CCA - COMPETENCE CENTRE
HTL Anichstraße

Diplomarbeit

HTL-Mensa-App

Mittagsmenüs mittels App online bestellen

Eingereicht von

Felix Haider
Patrick Klaric

Eingereicht bei

Höhere Technische Bundeslehr- und Versuchsanstalt Anichstraße

Abteilung für Wirtschaftsingenieure/Betriebsinformatik

Betreuer

Mag. Hans Vogt

Projektpartner

Innsbruck, April 2024

Abgabevermerk:

Betreuer/in:

Datum:

Gender-Erklärung

Aus Gründen der besseren Lesbarkeit wird in dieser Diplomarbeit die Sprachform des generischen Maskulinums angewandt und auf die zusätzliche Formulierung der weiblichen Form verzichtet. Es wird an dieser Stelle darauf hingewiesen, dass die ausschließliche Verwendung der männlichen Form explizit als geschlechtsunabhängig verstanden werden soll.

Danksagung

An dieser Stelle wollen wir uns bei allen Personen bedanken, die uns im Laufe der Diplomarbeit geholfen haben. Einen großen Dank wollen wir unserem betreuenden Lehrer Herrn Prof. Mag. Hans Vogt aussprechen, welcher uns bei Fragen und Problemen rund um diese Diplomarbeit stets mit Rat und Tat zur Seite stand. Anschließend wollen wir uns auch bei all unseren Freunden und Familien bedanken, welche uns ebenfalls stets geholfen haben. Hier gilt ein besonderer Dank an Oliver Brandstetter, der uns einen Server zur Verfügung gestellt hat.

Kurzfassung

Zur Mittagszeit entsteht in der Mensa immer eine lange Warteschlange, da mehr und mehr Schüler es bevorzugen, die Feinkost der Mensa zu genießen. Im Rahmen dieser Diplomarbeit versuchen wir, den Kaufvorgang der Mittagsmenüs zu digitalisieren und dadurch die Wartezeit zu verkürzen. Dafür wird nach der Bezahlung ein QR-Code generiert, der in der App oder über Mail aufgerufen werden kann. Diese QR-Codes können anschließend von dem Mensapersonal gescannt werden und zeigen ausgewählte Daten zur Bestellung an. Zusätzlich hat die Mensa, die Möglichkeit auf einem Terminal, die Statistiken der letzten Tage, Wochen und Monaten abzurufen und die Daten grafisch darstellen zu lassen.

Abstract

There is always a long queue in the canteen at lunchtime, as more and more pupils prefer to enjoy the canteen's food. As part of this diploma thesis, we are trying to digitise the process of buying lunch menus and thus reduce the waiting time. Therefore, a QR code is generated after payment, which can be called up in the app or via e-mail. These QR codes can then be scanned by the canteen staff and display selected data about the order. In addition, the canteen has the option of calling up the statistics of the last few days, weeks and months on a terminal and displaying the data graphically.

Projektergebnis

Das Projekttziel wurde größtenteils erreicht, sowohl die App als auch die Website sind funktionsfähig. Es sind nur ein paar Kleinigkeiten, die noch fehlen bzw. erweitert werden können, wie beispielsweise die Erstellung eines QR-Code-Generators und eines dazugehörigen Scanners oder das Übermitteln des QR-Codes über eine E-Mail. Außerdem fehlt noch die Einbindung der Bezahlung mit PayPal, die im Zuge dieser Diplomarbeit nur in einem Testprojekt ausprobiert wurde.

Erklärung der Eigenständigkeit der Arbeit

EIDESSTATTLICHE ERKLÄRUNG

Ich erkläre an Eides statt, dass ich die vorliegende Arbeit selbständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt und die den benutzten Quellen wörtlich und inhaltlich entnommenen Stellen als solche erkenntlich gemacht habe. Meine Arbeit darf öffentlich zugänglich gemacht werden, wenn kein Sperrvermerk vorliegt.

Ort, Datum Verfasser 1

Ort, Datum Verfasser 1

Inhaltsverzeichnis

1 Einleitung

1.1 Vertiefende Aufgabenstellung

Durch unsere jahrelange Erfahrung als Kunden der Mensa konnten wir nicht über das Problem der langen Warteschlange hinwegsehen. Deswegen versuchten wir die Warteschlange während der Mittagspausen in dieser Diplomarbeit zu verkürzen oder ganz abzuschaffen.

Der erste Schritt war es sich über vergleichbare Systeme in anderen Schulen zu informieren, wo wir auch schnell einige Unterschiede fanden. Der größte Unterschied war hierbei die Verwendung von Apps, um die gewünschten Speisen im Vorhinein (zum Beispiel am Anfang der Woche oder des Monats) auszuwählen und zu bezahlen.

Im nächsten Schritt machten wir uns auf die Suche nach der richtigen Sprache, das richtige Framework und den richtigen Betreuungslehrer zu suchen. Wir entschieden uns für C#, da wir beide damit am meisten gearbeitet hatten und unser Betreuungslehrer mit C# sehr vertraut ist. Zusätzlich wählten wir das .NET-Framework, um die App und die Website zu schreiben.

1.1.1 Patrick Klaric

Damit die Kunden die Mensa Menüs bestellen können, musste eine App programmiert werden, die die Menüs anzeigt, die jede Woche zum Verkauf verfügbar sind. Anschließend muss noch eine Möglichkeit geboten werden, die bestellten Menüs zu bezahlen; dafür wurde die Bezahlung mittels PayPal ausgesucht. Nach der Bestellung sollte es auch eine Möglichkeit geben, eine Art Bestellverlauf zu sehen, in dem der Status der Abholung angeschrieben ist. Zusätzlich wäre eine View geplant gewesen, die den QR-Code der Bestellung anzeigt.

1.1.2 Felix Haider

Damit die Menüs überhaupt in der App zur Verfügung stehen und diese verkauft werden können, muss der Mensa die Möglichkeit gegeben werden, diese Menüs einzugeben. Nach der Eingabe des Menüs wird dieses in der Datenbank gespeichert und je nach Datum in der App angezeigt. Falls eines dieser Menüs falsch eingegeben wurde, kann dieses ebenfalls auf der Website entweder gelöscht oder bearbeitet werden. Wenn nun eine neue Bestellung erstellt wird, kann die Mensa auf der Website nachsehen, für welches Menü wie viele Bestellungen vorhanden sind. Des Weiteren kann die Mensa verschiedene Statistiken über die verkauften Menüs auf der Website visualisieren.

2 Verwendete Technologien

2.1 MySQL

MySQL ist ein Datenbankmanagmentsystem, mit dem Daten in Tabellenform verwaltet und abgefragt werden können. Um die gespeicherten Daten aus der Datenbank abzufragen, wurde die MySQL-Workbench verwendet. Die MySQL-Workbench ist ein von Oracle entwickeltes Tool, um Datenbank-Designs zu erstellen, zu visualisieren, zu bearbeiten und zu verwalten.

2.1.1 Datenbankstruktur

Abbildung 2.1 zeigt die Struktur der Datenbank, die sich in vier zusammenhängende Tabellen teilt. Die ***persons***-Tabelle enthält die Nutzer, die sich angemeldet haben, während die ***menues***-Tabelle alle Menüs beinhaltet, die die Mensa wochenweise einträgt. Wichtig hierbei ist auch die Zwischentabelle ***menupersons***, die alle Bestellungen einer jeweiligen Person zu einem Menü widergeben. Zuletzt gibt es eine ***prices***-Tabelle, die für jedes Menü einen bestimmten Preis enthält, abhängig davon, ob ein Schüler oder ein Lehrer das Menü bestellen will.

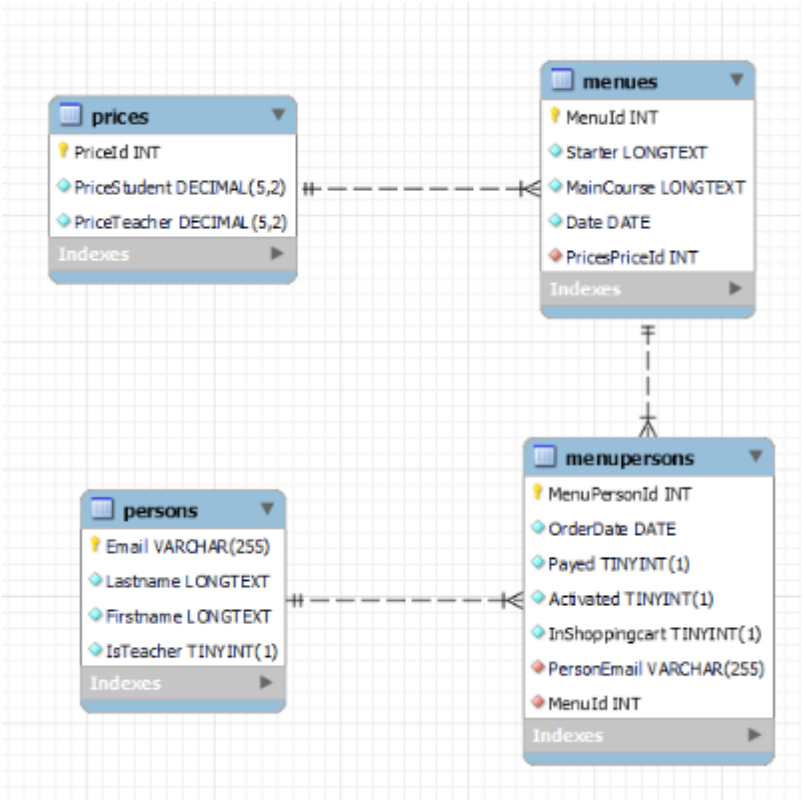

Abbildung 2.1: Hier wird die in der Diplomarbeit verwendete Datenbankstruktur gezeigt.

2.2 Visual Studio

Visual Studio ist eine integrierte Entwicklungsumgebung (IDE) von Microsoft, die zum Entwickeln von Computerprogrammen, Websites, Webanwendungen, Webdiensten und mobilen Anwendungen verwendet wird. Die IDE bietet Unterstützung für verschiedene Programmiersprachen, darunter C#, Visual Basic .NET, C++, JavaScript, TypeScript, Python und mehr. Visual Studio ermöglicht es Entwicklern, Code zu schreiben, zu debuggen, zu testen und zu veröffentlichen. In dieser Diplomarbeit wurde Visual Studio verwendet, da mit dieser IDE über den Verlauf der Schullaufbahn am meisten Erfahrung gesammelt wurde.

2.3 Insomnia/Postman

Sowohl Insomnia als auch Postman werden verwendet, um API-Aufrufe auf ihre Funktionalität zu testen. Im Falle dieser Diplomarbeit wurde Insomnia benutzt, um die API-Aufrufe für die selbst programmierten Controller zu testen. Da PayPal schon eine Verknüpfung für Postman hat, wurde Postman verwendet, um die PayPal API-Aufrufe zu testen.

2.4 GitHub

Um mit mehreren Personen an einem Projekt zu arbeiten, ist es wichtig, immer die aktuellste Version des Programms zu besitzen. Hierbei hilft GitHub, das Entwicklern erlaubt, ihren Quellcode zu verwalten und Versionskontrollen anzuwenden. Für diese Diplomarbeit wurden drei Äste (eng. Branches) benutzt, einerseits der immer vorhandene Master-Branch und anschließend noch ein App-Branch und ein Website-Branch. So konnten, ohne den Hauptcode zu verändern oder Konflikte auszulösen, an separaten Stellen im Code Veränderungen vorgenommen werden.

3 Handy-App

3.1 .NET MAUI

.NET MAUI ist ein Framework, um Handy (IOS und Android) und Desktop (Windows und MacOS) Anwendungen mittels XAML und C#-Code zu schreiben. .NET MAUI entstand aus Xamarin.Forms. In dieser Diplomarbeit wird näher auf MAUI im Zuge der Android-App-Entwicklung eingegangen, dies liegt daran, dass das Zielsystem Android ist. Damit .NET MAUI auf verschiedenen Plattformen (z.B. Android, IOS) kompilieren kann, wird jede Anwendung in die gleiche .NET Base Class Library (BCL) übersetzt; diese BCL trennt die Plattform von dem geschriebenen Code. Die BCL gibt jeder Basisklasse dieselbe Code-Logik, wobei die Benutzeroberfläche von der jeweiligen Plattform abhängig ist. So kann die Benutzeroberfläche für jede Plattform separat erstellt werden, indem man das spezifische Framework (für Android: .NET Android) verwendet. (vgl. davidbritch (n.d.*g*))

3.1.1 XAML

Steuerelemente: Die von MAUI zur Verfügung gestellte Toolbox bietet einige Möglichkeiten, ihre Ansichten (eng. Views) und die darin enthaltenen Daten in verschiedenen Layouts zu formatieren. In den nachstehenden Punkten werden die wichtigsten und am häufigsten verwendeten Views und Layouts erklärt.

Grid: ist ein Layout, um die darin liegenden Daten in Zeilen und Spalten einzuordnen; standardmäßig enthält das Raster nur eine Zeile und eine Spalte. Wie Grid in die Anwendung implementiert wurde und wie mehr Zeilen und Spalten verwendet werden können, wird in dem Kapitel 3.1.4 erklärt. (vgl. davidbritch (n.d.*d*))

StackLayout: ist ein Layout, um alle darin liegenden Kindelemente entweder vertikal oder horizontal zu orientieren, wobei die Elemente standardmäßig vertikal orientiert sind. Wie das StackLayout in die Anwendung implementiert wurde, wird in dem Kapitel 3.1.4 erklärt. (vgl. davidbritch (n.d.*f*))

ScrollView: Die ScrollView ist eine Ansicht, die es erlaubt, den Inhalt einer View vertikal zu scrollen. Durch dieses Feature wirkt die App dynamischer. Wie die ScrollView in die Anwendung implementiert wurde, wird in dem Kapitel 3.1.4 erklärt. (vgl. davidbritch (n.d.*e*))

CarouselView: Die CarouselView ist eine Ansicht, die die Möglichkeit bietet, alle geladenen Elemente horizontal auszurichten. Anschließend wird nur ein einzelnes Element am Bildschirm angezeigt, während die anderen Elemente zwar im Hintergrund geladen sind, aber nicht angezeigt werden. Wie die CarouselView in die Anwendung implementiert wurde, wird in dem Kapitel 3.1.4 erklärt. (vgl. davidbritch (n.d.*a*))

WebView: Durch die WebView kann eine Website mit ihrem enthaltenen CSS und JavaScript angezeigt werden, dafür ist nur eine URL nötig, die als Source dient. Im Zuge der Diplomarbeit wurde durch die WebView ein ASP.NET Core Projekt geladen, das für die Verarbeitung von PayPal Bezahlungen geschrieben wurde. (vgl. davidbritch (n.d.*h*))

Data binding: Durch User, die mit vorhandenen Views interagieren, können sich Daten ändern, die zum Beispiel in Datenbanken liegen. Falls sich solche Daten ändern sollten, muss die App das in ihren Views widerspiegeln. Damit diese Daten richtig in den Views angezeigt werden, muss die App Veränderungen in den Daten oder der View wahrnehmen können; für diesen Fall wird Data binding verwendet. Data binding verknüpft die Eigenschaften zweier Objekte; die Veränderung von einem Objekt bewirkt auch dieselbe Änderung auf das verknüpfte Objekt. Data binding kann sowohl in der .xaml Datei als auch in der .xaml.cs Datei verwendet werden, wobei es häufiger in der .xaml Datei vorkommt, da es die Views überschaubarer macht und die Anzahl der Zeilen verkürzt. In dieser Diplomarbeit wurde das Data binding so implementiert, dass ein Objekt immer in der .xaml Datei vorhanden ist und das zweite Objekt, aufgrund des verwendeten MVVM (Model-View-ViewModel) Design-Patterns in einer View Model enthalten ist. Weitere Informationen zu MVVM werden in dem hier folgenden Kapitel 3.1.2 genauer erläutert. (vgl. davidbritch (n.d.*c*))

3.1.2 MVVM

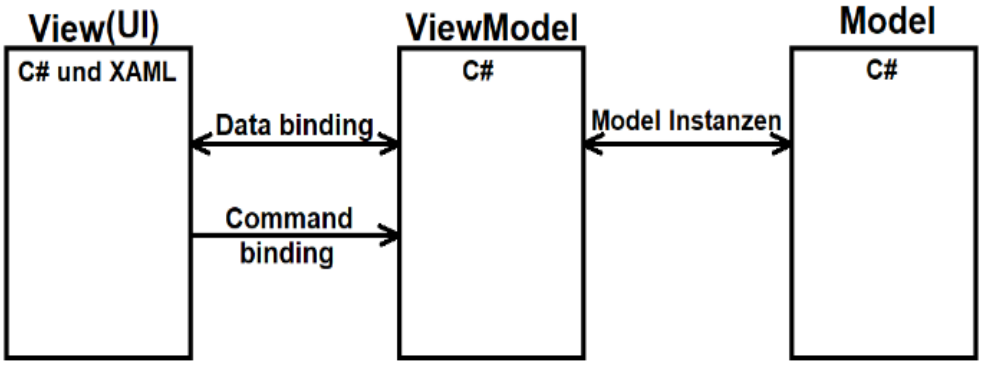

Abbildung 3.1: Hier wird das MVVM-Konzept abgebildet. Das Designmuster zeigt, wie das User-Interface von der App-Logik getrennt ist.

Model-View-ViewModel (MVVM) ist ein Designmuster, das verwendet wird, um das User-Interface (UI) von der App-Logik zu trennen. Diese strikte Trennung erlaubt es, das User-Interface leichter auszutauschen, während der Code gleich bleibt. Auf der Code Seite kann alter Code erweitert und neuer Code implementiert werden, außerdem erlaubt es diese Herangehensweise, Code leichter wiederzuverwenden. (vgl. michaelstonis (n.d.))

3.1.3 Struktur

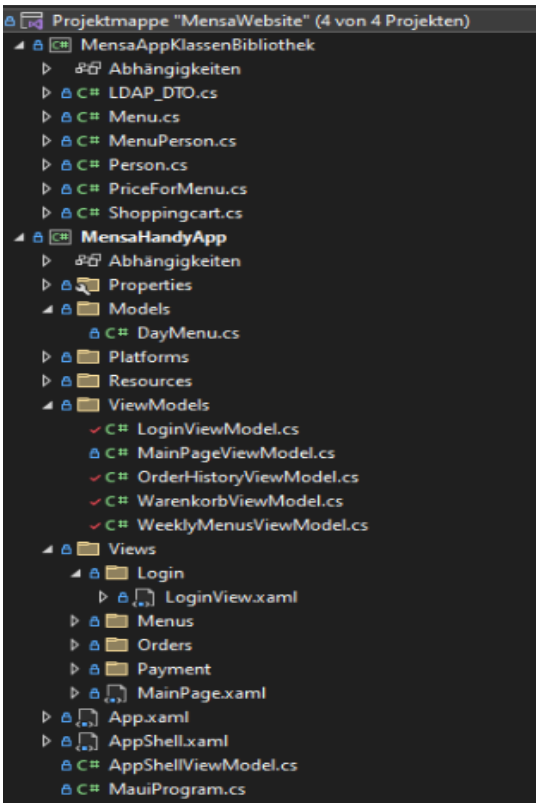

Abbildung 3.2: Zeigt eine Mögliche MVVM-Struktur beziehungsweise die verwendete Struktur des HandyApp-Teils in der Anwendung.

