifen können und es immer noch so übersichtlich wie auf dem Desktop aussieht.

Um den Ladeprozess im Frontend zu visualisieren, wir ein Ladebalken-Element geschrieben.

Das Frontend sollte über ein Logsystem verfügen, das einen CSV-Export ermöglicht.

### 2.3.4 Prozessschritte

|  |  |  |
| --- | --- | --- |
| **Tätigkeit** | **Arbeitsschritt** | **Aufwand in Stunden** |
| **1. Vorbereitung** | Analyse der Anforderungen | 1 |
|  | Erstellen des  Pflichtenhefts und des  Projektplans | 4 |
|  | Auswahl von Frameworks | 1 |
| **2. Implementierung** | **Backend** |  |
|  | Die Erstellung von Azure Funktionen CRUD Operationen | 15 |
|  | Unit Tests | 2 |
|  | **Frontend** |  |
|  | Erstellen aller benötigten Ansichten | 15 |
|  | Layout anpassen (Bootstrap) | 5 |
|  | Erstellen von Services | 3 |
| **3. Qualitätssicherung** | Tests von CRUD Operationen | 2 |
|  | Managementportal nach Fehlern geprüft | 1 |
| **4. Dokumentation** | Erstellen und Verfassen der Dokumentation | 17 |
|  | Layouts anpassen | 1 |
|  | Erstellung des Benutzerhandbuchs | 3 |
|  |  |  |
| **Summe** |  | **70** |

Tabelle : Prozessschritte

## 2.4 Vorbereitungsphase

### 2.4.1 Kick-off-Meeting

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

### 2.4.2 Auswahl des Frameworks

Es wurde entschieden, dass für die Logik - also für das Backend - Azure Funktionen verwendet werden, die anhand des .NET 6 Frameworks entwickelt werden. Grund dafür war, auf der Azure Cloud bereitstellen kann und sich somit Kosten spart. Man zahlt nur so viel, wie man die Azure Funktion auch benutzt.

Für die Erstellung der Webanwendung wird das Angular Framework verwendet, um den Vorteil von „Single Page Applications“ zu nutzen. Sie beinhalten einen optimierten Entwicklungsprozess und verbrauchen weniger Serverressourcen. Ein weiterer Grund hierfür war, dass ich bereits einige Erfahrungen mit Angular gesammelt habe. Für die Gestaltung wird Bootstrap als Framework verwendet, weil ich dort Expertise vorweise.

### Auswahl der Azure Services

Damit die Händlerinformation gespeichert, aktualisiert, gelöscht und abgerufen werden, wird der Azure Blob Storage verwendet. Dies hat folgenden Gründe:

1. Um Kosten zu sparen, weil im Vergleich zu SQL-Datenbanken, Serverkosten entfallen und lediglich der Datenspeicher sowie Lese- und Schreibvorgänge gezahlt werden muss.
2. Die Händlerdaten werden im JSON Format gespeichert, womit eine Planung und Festlegen auf das Schema der Datenbank entfällt.

# 3. Durchführung

## 3.1 Implementierungsphase

### 3.1.1 Servicebereitstellung

Das Backend läuft Lokal, in welcher die Verbindung zum „Azure Blob Storage“ emuliert ist. Dafür muss zunächst der „Azure Storage Explorer“ installiert werden und in den Azure Funktionen unter der „local.settings.json“-Datei sollte „UseDevelopmentStorage“ auf „true“ gesetzt werden. Somit wird die Verbindung lokal zum Azure Blob Storage aufgebaut.

Damit das Managementportal zukünftig in einer Azure Cloud gehostet werden kann, werden folgende Azure Ressourcen benötigt:

1. App Services -> Webanwendung hosten
2. Azure Blob Storage -> Händler Liste abspeichern
3. Function App -> Auslagerung von Logik mit Hilfe von Azure Funktionen
4. Application Insights -> Überwachung & Protokollierung der Webanwendung

### 3.1.2 Implementierung des Backends

#### 3.1.2.1 Implementierung der Klasse „MerchantAccountModel“

Um die Händlerdaten zu verarbeiten, wird die Klasse „MerchantAccountModel“ implementiert, die in den Azure Funktionen verwendet wird, um ankommende Händler später zu mappen. Die Klasse beschreibt, wie die Händler später auszusehen haben. Sie besteht aus mehreren Eigenschaften und der Syntax „JsonProperty“ dient dem Zweck, dass die Informationen über den Händler klein geschriebene Eigenschaften erkennen und dementsprechend mappen.

Abbildung : MerchantAccountModel in Form einer Klasse

Für die Nutzung der generische Liste wird die Using-Anweisung „System.Collections.Generic“ verwendet und um „JsonProperty“ zu benutzen, wird folgende Using-Anweisung verwendet: „Newtonsoft.Json“.

#### 3.1.2.2 Implementierung der Klasse „ResponseModel“

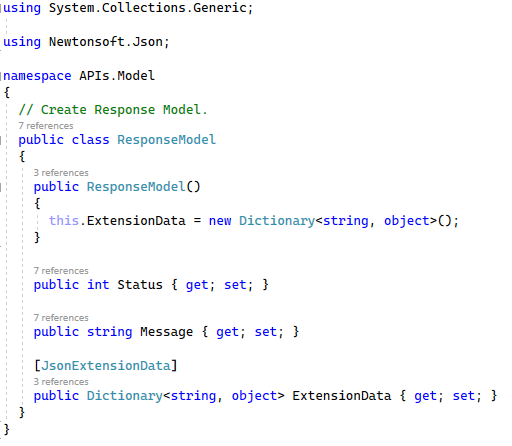
Die Klasse „ResponseModel“ wurde implementiert, um ein einheitliches Schema für Antwortnachricht zu verwenden. Dieses Schema wird bei den Fehlern, Erfolgen und weiteren Aktionen benutzt, um den Nutzer über die Antwort der Schnittstellen zu informieren.

Abbildung : 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 die Erstellung eines neuen Händlers wurde die statische Klasse „Create“ geschrieben, in welcher eine Methode deklariert wird. Darauf folgt immer ein Trigger, in unserem Fall ist das ein „HttpTrigger“, der nur die „Post“ Methode akzeptiert.

Beim Auslösen der Funktion über REST wird eine „HttpRequestMessage“ dem „req“ Parameter zugewiesen, die dann die notwendige Information enthält, welcher „RequestBody“, „Header“ geschickt wurde. Dann folgt der „Blob Output Binder“ „blobContainer“, der auf einen Container im „Azure Blob Storage“ verweist. Die nächsten zwei Parameter „ILogger“ sowie „ExecutionContext“ sind für die Protokollierung von Logs notwendig.

