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 2](#_Toc102549359)

[1.1 Projektumfeld 2](#_Toc102549360)

[1.2 Projektziel 3](#_Toc102549361)

[1.3 Projektbegründung 3](#_Toc102549362)

[1.4 Projektschnittstellen 3](#_Toc102549363)

[1.5 Projektabgrenzung 3](#_Toc102549364)

[1.6 Ansprechpartner 3](#_Toc102549365)

[2. Projektplanung 4](#_Toc102549366)

[2.1 Ablaufplan 4](#_Toc102549367)

[2.2 Ressourcenplanung 4](#_Toc102549368)

[2.3 Pflichtenheft 4](#_Toc102549369)

[2.3.1 Ist-/Soll-Analyse 4](#_Toc102549370)

[2.3.2 Musskriterien 5](#_Toc102549371)

[2.3.3 Wunschkriterien 5](#_Toc102549372)

[2.3.4 Prozessschritte 6](#_Toc102549373)

[2.4 Vorbereitungsphase 6](#_Toc102549374)

[2.4.1 Kick-off-Meeting 6](#_Toc102549375)

[2.4.2 Auswahl des Frameworks 6](#_Toc102549376)

[2.4.3 Auswahl der Azure Services 7](#_Toc102549377)

[3. Durchführung 7](#_Toc102549378)

[3.1 Implementierungsphase 7](#_Toc102549379)

[3.1.1 Servicebereitstellung 7](#_Toc102549380)

[3.1.2 Implementierung des Backends 8](#_Toc102549381)

[3.1.3 Unit Tests 14](#_Toc102549382)

[3.1.4 Implementierung des Frontends 17](#_Toc102549383)

# 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

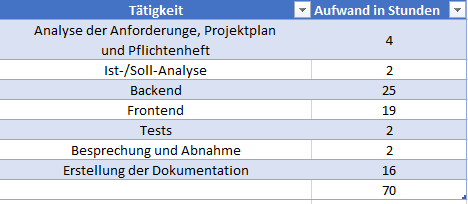
Anbei die Projektphasen inklusive Zeitaufwand für jeden Arbeitsschritt:

Tabelle 1: Projektphasen

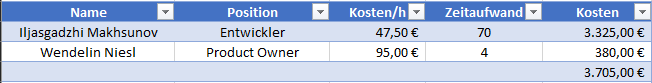
## 2.2 Ressourcenplanung

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

Arbeitsmittel:



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, bei den man den Händlern verwalten kann.

Es sollte möglich sein, dass alle Händler unter CSV Datei abzuspeichern, die für den Protokoll dienen sollten, um Fehler zu erkennen und dementsprechend schnell zu reagieren. Unter anderem sollten es für jeden Fehler eine eigene Fehlermeldung geben, der beschreiben soll, was falsch ausgeführt wurden.

Zukünftig sollten auch Gedanken gemacht werden, wie die Webanwendung gehostet wird und die einzelnen Services, wie die Azure Funktionen und der Azure Blob Storage. Anschließend sollte auch das Design angepasst werden, der PlanB. GmbH normen entspricht.

### 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, die man verwaltet kann. Zunächst sieht man nur die eindeutige ID des Händlers und zwei Knöpfe fürs aktualisieren und löschen. Ganz oben finden man noch einen weiteren Button, um einen neuen Händler einzutragen. Die Web Anwendung sollte in Angular geschrieben werden, weil sie Vorteile von „Single Page Application“ mit sich bringt und die leicht auf Azure gehostet werden kann. Die Azure Funktionen werden unter der .NET 6.0 Framework geschrieben.

### 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.

### 2.3.4 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

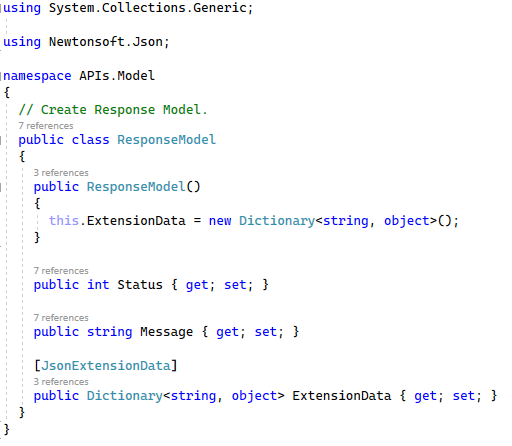
#### 3.1.2.1 Implementierung der Klasse „MerchantAccountModel“

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.

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.