Der Aufbau der Diplomarbeit strukturiert sich nach dem MVVM Muster. Nachdem das Projekt erstellt wurde, sollten für die Gliederung; die Ordner für Models, Views und ViewModels erstellt werden. In dieser Diplomarbeit wurden die Models in eine eigene Klassenbibliothek ausgelagert. Vorteile dieser Herangehensweise sind die Wiederverwendbarkeit der Klassen in anderen Projekten wie zum Beispiel dem Website-Teil; Außerdem kann die Funktionalität der Klassen leicht getestet werden. Weitere Punkte, die für die Auslagerung sprechen, sind das vereinfachte Arbeiten in Teams, da verschiedene Teile einer Klasse gleichzeitig erweitert werden können und alle Klassen an einem Punkt auffindbar und bearbeitbar sind.

View: Die View wird in XAML und C# geschrieben; Sie ist für das Aussehen des User-Interface verantwortlich und besteht typischerweise aus einer ContentPage und den darin liegenden Steuerelementen.

Model: Models sind Klassen, die aus reinem C#-Code bestehen; Sie werden genutzt, um mit der Datenbank zu interagieren und die Logik der App vorzugeben.

ViewModel: Das ViewModel besteht aus C#-Code und bildet die Verbindung zwischen der Programmlogik und dem User-Interface. Ein ViewModel besteht aus Objekten (eng. Properties) und Befehlen (eng. Commands); Durch Data binding wird die View über Änderungen informiert und kann die angezeigten Daten immer auf dem neuesten Stand halten. Das Laden solcher geänderten Daten aus zum Beispiel einer Datenbank benötigt Zeit, währenddessen muss die App aber immer noch auf Benutzereingaben reagieren können. Solche Methoden, die Zeit brauchen, um zu funktionieren, werden asynchron definiert, damit die App nicht blockiert wird.

3.1.4 Verwendung der Steuerelemente anhand der WeeklyMenusView.xaml Datei

Im folgenden Textabschnitt werden die XAML Steuerelemente anhand einer verkürzten Version der WeeklyMenusView.xaml und einem Screenshot der Handy-App erklärt.

```
1  <ScrollView>
2  <VerticalStackLayout>
3  <CarouselView x:Name="carusellMenus"  ItemsSource="{Binding DayMenus}">
4  <CarouselView.ItemTemplate>
5  <DataTemplate>
6  <VerticalStackLayout BackgroundColor="White" Padding="5">
7  <CollectionView ItemsSource="{Binding Menus}" SelectionMode="Single"
       SelectedItem="{Binding SelectedListItem}">
8  <CollectionView.ItemsLayout>
9      <GridItemsLayout Span="1" Orientation="Vertical" />
10 </CollectionView.ItemsLayout>
11 <CollectionView.ItemTemplate>
12 <DataTemplate>
13     <Border StrokeThickness="4" Padding="50" >
14         <Grid RowDefinitions="2*, 2*, 2*, 2*, 2*"
               ColumnDefinitions="1*, 1*"  Margin="5">
15             <Label Text="Vorspeiße:" HorizontalOptions="Start"
                   Grid.Row="1" Grid.Column="0"/>
16             <Label Text="{Binding Starter}" HorizontalOptions="Start"
                   Grid.Row="1" Grid.Column="1"/>
17         </Grid>
18     </Border>
19 </DataTemplate>
```

```
20 </CollectionView.ItemTemplate>
21 </CollectionView>
22 </VerticalStackLayout>
23 </DataTemplate>
24 </CarouselView.ItemTemplate>
25 </CarouselView>
26 </VerticalStackLayout>
27 </ScrollView>
```

Code 3.1: Zeigt eine verkürzte Version der WeeklyMenuesView.xaml, um bestimmte Inhalte der XAML Steuerelemente zu erklären.

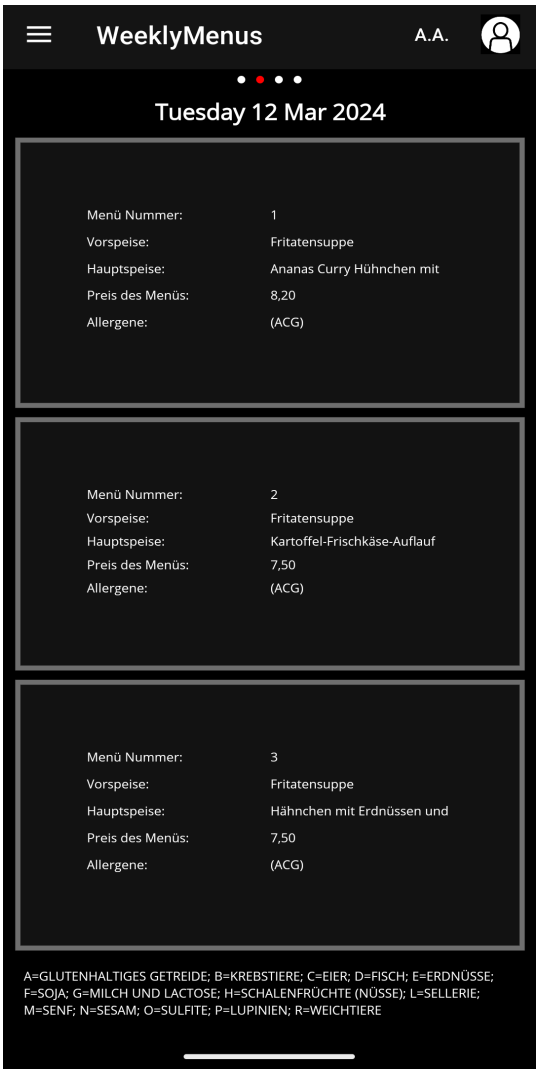

Abbildung 3.3: Diese Abbildung zeigt, wie der Code 3.1 in der HandyApp aussieht

Zuerst wurde eine ScrollView angelegt, damit zu den Daten, die außerhalb des derzeitigen Bildschirms liegen, gescrollt werden kann. Dieses Scrollen ist vertikal möglich; Um die geladenen Daten nun vertikal zu positionieren wird ein VerticalStackLayout benutzt. Anschließend wird die Hauptkomponente dieser View erstellt, die CarouselView, mit der der wöchentliche Menüplan geladen wird. Wie im Screenshot der Handy-App zu sehen ist, hat jeder Tag drei verschiedene Menüs, aus denen der User wählen kann . Damit so eine CarouselView funktioniert, benötigt sie eine Objektquelle (eng. ItemSource) und eine Objektvorlage (eng. ItemTemplate). Die ItemSource ist hierbei eine Liste, die alle Menüs, die zur Auswahl stehen, enthält (vgl. Code: DayMenus). Das ItemTemplate zeigt, wie die CarouselView aufgebaut ist; in diesem Fall ist ein weiteres VerticalStackLayout und darin die CollectionView vorhanden.

Die CollectionView ist ähnlich aufgebaut, sie benötigt eine ItemSource, ein ItemTemplate, einen SelectionMode (dt. Auswahlmodus) und ein ItemLayout (dt. Objektlayout). Der SelectionMode kann Einzelauswahl und Mehrfachauswahl sein; In diesem Beispiel wurde die Einzelauswahl (eng. Single) benutzt; das heißt, es kann nur ein Menü ausgewählt werden. Dieses wird mit einer Umrandung hervorgehoben. Anschließend kann das gewählte Menü dem Warenkorb hinzugefügt werden. Damit die CollectionView funktioniert, fehlt noch die ItemSource, die eine Liste aller Menüs ist, die in DayMenus gespeichert sind. Zusätzlich braucht die CollectionView, um zu funktionieren, noch das ItemLayout, das das Layout der CollectionView vorgibt; in diesem Fall ist es vertikal, mit nur einer Spalte. Zuletzt wird noch das ItemTemplate benötigt, das die Daten in Form von Labels in einem Grid anzeigt. Das Grid hat hierbei den Sinn, die Felder einheitlich und strukturierter zu machen.

3.1.5 Verwendung der ViewModels

Die App besteht im gesamten aus fünf ViewModels, diese beinhalten die Anzeige der Hauptseite, den Login, die Bezahlung, die Anzeige der bestellbaren Menüs, den Bestellverlauf. Da im Kapitel 3.1.4 bereits auf die WeeklyMenuView eingegangen wurde, wird in diesem Kapitel nicht auf dieses ViewModel eingegangen, sondern auf die Anderen.

3.1.6 MainPageViewModel und LoginViewModel:

Die MainPageViewModel besitzt einOrderHistoryViewModelen Knopf, der direkt auf die LoginView führt. In dieser View können Username und Passwort einge-

geben werden, mithilfe der Funktionen des LDAP-Controllers (referenz LDAP) kann die Authentifizierung und damit der Login durchgeführt werden. Sollte der Login funktioniert haben, werden alle Views auf sichtbar gestellt. Wie diese Sichtbarkeit nach dem Login aussieht, wird in Abbildung 3.4 gezeigt.

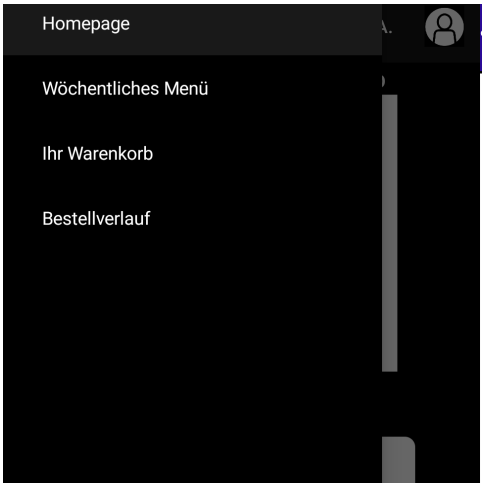

Abbildung 3.4: Diese Abbildung zeigt, die Sichtbarkeit der Views nach dem Login

Die wichtigste genutzte Funktion dieser ViewModels ist die ***MessagingCenter***-Klasse, wie diese funktioniert wird im folgenden Codesegment gezeigt. Es implementiert das Publish-and-Subscribe-Pattern, mit dem eine leichte Kommunikation zwischen den Views gegeben ist. So können Nachrichten als String vom Veröffentlicher (eng. Publisher) mittels ***MessaginCenter.Send();*** gesendet werden und alle Abonnenten (eng. Subscriber) erhalten mittels ***MessagingCenter.Subscribe<>();*** diese Nachricht und können damit in ihrer ViewModel Befehle ausführen.

```
1  //LoginViewModel:
2  MessagingCenter.Send(this, "LoginSuccess");
3
4  //MainPageViewModel:
5  MessagingCenter.Subscribe<LoginViewModel>(this, "LoginSuccess",
       (sender) =>
6  {
7      LoginSuccess();
8  });
9
10 public void LoginSuccess()
11 {
12     ShowLogoutButton = true;
13     ShowLoginButton = false;
```

```
14 }
15
16
17 //AppShell.xaml.cs:
18 MessagingCenter.Subscribe<LoginViewModel>(this, "LoginSuccess",
       (sender) =>
19 {
20     ShowLoggedInViews();
21 });
22
23 private void ShowLoggedInViews()
24 {
25     _vm.IsHomepageVisible = true;
26     _vm.IsWeeklyMenusVisible = true;
27     _vm.IsWarenkorbVisible = true;
28     _vm.IsOrderHistoryVisible = true;
29
30     _vm.IsLoginVisible = false;
31 }
```

Code 3.2: Diese Codesegmente zeigt, wie die MessagingCenter-Klasse String-Nachrichten verschickt und erhält mittels Subscribe und Send

Im Codsegement 3.2 wird gezeigt, wie nach einem erfolgreichen Login ***MessagingCenter.Send(this, "LoginSuccess");*** aufgerufen wird, also wird eine Nachricht "LoginSuccess" an alle Subscriber gesendet. Diese Nachricht wird auch von der AppShell.xaml.cs-Datei mittles ***MessagingCenter.Subscribe<LoginViewModel>(this, "LoginSuccess");*** empfangen. Die AppShell sorgt dann für das Einschalten der Sichtbarkeit von allen Views. "LoginSuccess" wird auch von der MainPageViewModel empfangen, diese schaltet den LoginButton aus und der LogoutButton wird stattdessen sichtbar.

3.1.7 WarenkorbViewModel:

Diese ViewModel hat aufgrund von Zeitmangels keine funktionierende Bezahlung, jedoch wurden sowohl die View als auch ViewModel so geschrieben, dass die Funktion leicht einfügbar ist. In dieser ViewModel ist die zurzeit wichtigste Funktion die ***GetShoppingCart()***-Methode. Wie im Codesegment 3.3 zu sehen ist, wird zuerst ein API-Request an Server mit der E-Mail gesendet. Dieser Request sendet als Response eine Liste von ***MenuPersons*** names ***mps***. Anschließend werden alle Bestellungen aller Personen gelöscht und neu hinzugefügt. Danach wird die Liste mps durchgegangen und nach allen Bestellungen gefiltert, die sich im Warenkorb befinden, also ***InShoppingCart*** sind. Als Nächstes wird der Preis

für jedes Menü festgelegt, je nachdem, ob der Benutzer ein Schüler oder Lehrer ist. Zum Schluss wird der Gesamtpreis (***ShoppingCartPriceDecimal***), der später für PayPal nötig ist, berechnet.

```
1  public async void GetShoppingCart()
2  {
3      List<MenuPerson> mps = await
           _client.GetFromJsonAsync<List<MenuPerson>>(url +
           "api/MenuPersonAPI/getAllOrderByUserEmail?mail=" +
           person.Email);
4
5      person.MenuPersons.Clear();
6      person.MenuPersons.AddRange(mps);
7
8      Shoppingcart = mps.Where(mp => mp.InShoppingcart).ToList();
9
10     person = await Person.LoadObject();
11
12     foreach(MenuPerson mp in Shoppingcart)
13     {
14         if (!person.IsTeacher)
15         {
16             IsTeacher = false;
17             IsStudent = true;
18             ShoppingCartPriceDecimal = ShoppingCartPriceDecimal +
                   mp.Menu.Prices.PriceStudent;
19         }
20         else if (person.IsTeacher)
21         {
22             IsTeacher = true;
23             IsStudent = false;
24             ShoppingCartPriceDecimal = ShoppingCartPriceDecimal +
                   mp.Menu.Prices.PriceTeacher;
25         }
26     }
27 }
```

Code 3.3: Diese Codesegmente zeigt, wie die WarenkorbViewModel alle Menüs Anzeigt, die sich derzeit im Warenkorb befinden und dazu ihre Preise

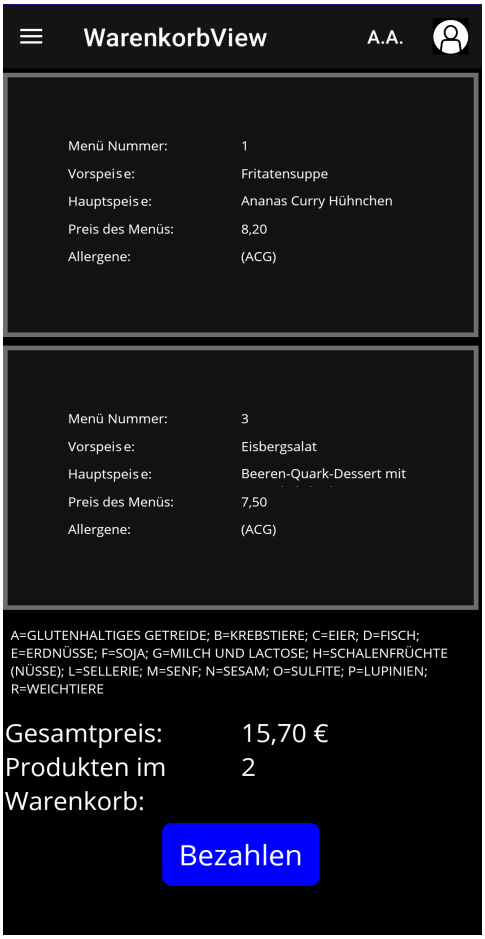

Abbildung 3.5: Diese Abbildung zeigt, wie der Warenkorb in der App aussieht

3.1.8 OrderHistoryViewModel:

Dieses ViewModel zeigt alle Bestellungen eines Benutzers an, dafür ist die Methode ***ShowOrder()*** verantwortlich. Wie im folgenden Codesegement 3.4 zu sehen ist, wird ein API-Request gesendet, der die E-Mail des Benutzers übergibt. Die Response wird in eine Liste von Bestellungen gespeichert. Danach wird die Liste umgekehrt, somit ist die neueste Bestellung die erste, die angezeigt wird. Zuletzt wird die Liste Element für Element durchgegangen und in die ***ObservableCollection Orders*** gespeichert, damit sie in der View angezeigt werden kann.

```
1 private async Task ShowOrder()
2 {
3     person = await Person.LoadObject();
```

```
4       List<MenuPerson> allOrders = new List<MenuPerson>();
5       allOrders = await _client.GetFromJsonAsync<List<MenuPerson>>(url +
            "api/MenuPersonAPI/getAllOrderByUserEmail?mail=" +
            person.Email);
6
7       allOrders.Reverse();
8       foreach (var order in allOrders)
9       {
10          Orders.Add(order);
11      }
12  }
```

Code 3.4: Dieses Codesegment zeigt, wie alle Bestellungen eines Benutzers angezeigt werden

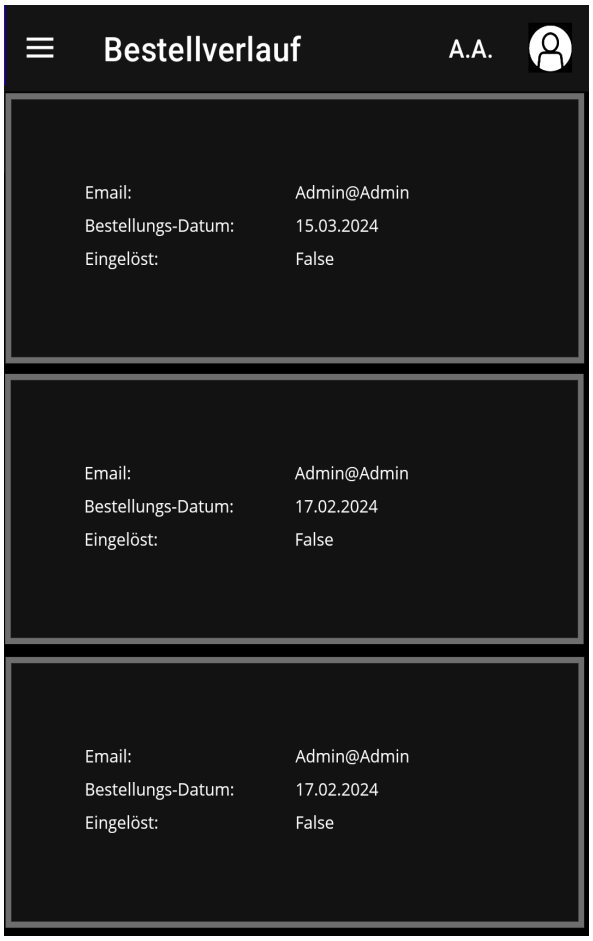

Abbildung 3.6: Diese Abbildung zeigt, wie der Bestellverlauf in der App aussieht

3.2 Entity Framework Core und API-Controller

3.2.1 Theoretische Grundlagen EF Core

Das Entity Framework Core (EF Core) ist ein Object Relation Mapper (ORM), der in .NET Programmen verwendet wird. Mit diesem ORM können Daten aus der Datenbank abgefragt und geändert werden, anschließend können diese Daten wieder in die Datenbank gespeichert werden. Das alles müsste eigentlich mit MySQL-Abfragen gemacht werden, durch EF Core kann die Datenbanken mit C#-Code verwaltet werden.