Abbildung : Azure Funktion, um Händlern zu erstellen

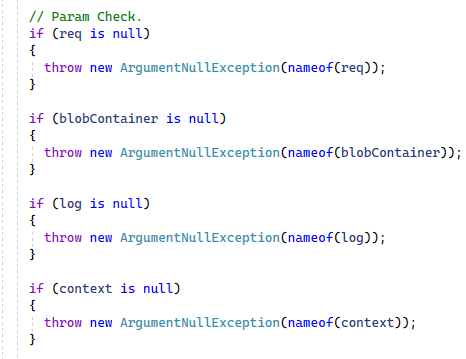
Es werden alle Parameter geprüft, weil Sie beim Ausführen von der Azure Funktion nicht „null“ sein dürfen.

Abbildung 5: Parametern nach Null Prüfungen

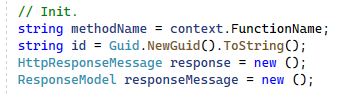
Hier werden die wichtigen Objekte/Attribute initialisiert.

Abbildung : Initialisierung von Attributen und Objekten

Die Zeile in Abbildung 7 enthält drei Funktionalitäten, einmal liest sie den „RequestBody“ des REST Aufrufs und mappt den „RequestBody“ mit unserer „MerchantAccountModel“ Klasse, siehe Abbildung 2. Anschließend mit dem Schlüsselwort „Result“ geben wir den Rückgabewert synchron zurück.

Abbildung : Einlesen von Inhalt der Anfrage, die von außen kommt

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

Abbildung : Prüfen, ob der kommende Inhalt etwas enthält

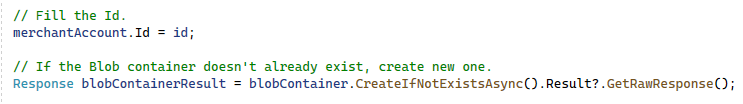
Als nächstes wird aus dem „Request Body“ die ID (Identifikator) gelesen und im „Azure Blob Storage“ ein neuer Container namens „merchants“ erstellt. Der ebenfalls, eine Rückmeldung zurückgibt, falls etwas schiefgelaufen ist.

Abbildung : Einen Blob Container erstellen

Nach Erstellen eines Containers, müssen als nächstes die Händlerdaten im Container abgelegt werden. Dazu wird zuerst ein neues Objekt von „MemoryStream“ erstellt. Als nächstes wird ein neuer "Blob Client" erstellt, damit ein neuer Blob mit dem ausgelesenem „id“ Parameter erzeugt werden kann. Dann wird der „MemoryStream“ mit dem Inhalt des „RequestBody“ geschrieben und auf die Position „0“ gesetzt, weil er sich nach der Schreiboperation an letzter Stelle befindet. Anschließend wird über die „UploadAsync“ Methode in dem „Azure Blob Storage“ ein neuer Eintrag eingetragen.

Abbildung : Händler in Blob Storage ablegen

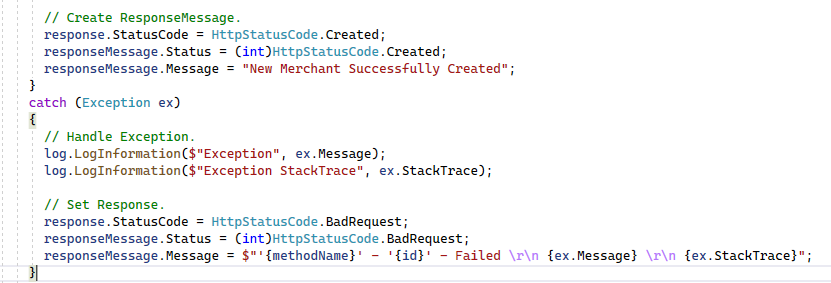
Beim erfolgreichen Abspeichern von Händlerinformationen wird eine Antwortnachricht definiert, so wie auch bei fehlgeschlagenen Speicheroperationen (Exception).

Abbildung : Antwortnachrichten definieren

**Read:**

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

Abbildung : Azure Funktion, um Händlern einzulesen

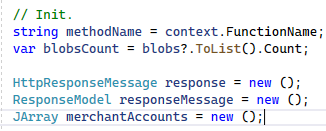
Bevor wir die Händler einlesen, werden noch einige Objekte/Attribute initialisiert.

Abbildung : Initialisierung von Attributen und Objekten

Das „JArray“ wird später für die Rückgabe benötigt.

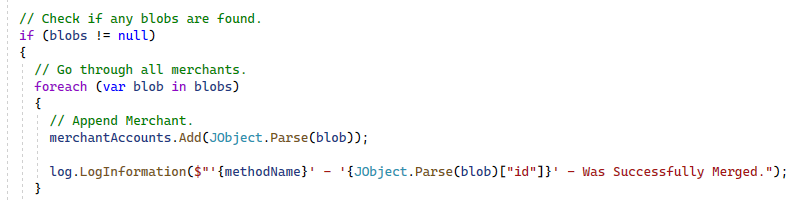
Wenn Einträge im Container gefunden werden, dann werden diese durchgegangen und der „JArray“ Liste über die „Add“ Methode hinzugefügt. Davor werden sie noch als „JObject“ geparsed, um später korrekt im JSON Format ausgegeben werden zu können.

Abbildung : Einen JSON von gefunden Händlern abspeichern

**Update:**

****Bei der statischen „Update“ Klasse, wird bei der „Update“ Methode ein „Blob Output Binder“, die „id“ mit Route definiert. Das hat den Hintergrund, dass der Benutzer beim Auslösen der Funktion, eine „id“ mitliefern wird. Somit kann man später mit der „id“ den jeweiligen Händler aus dem Blob Storage herauslesen und zurückgeben.

Abbildung : Azure Funktion, um Händler aktualisieren

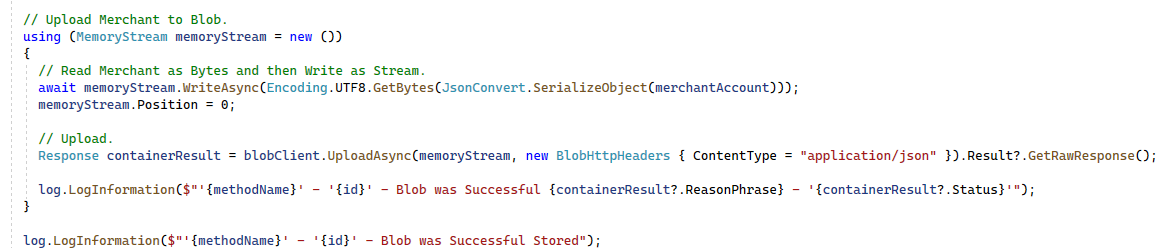
Wie auch beim „Create“, wird zuerst der Inhalt von „RequestBody“ eingelesen. Beim Aktualisieren haben wir bereits beim „Blob Output Binder“ den Pfad mit der „id“ übergeben „merchants/{id}“, deshalb können wir beim Einlesen den gleichen Schritten folgen wie auch beim „Read“.