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

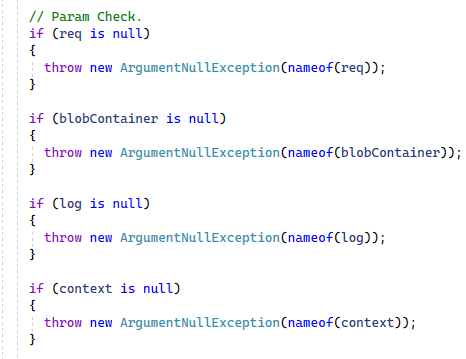
**Create:**

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

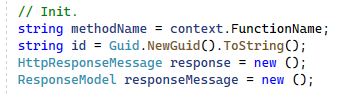
Für erstellen einen neuen Händler wurde die Klasse „Create“ zusammengeschrieben. Sie wird wie eine Methode deklariert und hat die Klasse „HttpResponseMessage“ als Rückgabe Wert. Darauf folg immer ein Trigger, in unserem Fall ist das der „HttpTrigger“ der nur die „Post“ Methode akzeptiert. Beim Auslösen 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.



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



Hier werden die wichtigen Objekte/Attribute initialisiert.



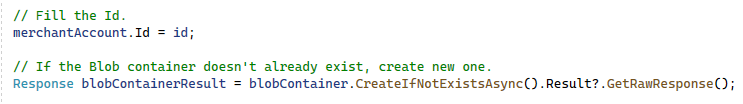
Die Zeile enthält drei Funktionen, einmal liest sie den „RequestBody“ der von außen gekommen ist und dann Mappt sie gleich den „RequestBody“ mit unserer „MerchantAccountModel“ klasse, die wir oben bereits implementiert haben. Anschließend mit „Result“ geben wir den Rückgabewert synchron zurück.



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



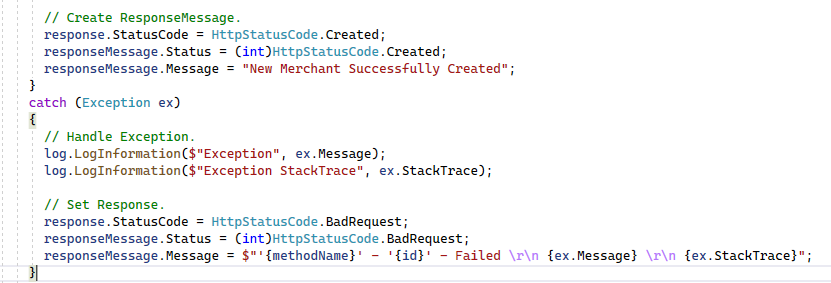
Als nächstes wird aus dem „Request Body“ die ID (Identifikator) übergeben und unter Blob Storage ein neuer Container namens „merchants“ erstellt. Der ebenfalls, eine Rückmeldung zurückgibt, falls etwas schiefgelaufen ist.



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 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).

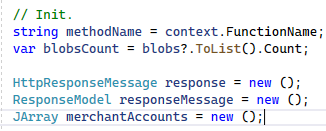


**Read:**

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“.

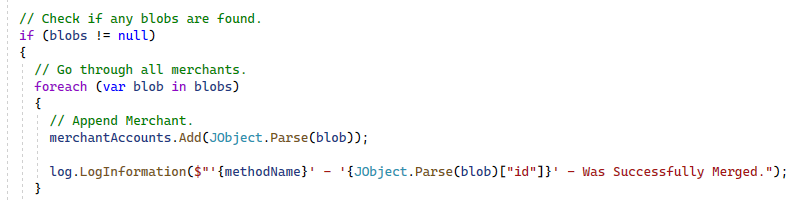


Bevor wir die Händler einlesen, werden noch einige Objekte/Attribute initialisiert.



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.

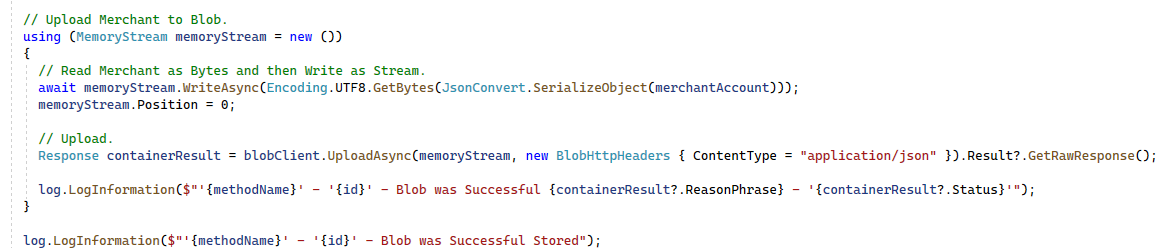


**Update:**

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.

****

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“.