Der Datenbank zugriff funktioniert in EF Core mithilfe eines Models, das von ***DbContext*** erbt. Das Model besteht aus Entitätsklassen und Kontextobjekten, diese Kontextobjekte repräsentieren eine Sitzung beziehungsweise eine Verbindung mit der Datenbank. Es gibt mehrere Ansätze wie das Model geschrieben werden kann, darunter zum Beispiel das Generieren eines Models aus einer bereits existierenden Datenbank; Diese Diplomarbeit nutzt nur den Ansatz ein Model zu erstellen und daraus mittels Migrationen (eng. migrations) eine Datenbank zu erstellen. Dieser Ansatz hat den Vorteil, dass bei Änderungen in der Struktur des Models auch die Datenbank durch weitere migrations geändert werden kann.

EF Core verwendet das Prinzip von Konvention über Konfiguration, das heißt es muss fast nichts konfiguriert werden, da Spalten, Tabellen und Datentypen aus der Struktur der Klasse, dem Klassennamen und dem Propertynamen gezogen werden. Konfigurationen der Tabellen oder Datenbank sind aber zum Beispiel durch Anmerkungen (eng. annotations) trotzdem möglich.

Damit EF Core überhaupt funktionieren kann, müssen einige NuGet-Pakte installiert werden. Darunter ***Microsoft.EntityFrameworkCore***, das den ORM enthält und darin auch die wichtigsten Typen, wie ***DbContext*** und ***DbSet***. Anschließend wird ***Microsoft.EntityFrameworkCore.Tools*** noch benötigt, um über die Paket-Manager-Konsole Befehle, wie ***Add-Migration*** oder ***Update-Database***, ausführen zu können. Schließlich wurde noch ***Pomelo.EntityFrameworkCore.MySql*** installiert, um auf die MySQL-Datenbank Zugriff zu bekommen. (vgl. Leismann (n.d.) & ajcvickers (n.d.)))

3.2.2 Verwendung von EF Core anhand der MenuContext-Klasse

Anhand des Codeteils 3.5; wird gezeigt, wie MenuContext von der DbContext Klasse, die von ***Microsoft.EntityFrameworkCore*** importiert wurde, erbt. Anschlie-

ßend wird von dem importierten Paket auch die DbSet<> Klasse verwendet, um Zugang auf eine Tabelle zu bekommen. Das heißt die Zeile ***public DbSet<Menu> Menues get; set;*** ermöglicht den Zugriff auf die gesamte Menü-Tabelle (**Menu**). Somit wird in den nächsten Zeilen der Zugriff auch für die Personen (***Persons***), die Bestellungen (***MenuPersons***) und die Preise (***Prices***) gewährt. Anschließend wird die ***OnConfiguring()***-Methode ausgeführt, mit der die Konfigurationen für die Datenbankverbindung festgelegt werden. Der ***connectionString*** gibt die wichtigsten Informationen an, die für die Verbindung zur MySQL-Datenbank benötigt werden; darunter fallen die Serveradresse, der Datenbankname, der Benutzername und das Passwort. Anschließend gibt ***UseMySql()*** an, dass der Pomelo-MySQL-Treiber für die Verbindung genutzt wird.

```
1   using   MensaAppKlassenBibliothek;
2 using Microsoft.EntityFrameworkCore;
3
4 namespace MensaWebsite.Models.DB
5 {
6     public class MenuContext : DbContext
7     {
8         public DbSet<Menu> Menues { get; set; }
9         public DbSet<Person> Persons { get; set; }
10        public DbSet<MenuPerson> MenuPersons { get; set; }
11        public DbSet<PriceForMenu> Prices { get; set; }
12
13        protected override void OnConfiguring(DbContextOptionsBuilder
              optionsBuilder)
14        {
15            string connectionString =
16                "Server=localhost;database=MensaApp;
17                user=root;password=password";
18            optionsBuilder
19                .UseMySql(connectionString,
20                ServerVersion.AutoDetect(connectionString));
21
22        }
23    }
24 }
```

Code 3.5: Die gezeigte Klasse erbt von der DbContext-Klasse in der Entity Framework Core-Bibliothek und stellt eine Verbindung zu einer MySQL-Datenbank her.

3.2.3 Theoretische Grundlagen API-Controller

Der API-Controller ermöglicht die Kommunikation und den Datenaustausch zwischen verschiedenen Softwarekomponenten. Mithilfe des Controllers können

HTTP-Requests versendet werden; anschließend wird eine HTTP-Response mit den Daten in Form von JSON als Antworten erhalten. Mit diesen Daten kann danach in zum Beispiel der .NET MAUI Anwendung weitergearbeitet werden. Ein API-Controller erbt von der ***ControllerBase*** Klasse und hat zwei annotations, die ihn ausmachen. Zuerst wird genau definiert, dass es sich um einen APIController handelt, indem **[*ApiController*]** über den Klassennamen geschrieben wird; diese annotation sorgt auch für das Formatieren der Antworten zu JSON. Anschließend wird noch eine Route benötigt, mit der angegeben wird, wie ein HTTP-Request einem bestimmten Controller zugeordnet ist. In dieser Diplomarbeit ist die Route immer ***[Route(" api/[controller]")]***, da es ein Platzhalter ist, der während der Laufzeit geändert wird. Wichtig ist auch, dass jede Controller-Methode eine Response im JSON-Format liefert, dafür ist das ***return new JsonResult();*** zuständig. Auch wichtig für eine Methode ist die Angabe; der Route und welche Art von HTTP-Request die Methode ist, dafür ist die annotation ***[HttpGet("Route")]*** zuständig.

3.2.4 Verwendung eines API-Controllers anhand des MenuAPIControllers

Zuerst wird der Controller mit seinen nötigen annotations und Vererbungen erstellt; Da ***[Route(" api/[controller]")]*** nur ein Platzhalter ist, wird während der Laufzeit die Route auf ***" api/MenuAPI"*** geändert. Anschließend wird eine private Instanz der MenuContext-Klasse erstellt; im Konstruktor wird ***_context*** gleich der Instanz gesetzt. Mit ***_context*** kann nun auf die Daten in der Datenbank zugegriffen werden. Als Nächstes wird eine Instanz von ***JsonSerializerOptions*** erstellt und die Einstellung ***ReferenceHandler.IgnoreCycles*** hinzugefügt. Mit dieser Einstellung soll der Serializer Objekte erkennen, die sich gegenseitig referenzieren und somit Endlosschleifen verhindern, beziehungsweise die Referenzierung nach dem ersten Mal ignorieren. Die Methode ***AsyncGetAllMenues*** zeigt, wie ein Request versendet wird, der die Menüs inklusive der Preise aus der Datenbank fordert; die Antwort beinhaltet alle Menüs als JSON. Damit der Controller weiß, wo die Methode ist, wird eine Route mitgegeben, die ***"getAllMenus"*** ist; somit befindet sich die Methode bei ***api/MenuAPI/getAllMenus***. Zudem wird noch HttpGet mitgegeben, das heißt, dass diese Methode auf HTTP GET-Anforderungen reagiert.

```
1    [Route("api/[controller]")]
2 [ApiController]
3 public class MenuAPIController : ControllerBase
4 {
5
```

```
6       private MenuContext _context = new MenuContext();
7
8
9       JsonSerializerOptions options = new JsonSerializerOptions()
10      {
11          ReferenceHandler = ReferenceHandler.IgnoreCycles
12      };
13
14
15      public MenuAPIController(MenuContext context)
16      {
17          this._context = context;
18      }
19
20
21
22      [HttpGet("getAllMenus")]
23      public async Task<IActionResult> AsyncGetAllMenues()
24      {
25          return new JsonResult(await
                this._context.Menues.Include("Prices").ToListAsync(),
                options);
26      }
27
28  }
```

Code 3.6: Zeigt den Aufbau des MenuAPIControllers, der eine Methode getAllMenus besitzt, die alle Menüs aus der Datenbank abruft und als JSON-Resultat zurückgibt.

3.3 LDAP

3.3.1 Theoretische Grundlagen

LDAP, kurz für Lightweight Directory Access Protocol, ist ein Protokoll, mit dem auf die Daten eines aktiven Verzeichnisses (eng. Active Directories) zugegriffen werden kann. Durch LDAP-Befehle können Daten verändert werden oder eine Authentifizierung eingebaut werden. Mit so einer Authentifizierung kann zum Beispiel; nach einer erfolgreichen Anmeldung; auf bestimmte Daten von einem Server zugegriffen werden. (vgl. *What Is LDAP & How Does It Work? | Okta* (n.d.))

3.3.2 Aufbau eines Active Directory

Ein Active Directory wird verwendet, wenn statische Daten wie Nutzernamen, E-Mails oder Passwörter langfristig gespeichert werden müssen. Begriffe, die in dieser Diplomarbeit bei der Nutzung von Active Directories fallen werden, sind DN, sn, givenName, Domain und Tree. Darum folgt hier eine kurze Erklärung zu diesen Begriffen. (vgl. *Active Directory einfach erklärt | prosec-networks.com* (n.d.))

Distinguished Name (DN): repräsentiert den vollständigen Pfad eines Objekts in einem Active Directory und ist damit eine eindeutige Kennung. (vgl. *IBM Documentation* (n.d.))

sAMAccountName (sn): ist ein Attribut des Benutzerobjekts und stellt den Benutzerkontonamen dar; dieser muss innerhalb einer Domäne eindeutig sein. (vgl. ebd.)

givenName: enthält den Vornamen eines Benutzers. Der givenName ist zusammen mit dem sn verantwortlich, den vollständigen Namen eines Benutzers zu bilden. (vgl. ebd.)

Domain: ist eine logische Gruppierung von Benutzern, die denselben Sicherheitsrichtlinien und Einstellungen folgen müssen. Als Beispiel könnten Lehrer und Schüler in zwei verschiedenen Domänen sein, die sich durch verschiedene Sicherheitsrichtlinien und Einstellungen unterscheiden. (vgl. ebd.)

Tree: bezieht sich auf die Hierarchie von verschiedenen Domänen, die miteinander vertraut sind. Hier könnte also die Schule XY eine Tree für die Schüler-Domain und Lehrer-Domain anlegen. (vgl. ebd.)

3.3.3 Verwendung der Request und Response

Für die Verbindung zum Active Directory mit LDAP wurde eine HTTP Get-Methode benutzt, die als Einstiegspunkt für die anderen drei Methoden verwendet wird. Alle verwendeten Methoden sind asynchron, da zwischen dem Request und der Response eine gewisse Zeit vergeht und das Programm währenddessen nicht blockiert werden soll.

Damit es möglich ist, Daten aus dem Active Directory zu bekommen, müssen einige Parameter gegeben sein. Die benötigten Parameter für die erste Methode sind die IP-Adresse und das Passwort für den Server. Diese Parameter werden

genutzt, um zu verifizieren, dass das Active Directory ansprechbar ist und eine Verbindung eingegangen werden kann.

Wichtig ist auch der angegebene Port, wie aus dem Codeteil 3.7 zu entnehmen ist. Im Normalfall ist der Port für das LDAP-Protokoll entweder 636 für TLS-gesicherte Verbindungen oder, wie im Fall der in der Diplomarbeit verwendeten Konfiguration, der Port 389, da eine ungesicherte Verbindung hergestellt wird.

```
1  private async Task<LdapConnection> AsyncConnectToLDAP(string
     ldapServer, string ldap_password)
2  {
3      LdapConnection ldapConnection = null;
4
5      try
6      {
7          var credentials = new NetworkCredential("MensaLDAP",
               ldap_password, "SYNCHTLINN");
8          var serverId = new LdapDirectoryIdentifier(ldapServer, 389,
               false, false);
9          ldapConnection = new LdapConnection(serverId, credentials);
10         ldapConnection.Bind(credentials);
11         return ldapConnection;
12     }
13     catch (Exception e)
14     {
15         return null;
16     }
17     return ldapConnection;
18 }
```

Code 3.7: Erstellung einer asynchronen Verbindung zu einem Active Directory mittels IP, Passwort und Port (389).

Anschließend greift sofort die zweite Methode, die im Codeteil 3.8 gezeigt wird. In dieser Methode kann die Authentifizierung leicht erkannt werden, für diese Authentifizierung werden aber einige Parameter benötigt, diese wären, die schon vorhandene Verbindung, einen Nutzernamen und ein Passwort.

Im Codesegment zeigt die Zeile ***var conn = new LdapConnection(serverId, credentials);*** wie eine neue Verbindung, jetzt aber mit einem Nutzer, erstellt wird. Anschließend wird mit ***conn.Bind(credentials);*** versucht, den Nutzer zu binden, also zu sehen, ob dieser Nutzer im Active Directory vorhanden ist. Die ***AsyncValidateUser*** Methode liefert Wahr (eng. True) zurück, wenn die Verbindung erfolgreich war, also der übergebene Nutzername und das Passwort in dem Active Directory vorhanden sind. Am Ende wird die Verbindung mittels ***conn.Dispose()*** geschlossen.

```
1   private   async Task<bool> AsyncValidateUser(LdapConnection
      ldapConnection, string username, string password)
2 {
3       var credentials = new NetworkCredential(username, password,
            "SYNCHTLINN");
4       var serverId = new LdapDirectoryIdentifier("10.10.80.42", 389,
            false, false);
5       var conn = new LdapConnection(serverId, credentials);
6
7       try
8       {
9           conn.Bind(credentials);
10          return true;
11      }
12      catch (Exception e)
13      {
14          return false;
15      }
16
17      conn.Dispose();
18 }
```

Code 3.8: Zeigt eine asynchrone Methode, die überprüft, ob ein Nutzer sich in diesem Active Directory befindet. Ist dies der Fall, gibt die Methode True zurück und schließt die Verbindung zum Active Directory.

Sollte die Verbindung mit dem angegebenen Nutzer funktioniert haben, wird die dritte und letzte Methode ausgeführt. Da nicht bekannt ist, ob der Nutzer ein Schüler oder ein Lehrer ist, wird die Methode zweimal ausgeführt und somit der Benutzer in der Domain der Lehrer und anschließend in der Domain der Schüler gesucht. Dies kann im Codesegment 3.9 anhand des Parameters ***baseDn*** erkannt werden, der den Ort angibt, wo sich der Lehrer-Tree beziehungsweise der Schüler-Tree befindet.

Findet die Methode den Anwender, so gibt sie den Vornamen und den Nachnamen zurück, dies kann im Code 3.9 gut an der Zeile ***entry.Attributes["givenName"][0]***, die den Vornamen ausgibt und ***entry.Attributes["sn"][0]***, die den Nachnamen ausgibt, erkannt werden.

```
1 private async Task<SearchResultEntryCollection>
      AsyncSearchUser(LdapConnection ldapConnection, string baseDn,
      string username)
2 {
```

```
3       var searchRequest = new SearchRequest(baseDn,
            $"(&(objectClass=person)(sAMAccountName={username}))",
            SearchScope.Subtree, null);
4
5       try
6       {
7           var searchResponse = (SearchResponse)ldapConnection
                .SendRequest(searchRequest);
8           if (searchResponse.Entries.Count == 1)
9           {
10              return searchResponse.Entries;
11          }
12          else
13          {
14              return null;
15          }
16      }
17      catch (LdapException e)
18      {
19          return null;
20      }
21  }
```

Code 3.9: Dieser Code sucht im Active Directory im Lehrer-Tree und im Schüler-Tree nach dem angegebenen Nutzer; Wenn der Nutzer gefunden wurde, gibt diese Methode den Vornamen und Nachnamen zurück.

Abschließend wird noch nachgeschaut, ob die Person in der Lehrer-Domain war oder im Schüler-Domain. Mit diesen Informationen kann nun eine Person erstellt werden, die die Felder ***IsTeacher***, ***FirstName*** und ***LastName*** füllt. Die erstellte Person wird dann als Response an die LoginViewModel gesendet, die mit den Informationen weiterarbeitet.

3.4 PayPal

3.4.1 Theoretische Grundlagen

Damit die Nutzer nach der Auswahl der gewünschten Menüs, diese auch bezahlen können, wurde für die Diplomarbeit entschieden, die Bezahlung über PayPal zu regeln. Damit die Bezahlung im Programm getestet werden kann, ohne echtes Geld zu überweisen, bietet PayPal die Möglichkeit, ein Sandbox-Konto (eng. Sandbox-Account) zu erstellen. Mithilfe des Sandbox-Accounts kann dann ebenfalls die Erstellung einer Bestellung (eng. Order) und die Erfassungen

(eng. Capture) der Bezahlung für diese Bestellung getestet werden. Der erstellte Sandbox-Account gliedert sich in zwei Teile, das Geschäftskonto (eng. Buisness-Account), das Geld von den Kunden erhält und das persönliche Konto (eng. Personal-Account), das Geld an das Geschäftskonto sendet. (vgl. *Part 3: What is Sandbox? | Create PayPal Sandbox Account | PayPal Tutorial With .NET Application* (n.d.))

Aufgrund von Zeitmangel wurde dieser Teil nur als ein Testprogramm geschrieben und nicht in die App implementiert.