Abbildung : Aktualisierten Händler auf Blob Storage hochladen

**Delete:**

Die statische „Delete“ Klasse und ihre „Delete“ Funktion wird analog zur „Update“ implementiert, weil wir den „BlobClient“ mit der richtigen „id“ benötigen, um den Händler aus dem Container löschen zu können.

Abbildung : Azure Funktion, um Händler zu löschen

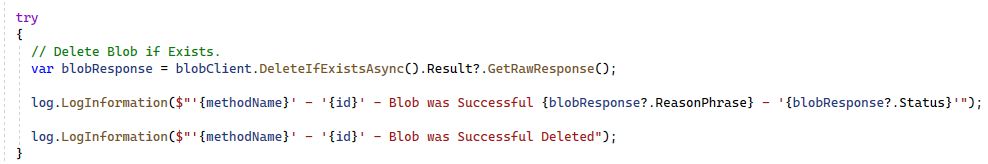
Mit der „DeleteIfExistAsync“ Methode löschen wir den ausgewählten Händler aus dem Container, den wir anhand der „id“ mitgeben.

Abbildung : 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 einige Unit Tests erstellt. Die Unit Tests müssen so aussehen und aufgebaut werden, wie die Azure Funktionen. Alle Parameter müssen entweder gemockt (simuliert) oder initialisiert werden. Zum Beispiel beim Erstellen eines neuen Händlers sieht der Unit Test folgendermaßen aus:

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

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

Für den „Azure Blob Storage“ müssen wir einen gemockten „BlobContainerClient“ deklarieren. Dafür muss die Using-Anweisung „Moq“ eingetragen werden, die ein Simulieren von Objekten und ihres Verhaltens ermöglicht.

Dann initialisieren wir die Standardwerte für Attribute/Objekte, was durch die vordefinierte Methode „Init“ ermöglicht wird. Diese wird jeweils vor der eigentlichen „TestMethod“ ausgeführt, um alle notwendige Werte zu setzen.

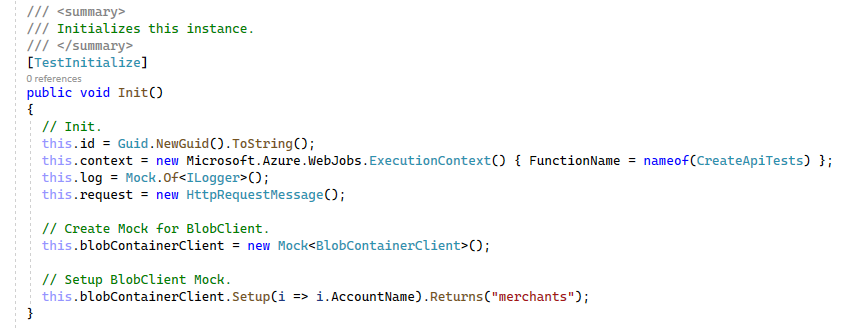
Wie man einen Mock einstellt und konfiguriert, findet man im Glossar.

Abbildung : Attribute und Objekte Initialisieren

Anschließend folgt die „TestMethod“, die unsere Azure Funktion mit den richtigen Parametern ausführt. Davor wird die „RequestMessage“ mit beispielhaften Händlerdaten vorbereitet. Am Ende sind dann die „Asserts“ zu sehen, die anhand der Überprüfung von Resultaten aussagen, ob der Test erfolgreich durchgelaufen ist. Einmal wird hier geprüft, ob „result“ nicht „null“ ist und ob der Statuscode dem erwarteten „201 Created“ entspricht.

Abbildung : Unit Tests Beispiel, um Azure Funktion zu testen

### 3.1.4 Implementierung des Frontends

Das Frontend wurde wie bereits erwähnt mit Angular umgesetzt. Die Startseitenansicht, besteht aus HTML, also aus einer Überschrift, Text und ein paar Icons.

Der wesentlich komplexere Teil der Arbeit bestand in der Entwicklung der Händler-Verwaltungsanwendung, die eine Tabelle dargestellt hat und mehrere Funktionen enthält:

* Anzeige der Händlerliste
* Bearbeiten, Erstellen und Löschen Button

Ein Bild, das Text enthält.

Automatisch generierte Beschreibung**Interface:** Es wird ein Interface für die Händler verwendet, das Eigenschaften enthält, die aus dem Frontend ins Backend geschickt werden. Unten wird noch eine Methode definiert, um Standardwerte für die Händler zu erstellen.

Abbildung : 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.

Ein Bild, das Text enthält.

Automatisch generierte BeschreibungZuerst werden wichtige Attribute initialisiert, wie der Host, die Header und Contenttype. Dann ist eine Methode zu sehen, die das Händlerobjekt setzt, der von der Anwendung später ausgewählt werden kann.

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

Ein Bild, das Text enthält.

Automatisch generierte BeschreibungAls nächstes werden nacheinander die CRUD Operatoren implementiert. Diese werden später in den Ansichten bei Klick Aktionen oder beim Laden der Seite verwendet. Die Anfragen lösen die entsprechenden Routen in der backend Logik aus.

Abbildung : CRUD Operationen

Im Folgenden findet sich der Aufbau der Parent und Child Komponenten. Die Parent Komponente „merchant-accounts“ enthält eine Tabelle, in der einzelne Händler aufgelistet sind. Durch einen Klick Button wird die Child Komponente „merchant-card“ aufgerufen, die wiederum einzelne Eigenschaften des Händlers enthält, die manipuliert werden können.

**Parent Komponente:**

Ein Bild, das Text enthält.

Automatisch generierte BeschreibungHier wird zunächst der Händler initialisiert und der Konstruktor des Services deklariert, um auf den erstellten Service zugreifen zu können.

Abbildung : Eltern Komponente - Initialisierung von Attributen und Service

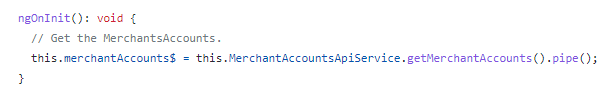
Beim erstmaligen Laden der Webseite wird die Methode „ngOnInit“ ausgeführt. Dadurch wird beim Laden der Ansicht eine Anfrage an unsere „Read“ Azure Funktionen gemacht und alle Händlern angefragt.

Abbildung : Aufruf zum Backend, um Händler Liste zu bekommen

Ein Bild, das Text enthält.

Automatisch generierte BeschreibungLetztendlich wird durch einen Klick auf den Button „Edit“ eine Methode ausgelöst, die den Zugriff auf den ausgewählten Händlern einer anderen Komponente ermöglicht.