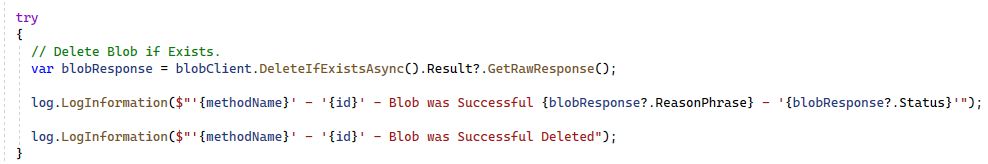


**Delete:**

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.



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



### 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:

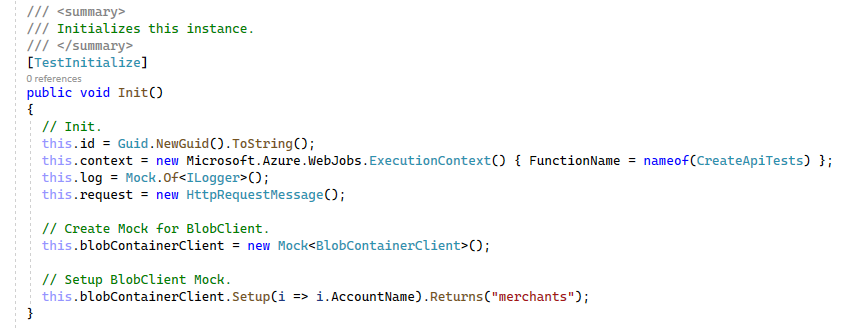
Zuerst werden folgende Attribute deklariert, die beim später fürs testen verwendet werden.



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.



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.



### 3.1.4 Implementierung des Frontends

Ein Bild, das Text enthält.

Automatisch generierte Beschreibung

Ein Bild, das Text enthält.

Automatisch generierte Beschreibung

Ein Bild, das Text enthält.

Automatisch generierte Beschreibung

Ein Bild, das Text enthält.

Automatisch generierte Beschreibung

Ein Bild, das Text enthält.

Automatisch generierte Beschreibung

Ein Bild, das Text enthält.

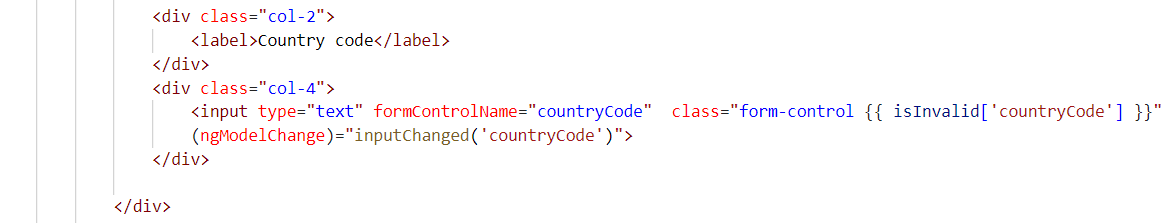
Automatisch generierte Beschreibung

Ein Bild, das Text enthält.

Automatisch generierte Beschreibung

Ein Bild, das Text enthält.

Automatisch generierte Beschreibung



Ein Bild, das Text enthält.