3.4.2 PayPal-API Request und Response

Als Erstes wurde die Funktion der von PayPal programmierten API-Requests getestet. Mithilfe von Postman konnte eine neue Instanz der API-Befehle erstellt werden (eng. Fork collection). Mit dieser Instanz konnten anschließend; unter dem Ordner ***Orders***, die Befehle ***Bestellung Erstellen*** (eng. ***Create order***), ***Erfassung der Bezahlung für die Bestellung*** (eng. ***Capture payment for order***) getestet werden. Zusätzlich gibt es noch einen API-Request, der getestet werden muss, diese Methode liefert den ***Zugangstoken*** (***get access token***), mit dem der Aufruf von PayPal API-Requests möglich gemacht wird.

3.4.3 Javascript

Hauptsächlich wird Javascript verwendet, um die Funktion der PayPal Knöpfe zu erstellen. Die Grundstruktur des verwendeten Codes wurde von der PayPal-Website, beziehungsweise der PayPal-API übernommen (vgl. *JavaScript SDK reference* (n.d.)), anschließend wurde der Code mit einem YouTube-Video (vgl. *How to Integrate PayPal Payments in ASP.NET Web Applications with Razor Pages | The Complete Guide - YouTube* (n.d.)) erweitert.

3.4.4 Verwendung der Request/Response

Zuerst wird ein access token erstellt; dieser benötigt, wie im Code 3.10 gezeigt wird, einige Parameter. Einer dieser Parameter ist das ***urlaccesstoken***, das sich aus ***https://api-m.sandbox.paypal.com*** und ***/v1/oauth2/token*** zusammensetzt; Es zeigt, wohin der API-Request mit HTTP-Post versendet wird. Anschließend ist die ***PayPalClientId***, also die von PayPal vergebene eindeutige Kennzeichnung eines Nutzers, anzugeben. Letztlich wird das ***PayPalSecret*** also das Passwort des

Geschäftskontos, das die Rechnung ausstellt, angegeben. Nach diesem Aufruf wird von dem PayPal Server ein access token zurückgesendet.

```
1  private String GetPayPalAccessToken()
2  {
3      string accesToken = "";
4      string urlaccesstoken = PayPalUrl + "/v1/oauth2/token";
5
6      using (var client = new HttpClient())
7      {
8          string credentials64 = Convert.ToBase64String(
9              Encoding.UTF8.GetBytes(PayPalClientId + ":" +
                   PayPalSecret));
10         client.DefaultRequestHeaders.Add("Authorization", "Basic " +
               credentials64);
11
12         var requestMessage = new HttpRequestMessage(HttpMethod.Post,
               urlaccesstoken);
13         requestMessage.Content = new
               StringContent("grant_type=client_credentials", null,
               "application/x-www-form-urlencoded");
14
15         var responeseTask = client.SendAsync(requestMessage);
16         responeseTask.Wait();
17
18         var result = responeseTask.Result;
19         var readTask = result.Content.ReadAsStringAsync();
20         readTask.Wait();
21
22         var strResponse = readTask.Result;
23         var jsonResponse = JsonNode.Parse(strResponse);
24         accesToken = jsonResponse["access_token"]?.ToString() ?? "";
25     }
26     return accesToken;
27 }
```

Code 3.10: Mithilfe von bereitgestellten Anmeldeinformationen authentifiziert sich diese Methode bei PayPal, um einen Accesstoken zu erhalten und diesen als Zeichenkette zu retournieren.

Nun kann die Methode, die für das eigentliche Bezahlen zuständig ist, ausgeführt werden. Diese benötigt, wie in der Abbildung OnPostCreateOrder erkennbar ist, den Gesamtpreis aller Waren (=***Total***) und wie viele Produkte gekauft wurden (=***ProductIdentifiers***). Bei der Erstellung des API-Anfragekörpers (eng. API-RequestBody) wird zusätzlich noch der dreistellige Währungscode (=***currency_code***) und die Absicht (=***intent***) benötigt. Im Falle der Diplomarbeit wurde der currency_code auf EUR gestellt, da Euro abgebucht werden sollen und der intent auf capture gesetzt, da die Verbuchung der Bestellung unmittelbar

nach der Bezahlung stattfinden soll. Zuletzt wird dieser RequestBody an die URL ***https://api-m.sandbox.paypal.com/v2/checkout/orders*** gesendet. Die Antwort wird zuerst als Zeichenkette gelesen, danach in ein JSON umgewandelt und schließlich kann die orderId gelesen werden. Mit dieser Auftragsnummer (=orderId), wird in der nächsten Methode weitergearbeitet.

```
1  public JsonResult OnPostCreateOrder()
2  {
3      Total = TempData["Total"]?.ToString() ?? "";
4      ProductIdentifiers = TempData["ProductIdentifiers"]?.ToString() ??
           "";
5      TempData.Keep();
6
7      JsonObject orderRequestBody = CreateRequestBody();
8
9      string accessToken = GetPayPalAccessToken();
10     string url = PayPalUrl + "/v2/checkout/orders";
11
12     string orderId = "";
13     using (var client = new HttpClient())
14     {
15         client.DefaultRequestHeaders.Add("Authorization", "Bearer " +
               accessToken);
16
17         var requestMessage = new HttpRequestMessage(HttpMethod.Post,
               url);
18         requestMessage.Content = new
               StringContent(orderRequestBody.ToString(), null,
               "application/json");
19
20         var responseTask = client.SendAsync(requestMessage);
21         responseTask.Wait();
22
23         var result = responseTask.Result;
24         var readTask = result.Content.ReadAsStringAsync();
25         readTask.Wait();
26
27         var strResponse = readTask.Result;
28         var jsonResponse = JsonNode.Parse(strResponse);
29         orderId = jsonResponse["id"]?.ToString() ?? "";
30     }
31     var response = new { Id = orderId };
32     return new JsonResult(response);
33 }
```

Code 3.11: Sendet einen HTTP-Post-Request an die Paypal-Sandbox-Adresse mit der Erweiterung v2/checkout/orders. Als Response erhält die Methode einen String, der in JSON umgewandelt wird; dieser enthält die OrderId, die retourniert wird.

Mithilfe der Auftragsnummer und dem Zugangstoken; sendet die Methode OnPostCompleteOrder, wie im Codesegment 3.12 zu sehen ist, einen API-Request. Die Adresse, an die gesendet wird, ist ***https://api-m.sandbox.paypal.com/v2/checkout/orders/orderId/capture***, sie wird verwendet, um ein capture auf die mitgegebene orderId zu setzen. Sollte dieses capture erfolgreich sein, wird die Methode ***ChangeStatusPayed*** ausgeführt, die den Bestellungsstatus in der Datenbank auf "SUCCESS" stellt und die Auftragsnummer einträgt.

```
1  public JsonResult OnPostCompleteOrder([FromBody] JsonObject data)
2  {
3      var orderID = data["orderID"]!.ToString();
4
5      string accesToken = GetPayPalAccessToken();
6      string url = PayPalUrl + "/v2/checkout/orders/" + orderID +
           "/capture";
7
8      using (var client = new HttpClient())
9      {
10         client.DefaultRequestHeaders.Add("Authorization", "Bearer " +
               accesToken);
11
12         var requestMessage = new HttpRequestMessage(HttpMethod.Post,
               url);
13         requestMessage.Content = new StringContent("", null,
               "application/json");
14
15         var responseTask = client.SendAsync(requestMessage);
16         responseTask.Wait();
17
18         var result = responseTask.Result;
19
20         var readTask = result.Content.ReadAsStringAsync();
21         readTask.Wait();
22
23         var strResponse = readTask.Result;
24         var jsonResponse = JsonNode.Parse(strResponse);
25
26         string paypalOrderStatus = jsonResponse["status"]?.ToString()
               ?? "";
27         string paypalOrderId = jsonResponse["id"]?.ToString() ?? "";
28
29         if (paypalOrderStatus == "COMPLETED" && paypalOrderId != "")
30         {
31             string success_message = "SUCCESS";
32             ChangeStatusPayed(success_message, paypalOrderId);
33             TempData.Clear();
34             return new JsonResult("success");
```

```
35          }
36      }
37
38      return new JsonResult("");
39  }
```

Code 3.12: Die Methode sendet einen HTTP-Post-Request an die PayPal-Sandbox-Adresse mit /v2/checkout/orders/orderId/capture, um die Bestellung abzuschließen und den Status sowie die OrderId in der Datenbank zu aktualisieren.

Sollte an einem dieser Punkte etwas schieflaufen, greift die Methode OnPostCancelOrder. Diese Methode ruft, wie in der Codestelle 3.13 zu sehen, ChangeStatusPayed auf und ändert den Bestellstatus in der Datenbank auf “CANCELED”.

```
1  public JsonResult OnPostCancelOrder([FromBody] JsonObject data)
2  {
3      var orderID = data["orderID"]!.ToString();
4      string cancle_message = "CANCELED";
5      ChangeStatusPayed(cancle_message, orderID);
6      return new JsonResult("");
7  }
```

Code 3.13: Falls die Erstellung der Bestellung scheitern sollte, setzt diese Methode den Status in der Datenbank auf "CANCELED".

4 Website

Die in diesem Diplomarbeitsprojekt entwickelte Website wurde mit dem ASP.NET Core Web-App MVC Framework entwickelt. Sie dient der Mensa dazu, Menüs anzulegen, in einer Datenbank zu speichern und all diese gespeicherten Menüs mittels der JavaScript Bibliothek DataTables zu visualisieren. Desweiteren können die Menüs auf der Website bearbeitet oder sogar aus der Datenbank gelöscht werden. Eine weitere Funktion dieser Anwendung ist es, die Anzahl der vorhandenen Bestellungen eines Menüs anzuzeigen. Zugleich gibt es die Möglichkeit, bestimmte Statistiken aus der Datenbank auszulesen und in einem Diagramm wiederzugeben. Diese Funktion wurde unter Verwendung der JavaScript-Bibliothek Chart.js implementiert.

4.1 ASP.NET Core Web-App MVC

ASP.NET Core ist ein von Microsoft entwickeltes Framework für die Entwicklung von modernen, cloud-basieten, internetverbundenen Anwendungen wie Web-Apps, Webdienste, IoT-Apps und mobile Back-Ends. ASP.NET Core ist eine Neuauflage von ASP.NET und zielt darauf ab, durch die Änderungen der Architektur eine modulare und performantere Web-Entwicklungsplattform zu bieten (vgl. tdykstra (n.d.)).

ASP.NET Core ist Teil der .NET-Plattform, was bedeutet, dass Entwickler die Vorteile der .NET-Standardbibliothek und anderer .NET-Features wie Entity Framework Core für Datenbanken und C# für die Programmierung nutzen können.

4.1.1 MVC-Konzept

Bei diesem Konzept wird die ganze Applikation in Modelle (eng. Models), Ansichten (eng. Views) und Steuerungen (eng. Controllers) unterteilt. Zum einen dient diese Trennung zur Strukturierung des Projekts und zum anderen zur

Aufteilung der Web-Anwendung in einen Frontend- und einen Backend-Teil. Diese Trennung von Frontend und Backend vereinfacht das Testen der jeweiligen Gebiete und die Entwickler können voneinander unabhängig die Anwendung erweitern, deswegen ist dieses Framework bei den Entwicklern weit verbreitet (vgl. Mehta (n.d.)).

Model (M): Hierbei handelt es sich um C#-Klassen, die von den Controllern zur Weiterverarbeitung genützt werden.

View (V): Die Views sind im Grunde die HTML-Seiten, die am Client zu sehen sind und bestehen größtenteils aus HTML, CSS und ebenfalls ein wenig aus Razor (C#).

Controller (C): Bei den Controllern handelt es sich um die Logik hinter der Applikation. Bei einem Web-Aufruf kümmert sich der Controller darum, dass die richtigen Models verwendet werden und die richtigen Views an den Client übergeben werden.

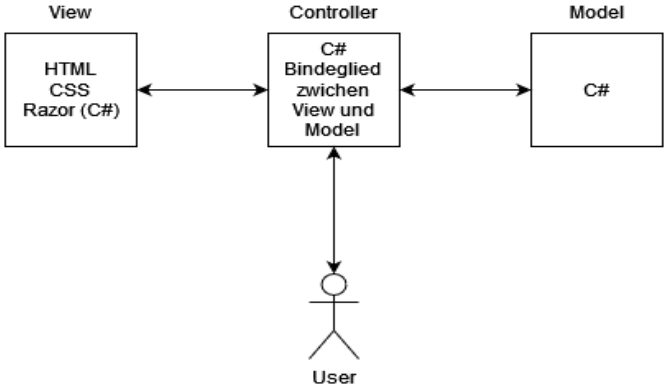

Abbildung 4.1: Hier wird das MVC-Konzept abgebildet. Wenn ein User Daten an den Controller sendet, wird entweder direkt eine Änderung gemacht und an die View weitergeleitet oder die Daten werden über das Model verarbeitet und dann an die View weitergeleitet.

4.1.2 Aufbau eines ASP.NET Core MVC Projekts

Wenn ein neues ASP.NET Core Web App Projekt erstellt wird, werden für die ***Models***, die ***Views*** und die ***Controllers*** eigene Ordner erstellt.

Ebenfalls wird noch ein ***wwwroot*** Ordner erstellt. In diesem globalen Ordner werden die JavaScript-Dateien für die Dynamisierung der HTML-Seiten im ***js***

Ordner, die CSS-Dateien für die Formatierung der HTML-Seiten im ***css*** Ordner und verschiedene Bibliotheken, wie zum Beispiel Bootstrap oder jQuery im ***lib*** Ordner gespeichert.

Desweiteren wird beim Erstellen eines neuen Projekts ein ***Shared*** Ordner erstellt. In diesem Ordner wird standardmäßig eine View mit dem Namen ***Layout.cshtml*** erstellt. Diese View ist, wie der Name schon verrät, einerseits für das Layout zuständig und andererseits werden dort auch die JavaScript-Bibliotheken global eingebunden, sodass diese Bibliotheken für alle Views verwendbar sind. Ein Beispiel dafür ist die ***jQuery***-Bibliothek. Es können aber auch sogenannte ***PartialViews*** in dem Shared Ordner gespeichert werden. Diese ***PartialViews*** sind dazu da, um einen Teil einer View, der möglicherweise in einer anderen View ebenfalls vorkommt, auszulagern. Dadurch kann dann einfach an der gewünschten Stelle diese ***PartialView*** eingebunden werden.

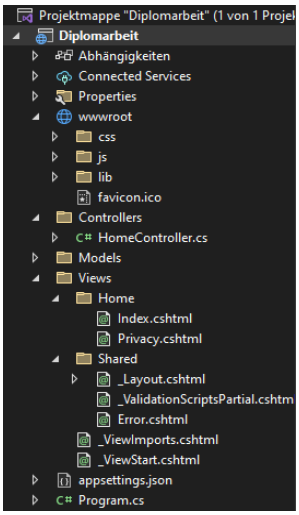

Abbildung 4.2: Ordnerstruktur eines ASP.NET Core Web App Projekts

Wird ein Controller erstellt, nennt man ihn ***NameController.cs***, dabei kann man für ***Name*** einen beliebigen Namen wählen. Bei den Views ist es dann wichtig, einen Unterordner mit diesem Namen zu erstellen und dort die Views zu speichern. Dies ist beispielsweise in der Abbildung 4.2 so der Fall, wo zum ***HomeController.cs*** ein dazugehöriger ***Home*** Ordner in den Views erstellt wurde.

In einem Controller befinden sich mehrere Methoden, die Actions genannt werden. Diese Methoden verlangen als Rückgabewert eine View. Wenn man nur ein View-Objekt ohne Parameter zurückgibt, dann sucht der Controller aus dem Viewordner eine View, die dem Namen der Action entspricht. Man kann dem

View-Objekt allerdings einen String-Parameter übergeben, der dem Namen der gewünschten View entspricht; dann wird diese View vom Controller verwendet. Im HomeController aus der Abbildung 4.2 sind zwei Actions vorhanden. Einmal eine Action mit dem Namen ***Index()*** und eine weitere mit dem Namen ***Privacy()***. Wenn jetzt beispielsweise in der ***Index()***-Methode nur ein ***return View();*** vorhanden ist, dann sucht der Controller in ***Views/Home*** eine Datei mit dem Namen ***Index.cshtml*** und wird diese an den Client weiterleiten. Man könnte aber dem ***return*** einen String-Parameter hinzufügen, zum Beispiel ***return View(„Name“);*** dann würde der Controller nach einer View mit dem Namen ***Name.cshtml*** suchen und nicht fündig werden.

4.1.3 Middleware

Die Middleware ist eine kleine Software, die sich um die Requests bzw. die Responses kümmert. Wenn man beispielsweise einen Web-Aufruf durchführt mit der URL

https://www.server.at/Controllername/Actionname

sucht die Middleware den Controller mit dem ***Controllernamen*** im Controller-Ordner und sucht die zum Request dazugehörige Action mit dem ***Actionnamen***. Nachdem der Controller dann eine fertige View erstellt hat, übergibt er diese an die Middleware. Die Middleware erstellt dann eine Response und schickt diese an den Client zurück.

Eine weitere Aufgabe der Middleware ist es, die erstellten Views in puren HTML-Code zu übersetzten, da es sich bei den Views um Razor-Pages handelt und diese deshalb C#-Code enthalten können.

Die Middleware hat noch viel mehr Einsatzmöglichkeiten, wie zum Beispiel das Session-Handling, das in diesem Diplomarbeitsprojekt nicht gebraucht wurde oder das Exception-Handling, das ebenfalls von der Middleware gehandhabt wird.

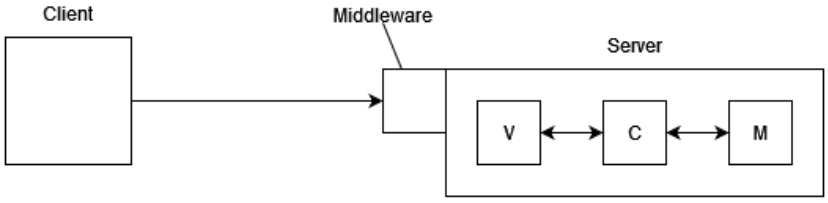

Abbildung 4.3: Hier ist ein Request vom Client zum Server zu sehen. Dabei läuft der Request über die Middleware, die im Grunde den Übersetzer spielt.

4.2 Speichern von Menüs

Um überhaupt irgendein Menü online verkaufen zu können, muss die Mensa zuvor im Stande sein, ein Menü anzulegen. Diese Möglichkeit ein Menü anzulegen und ebenfalls in einer Datenbank zu speichern, ist nur eine der vielen Funktionen dieser Website.