Abbildung : Methode die einen Händler hinzufügt

Ein Bild, das Text enthält.

Automatisch generierte BeschreibungIn dieser Ansicht wird eine dreispaltige Tabelle definiert und mit der „ngFor directive“ iterativ die gelesenen Listeneinträge aus dem Backend befüllt. In unserem Fall wird die „id“ dargestellt und jeder Händler wird um einen „Edit“ Button ergänzt, der auf unsere Child Komponente weiterleitet. Die Child Komponente übernimmt den ausgewählten Händler und der URL-Pfad wird im Browser auf „/merchant/{id}“ abgeändert.

Abbildung : Ansicht von der Tabelle mit Händlern

**Child Komponente:**

Die Child Komponente enthält die „Form“ Eigenschaft, damit wir die Inputs und Labels nutzen können, die in der „merchantAccountForm“ im TypeScript definiert sind.

Es folgen „ng-container“, die Input Felder einer Form-Control enthalten. Sobald diese erfüllt ist, wird der Input angezeigt. Das hat den Hintergrund, dass keine zwei Formulare für die Erstellung und Aktualisierung von Händlern erstellt implementiert werden müssen. Somit kann die URL nach dem „id“ Parameter geprüft werden, um eine Bearbeitung am Händler vorzunehmen.

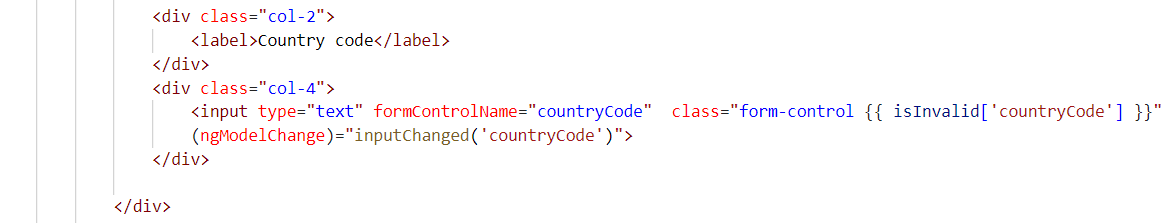
Mit Hilfe des Frameworks „Bootstrap“, erstelle ich jeweils eine Reihe aus zwei Spalten mit Labeln und vier Spalten mit Input-Feldern. Damit wird erreicht, dass das Label und der Input nebeneinander dargestellt werden. Für jeden Input gibt es eine „inputChanged“ Methode, die mithilfe des Angular Binder „ngModelChange“ immer ausgeführt wird, wenn sich das Input Feld geändert hat. So kann ich den Input in Typescript prüfen und den Benutzer benachrichtigen, ob die Eingabe auch im richtigen Format ist.

Abbildung : Hinzufügen von "countryCode" Eigenschaft

Ein Bild, das Text enthält.

Automatisch generierte BeschreibungBeim „ngOnInit“ werden einige Dinge geprüft und initialisiert. Hier angeführt ist der Check mit der „id“, bei dem ich sicherstellen kann, ob der Benutzer bei der Ansicht auf den „Edit“ oder „Create“ Button geklickt hat. Unter anderem wird der Händler via GET Request auf den Webservice angefordert.

Abbildung : Typescript - Id nach Existenz prüfen

Ein Bild, das Text enthält.

Automatisch generierte BeschreibungWie weiter oben bereits erwähnt, wird eine Form erstellt und auf die „merchantAccountForm“ referenziert, womit die Form in der Ansicht verwenden kann. Hierzu wird eine „Form Group“ erstellt, in der wir bestimmen können, wie die einzelnen Inputs sich zu verhalten haben.

Abbildung : React Formular - Validierung von einzelnen Feldern

Die Input Felder werden mit „Validators“ geprüft. Sollte ein Feld nicht gesetzt sein oder der Inhalt das falsche Format haben, wird das Feld rot leuchten. Sollten die „Validators“ keine Fehler finden, wird das grün leuchten. Die Farbgebung wird mit „Bootstrap“ umgesetzt.

Ein Bild, das Text enthält.

Automatisch generierte BeschreibungAnschließend wird durch Klick auf den „createButton“ via POST Request an den Webservice ein Händler erstellt. Selbiges gilt auch für den „deleteButton sowie den updateButton“

Abbildung : Methoden für CRUD Operationen

## 3.2 Logging mit Hilfe von ILogger

Die Logs, die beim Backend erstellt werden, dienen der Protokollierung von Fehlern und zur Prüfung, ob die Aktionen erfolgreich ausgeführt wurden. In den folgenden Abbildungen sind einige Logs aufgelistet, um ein Bild zu vermitteln.

Ein Bild, das Text enthält.

Automatisch generierte BeschreibungZunächst werden alle Azure Funktionen aufgelistet, die geladen wurden.

Abbildung : Backend Ausführung der Azure Funktionen

Sobald eine davon ausgelöst wird, können wir die einzelnen Logs verfolgen.

Ein Bild, das Text enthält.

Automatisch generierte Beschreibung**Create Funktion:**

Abbildung : Create Azure Funktion Logs

Ein Bild, das Text enthält.

Automatisch generierte Beschreibung**Read Funktion:**

Abbildung : Read Azure Funktion Logs

**Ein Bild, das Text enthält.

Automatisch generierte BeschreibungUpdate Funktion:**

Abbildung : Update Azure Funktion Logs

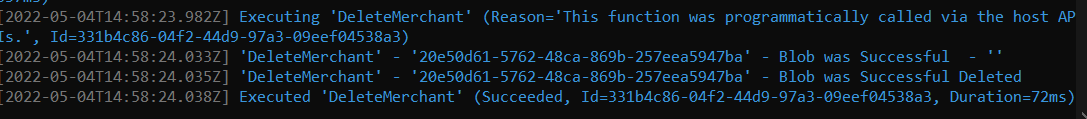
**Delete Funktion:**

Abbildung : Delete Azure Funktion Logs

## 3.3 Qualitätssicherung

Um die Qualität zu garantieren und sicherzustellen, dass zwischen den Prozessen alles sauber läuft, wurde viele Prozesse manuell mit Hilfe von Visual Studio Code (anhand der Erweiterung „REST-Client“) getestet. Natürlich können diese ebenfalls über das weit verbreitete Tool „Postman“ gemacht werden. Für die einzelnen Funktionen im Backend wurden Unit Tests geschrieben. Aufgrund zeitlicher Einschränkungen wurden keine automatisierten End-to-End Tests geschrieben, die den Prozess von Anfang bis Ende testen. Deswegen musste ich manuelle Klick-Tests ausführen und davon ausgehen, dass bei weiteren Änderungen an der Applikation keine Funktionalitäten beeinträchtigt wurden.