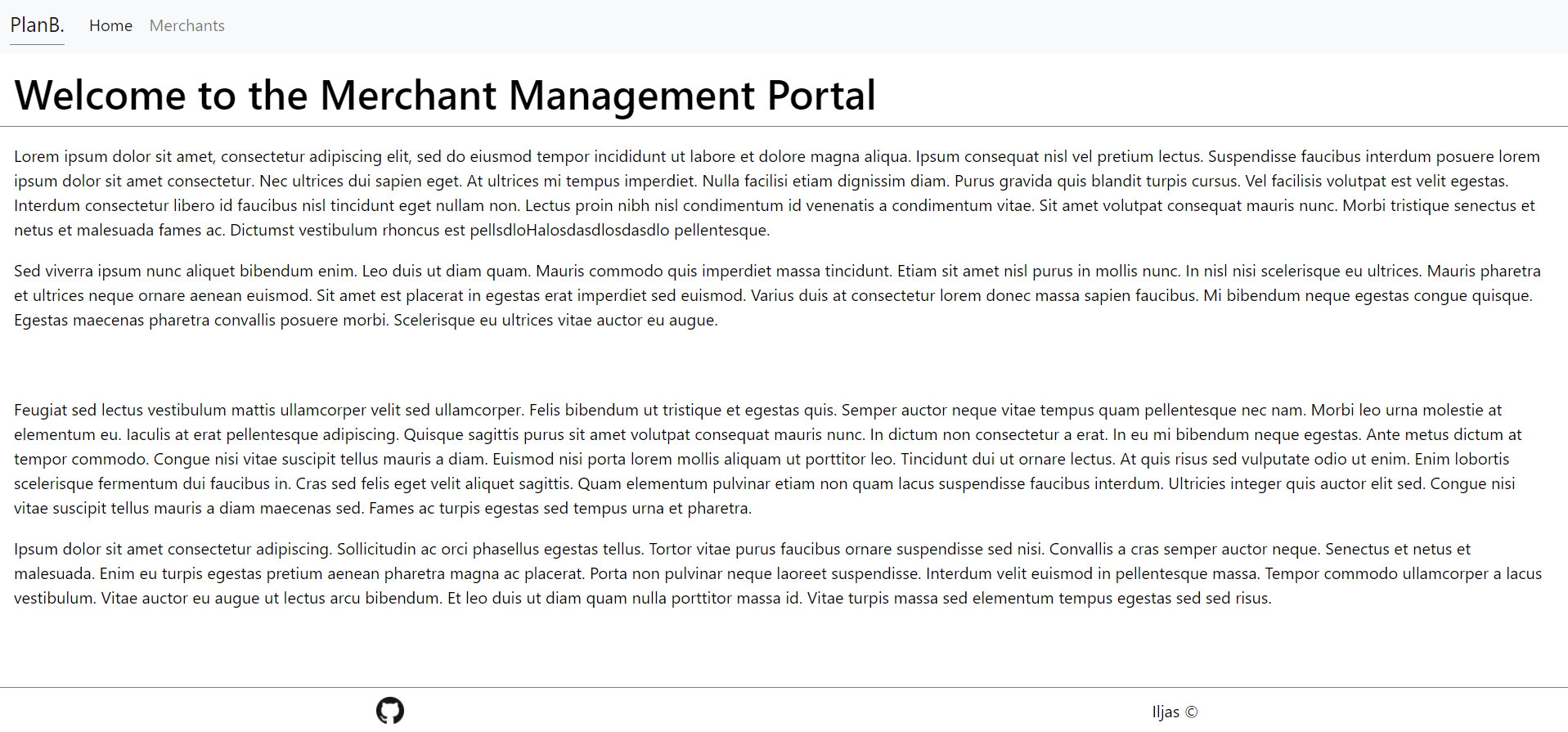
Automatisch generierte Beschreibung

Ein Bild, das Text enthält.

Automatisch generierte Beschreibung

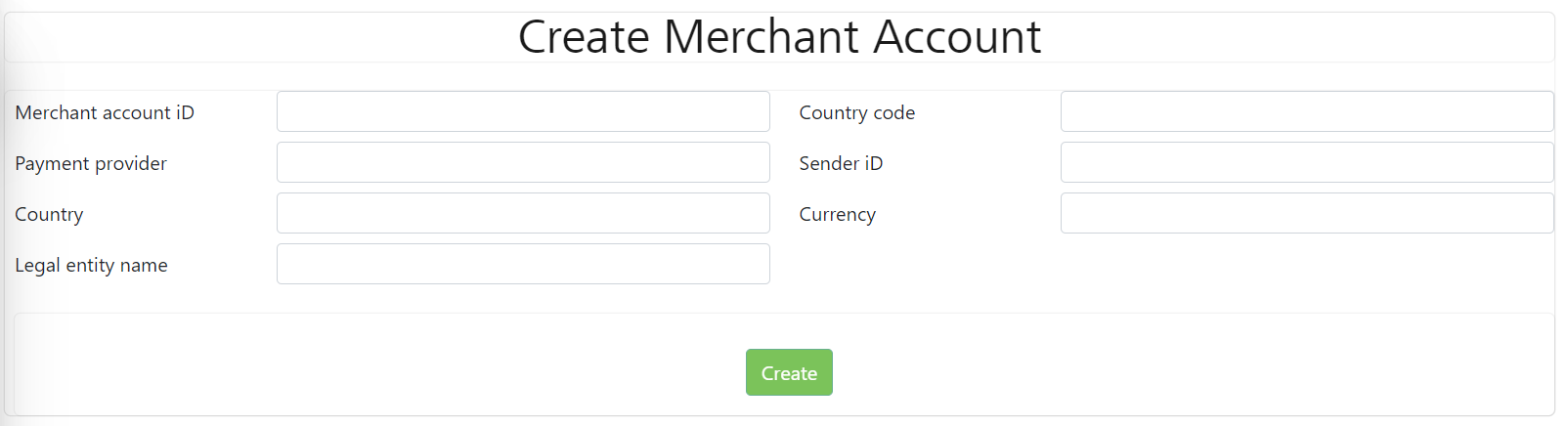
Ein Bild, das Text enthält.

Automatisch generierte Beschreibung



Ein Bild, das Tisch enthält.

Automatisch generierte Beschreibung



Ein Bild, das Text enthält.

Automatisch generierte Beschreibung