4.2.1 Controller

Die Web-Anwendung benötigt eine Logik und dafür sind die Controller zur Stelle. Um einen besseren Überblick über das Projekt zu haben, wird eine Web-Anwendung meistens in mehrere Controller, die sich normalerweise jeweils mit einem Themengebiet befassen, unterteilt. In dieser Diplomarbeit wurden ebenfalls ein Controller für die Menüs mit dem Namen ***MenuController.cs*** und ein Controller für die Bestellungen mit dem Namen ***OrderController.cs*** erstellt.

Für das Speichern der Menüs wird im ***MenuController.cs*** eine Action mit dem Namen ***SaveMenus()*** erstellt. In Zeile 1 im Codeabschnitt 4.2 ist eine sogenannte Annotation vorhanden. Diese Annotation sagt aus, bei welcher Art des Requests die darunter stehende Action ausgeführt wird. Ist vor einer Action keine Annotation vorhanden, so wird diese Action standardmäßig mit einem GET-Request ausgeführt.

Im ***Layout.cshtml*** existiert oben in der Navigationsleiste ein Button namens ***Menü hinzufügen***, der beim Drücken die ***SaveMenus()***-Methode mit einem GET-Request aufruft (siehe Codeabschnitt 4.1).

```
<a class="nav-link text-dark" asp-controller="Menu"
    asp-action="SaveMenus">Menüs hinzufügen</a>
```

Code 4.1: Hier ist der Button für das Speichern eines Menüs in der Navigationsleiste in der Datei Layout.cshtml.

Wird die ***SaveMenus()***-Action mit einem GET-Request ausgeführt, so holt sich die Methode den ersten Preis aus der Datenbank und speichert den Schüler- und den Lehrerpreis in ein ***MenuDTO*** (Zeilen 6-8 in Codeabschnitt 4.2). ***DTOs*** (Data Transfer Object) sind Objekte, die nur die benötigten Felder besitzen und zum Transfer vom Backend zum Frontend verwendet werden, um die zu transferierende Datenmenge zu reduzieren. Ebenfalls wird in diesem ***MenuDTO*** noch das aktuelle Datum gespeichert, wie in Zeile 9 in Codeabschnitt 4.2 zu sehen ist.

Zum Schluss wird die View zurückgegeben. Dieser View kann ein ***Modell*** (engl. Model), in Form beliebiger Daten mitgegeben werden. Der Datentyp dieser Daten muss zuerst am Anfang der View mit ***@model Name des Datentyps*** festgelegt werden, dann kann in der View auf diese übergebenen Daten auch mit ***@Model*** darauf zugegriffen werden. In der ***SaveMenus.cshtml***-View wurde am Anfang ebenfalls ein Model mit dem Datentyp ***MenuDTO*** eingebunden, dadurch kann im Controller nun in der ***SaveMenus()***-Action die View mit einem Parameter des Datentyps ***MenuDTO***, in welcher der Preis und das aktuelle Datum gespeichert sind, übergeben werden.

```
1  [HttpGet]
2  public async Task<IActionResult> SaveMenus()
3  {
4      int priceId = 1;
5      MenuDTO menuDTO = new MenuDTO();
6      PriceForMenu price = await _context.Prices.FindAsync(priceId);
7      menuDTO.PriceStudent = price.PriceStudent;
8      menuDTO.PriceTeacher = price.PriceTeacher;
9      menuDTO.Date = DateOnly.FromDateTime(DateTime.Now);
10     return View(menuDTO);
11 }
```

Code 4.2: Die GET-Variante der SaveMenus()-Methode, um die View zum Speichern der Menüs anzuzeigen.

Wird nach dem Eingeben der Daten der Speichern-Button betätigt, werden die eingegebenen Daten mit einem POST-Request an die ***SaveMenus()***-Methode mit der POST-Annotation gesendet. Diese eingegebenen Daten werden als Parameter von der Methode erwartet.

Zuerst werden die übergebenen Daten auf ihre Richtigkeit überprüft. Diese Überprüfung von falschen Eingaben wird auch ***Validation*** genannt. Es gibt zwei

Arten von Validation. Einerseits gibt es die serverseitige Validation, die die Daten erst überprüft, wenn sie schon abgeschickt worden sind. Andererseits gibt es die clientseitige Validation, die die Daten nach der Eingabe mittels Javascript überprüft und direkt (falls nötig) eine Fehlermeldung ausgibt. Bei der Validation in der Action handelt es sich um eine serverseitige Validation, da die Daten vom Client zum Server geschickt worden sind. In Codeabschnitt 4.3 ist die Validierung für die Vorspeise zu sehen. Dort wird überprüft, ob überhaupt etwas eingegeben worden ist oder ob die Eingabe mindestens aus drei Zeichen besteht. Falls dies nicht der Fall ist, wird eine Error Message erstellt. Die Erstellung dieser Error Message ist in Zeile 3 im Codeabschnitt 4.3 zu sehen. Dabei ist zu beachten, dass der erste Parameter für den Namen der Error Message steht und beim zweiten Parameter handelt es sich um die Nachricht, die angezeigt werden soll. Um diese Error Message in der View anzuzeigen, muss nur der Helper Tag ***asp-validation-for*** mit dem Namen der Error Message aufgerufen werden. Beispiel dazu ist im Kapitel 4.2.2 View zu sehen.

```
1 if (menuDTO.Starter == null || menuDTO.Starter.Trim().Length < 3)
2 {
3     ModelState.AddModelError("Starter", "Vorspeise muss mehr als 3
      Zeichen enthalten!");
4 }
```

Code 4.3: Die Überprüfung der übergebenen Daten auf ihre Richtigkeit. Hier wird überprüft, ob überhaupt etwas eingegeben worden ist oder die Länge mindestens aus drei Zeichen besteht.

Ist die Validation abgeschlossen, wird zunächst überprüft, ob eine solche Error Message erstellt worden ist (Zeile 1 in Codeabschnitt 4.4). Ist dies der Fall, wird die View mit den eingegebenen Daten wieder zurückgegeben und dem Client wird diese Fehlermeldung angezeigt. Wenn keine neue Error Message erstellt worden ist, wird die Datenbank aufgerufen und die Daten darin gespeichert.

Um die Datenbank aufrufen zu können, muss ganz oben im Controller ***MenuContext*** eingebunden werden. Mit dem ***_context.Menues.Add(Menu)*** wird das eingegebene Menü der Datenbank hinzugefügt, aber noch nicht gespeichert. Es wird erst gespeichert, wenn ***_context.SaveChangesAsync()*** aufgerufen wird, wie in Zeile 13 im Codeabschnitt 4.4 zu sehen ist. Zudem wird dabei gleich überprüft, ob es funktioniert hat, indem geschaut wird, ob der Rückgabewert dieser Methode gleich 1 ist.

Im Nachhinein wird überprüft, ob das Speichern erfolgreich war und dazu eine Erfolgs-/Misserfolgsnachricht an den Client gesendet. Diese Nachricht wird in einem ***TempData*** gespeichert (Zeile 22 und 27 in Codeabschnitt 4.4). Ein ***TempData*** wird verwendet, um Daten von einer View zu einem Controller, von

einem Controller zu einer View oder von einer Action zu einer anderen Action zu senden. Diese Daten sind nur temporär in ***TempData*** gespeichert und werden nach dem Verwenden automatisch wieder gelöscht (vgl. *TempData in ASP.NET MVC* (n.d.)).

```
1  if(ModelState.IsValid)
2  {
3      try
4      {
5          Menu menu = new Menu()
6          {
7              Prices = await
                   _context.Prices.FindAsync(menuDTO.WhichMenu),
8              Starter = menuDTO.Starter,
9              MainCourse = menuDTO.MainCourse,
10             Date = menuDTO.Date
11         };
12         _context.Menues.Add(menu);
13         isSuccess = (await _context.SaveChangesAsync()) == 1;
14     }
15     catch (Exception ex)
16     {
17         Console.WriteLine(ex.ToString());
18     }
19
20     if (isSuccess)
21     {
22         TempData["SuccessAlert"] = "Menü wurde erfolgreich
               hinzugefügt!";
23         return RedirectToAction("SaveMenus");
24     }
25     else if (!isSuccess)
26     {
27         TempData["NoSuccessAlert"] = "Menü konnte nicht gespeichert
               werden!";
28         return RedirectToAction("SaveMenus");
29     }
30 }
```

Code 4.4: Überprüfung, ob eine Error Message erstellt worden ist; ist das nicht der Fall, wird das eingegebene Menü in der Datenbank gespeichert.

4.2.2 View

Damit in der Web-Anwendung auch etwas zu sehen ist, werden die Views verwendet. Um in Visual Studio eine neue View zu erstellen, gibt es die Möglichkeit,

im Controller eine neue Action, also eine Methode, zu definieren und im Nachhinein auf die Action rechts zu klicken und "Ansicht hinzufügen" auszuwählen. Daraufhin erstellt Visual Studio selbst einen passenden Unterordner im Views Ordner mit dem Controllernamen sowie eine passende View, die den gleichen Namen wie die Action besitzt. Natürlich kann dies auch händisch gemacht werden, dabei sollte beachtet werden, dass in dem Views Ordner ein Unterordner mit dem Namen des Controllers erstellt wird und dort können die Views gespeichert werden.

Zum Erstellen der View für das Speichern eines Menüs wurde zuerst eine neue Action mit dem Namen ***SaveMenus()*** erstellt und danach der oben genannte Schritt durchgeführt, um die dazugehörige View zu erstellen. Im Endeffekt sollte eine Datei mit dem Namen ***SaveMenus.cshtml*** in ***Views/Menu*** erstellt worden sein. In diese Datei wird mit HTML ein Formular für die Eingabe des zu speichernden Menüs einprogrammiert.

Wie schon im Kapitel 4.1.2 Aufbau eines ASP.NET Core MVC Projekts erklärt wurde, können Teile einer View in eine PartialView ausgelagert werden; dies geschieht hier mit dem Formular, das in die Datei ***_SaveMenuFormPatrialView.cshtml*** gespeichert wird. Dieses Auslagern bietet sich an, weil dasselbe Formular nochmals zum Bearbeiten des Menüs benötigt wird.

Das Menü besteht aus den Feldern (engl. Properties) ***MenuId***, ***Starter***, ***MainCourse***, ***Price*** und ***Date***. Für diese Properties wurde jeweils ein Eingabefeld erstellt. Ein solches Eingabefeld ist in Codeabschnitt 4.5 zu sehen. Mit der 4. Codezeile wird das Eingabefeld erstellt; dabei ist zu beachten, dass ein sogenannter ***Tag Helper***, zum Beispiel ***asp-for***, verwendet wurde, um zu sagen, zu welchem Parameter in der Action der eingegebene Text gehört.

In der Codezeile 7 wird mit ****** ein Bereich geschaffen, in dem es möglich ist, bei falschen Eingaben eine Fehlermeldung, wie zum Beispiel, dass die Eingabe zu kurz ist, ausgegeben werden kann. Der Tag Helper ***asp-validation-for*** ist sozusagen dazu da, um zu sagen, dass die Fehlermeldung für die Vorspeise an dieser Stelle ausgegeben werden soll.

Im Grunde genommen sieht der Code für die anderen Properties gleich aus wie im Codeabschnitt 4.5, außer für MenuId, denn für diese wird kein Eingabefeld benötigt, da diese in der Datenbank automatisch vergeben wird, und für den Preis, aber dies wird im Kapitel 4.2.3 Handhabung mit dem Preis erklärt.

Zum Schluss der PartialView wird ein Speichern-Button eingefügt, der beim Drücken einen ***HttpPost*** request an den Controller sendet, damit die eingegebenen Daten in die Datenbank gespeichert werden können.

```
1 <div class="form-group row">
2     <label class="col-sm-2 col-form-label">Vorspeise:</label>
3     <div class="col-sm-6">
4         <input asp-for="Starter" placeholder="Suppe"
              class="form-control"/>
5     </div>
6     <div class="col-sm-4">
7         <span asp-validation-for="Starter"></span>
8     </div>
9 </div>
```

Code 4.5: Eingabefeld für die Vorspeise aus der _SaveMenuFormPartialView.cshtml

Diese PartialView für das Formular wird dann in der View ***SaveMenus.cshtml*** innerhalb eines ***<form>***-Tags (Zeile 1 und 3 im Codeabschnitt 4.6) mit dem ***<partial>***-Tag (Z. 2 in Codeabschnitt 4.6) aufgerufen. Im ***<form>*** Tag sind wieder ***Helper Tags*** vorhanden. Zum einen gibt es den ***asp-controller***-Tag, bei dem der Name des Controllers angegeben wird. Zum anderen gibt es den ***asp-action***-Tag, bei dem der Name der Action angegeben wird. Dadurch ist bekannt, zu welchem Controller und zu welcher Action der Request gesendet wird, nachdem der Speichern-Button getätigt wurde. Des Weiteren wird noch mit dem ***method***-Tag die Art des Requests übergeben. Im ***<partial>***-Tag wird lediglich der Name der PartialView und ein Model übergeben.

```
1 <form asp-controller="menu" asp-action="SaveMenus" method="post">
2     <partial name="_SaveMenuFormPartialView" model="@Model" />
3 </form>
```

Code 4.6: Aufruf der PartialView ***_SaveMenuFormPartialView.cshtml*** in der View ***SaveMenus.cshtml***

4.2.3 Handhabung mit dem Preis

In der Mensa gibt es drei Arten von Menüs: Menü 1, 2 und 3. Diese Menüs haben jeweils einen eigenen Schüler- und einen eigenen Lehrerpreis. Deshalb wurde in dieser Diplomarbeit der Preis in einer eigenen Tabelle in der Datenbank gespeichert, damit dieser Preis von der Art des Menüs abhängig wird. Wird nun beim Eingeben eines Menüs die Art des Menüs geändert, so wird automatisch der dazugehörige Preis aus der Datenbank geholt und angezeigt.

Dazu wurde im Shared Ordner eine PartialView ***_PricesPartialView.cshtml*** erstellt, in der einfach der Schüler- bzw. der Lehrerpreis angezeigt wird. Dabei gibt es für beide Preise jeweils ein ***<label>***, wo der Preis ausgegeben wird und ein

<input>, wobei der Type davon ***hidden*** ist und deshalb nicht am Client angezeigt wird. Dieses Feld wird benötigt, um dort den Preis zu lagern, damit, wenn das ausgefüllte Formular abgeschickt wird, auch der richtige Preis mitgeschickt wird. Diese PartialView für den Preis wird in der ***_SaveMenusFormPartialView.cshtml*** in einem *** *** mit ***id="prices"*** mit dem ***<partial>***-Tag aufgerufen.

Wenn die Art des Menüs geändert wird, soll der richtige Preis aus der Datenbank geholt werden und am Client angezeigt werden. Dazu wurde einerseits die Javascript Bibliothek ***jQuery*** verwendet und andererseits ***AJAX*** (Asynchronous Javascript And XML) verwendet. AJAX ermöglicht es, im Hintergrund einen asynchronen Datenaustausch mit dem Server durchzuführen, um Teile der Seite zu aktualisieren, ohne die ganze Seite neu laden zu müssen.

Um jQuery und AJAX verwenden zu können, muss in der View am Ende ein ***@section Scripts{}*** vorhanden sein. In diesem Abschnitt können Javascript-Dateien eingefügt werden, aber auch einfach mit dem ***<script>***-Tag Javascript programmiert werden. In einem solchen ***<script>***-Tag wird auch der AJAX-Call für das Holen des Preises hineingeschrieben.

Wie im Codeabschnitt 4.7 zu sehen ist, wird in der 1. Zeile mit jQuery abgefragt, ob sich die Art des Menüs geändert hat. Ist dies der Fall, so wird der Value von diesem Feld geholt und ein AJAX-Call getätigt. Dieser Call benötigt mehrere Felder. Zuerst wird die ***url*** übergeben, um festzustellen, zu welcher Action in welchem Controller dieser Aufruf gesendet wird und mit welchem ***type***. Der ***dataType*** gibt an, was beim Zurücksenden erwartet wird, wobei dies wie im Codeabschnitt 4.7 eine HTML-Seite oder einfach nur ein JSON mit beliebigen Daten sein kann. Bei ***data*** können Daten mitgesendet werden, die beim Controller in der Action als Parameter erwartet werden. Zum Schluss gibt es noch ***success*** und ***error***. ***success*** wird ausgeführt, wenn der AJAX-Call erfolgreich war und der richtige Datentyp zurückgegeben wurde. Wenn der Call fehlerhaft war, wird ***error*** ausgeführt; dort kann eine beliebige Fehlermeldung ausgegeben werden (vgl. Startup & Code (n.d.)).

```
1  $('#WhichMenu').change(function () {
2      var whichMenu = $('#WhichMenu').val();
3      $.ajax({
4          url: '@Url.Action("_PricesPartialView","Menu")',
5          dataType: 'html',
6          type: 'GET',
7          data: {
8              priceId: whichMenu
9          },
10         success: function (data) {
11             $('#prices').html(data).fadeIn();
12         },
```

```
13          error: function () {
14              alert('Error');
15          }
16      });
17  });
```

Code 4.7: Beim Ändern der Art des Menüs wird ein AJAX-Call aufgerufen, der den richtigen Preis in der Datenbank holt und auf der Seite den geholten Preis anzeigt.

Problematik mit dem Auslagern des Preises in einer eigenen Tabelle ist, dass, wenn ein Preis nun geändert wird, im Nachhinein der alte Preis nicht mehr bekannt ist. Damit der alte Preis bekannt bleibt, müsste man noch ein Start- und Enddatum hinzufügen, damit man herausfinden kann, zu welchem Zeitraum dieser Preis gegolten hat.

Abbildung 4.4: Hier ist das fertige Formular für das Hinzufügen eines Menüs zu sehen.

4.3 Menüs anzeigen

Eine weitere Funktion dieser Website ist es, alle gespeicherten Menüs in einer Tabelle anzuzeigen. Dafür wird die Javascript Bibliothek ***DataTables*** verwendet, da diese Bibliothek es ermöglicht, sehr einfach Tabellen mit integrierter Suchmöglichkeit oder Sortiermöglichkeit zu erstellen (vgl. *DataTables | Javascript table library* (n.d.)). Des Weiteren kann man ein Menü auswählen, um es zu bearbeiten oder sogar zu löschen.

4.3.1 Eine Tabelle erstellen mit DataTables

Zuerst wurde wieder ein neuer Button in der ***Layout.cshtml*** mit dem Namen ***Alle Menüs***, der beim Betätigen die Action ***ShowAllMenus()*** ausführt, hinzugefügt.