**Manuelles testen:**

Ein Bild, das Text enthält.

Automatisch generierte BeschreibungFür die manuellen Tests wurde der Visual Studio Code Editor verwendet. So sieht ein „HTTP-Request“ zur Erstellung eines neuen Händlers aus:

Abbildung : Http Anfrage an Backend mit der Händler Information

Ein Bild, das Text enthält.

Automatisch generierte BeschreibungAnhand der Antwort des Backends, können wir sicherstellen, dass die „Create“ Azure Funktion erfolgreich durchgelaufen ist.

Abbildung : Antwort von Backend aus der Anfrage oben

# 4. Projektergebnis

## 4.1 Zeitlicher Rahmen

Bis auf einen konnten alle Muss-Kriterien umgesetzt werden. Die Händlerliste ist bearbeitbar und man kann einen Händler löschen und aktualisieren. Wegen des beschränkten zeitlichen Rahmens konnte das Popup mit den Fehlermeldungen nicht umgesetzt werden. Diese Fehlermeldungen sind jedoch in den Backend Logs sichtbar und können nach Bedarf jederzeit eingesehen sowie erweitert werden.

## 4.1 Soll- /Ist-Vergleich

Die restlichen Muss-Kriterien konnten umgesetzt werden. Die Webanwendung läuft lokal fehlerlos, die Kommunikation zwischen den Ansichten von Webanwendung und die Backend Logik funktioniert ebenfalls wie gewünscht. Es werden einzelne Händler im „Azure Blob Storage“ abgelegt und gepflegt, so dass sie jederzeit abgerufen und bearbeitet werden können.

Die Wunsch-Kriterien konnten nur teilweise umgesetzt werden. Hier konnte das mobile Design implementiert werden, womit die Webanwendung in allen Bildschirmgrößen übersichtlich dargestellt wird. Es konnte jedoch weder ein Ladebalken-Icon implementiert noch ein CSV-Protokoll erstellt werden – beides ist aus zeitlichen Gründen nicht umsetzbar gewesen.

# 5. Benutzerhandbuch

## Vorwort

Um das Managementportal bestmöglich und fehlerfrei nutzen können, empfiehlt sich im Vorfeld bei einem einen Anwendungsentwickler zu melden, um die Voreinstellungen korrekt vorzunehmen. Grund dafür ist, dass die Webanwendung mit allen Verbindungen lokal konfiguriert und installiert werden muss.

Zuerst sollten einige Dinge geprüft werden:

1. Die Version von Node.js sollte entweder höher oder gleich 16 sein. Um zu prüfen welche Version installiert ist, geben Sie bitte in CMD/Powershell **node -v** ein. Falls Sie Node.js installieren müssen, finden Sie dies unter dem Link: <https://nodejs.org/en/download/>.
2. Nun wird die „Angular CLI“ benötigt, diese können Sie unter dem Befehl installieren: **npm install -g @angular/cli@13**
3. Im Angular Projekt sollten nun die „NPM-Pakete“ via CLI installiert werden: **npm install**
4. Mit dem Befehl können Sie die Webanwendung starten: **ng serve –o**
5. Als nächstes wird der „Azure Storage Explorer“ benötigt, den Sie installieren müssen: <https://azure.microsoft.com/de-de/features/storage-explorer/>
6. Anschließen sollten die Azure Funktionen gestartet werden, davor müssen Sie noch eine „**local.settings.json**“ mit folgendem Inhalt anlegen:

Ein Bild, das Text enthält.

Automatisch generierte Beschreibung

Abbildung : Die Konfiguration "local.setting.json" Datei

Es werden Einstellungen wie die „Azure Blob Storage“ Verbidung verwenden werden sollte eingestellt, in unseren Fall „UseDevelopmentStorage“, der die lokale Verbindung zwischen „Blob Storage Explorer“ aufbaut. Damit die Kommunikation zwischen den Frontend und Backend wegen den CORS nicht scheitert, wird vorerst die „CORS“-Policy auf den Wert „\*“ gesetzt.

## 5.2 Starten

Es müssen die Webanwendung und die Azure Funktionen mit „Blob Storage Explorer“ lokal ausgeführt werden, bevor Sie das Managementportal fehlerfrei nutzen können.

## Navigation

### 5.3.1 Startseite

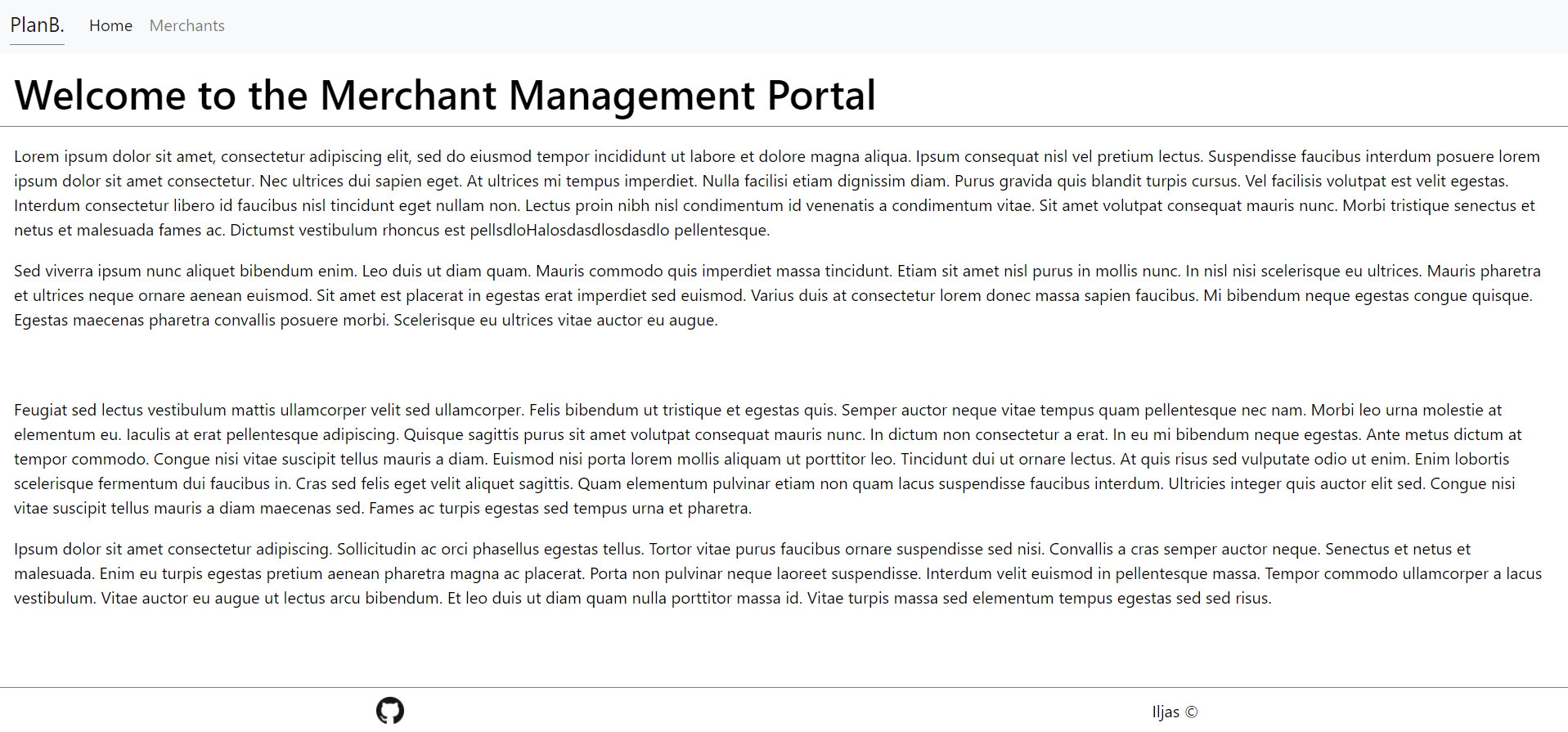
Nach dem Sie die Webanwendung und die Azure Funktionen gestartet haben, werden Sie in Ihren Standardbrowser die Startseite sehen, die wir bei dem letzten Schritt ausgeführt haben.

Abbildung : Startseite des Managementportals

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

Ein Bild, das Tisch enthält.

Automatisch generierte BeschreibungWenn Sie zu den Händlern navigieren, dann können Sie folgenden Ereignissen sehen.

Abbildung : Händler Verwaltungsliste

1

3

2

Auf der linken Seite (Markierung 1) können Sie die Händler aufgelistet sehen, die aufsteigend nummeriert werden und Sie können auch ihre Id (Merchant Id) sehen. Auf der rechten Seite sehen Sie mehrere Buttons zur Auswahl. Der Button „Create“ (Markierung 2) leitet Sie auf ein leeres Formular weiter, womit sie dann einen neuen Händler erstellen können. Die Buttons mit dem Label „Edit“ leiten Sie ebenfalls auf ein Formular weiter, das bereits mit Händlerdaten vorausgefüllt ist. Sie können damit einzelne Händler verwalten.

## 5.5 Händler Erstellen

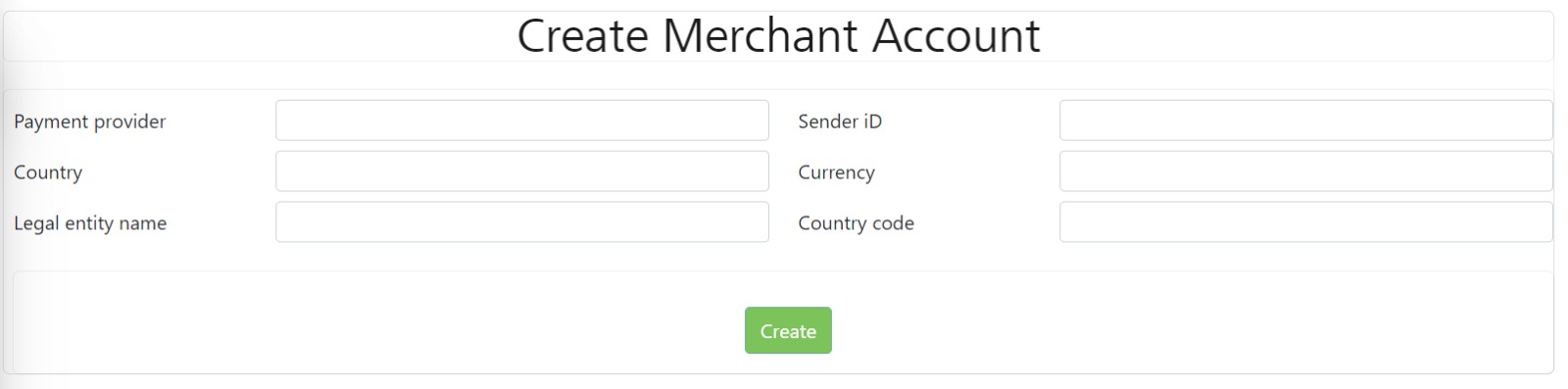
Sobald Sie auf „Create“ geklickt haben, werden Sie ebenfalls weitergeleitet und Sie können ein leeres Formular sehen. Dieses besteht - wie herkömmliche Formulare - aus Labels/Inputs und einen Button.

Abbildung : Einen neuen Händler erstellen

Bei manchen Feldern wurde auch geprüft, ob die Inhalte „null“ sind oder ob sie eine bestimmte Länge haben sollten. Das Feld „Currency“ sollte beispielsweise aus genau drei Zeichen bestehen.

D:\downloads\WhatsApp Image 2022-05-04 at 15.05.54.jpegSobald dieser richtig ausgefüllt ist, wird auch dementsprechend die Ansicht grün leuchten.

Abbildung : Feld "Currency" entspricht den Validierungen

Abbildung : Feld "Currency" entspricht nicht der Validierung

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

## 5.6 Händler Bearbeiten

Ein Bild, das Text enthält.

Automatisch generierte BeschreibungDie Händler können auch bearbeitet werden, also einzelne Felder können aktualisiert werden. Nachdem gewünschte Felder bearbeitet wurden, müssen sie mit dem Klick auf Button „Update“ bestätigt werden. Wenn Sie vorhaben, einen Händler zu löschen, erfolgt dies durch den „Delete“ Button.