Die ***ShowAllMenus()***-Action holt sich aus der Datenbank alle vorhandenen Menüs und sortiert sie mit ***.OrderByDescending(m=>m.MenuId)*** nach der MenuId absteigend, damit am Client das zuletzt hinzugefügte Menü ganz oben steht. Danach werden alle Menüs mit einer ***foreach()***-Schleife durchiteriert und die Menüs zu einem MenuDTO geändert und in eine Liste von MenuDTOs gespeichert (siehe Codeabschnitt 4.8). Zum Schluss wird die View mit der Liste von MenuDTOs als Parameter zurückgegeben.

```
1  menus = await _context.Menues.Include(m =>
       m.Prices).OrderByDescending(m => m.MenuId).ToListAsync();
2  foreach(Menu menu in menus)
3  {
4      MenuDTO menuDTO = new();
5      menuDTO.MenuId = menu.MenuId;
6      menuDTO.WhichMenu = menu.Prices.PriceId;
7      menuDTO.Starter = menu.Starter;
8      menuDTO.MainCourse = menu.MainCourse;
9      menuDTO.Date = menu.Date;
10     menuDTO.PriceStudent = menu.Prices.PriceStudent;
11     menuDTO.PriceTeacher = menu.Prices.PriceTeacher;
12
13     menuDTOList.Add(menuDTO);
14 }
```

Code 4.8: Menüs werden aus der Datenbank geholt, nach MenuId absteigend sortiert und in eine Liste von MenuDTO's gespeichert.

In der View muss zuerst ganz oben ein ***@model List<MenuDTO>*** definiert werden, damit der Zugriff auf die übergebenen Daten gewährt wird. Danach werden drei Buttons erstellt. Der Erste davon ist dafür zuständig, dass die Seite beim Drücken neu geladen wird. Dies geschieht, indem mit jQuery abgefragt wird, ob der Button betätigt wurde und im Nachhinein wird in Javascript der Befehl ***location.reload();*** ausgeführt. Wird der zweite Button gedrückt, wird man zum Formular für das Bearbeiten eines Menüs weitergeleitet. Wie das funktioniert, wird im Kapitel 4.3.2 Menü bearbeiten erklärt. Der dritte Button ist dafür da, um das ausgewählte Menü aus der Datenbank zu löschen. Wie das funktioniert, wird im Kapitel 4.3.3 Menü löschen erklärt.

Um DataTables nutzen zu können, muss zuerst eine Tabelle mit den ganzen Einträgen erzeugt werden. Diese Tabelle benötigt noch eine Id, damit sie später

im Javascript-Teil angesprochen werden kann. Es ist wichtig, wie im Codeabschnitt 4.9 in Zeile 2-11, einen ***<thead>*** mit den Überschriften der einzelnen Felder zu machen, ansonsten wird die Tabelle von DataTables nicht erkannt und es funktioniert nicht. Im ***<tbody>*** werden mit ***@foreach*** alle MenuDTOs in dem Model durchiteriert und die Felder in der gleichen Reihenfolge wie die Überschriften ausgegeben (siehe Codeabschnitt 4.9 Zeile 12-25).

```
1  <table id="ShowAllMenues" class="table table-striped"
       style="width:100%">
2      <thead>
3          <tr>
4              <td> ID </td>
5              <td> Menü </td>
6              <td> Vorspeise </td>
7              <td> Hauptspeise </td>
8              <td> Preis Schüler </td>
9              <td> Preis Lehrer </td>
10             <td> Datum </td>
11     </thead>
12     <tbody>
13         @foreach (MenuDTO m in Model)
14         {
15             <tr>
16                 <td>@m.MenuId</td>
17                 <td>@m.WhichMenu</td>
18                 <td>@m.Starter</td>
19                 <td>@m.MainCourse</td>
20                 <td>@m.PriceStudent</td>
21                 <td>@m.PriceTeacher</td>
22                 <td>@m.Date</td>
23             </tr>
24         }
25     </tbody>
26 </table>
```

Code 4.9: Hier wird die Tabelle mit allen vorhandenen Menüs aus der Datenbank erzeugt.

Zuerst wurde die Bibliothek installiert und im ***wwwroot/js*** Ordner gespeichert. Da DataTables eine Javascript-Bibliothek ist, muss das Erstellen des DataTables im ***@section Scripts{}*** erfolgen. Um diese Bibliothek verwenden zu können, muss sie zuerst eingebunden werden.

```
1  <script src="~/js/datatables.min.js"></script>
```

Code 4.10: Hier wird das Einbinden der Javascript-Bibliothek DataTables gezeigt.

Nachdem die Bibliothek eingebunden wurde, kann die Tabelle mit DataTables erzeugt werden. Dies geschieht mit ***$('#ShowAllMenues').DataTables({});***. Wie

im Codeabschnitt 4.11 zu sehen ist, können dann verschiedene Einstellungen festgelegt werden. Das ***select: true*** zum Beispiel sagt aus, dass man Zeilen in der Tabelle selektieren kann. Da Englisch standardmäßig für den DataTable verwendet wird, kann die Sprache auch mit der Einstellung ***language*** geändert werden (siehe Codeabschnitt 4.11 Zeile 3-13).

```
1  var table = $('#ShowAllMenues').DataTable({
2      select: true,
3      "language": {
4          "lengthMenu": "_MENU_ Zeilen pro Seite",
5          "zeroRecords": "Es wurde kein Eintrag gefunden!",
6          "info": "Seite _PAGE_ von _PAGES_",
7          "infoEmpty": "Es wurde kein Eintrag gefunden!",
8          "infoFiltered": "(von _MAX_ Einträge)",
9          "search": "Suchen:",
10         "paginate": {
11             "next": "->",
12             "previous": "<-"
13         },
14         "select": {
15             "rows": {
16
17                 "1": "1 Zeile ausgewählt"
18             }
19         }
20     },
21     order: [0, 'desc']
22 });
```

Code 4.11: Hier wird DataTables initialisiert und ein paar Einstellungen werden vorgenommen.

Der Bearbeiten-Button und der Löschen-Button sind am Anfang deaktiviert und können nicht gedrückt werden. Erst wenn ein Eintrag in der Tabelle ausgewählt wird, werden die beiden Buttons aktiviert und können gedrückt werden (siehe Codeabschnitt 4.12).

```
1  table.on('select deselect', function () {
2      var selectedRows = table.rows({ selected: true }).count();
3
4      if (selectedRows === 1) {
5          $('#deleteMenu').prop('disabled', false);
6          $('#editMenu').prop('disabled', false);
7      } else {
8          $('#deleteMenu').prop('disabled', true);
9          $('#editMenu').prop('disabled', true);
10     }
11 });
```

Code 4.12: Wenn ein Eintrag in der Tabelle selektiert wird, werden der Bearbeiten-Button und der Löschen-Button aktiviert.

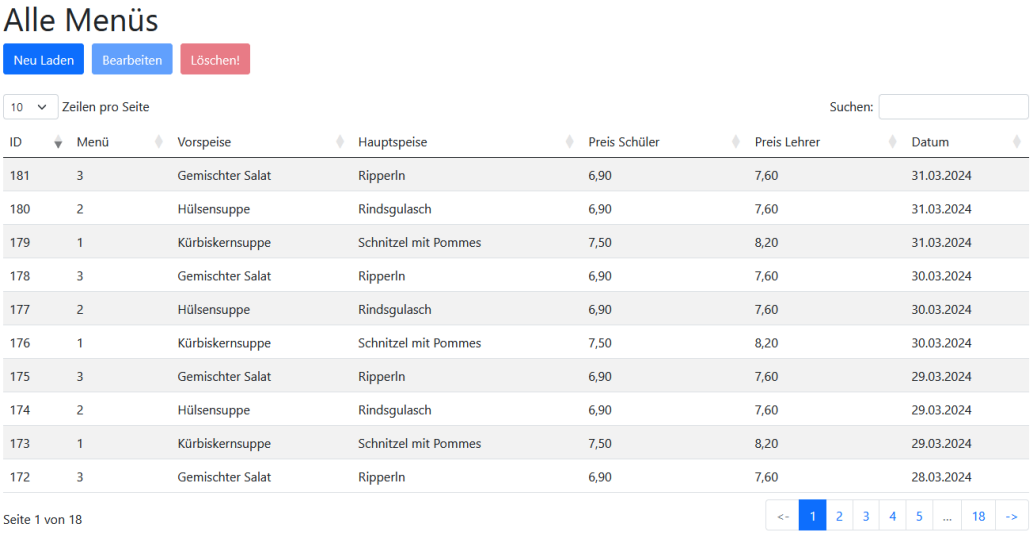

Abbildung 4.5: So sieht die fertige View aus, die alle Menüs aus der Datenbank anzeigt.

4.3.2 Menü bearbeiten

Wird der Bearbeiten-Button getätigt, dann wird die Action, die das Bearbeitungsformular anzeigt, aufgerufen. Diese Action bekommt die MenuId des ausgewählten Menüs mitgesendet und holt sich das passende Menü zu dieser MenuId aus der Datenbank. Dieses Menü wird dann wieder in ein MenuDTO umgewandelt und wird mit der View als Parameter zurückgegeben, sodass das Bearbeitungsformular mit dem zu bearbeitenden Menü beim Client zu sehen ist.

Die View sieht im Prinzip ähnlich aus wie beim Speichern eines Menüs; es wird dort wieder die ***_SaveMenusFormPartialView.cshtml*** aufgerufen wie im Kapitel 4.2.2 View. Für den Preis muss unten im ***@section Scripts{}***-Bereich wieder ein AJAX-Call wie im Kapitel 4.2.3 Handhabung mit dem Preis durchgeführt werden.

Wird nun wieder der Speichern-Button betätigt, dann wird die Post-Variante der ***EditMenu()***-Action aufgerufen. In dieser Action wird zuerst wieder eine serverseitige Validation durchgeführt, wie beim Speichern eines Menüs. Im Nachhinein wird das übergebene Menü mit dem zu bearbeitenden Menü ausgetauscht und danach mit ***_context.SaveChangesAsync()*** in der Datenbank gespeichert (siehe Codeabschnitt 4.13). War das Speichern erfolgreich, so wird im ***TempData*** wieder eine Erfolgs-/Misserfolgsnachricht gespeichert und an die View zurückgegeben, um diese Nachricht beim Client erscheinen zu lassen.

```
1 var menuToEdit = _context.Menues.FirstOrDefault(m => m.MenuId == id);
2 if (menuToEdit != null) {
3     menuToEdit.MenuId = id;
4     menuToEdit.Prices = await
          _context.Prices.FindAsync(menuDTO.WhichMenu);
5     menuToEdit.Starter = menuDTO.Starter;
6     menuToEdit.MainCourse = menuDTO.MainCourse;
7     menuToEdit.Date = menuDTO.Date;
8 }
9 isSuccess = (await _context.SaveChangesAsync()) == 1;
```

Code 4.13: Hier wird das Menü mit der übergebenen Id in der Datenbank gesucht, danach durch die eingegebenen Daten ersetzt und zum Schluss in der Datenbank gespeichert.

4.3.3 Menü löschen

Für das Löschen eines Menüs gibt es in der View einen ***<form>***-Teil, in dem der Helper Tag ***asp-controller*** und ***asp-action*** vorhanden sind. In diesem ***<form>***-Tag ist einerseits ein ***<input>***-Tag vorhanden, wobei der Type ***hidden*** ist, damit es beim Client nicht sichtbar ist, und andererseits ist der Löschen-Button vorhanden (siehe Codeabschnitt 4.14). In diesem ***<input>***-Tag wird beim Betätigen des Löschen-Buttons vorübergehend die Id des Menüs mit Javascript aus der Tabelle gespeichert und im Nachhinein die Action ***DeleteMenuFromDatabase()*** aufgerufen.

```
<form asp-controller="menu" asp-action="DeleteMenuFromDatabase"
    method="post">
    <input id="IdToDelete" name="Id" type="hidden"/>
    <button id="deleteMenu" class="btn btn-danger btn-md"
        disabled>Löschen!</button>
</form>
```

Code 4.14: Hier ist der <form>-Teil mit dem versteckten <input>-Feld und dem Löschen-Button zu sehen.

Wird nun die ***DeleteMenuFromDatabase()***-Action aufgerufen, wird mit der mitgegebenen MenuId das Menü aus der Datenbank geholt, mit der ***Remove()***-Methode entfernt und mit ***SaveChangesAsync()*** aus der Datenbank gelöscht. Im Nachhinein werden wieder alle Menüs aus der Datenbank geholt, in ein MenuDTO gespeichert und dann an die ***ShowAllMenus()***-View mit dem MenuDTO als Parameter übergeben, damit das gelöschte Element nicht mehr angezeigt wird. Ebenfalls wird in einem TempData wieder eine Erfolgs-/Misserfolgnachricht gespeichert, die dem Client angezeigt wird.

```
var menuToDelete = await _context.Menues.FindAsync(Id);
_context.Menues.Remove(menuToDelete);
isSuccess = (await _context.SaveChangesAsync()) == 1;
menus = await _context.Menues.Include("Prices").OrderByDescending(m
    => m.MenuId).ToListAsync();
foreach (Menu menu in menus)
{
    MenuDTO menuDTO = new();
    menuDTO.MenuId = menu.MenuId;
    menuDTO.WhichMenu = menu.Prices.PriceId;
    menuDTO.Starter = menu.Starter;
    menuDTO.MainCourse = menu.MainCourse;
    menuDTO.Date = menu.Date;
    menuDTO.PriceStudent = menu.Prices.PriceStudent;
    menuDTO.PriceTeacher = menu.Prices.PriceTeacher;

    menuDTOList.Add(menuDTO);
}
```

Code 4.15: Hier wird das Menü mit der übergebenen Id aus der Datenbank gelöscht und im Nachhinein werden alle Menüs aus der Datenbank geholt und in ein MenuDTO gespeichert.

4.4 Bestellungen anzeigen

Damit die Mensa nachsehen kann, wie viele Bestellungen es zu einem Menü gibt, wurde eine weitere Funktion entwickelt. Bei dieser Funktion kann ein Datum ausgewählt werden und für dieses ausgewählte Datum werden alle drei Menüs mit der Anzahl der Bestellungen angezeigt. Damit, falls eine neue Bestellung hinzugefügt wird, auch die richtige Anzahl an Bestellungen angezeigt wird, wird bei dieser Funktion der Website alle zehn Sekunden ein AJAX-Aufruf getätigt, der die Daten aus der Datenbank holt und wieder dem Client anzeigt.

4.4.1 Anzeigen der Menüs mit der Anzahl der Bestellungen

Zuerst wurde wieder ein neuer Button in der ***Layout.cshtml*** mit dem Namen Bestellungen, der beim Betätigen die Action ***ShowAllOrders()*** ausführt, hinzugefügt.

Die ***ShowAllOrders()***-Methode holt sich aus der Datenbank alle drei Menüs mit dem aktuellen Datum und sortiert diese Menüs nach der Art des Menüs, damit das ***Menü 1*** an der ersten Stelle erscheint (siehe Codeabschnitt 4.16). Diese drei Menüs werden in einer Liste von ***Menu*** gespeichert. Daraufhin wird die View mit der Liste als Parameter an den Client weitergeleitet.

```
1 menus = await
    _context.Menues.Include("MenuPersons").Include("Prices").Where(m
    => m.Date == DateOnly.FromDateTime(DateTime.Now)).OrderBy(m =>
    m.Prices.PriceId).ToListAsync();
```

Code 4.16: Die drei Menüs mit dem aktuellen Datum werden aus der Datenbank geholt und nach der Art des Menüs sortiert.

Die dazugehörige View ***ShowAllOrders.cshtml*** erwartet als ***@Model*** eine Liste von ***Menu***. Diese Liste wird überprüft, ob darin überhaupt etwas vorhanden ist. Ist etwas in der Liste vorhanden, dann wird die PartialView ***_ShowOrdersMenus.cshtml*** aufgerufen, worin die Liste durchiteriert wird und für jedes Menü, mittels Bootstrap, eine ***Card*** erstellt wird (siehe Codeabschnitt 4.17 Zeile 1). In den Zeilen 5-7 wird ein ***card-header*** erstellt. In diesem Bereich wird die Art des Menüs ausgegeben. Im Nachhinein wird ein ***card-body*** erstellt, in dem einerseits überprüft wird, ob eine Bestellung vorhanden ist. Falls keine Bestellung vorhanden ist, wird 0 ausgegeben, ansonsten die Anzahl an vorhandenen Bestellungen. Andererseits werden die einzelnen Felder des Menüs nochmals ausgegeben (vgl. Jacob Thornton (n.d.)).

```
@foreach (Menu m in Model)
{
<div id="orders" class="col-md-4">
    <div class="card mb-4 box-shadow">
        <div class="card-header">
            <h4 class="my-0 font-weight-normal">Menü
                @m.Prices.PriceId</h4>
        </div>
        <div class="card-body">
                <h1 class="card-title">Bestellungen: @(m.MenuPersons
                    == null ? "0" : m.MenuPersons.Count)</h1>
                <table>
                    <tr>
                        <td><strong>Vorspeise: </strong></td>
                        <td style="padding-left: 50px;">@m.Starter</td>
                    </tr>
                    <tr>
                        <td><strong>Hauptspeise: </strong></td>
                        <td style="padding-left:
                            50px;">@m.MainCourse</td>
                    </tr>
                    <tr>
                        <td><strong>Preis Schüler: </strong></td>
                        <td style="padding-left:
                            50px;">@m.Prices.PriceStudent</td>
                    </tr>
                    <tr>
                        <td><strong>Preis Lehrer: </strong></td>
                        <td style="padding-left:
                            50px;">@m.Prices.PriceTeacher</td>
                    </tr>
                </table>
        </div>
    </div>
</div>
}
```

Code 4.17: Hier wird eine Liste von Menu durchiteriert und mit Bootstrap eine Card erstellt. In dieser Card wird einerseits die Anzahl der vorhandenen Bestellungen angezeigt und andererseits das Menü ausgegeben.

Damit nicht nur die Bestellungen von heute angesehen werden können, wurde noch ein Auswahlfeld hinzugefügt, um das gewünschte Datum auszuwählen und die dazugehörige Anzahl der Bestellungen auszugeben (siehe Codeabschnitt 4.18). Dafür wurde zuerst ein ***<form>***-Tag verwendet. Darin wird wieder der Controller und die Action angegeben, zu welcher der POST-Request gesendet werden soll, wenn der Submit-Button gedrückt worden ist. Im Nachhinein wird ein ***<input>***-Tag verwendet, um das gewünschte Datum auszuwählen.

Wird nun der Submit-Button getätigt, so wird die ***ShowAllOrders()***-Action mit der ***HttpPOST***-Annotation aufgerufen. Diese Methode erwartet als Parameter ein Datum. Dabei handelt es sich um das gewünschte Datum. Mit diesem übergebenen Datum werden aus der Datenbank die dazugehörigen Menüs geholt, wieder nach der Art des Menüs sortiert und in eine Liste von ***Menu*** gespeichert. Zum Schluss wird wieder die View mit dieser Liste als Parameter an den Client weitergeleitet.

```
1  <form asp-controller="order" asp-action="ShowAllOrders" method="post">
2      <div class="row justify-content-center">
3          <label class="col-sm-1 col-form-label">Datum:</label>
4          <div class="col-sm-2">
5              <input id="date" name="Date" placeholder="Tag-Monat-Jahr"
                   class="form-control datepicker"
                   value="@Model[0].Date"/>
6          </div>
7          <div class="col-sm-9">
8              <button id="datePickerBtn" type="submit" class="btn
                   btn-success btn-sm">Bestätigen!</button>
9          </div>
10     </div>
11 </form>
```

Code 4.18: Hier wird das Auswahlfeld für das Datum erstellt.