Abbildung : Vorhandenen Händler bearbeiten

# 6. Tabellenverzeichnis

[Tabelle 1: Projektphasen 5](#_Toc102673685)

[Tabelle 2: Arbeitsmittel 5](#_Toc102673686)

[Tabelle 3: Personalkosten 6](#_Toc102673687)

[Tabelle 4: Prozessschritte 7](#_Toc102673688)

# 7. Abbildungsverzeichnis

[Abbildung 1: PlanB. GmbH 3](file:///C:\Users\IljasgadzhiM\Desktop\Iljas_Makhsunov_Dokumentation.docx#_Toc102673689)

[Abbildung 2: MerchantAccountModel in Form einer Klasse 9](file:///C:\Users\IljasgadzhiM\Desktop\Iljas_Makhsunov_Dokumentation.docx#_Toc102673690)

[Abbildung 3: ResponseModel in Form einer Klasse 10](file:///C:\Users\IljasgadzhiM\Desktop\Iljas_Makhsunov_Dokumentation.docx#_Toc102673691)

[Abbildung 4: Azure Funktion, um Händlern zu erstellen 11](file:///C:\Users\IljasgadzhiM\Desktop\Iljas_Makhsunov_Dokumentation.docx#_Toc102673692)

[Abbildung 5: Parametern nach Null Prüfungen 11](file:///C:\Users\IljasgadzhiM\Desktop\Iljas_Makhsunov_Dokumentation.docx#_Toc102673693)

[Abbildung 6: Initialisierung von Attributen und Objekten 12](file:///C:\Users\IljasgadzhiM\Desktop\Iljas_Makhsunov_Dokumentation.docx#_Toc102673694)

[Abbildung 7: Einlesen von Inhalt der Anfrage, die von außen kommt 12](file:///C:\Users\IljasgadzhiM\Desktop\Iljas_Makhsunov_Dokumentation.docx#_Toc102673695)

[Abbildung 8: Prüfen, ob der kommende Inhalt etwas enthält 12](file:///C:\Users\IljasgadzhiM\Desktop\Iljas_Makhsunov_Dokumentation.docx#_Toc102673696)

[Abbildung 9: Einen Blob Container erstellen 12](file:///C:\Users\IljasgadzhiM\Desktop\Iljas_Makhsunov_Dokumentation.docx#_Toc102673697)

[Abbildung 10: Händler in Blob Storage ablegen 13](file:///C:\Users\IljasgadzhiM\Desktop\Iljas_Makhsunov_Dokumentation.docx#_Toc102673698)

[Abbildung 11: Antwortnachrichten definieren 13](file:///C:\Users\IljasgadzhiM\Desktop\Iljas_Makhsunov_Dokumentation.docx#_Toc102673699)

[Abbildung 12: Azure Funktion, um Händlern einzulesen 14](file:///C:\Users\IljasgadzhiM\Desktop\Iljas_Makhsunov_Dokumentation.docx#_Toc102673700)

[Abbildung 13: Initialisierung von Attributen und Objekten 14](file:///C:\Users\IljasgadzhiM\Desktop\Iljas_Makhsunov_Dokumentation.docx#_Toc102673701)

[Abbildung 14: Einen JSON von gefunden Händlern abspeichern 15](file:///C:\Users\IljasgadzhiM\Desktop\Iljas_Makhsunov_Dokumentation.docx#_Toc102673702)

[Abbildung 15: Azure Funktion, um Händler aktualisieren 15](file:///C:\Users\IljasgadzhiM\Desktop\Iljas_Makhsunov_Dokumentation.docx#_Toc102673703)

[Abbildung 16: Aktualisierten Händler auf Blob Storage hochladen 16](file:///C:\Users\IljasgadzhiM\Desktop\Iljas_Makhsunov_Dokumentation.docx#_Toc102673704)

[Abbildung 17: Azure Funktion, um Händler zu löschen 16](file:///C:\Users\IljasgadzhiM\Desktop\Iljas_Makhsunov_Dokumentation.docx#_Toc102673705)

[Abbildung 18: Händler aus dem Blob Storage entfernen 17](file:///C:\Users\IljasgadzhiM\Desktop\Iljas_Makhsunov_Dokumentation.docx#_Toc102673706)

[Abbildung 19: Deklaration von Attributen, die für Unit Tests benötigt werden 17](file:///C:\Users\IljasgadzhiM\Desktop\Iljas_Makhsunov_Dokumentation.docx#_Toc102673707)

[Abbildung 20: Attribute und Objekte Initialisieren 18](file:///C:\Users\IljasgadzhiM\Desktop\Iljas_Makhsunov_Dokumentation.docx#_Toc102673708)

[Abbildung 21: Unit Tests Beispiel, um Azure Funktion zu testen 18](file:///C:\Users\IljasgadzhiM\Desktop\Iljas_Makhsunov_Dokumentation.docx#_Toc102673709)

[Abbildung 22: Ein Interface für die Händler 19](file:///C:\Users\IljasgadzhiM\Desktop\Iljas_Makhsunov_Dokumentation.docx#_Toc102673710)

[Abbildung 23: Erstellen eines Service, um die Verbindung zwischen Frontend und Backend herzustellen 20](file:///C:\Users\IljasgadzhiM\Desktop\Iljas_Makhsunov_Dokumentation.docx#_Toc102673711)

[Abbildung 24: CRUD Operationen 20](file:///C:\Users\IljasgadzhiM\Desktop\Iljas_Makhsunov_Dokumentation.docx#_Toc102673712)

[Abbildung 25: Eltern Komponente - Initialisierung von Attributen und Service 21](file:///C:\Users\IljasgadzhiM\Desktop\Iljas_Makhsunov_Dokumentation.docx#_Toc102673713)

[Abbildung 26: Aufruf zum Backend, um Händler Liste zu bekommen 21](file:///C:\Users\IljasgadzhiM\Desktop\Iljas_Makhsunov_Dokumentation.docx#_Toc102673714)

[Abbildung 27: Methode die einen Händler hinzufügt 21](file:///C:\Users\IljasgadzhiM\Desktop\Iljas_Makhsunov_Dokumentation.docx#_Toc102673715)

[Abbildung 28: Ansicht von der Tabelle mit Händlern 22](file:///C:\Users\IljasgadzhiM\Desktop\Iljas_Makhsunov_Dokumentation.docx#_Toc102673716)

[Abbildung 29: Hinzufügen von "countryCode" Eigenschaft 23](file:///C:\Users\IljasgadzhiM\Desktop\Iljas_Makhsunov_Dokumentation.docx#_Toc102673717)

[Abbildung 30: Typescript - Id nach Existenz prüfen 23](file:///C:\Users\IljasgadzhiM\Desktop\Iljas_Makhsunov_Dokumentation.docx#_Toc102673718)

[Abbildung 31: React Formular - Validierung von einzelnen Feldern 24](file:///C:\Users\IljasgadzhiM\Desktop\Iljas_Makhsunov_Dokumentation.docx#_Toc102673719)

[Abbildung 32: Methoden für CRUD Operationen 24](file:///C:\Users\IljasgadzhiM\Desktop\Iljas_Makhsunov_Dokumentation.docx#_Toc102673720)

[Abbildung 33: Backend Ausführung der Azure Funktionen 25](file:///C:\Users\IljasgadzhiM\Desktop\Iljas_Makhsunov_Dokumentation.docx#_Toc102673721)

[Abbildung 34: Create Azure Funktion Logs 25](file:///C:\Users\IljasgadzhiM\Desktop\Iljas_Makhsunov_Dokumentation.docx#_Toc102673722)

[Abbildung 35: Read Azure Funktion Logs 25](file:///C:\Users\IljasgadzhiM\Desktop\Iljas_Makhsunov_Dokumentation.docx#_Toc102673723)

[Abbildung 36: Update Azure Funktion Logs 26](file:///C:\Users\IljasgadzhiM\Desktop\Iljas_Makhsunov_Dokumentation.docx#_Toc102673724)

[Abbildung 37: Delete Azure Funktion Logs 26](file:///C:\Users\IljasgadzhiM\Desktop\Iljas_Makhsunov_Dokumentation.docx#_Toc102673725)

[Abbildung 38: Http Anfrage an Backend mit der Händler Information 27](file:///C:\Users\IljasgadzhiM\Desktop\Iljas_Makhsunov_Dokumentation.docx#_Toc102673726)

[Abbildung 39: Antwort von Backend aus der Anfrage oben 27](file:///C:\Users\IljasgadzhiM\Desktop\Iljas_Makhsunov_Dokumentation.docx#_Toc102673727)

[Abbildung 40: Die Konfiguration "local.setting.json" Datei 29](file:///C:\Users\IljasgadzhiM\Desktop\Iljas_Makhsunov_Dokumentation.docx#_Toc102673728)

[Abbildung 41: Startseite des Managementportals 30](file:///C:\Users\IljasgadzhiM\Desktop\Iljas_Makhsunov_Dokumentation.docx#_Toc102673729)

[Abbildung 42: Händler Verwaltungsliste 30](file:///C:\Users\IljasgadzhiM\Desktop\Iljas_Makhsunov_Dokumentation.docx#_Toc102673730)

[Abbildung 43: Einen neuen Händler erstellen 31](file:///C:\Users\IljasgadzhiM\Desktop\Iljas_Makhsunov_Dokumentation.docx#_Toc102673731)

[Abbildung 44: Feld "Currency" entspricht den Validierungen 31](file:///C:\Users\IljasgadzhiM\Desktop\Iljas_Makhsunov_Dokumentation.docx#_Toc102673732)

[Abbildung 45: Feld "Currency" entspricht nicht der Validierung 31](file:///C:\Users\IljasgadzhiM\Desktop\Iljas_Makhsunov_Dokumentation.docx#_Toc102673733)

[Abbildung 46: Vorhandenen Händler bearbeiten 32](file:///C:\Users\IljasgadzhiM\Desktop\Iljas_Makhsunov_Dokumentation.docx#_Toc102673734)

# 8. Glossar

**ESB:**

„Enterprise Service Bus“ ist ein erfolgreiches Projekt bei der PlanB. GmbH. Die stellen eine sichere Verbindung zwischen zwei System dar und sichern sorgfältig Ihre Daten.

**Azure Funktionen:**

Sind Funktionen, die von einem „Triggern“ ausgelöst werden, zum Beispiel durch eine Anfrage („HttpTrigger“), durch eine bestimmte Zeit („TimerTrigger“) usw. Nach dem Auslösen wird der Code Stück mit beliebiger Logik ausgeführt.

**JSON:**

“JavaScript Object Notation” ist ein Datenstruktur Format, indem werden Objekte, Arrays und sonstige Eigenschaften definiert. Wird hauptsächlich für den Datenaustausch zwischen Anwendungen verwendet.

**HTTP-Trigger:**

Ist einer von Trigger arten die Azure Funktionen anbietet. Die wird von „HTTP-Request“ ausgelöst. Es werden von „HTTP-Clients“ zum Beispiel von „Browser“ an den „HTTP-Server“ Anfragen geschickt, die nach dem „HTTP-Request“ Schema aufgebaut sind.

**Blob:**

Ausgeschrieben ein „Binary Large Objects“ sind Große binäre Datenobjekte, wie zum Beispiel Bildern/Grafiken oder Videos.

**Blob Storage:**

Speichermedium, in dem alle mögliche Objekte abgespeichert werden können. Zum Beispiel JSON Objekte.

**Container:**

Die Container haben den Zweck in Azure Storage die Blobs zu separieren.

**BlobClient:**

Ist eine Klasse, die vordefiniert ist und gewährt und den Zugriff auf Azure Storage Container und deren Blobs, um die zu manipulieren.

**CRUD-Operationen:**

CRUD-Operationen (Create, Read, Update, Delete) sind Datenoperationen die für Erstellen, Lesen, Aktualisieren, Löschen stehen. Das sind grundlegende Aktionen, die für verschiede Zwecken benötigt werden. Zum Beispiel bei der Datenbank einen neuen Eintrag hinzufügen.

**CSV-Datei:**

DIe „Comma seperated values“ sind Dateien, die nach bestimmter Struktur aufgebaut sind. Zum Beispiel werden einzelne Datensätze durch ein Komma, Tabulatur etc. getrennt

**Azure Cloud:**

Die Azure Cloud wird von Microsoft angeboten, um verschiedene Lösungsansätzen umzusetzen, dazu stehen verschiede Dienste und Service zu Verfügung.

**Managementportal:**

Zum Beispiel ein Webanwendungsportal, bei dem man verschiedene Information/Daten verwaltet.

**Schnittstellen:**

Ermöglicht die Interaktion bzw. Kommunikation zwischen Software und einem externen System.

**Logs:**

Dient zu der Nachverfolgung des Status der Anwendung, ob irgendwelchen Fehlern, Attacken etc. stattgefunden haben.

**Moq:**

Der dient als externe Bibliothek, den man hauptsätzlich für simulieren der Daten verwendet. Mehr Informationen können hier gefunden werden: <https://softchris.github.io/pages/dotnet-moq.html#add-unit-tests>

**Typescript:**

Ist eine Programmiersprache, die von Microsoft entwickelt wurde. Die ist eine Erweiterung von Javascript, die bietet uns an, dass wir mit Klassen, Vererbung usw. arbeiten können. Der Typescript wird am Ende im Browser zum Javascript kompiliert.

**App Services:**

Der Service ermöglicht uns unsere Applikation zu hosten. Zum Beispiel eine Webanwendung.

**Azure Blob Storage:**

Ermöglicht uns die einzelne Blobs abzuspeichern und die zu verwalten.

**Function App:**

Hier werden Azure Funktionen abgelegt, das heißt der Code unabhängig von gewählter Sprache (unterstützen Sprachen: C#, Powershell, Python, und Javascript/Typescript) wird problemlos ausgeführt.

**Application Insights:**

Ist ein Azure Service, der die Anwendung überwacht. Es werden verschiedene Log-Nachricht geloggt, die man verfolgen kann, um seltsame Aktionen rechtzeitig zu erkennen.

**Product Owner:**

Wird Person bezeichnet, die für den Ablauf des Projektes verantwortlich ist.

# Ein Bild, das Text enthält. Automatisch generierte Beschreibung9. Anlagen