4.4.2 Alle zehn Sekunden die Anzahl der Bestellungen updaten

Wenn eine neue Bestellung getätigt wird, wird diese nicht angezeigt, deshalb wird alle zehn Sekunden ein AJAX-Aufruf gemacht, um die neue Anzahl an Bestellungen aus der Datenbank zu holen und mit der alten Anzahl auszutauschen.

Zuerst wird im ***@section Scripts{}***-Teil wieder ein ***<script>***-Bereich erstellt, in dem der Javascript-Teil hineinkommt, um alle zehn Sekunden einen AJAX-Aufruf zu machen. In diesem Bereich wird eine Methode mit dem Namen ***reloadEveryTenSeconds()*** aufgerufen. In dieser Methode wird die globale Methode ***setTimeout()***

aufgerufen. Diese Methode führt einen Code-Teil aus, nachdem die eingestellte Zeit ausläuft. In den Zeilen 3 und 4 in Codeabschnitt 4.19 ist zu sehen, dass zwei Methoden nach zehn Sekunden ausgeführt werden. Einerseits führt sich die Methode selbst nochmals aus, damit die Seite nicht nur einmal nach zehn Sekunden neu geladen wird, sondern alle zehn Sekunden. Andererseits wird die Methode ***reloadPage()*** aufgerufen. In dieser Methode wird der AJAX-Aufruf veranlasst (siehe Codeabschnitt 4.20).

```
1 function reloadEveryTenSeconds() {
2     setTimeout(function () {
3         reloadPage();
4         reloadEveryTenSeconds();
5     }, 10000);
6 }
7 reloadEveryTenSeconds();
```

Code 4.19: In dieser Methode wird wird mit setTimeout() die reloadPage()-Methode alle zehn Sekunden aufgerufen.

In der Methode ***reloadPage()*** wird zuerst das ausgewählte Datum gespeichert, das später bei dem AJAX-Aufruf mitgegeben wird. Im Nachhinein wird die momentane Uhrzeit gespeichert und in einen ***<label>***-Tag hineingeschrieben, wann die Seite zuletzt neu geladen worden ist (siehe Zeile 5 in Codeabschnitt 4.20). Beim AJAX-Aufruf wird die Action ***_ShowOrdersMenus()*** aufgerufen; dabei wird das gewünschte Datum mitgegeben. In dieser Action werden zu dem mitgegebenen Datum die drei Menüs aus der Datenbank gesucht, nach der Art des Menüs sortiert und in einer Liste von ***Menu*** gespeichert. Danach wird die PartialView mit der Liste als Parameter zurückgegeben und in der View am Client ausgetauscht.

```
function reloadPage() {
    let date = $('#date').val();
    const d = new Date();
    let lastRefresh = d.toLocaleTimeString();
    document.getElementById("lastRefresh").innerHTML = "Zuletzt neu
        geladen um " + lastRefresh;
    $.ajax({
        type: "GET",
        url: "@Url.Action("_ShowOrdersMenus", "Order")",
        dataType: "html",
        data: { orderDate: date },
        success: function(data){
            $("#orders").html(data).fadeIn();
        }
    });
}
```

Code 4.20: In der reloadPage()-Methode wird das ausgewählte Datum geholt, die Zeit gespeichert, wann es zuletzt neu geladen wurde, und ein AJAX-Aufruf getätigt, um die Anzahl der Bestellungen zu updaten.

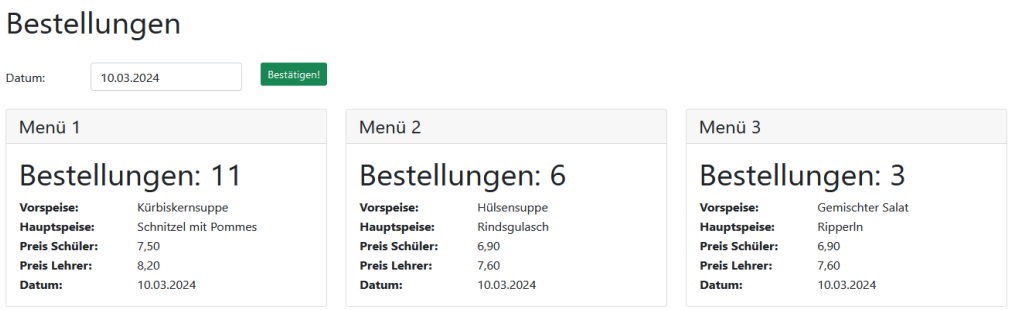

Abbildung 4.6: So sieht die fertige View für das Anzeigen der Bestellungen aus.

4.5 Statistiken anzeigen

Eine weitere Funktion dieser Web-Anwendung ist es, verschiedene Statistiken zu den in der Datenbank gespeicherten Daten zu erstellen und zu visualisieren. Dafür wurde die Javascript-Bibliothek ***Chart.js*** verwendet. Diese Bibliothek ermöglicht es, verschiedene Arten von Diagrammen zu erstellen. In dieser Diplomarbeit wurden einerseits Balkendiagramme (engl. ***Bar***-Chart) und andererseits ein Donut-Diagramm (engl. ***Doughnut***-Chart) angewendet. So wurden zum Beispiel alle bestellten Menüs und die nicht abgeholten Menüs in einem ***Bar***-Chart dargestellt (vgl. *Chart.js | Chart.js* (n.d.)).

Zuerst wurde wieder ein neuer Button in der ***Layout.cshtml*** mit dem Namen ***Statistiken*** hinzugefügt, der beim Betätigen die Action ***showStatistic()*** ausführt. Wenn der Button gedrückt wird, wird die ***showStatistic.cshtml*** aufgerufen. In dieser View sind zuerst nur ein ***Menü***-Button und ein ***Bestellungen***-Button; wenn diese gedrückt werden, wird jeweils eine PartialView angezeigt mit nochmals verschiedenen Buttons. Diese Buttons geben an, welches Diagramm aufgerufen werden soll. Des Weiteren gibt es noch die Auswahlmöglichkeit, ob die Statistik über das ganze Monat oder die ganze Woche verläuft.

4.5.1 Bestellungen der heutigen Menüs

In der ersten Statistik werden alle bestellten Menüs von heute in einem ***Doughnut***-Chart angezeigt. Dafür wird in der View im Javascript-Teil ein AJAX-Aufruf gemacht, wenn der Button ***Heute verkaufte Menüs*** gedrückt wird. Davor wird aber überprüft, ob schon ein Diagramm vorhanden ist. Ist dies der Fall, wird das schon vorhandene Diagramm gelöscht, damit das neue Diagramm gezeichnet werden kann (siehe Codeabschnitt 4.21).

```
1 if (activeChart) {
2     activeChart.destroy();
3 }
```

Code 4.21: In ***activeChart*** ist das momentane Diagramm gespeichert. Wenn ein Diagramm vorhanden ist, wird dieses mit der ***destroy()***-Methode gelöscht.

Im Nachhinein wird mittels AJAX im Controller die Methode ***getMostSoldMenus()*** aufgerufen. In dieser Methode werden zuerst alle Menüs, die am heutigen Tag existieren, aus der Datenbank geholt und in einer Liste gespeichert (siehe Codeabschnitt 4.22 Zeile 3). Von diesen Menüs wird danach die Anzahl an Bestellungen in einem Array gespeichert. Daraufhin wird dieses Array mit ***new JsonResult()*** zu einem JSON umgewandelt und zurück an den AJAX-Aufruf gesendet, damit mit diesen Daten in Javascript das gewünschte ***Doughnut***-Chart erstellt werden kann.

```
1 try
2 {
3     menus = await
          _context.Menues.Include("MenuPersons").Include("Prices")
          .Where(m => m.Date ==
          DateOnly.FromDateTime(DateTime.Now)).OrderBy(m =>
          m.Prices.PriceId).ToListAsync();
4 }
5 catch (Exception ex)
6 {
```

```
7       Console.WriteLine(ex.ToString());
8   }
9   var countMenus = new int[3]
10  {
11      menus[0].MenuPersons.Count,
12      menus[1].MenuPersons.Count,
13      menus[2].MenuPersons.Count
14  };
15  return new JsonResult(Ok(countMenus));
```

Code 4.22: Hier ist die Methode ***getMostSoldMenus()*** zu sehen. In dieser Methode werden alle heutigen Menüs aus der Datenbank geholt und danach von jedem einzelnen Menü die Anzahl an Bestellungen in ein Array gespeichert, mit ***new JsonResult()*** zu einem JSON umgewandelt und an den AJAX-Aufruf zurückgesendet.

Wird im Controller das Array richtig erstellt und an den AJAX-Aufruf zurückgesendet, so wird das ***success***-Feld ausgeführt. In diesem Feld wird dann mit ***new Chart()*** ein neues Diagramm erstellt und in ***activeChart*** gespeichert. Als erster Parameter muss das Element übergeben werden, in welches das Diagramm gezeichnet werden soll. Dafür wurde zuerst ein ***<canvas>***-Element erzeugt und dieses wird mit Javascript in einer globalen Variable ***ctx*** gespeichert. Als zweiten Parameter werden verschiedene Einstellungen übergeben. So kann zum Beispiel mit ***type*** die Art des Charts festgelegt werden oder in ***data*** werden die dafür benötigten Daten übergeben. Zusätzlich können auch visuelle Einstellungen festgelegt werden, wie zum Beispiel die Größe des Charts oder die Farben. Dies kann in ***options*** eingestellt werden (siehe Codeabschnitt 4.23).

```
1  success: function (soldMenus) {
2      activeChart = new Chart(ctx, {
3          type: 'doughnut',
4          data: {
5              labels: ['Menü 1', 'Menü 2', 'Menü 3'],
6              datasets: [{
7                  data: soldMenus.value
8              }]
9          },
10         options: {
11             aspectRatio: 3
12         }
13     });
14 }
```

Code 4.23: In dem ***success***-Feld von dem AJAX-Aufruf wird mit ***Chart.js*** ein neues Diagramm mit dem ***type: 'doughnut'*** erstellt und das übergebene Array aus der ***getMostSoldMenus()***-Methode wird als ***data*** verwendet.

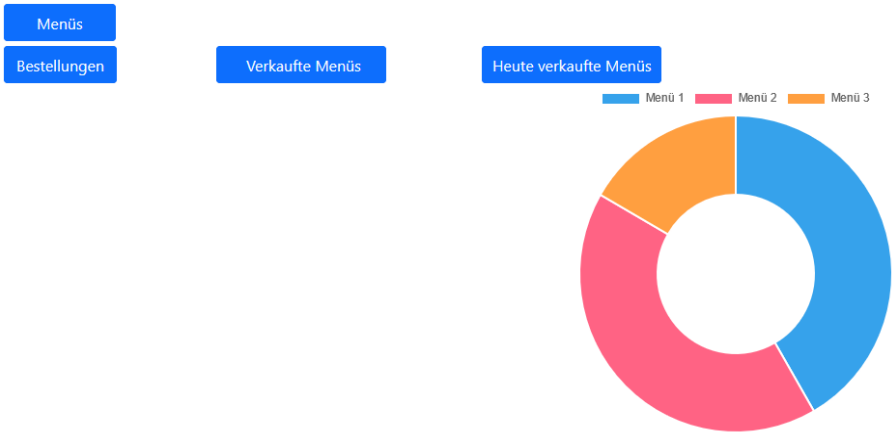

Abbildung 4.7: So sieht das fertige ***Doughnut***-Chart zum Schluss aus.

4.5.2 Anzahl an bestellten Menüs

Im Prinzip funktioniert dies gleich wie im vorhergehenden Kapitel Bestellungen der heutigen Menüs 4.5.1, nur mit dem Unterschied, dass es die Möglichkeit gibt auszuwählen, ob die Anzahl der bestellten Menüs für das aktuelle Monat oder nur für die aktuelle Woche in einem ***Bar***-Chart angezeigt werden soll. Deshalb wird bei dem AJAX-Aufruf noch ein String mitgegeben, ob es sich um das monatliche Chart oder um das wöchentliche Chart handelt. Im Controller wird dieser String überprüft und die richtigen Daten werden aus der Datenbank geholt (siehe Codeabschnitt 4.24).

```
1 if (monthlyOrWeekly == "monthly")
2 {
3     menus = await
          _context.Menues.Include("Prices").Include("MenuPersons")
          .Where(m => m.Date.Month ==
          DateOnly.FromDateTime(DateTime.Now).Month).ToListAsync();
4 }
5 else if (monthlyOrWeekly == "weekly")
6 {
7     menus = await
          _context.Menues.Include("Prices").Include("MenuPersons")
          .Where(m => getWholeWeek().Contains(m.Date)).ToListAsync();
8 }
```

Code 4.24: Hier wird überprüft, ob die Daten für das ganze Monat oder für die ganze Woche aus der Datenbank geholt werden sollen.

Die aus der Datenbank geholten Daten werden daraufhin nach der Art des Menüs aufgeteilt und jeweils in einer Liste gespeichert. Diese Listen werden im Nachhinein jeweils durchiteriert und in einer Liste von ***forChartDTO***, welches nur je ein Feld für die x- und y-Achse enthält, gespeichert. Das Datum des Menüs wird immer in das Feld für die x-Achse und die Anzahl der Bestellungen wird in das Feld für die y-Achse gespeichert (siehe Codeabschnitt 4.25).

```
1 foreach (Menu m in menu1)
2 {
3     forChartDTO d = new();
4     d.x = m.Date.ToString("dd.MM.yyyy");
5     d.y = m.MenuPersons.Count;
6     dataForMenu1.Add(d);
7 }
```

Code 4.25: Hier wird eine Liste von Menu durchiteriert und in ein ***forCharDTO*** gespeichert. Das Datum wird in das Feld für die x-Achse und die Anzahl an Bestellungen in das Feld für die y-Achse gespeichert.

Wenn die ***foreach()***-Schleife durch alle drei Menüs fertig iteriert hat, werden diese neuen Listen von ***forChartDTO*** in ein JSON umgewandelt und dem AJAX-Aufruf zurück gesendet.

Das ***Bar***-Chart wird gleich erstellt, wie im Kapitel **Bestellungen der heutigen Menüs**, nur wurde der ***type*** von ***doughnut*** zu ***bar*** geändert. Des Weiteren werden bei ***labels*** in ***data*** die ganzen Daten von diesem Monat übergeben. Bei ***datasets*** wird die Anzahl der Bestellungen für jedes einzelne Menü mit übergeben. Zum Schluss können wieder bestimmte Einstellungen festgelegt werden, wie zum Beispiel die Größe des Charts oder wie viele Schritte zwischen den Werten auf der y-Achse erfolgen (siehe Codeabschnitt 4.26).

```
1  activeChart = new Chart(ctx, {
2      type: 'bar',
3      data: {
4          labels: x,
5          datasets: [{
6              label: 'Menü 1',
7              data: Menu1_y
8          },
9          {
10             label: 'Menü 2',
11             data: Menu2_y
12         },
13         {
14             label: 'Menü 3',
15             data: Menu3_y
16         }]
```

```
17      },
18      options: {
19          aspectRatio: 4,
20          scales: {
21              y: {
22                  ticks: {
23                      stepSize: 1,
24                  }
25              }
26          }
27      }
28  });
```

Code 4.26: Es wird mit den mitgegebenen Daten aus dem Controller ein neues ***Bar***-Chart erstellt.

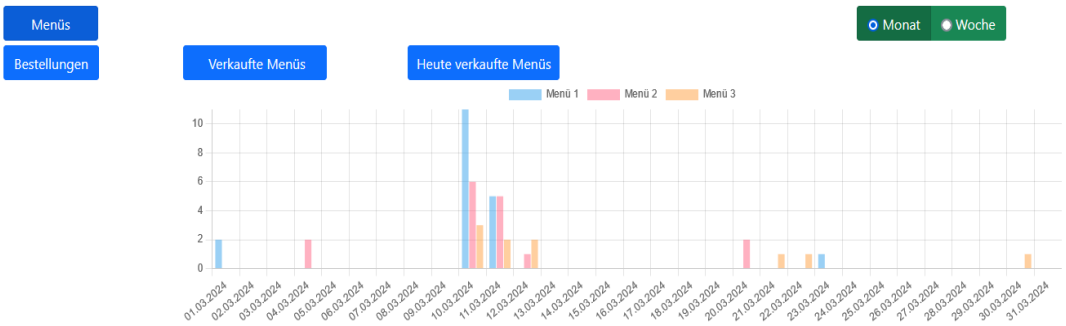

Abbildung 4.8: So sieht das fertige ***Bar***-Chart aus, für die Anzahl der Bestellungen über das ganze Monat hinweg. Oben rechts könnte man noch auswählen, ob man das ***Bar***-Chart nur über die ganze Woche hinweg anzeigen möchte.

4.5.3 Nicht abgeholte Menüs

Bei dieser Funktion der Web-Anwendung wird die Anzahl an Bestellungen damit verglichen, wie viele von diesen Bestellungen noch nicht abgeholt worden sind. Dies funktioniert im Grunde gleich wie in Kapitel Anzahl der bestellten Menüs 4.5.2, nur mit dem Unterschied, dass bei dem AJAX-Aufruf zusätzlich noch die Art des Menüs mitgegeben wird. Im Controller werden dann zu dieser Art des Menüs alle Bestellungen aus der Datenbank geholt und in einer Liste gespeichert. Diese Liste wird durchiteriert und dabei jede Bestellung darauf überprüft, ob sie schon abgeholt worden ist. Das Ergebnis davon und generell die Anzahl an Bestellungen werden im Nachhinein in ein JSON umgewandelt und an den AJAX-Aufruf zurückgesendet. Im ***success***-Feld des AJAX-Aufrufes wird dann das ***Bar***-Chart erstellt und gezeichnet.

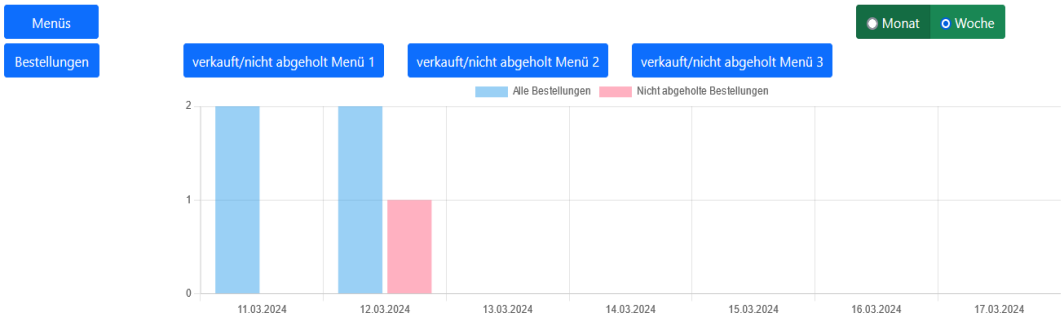

Abbildung 4.9: So sieht das fertige ***Bar***-Chart aus, für den Vergleich von vorhandenen Bestellungen mit den nicht abgeholten Bestellungen über die ganze Woche hinweg. Oben rechts könnte man noch auswählen, ob man das ***Bar***-Chart über das ganze Monat hinweg anzeigen möchte.

5 Möglichkeiten zur Erweiterung des Projekts

In Bezug auf diese Diplomarbeit konnten nicht alle vorab festgelegten Ziele erreicht werden. Dies ist einerseits auf die begrenzte Zeitvorgabe zurückzuführen und andererseits auf die Herausforderungen, die während des Prozesses auftraten. Auch wenn nicht alle Ziele erreicht wurden, sind uns trotzdem weitere Funktionen eingefallen, die der Diplomarbeit von Nutzen sein könnten.

Geplant war es, im Falle einer erfolgreich bezahlten Bestellung in der App, einen QR-Code mit den Bestellinformationen einerseits in der App einsehen und andererseits mittels einer E-Mail-Nachricht an die zum Account verknüpfte E-Mail-Adresse senden zu können. Dieser QR-Code könnte dann von dem Mensapersonal mittels eines QR-Code-Scanners eingescannt werden und von dem Mensapersonal als abgeholt gekennzeichnet werden. Diese Funktion wurde aus zeitlichen Gründen nicht programmiert.

Zusätzlich geplant war auch, die Bezahlung mittels PayPal in der WarenkorbView zu tätigen. Aufgrund des Zeitmangels konnte die Implementierung von Paypal nicht fertig durchgeführt werden; deshalb wurde PayPal nur in einem externen Programm getestet.

Die Mensa muss für jedes Gericht Allergene angeben. Dies könnte beim Speichern des Menüs eingebaut werden, indem die Möglichkeit existiert, die Allergene, die für das zu speichernde Menü notwendig sind, auszuwählen. Diese Allergene würden im Nachhinein in der Datenbank gespeichert werden und in der App für jedes Menü zu sehen sein. In der App wurde die Visualisierung der Allergene schon eingebaut, jedoch ist diese nur statisch einprogrammiert. Die Allergene müssten noch in der Website beim Hinzufügen eines neuen Menüs eingebaut werden.

Zum Schluss wären noch einige Verbesserungen der Benutzerfreundlichkeit möglich gewesen. Darunter die Farb- und Textwahl für angenehmeres Lesen,

sanftere und kontrolliertere Wischfunktionen für das Bedienen der WeeklyMenu-View und schließlich noch Ladebalken, die das Warten auf eine HTTP-Response signalisieren.

6 Arbeitsnachweis Diplomarbeit

Tabelle 6.1: Arbeitsnachweis: Felix Haider

Datum	Stunden	Beschreibung
09.07.2023	5	Dokumentationen zu ASP.NET Core Web App MVC reingelesen
10.07.2023	3	Dokumentationen zu ASP.NET Core Web App MVC reingelesen
11.07.2023	4	Dokumentationen zu ASP.NET Core Web App MVC reingelesen
13.07.2023	2	Benötigten Programme (VS, MySqlWorkbench) heruntergeladen
14.07.2023	3	Datenbank erstellt und versucht Daten auf ersten View auszugeben
17.07.2023	4	View SafeMenues angefangen
18.07.2023	5	SafeMenu Eingabe Felder hinzugefügt
19.07.2023	3	Speichern Button mit Safe Methode in WebApi
20.07.2023	5	Mit Partial Views herumprobiert
21.07.2023	4	Weiter mit Partial Views herumprobiert
24.07.2023	4	SafeMenu ausgebessert
25.07.2023	3	SafeMenu SuccessMessage hinzugefügt
26.07.2023	4	View ShowAllOrders angefangen
27.07.2023	5	DatePicker hinzugefügt
28.07.2023	4	Für das ausgewählte Datum die Menüs anzeigen lassen
31.07.2023	3	ShowAllOrders die Formatierung der Menüs ausgebessert
01.08.2023	4	View ShowAllMenues angefangen
02.08.2023	4	DataTable eingelesen und angewendet
03.08.2023	3	bei DataTable select hinzugefügt
04.08.2023	4	DataTable erweitert mit deutscher Beschriftung
13.08.2023	3	Delete Button hinzugefügt bei DataTable

Datum	Stunden	Beschreibung
14.08.2023	2	Den DeleteButton disablen/enablen
19.08.2023	1	DeleteButton ausgebessert
20.08.2023	2	Delete Methode bei WebApi hinzugefügt
21.08.2023	1	View ShowAllMenues verfeinert
09.09.2023	2	Show Orders verbessert und Startseite geändert
11.09.2023	2	ShowAllMenus Select Row ausgebessert
14.09.2023	1	Fehler bei Datenbank auf dem Laptop gefixxt
16.09.2023	2	Bei GetMenuByDate nach WhichMenu sortiert
17.09.2023	2	Ganze Projekt gemerged
19.09.2023	1	Edit button hinzugefügt
20.09.2023	2	ViewEditMenu hinzugefügt
22.09.2023	2	Andere Methode für EditMenu versuchen, statt ViewEditMenu
26.09.2023	2	EditMenu; die Daten werden in SafeMenu angezeigt; Price Nachkommastellen überarbeitet
02.10.2023	2	Menü bearbeiten in WebApi
30.10.2023	3	WebAPI in Website implementieren; Controller und DBContext in Website kopiert
31.10.2023	3	Menü speichern, alle Menü anzeigen, Menü löschen und Menü bearbeiten geändert für DB
01.11.2023	2	Validation für Menü speichern und bearbeiten;
15.11.2023	1	Die View der Bestellungen wird alle 10 Sekunden refresht
16.11.2023	2	Versucht die Bestellungen richtig anzeigen zu lassen
20.11.2023	1	Ajax-Aufruf umschreiben, damit View refresht werden kann
03.12.2023	4	Für Preis eigene Tabelle erstellt und versucht Menüs zu speichern und Bearbeiten
04.12.2023	3	SafeMenu: Wenn WhichMenu getauscht wird, wird der Preis geändert
08.12.2023	3	Eigene Preis Tabelle fertig gestellt und alles fertig geändert
16.12.2023	2	Preis kann man nun bearbeiten und die Orders werden jetzt alle 10 Sekunden refreshed
17.12.2023	2	WebAPI kann man nun auch über HTTPS nutzen
23.12.2023	1	Beim Löschen eines Menüs funktioniert die SuccessMessage jetzt

Datum	Stunden	Beschreibung
02.01.2024	3	Alles zusammen gemerged; Controller für API geändert;
03.01.2024	2	Menü löschen geändert; Chart.js eingelesen und probiert
04.01.2024	3	Erstes Pie Chart mit verkauften Menüs erstellt
14.01.2024	2	Liniendiagramm für Menü 1 anzahl der Bestellungen und Diplomschrift erstellt
07.02.2024	4	Layout der Chart View überarbeitet
14.02.2024	4	Bar-Chart für verkaufte Menüs
15.02.2024	3	LinienChart zum Vergleich von abgeholten und bezahlten Menüs
20.02.2024	2	Monthly Weekly bei Statistic
21.02.2024	2	Erster Entwurf von Statistic fertig
22.02.2024	1	Diplomschrift
24.02.2024	5	Diplomschrift
25.02.2024	4	Diplomschrift
02.03.2024	3	Diplomschrift
03.03.2024	4	Diplomschrift
09.03.2024	3	Diplomschrift
10.03.2024	3	Diplomschrift
13.03.2024	2	Endstand des Programmes auf Github gemerged
Summe	180	Diplomschrift fertig, ausgedruckt, projekt auf USB-Stick geladen

Tabelle 6.2: Arbeitsnachweis: Patrick Klaric

Datum	Stunden	Beschreibung
02.06.2023	1	Abstract schreiben
07.08.2023	2	Vorbereitung: MySQL & VisualStudio download, Github, Projekt erstellen. Remote Desktop-Verbindung zum Schulserver Testen
09.08.2023	3	ShowDailyMenuView und ViewModel
10.08.2023	4	ShowWeeklyMenuView und ViewModel
11.08.2023	1	MenuController: DailyMenu
14.08.2023	3	MenuController: Aus DateTime.Now die Wochennummer bekommen

Datum	Stunden	Beschreibung
15.08.2023	2	MenuController: Aus der Wochennumer jeden Tag der Woche bekommen und in eine Liste speichern
16.08.2023	2	MenuController: Über die Liste die Menüs aller Tage bekommen
17.08.2023	1	MenuController abgeschlossen
23.08.2023	2	Wochenmenüliste nach „Whichmenu“ und „Date“ zu sortieren
13.09.2023	3	Von WeeklyMenuView Datum bekommen
15.09.2023	2	Aus dem Datum die richtige DailyMenuView aufbauen
17.09.2023	3	Erster Projekt Merge
13.10.2023	4	Einbau der CarusellView
14.10.2023	1	Problem: Handy (Kabel) & Handy (Simuliert) erhalten keine Daten in View über {Databinding Menus}
15.10.2023	1	Problemsuche
16.10.2023	1	Lösung: Handy hat keine Verbindung zur DB ->Statische Einträge verwenden bis Lösung gefunden (z.B. externen Server)
20.10.2023	3	Design der View;
28.10.2024	1	CarusellView Komplett eingebaut und anzeige der Daten möglich
29.10.2024	1	Problem: Nichtgewünschte Ausrichtung (Für jeden Eintrag wird neue Seite in der CarousellView erstellt: Montag: 3x Menü 1; Dienstag: 3x Menü 2; Mittwoch: 3x Menü 3)
30.10.2023	1	Lösungsuche
31.10.2023	2	Lösung durch Einbau von neuer Klasse dayMenu. Jeder Tag ist eine Liste von 3 Einträgen
02.11.2023	4	Menu kann von Carusellview an Warenkorb gesendet werden mit Alert zur auswahl(Ja, Nein)
03.11.2023	4	Beim Warenkorb können Menüs gelöscht werden mit Alert zur auswahl(Ja, Nein)
06.11.2023	2	Orders erstellen und Menüs speichern, alles vorbereiten für umstieg auf Raspberrypi-Server
17.11.2023	8	Orders geht komplett

Datum	Stunden	Beschreibung
24.11.2023	2	Einlesen in PayPal, Bildschirmdrehung bei Handydrehung ausgeschalten
22.12.2023	1	Menu-Person(=Order) Klassen erstellen und Migration alte Models löschen (Order, Warenkorb)
28.12.2023	10	Controller Person erstellen, Controller MenuPerson(Order) erstellen, MenuController umschreiben
30.12.2023	2	Views umschreiben + Debugging
31.12.2023	3	Weekly Menus und Warenkorb funktionieren
01.01.2024	3	OrderHistory funktioniert
03.01.2024	4	Lösen von Fehlern des letzten Sprints(Doppelte Items im Warenkorb erlauben, Items aus Warenkorb löschbar, neues Item im Warenkorb wird geladen)
04.01.2024	2	Anmeldungsview erstellt (ohne LDAP)
05.01.2024	3	Appshell bearbeitet damit erst bei korrekter Anmeldung die Views angezeigt werden
06.01.2024	3	User Icon rechts oben erstellt und nach erfolgreichem Login, Initialen neben User Icon schreiben
12.01.2024	2	LDAP-Testprogramm geschrieben, keine Verbindung möglich
14.01.2024	2	LDAP-Verbindung funktioniert auf C# Testprojekt
15.01.2024	3	LDAP-Verbindung funktioniert auf Mensa Projekt
18.01.2024	5	Letzte fixes und Anmeldung funktioniert
19.01.2024	3	Fehler ausgebessert: Programmabsturz bei falscher Eingabe und falsche Preise bei Lehrer Login
31.01.2024	4	In PayPal eingelesen erste Videos angeschaut und Developer/Sandbox Account gemacht
05.02.2024	2	Weitere Videos angeschaut und mit Postman die API-Calls getestet
06.02.2024	1	Versucht ein Testprogramm zu schreiben
07.02.2024	3	Mehrere verschiedene Testprogramme versucht zu schreiben alle ohne Erfolg

Datum	Stunden	Beschreibung
08.02.2024	2	Video: https://www.youtube.com /watch?v=qLXDsoYOopU&t=1731s angeschaut veruscht den Controller zu schreiben
11.02.2024	2	Controller geht nicht, deswegen einfach mit eine Website
14.02.2024	3	Angefangen den Anatz einzubauen
15.02.2024	2	Eine funktionierende PayPal Website mit asp.net Core gemacht
16.02.2024	6	Integration der Website in .net Maui Programm, dafür benötigt: Anzahl der Produkte und Gesamtwert des Waenkorbs
17.02.2024	1	Aufgrund von Zeitmangel integration von PayPal nicht fertig gemacht und lieber angefangen zu schreiben
18.02.2024	1	Erstellung der Diplomschrift in Latex
19.02.2024	1	Grundlegende Struktur der Diplomarbeit festgelegt
20.02.2024	3	Diplomschrift
21.02.2024	2	Diplomschrift
22.02.2024	2	Diplomschrift
23.02.2024	2	Diplomschrift
24.02.2024	5	Diplomschrift
25.02.2024	5	Diplomschrift
01.03.2024	3	Diplomschrift
02.03.2024	3	Diplomschrift
03.03.2024	3	Diplomschrift
09.03.2024	4	Diplomschrift
10.03.2024	5	Diplomschrift
11.03.2024	1	Diplomschrift
12.03.2024	1	Diplomschrift
13.03.2024	3	Merge des Projekts und abschließendes lesen der Diplomschrift
Summe	180	Diplomschrift fertig, ausgedruckt, projekt auf USB-Stick geladen

Appendix

Tabellenverzeichnis

Abbildungsverzeichnis

Listings

Literaturverzeichnis

Active Directory einfach erklärt | prosec-networks.com (n.d.).
URL: *https://www.prosec-networks.com/blog/active-directory-einfach-erklart/*

ajcvickers (n.d.), 'Überblick über entity framework core – EF core'.
URL: *https://learn.microsoft.com/de-de/ef/core/*

Chart.js | Chart.js (n.d.).
URL: *https://www.chartjs.org/docs/latest/*

DataTables | Javascript table library (n.d.).
URL: *https://datatables.net/*

davidbritch (n.d.*a*), 'CarouselView - .NET MAUI'.
URL: *https://learn.microsoft.com/de-de/dotnet/maui/user-interface/controls/carouselview/?view=net-maui-8.0*

davidbritch (n.d.*b*), 'Commanding - .NET MAUI'.
URL: *https://learn.microsoft.com/en-us/dotnet/maui/fundamentals/data-binding/commanding?view=net-maui-8.0*

davidbritch (n.d.*c*), 'Data binding - .NET MAUI'.
URL: *https://learn.microsoft.com/en-us/dotnet/maui/fundamentals/data-binding/?view=net-maui-8.0*

davidbritch (n.d.*d*), 'Grid - .NET MAUI'.
URL: *https://learn.microsoft.com/en-us/dotnet/maui/user-interface/layouts/grid?view=net-maui-8.0*

davidbritch (n.d.*e*), 'ScrollView - .NET MAUI'.
URL: *https://learn.microsoft.com/en-us/dotnet/maui/user-interface/controls/scrollview?view=net-maui-8.0*

davidbritch (n.d.*f*), 'StackLayout - .NET MAUI'.
URL: *https://learn.microsoft.com/en-us/dotnet/maui/user-interface/layouts/stacklayout?view=net-maui-8.0*

davidbritch (n.d.*g*), 'Was ist .NET MAUI? - .NET MAUI'.
URL: *https://learn.microsoft.com/de-de/dotnet/maui/what-is-maui?view=net-maui-8.0*

davidbritch (n.d.*h*), 'WebView - .NET MAUI'.
URL: *https://learn.microsoft.com/en-us/dotnet/maui/user-interface/controls/webview?view=net-maui-8.0*

Grundlagen - ASP.NET MVC - ASP.NET - Homepage-Webhilfe (n.d.).
URL: *https://www.homepage-webhilfe.de/ASP-NET/MVC/*

How to Integrate PayPal Payments in ASP.NET Web Applications with Razor Pages | The Complete Guide - YouTube (n.d.).
URL: *https://www.youtube.com/watch?v=qLXDsoYOopU*

IBM Documentation (n.d.).
URL: *https://www.ibm.com/docs/fi/api-connect/5.0.x?topic=definition-ldap-authentication*

Jacob Thornton, {and} Bootstrap, M. O. (n.d.), 'Cards'.
URL: *https://getbootstrap.com/docs/4.0/components/card/*

JavaScript SDK reference (n.d.).
URL: *https://developer.paypal.com/sdk/js/reference/*

Leismann, J. C. (n.d.), 'Entity framework in c#'.
URL: *https://csharp-hilfe.de/entity-framework-in-csharp/*

Mehta, P. (n.d.), 'What is ASP.NET core MVC? - the complete guide 2024'.
URL: *https://positiwise.com/blog/introduction-asp-net-core-mvc*

michaelstonis (n.d.), 'Model view ViewModel - .NET'.
URL: *https://learn.microsoft.com/de-de/dotnet/architecture/maui/mvvm*

Part 3: What is Sandbox? | Create PayPal Sandbox Account | PayPal Tutorial With .NET Application (n.d.).
URL: *https://www.youtube.com/watch?v=wwr2HWh14NI*

Startup & Code (n.d.), 'Partial view and data refresh #5'.
URL: *https://www.youtube.com/watch?v=zYqQOIES4ME*

tdykstra (n.d.), 'Übersicht über ASP.NET core'.
URL: *https://learn.microsoft.com/de-de/aspnet/core/introduction-to-aspnet-core?view=aspnetcore-8.0*

TempData in ASP.NET MVC (n.d.).
URL: *https://www.tutorialsteacher.com/mvc/tempdata-in-asp.net-mvc*

tutorialsEU - C# (n.d.), 'Use jQuery AJAX in ASP.NET CORE 6? you NEED to see how it´s done!'.
URL: *https://www.youtube.com/watch?v=dXutAlml$_x$E*

What Is LDAP & How Does It Work? | Okta (n.d.).
URL: *https://www.okta.com/identity-101/what-is-ldap/*