APP-SCHIENE - ABGABE SPRINT 2

INHALT

1 GENERELLE NEUERUNGEN

Beide Apps verfügen seit Sprint 2 über einen Login. (Da bei jeder aufgenommenen Bestellposition der jeweilige Kellner in der Datenbank miterfasst wird.)

Beide Apps sind mit folgenden Zugangsdaten erreichbar:

Benutzername	Passwort
wat	wat

2 ORDERMAN

Von der Funktionalität her hat sich in Sprint 2 nicht atemberaubend viel getan. Der Funktionalitätsumfang ist noch der gleiche wie im vorigen Sprint. (Deshalb keine Screenshots)

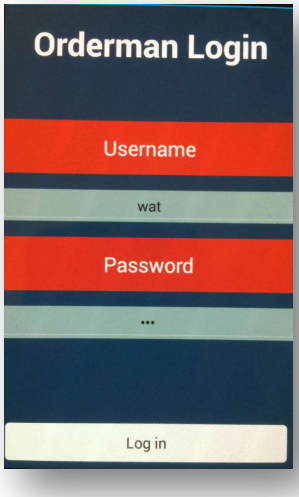
2.1 Funktionalitätsumfang

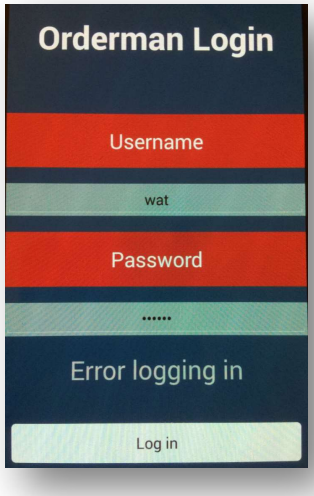
Wie gesagt, gleich wie in Sprint 1.

- Tischübersicht wird dargestellt
- Bestellungsübersicht
- Bestellungspositionen können hinzugefügt werden
- Bestellungspositionen können einer Rechnung zugewiesen werden (so können Bestellungen getrennt gezahlt werden)
- Bei Bezahlung können (werden) Coupons (Getränk bzw. Essen) berücksichtigt

2.2 Neuerungen (Sprint 2)

2.2.1 Login





Eine der kleineren Änderungen ist das Login mit dazugehöriger Error-Message bei fehlgeschlagener Authentifizierung.

Nach erfolgreicher Authentifizierung wird temporär ein Access-Token gespeichert, der für die Authentifizierung der Token-basierten Webservices dient.

2.2.2 MVVM – Pattern

2.2.2.1 Allgemein

MVVM -> Model – View - ViewModel

2.2.2.1.1 Model

Das Model stellt eine Abbildung der Daten dar, welche für die visuelle Anwendung benötigt werden.
Es enthält also Daten, die dem Benutzer zur Verfügung gestellt und von ihm manipuliert werden.

Beispiel Screenshots aus unserem Projekt:

```
namespace OrderM8_mvvm.Model
{
    public class OrderEntry
    {
        public bool cancelled { get; set; }
        public bool coupon { get; set; }
        public int fkBill { get; set; }
        public int fkProduct { get; set; }
        public int fkTable { get; set; }
        public int fkUser { get; set; }
        public int idOrderEntry { get; set; }
        public string note { get; set; }

        public OrderEntry(bool _cancelled, bool _coupon, int _fkBill, int _fkProduct,
                          int _fkTable, int _fkUser, int _idOrderEntry, string _note)
        {
            cancelled = _cancelled;
            coupon = _coupon;
            fkBill = _fkBill;
            fkProduct = _fkProduct;
            fkTable = _fkTable;
            fkUser = _fkUser;
            idOrderEntry = _idOrderEntry;
            note = _note;
        }
    }
}
```

```
namespace OrderM8_mvvm.Model
{
    public class OrderEntryProductWrapper
    {
        public OrderEntry orderEntry { get; set; }
        public Product product { get; set; }

        public OrderEntryProductWrapper(OrderEntry _orderEntry, Product _product)
        {
            orderEntry = _orderEntry;
            product = _product;
        }
    }
```

2.2.2.1.2 View

Stellt wie erwartet nur die Daten dar. Im Code-Behind der View wird so wenig Code als möglich geschrieben. Grundsätzlich wird mit DataBinding gearbeitet und so die Möglichkeit geschaffen, ohne größeren Aufwand die View auszutauschen.

2.2.2.1.2.1 Vergleich MVC vs. MVVM

Beispiel: Code-Behind von **TableDetailPay** View.
View ist in beiden Pattern fast ident (bis auf die gemachten Bindings im MVVM-Pattern)

```
namespace XamOrderM8_Orderman.Views
{
    #region TableDetailPay

    public partial class TableDetailPay : ContentPage
    {
        #region Fields / Properties

        public Table selectedTable;
        public ObservableCollection<OrderItem> observableColOrderItem;
        public Bill currentBill;

        #endregion

        #region Constructor

        public TableDetailPay(Table paramTable)
        {
            InitializeComponent();

            selectedTable = paramTable;

            currentBill = new Bill();
        }

        #endregion

        #region Methods

        private void InitData()
        {
            lblTableNumber.Text = selectedTable.Tischnummer + "";

            lblOrderCount.Text = currentBill.ListOrderItems.Count + "";

            observableColOrderItem = new ObservableCollection<OrderItem>(selectedTable.ListOrderItem);
            listOrders.ItemsSource = observableColOrderItem;
        }

        #endregion

        #region Eventhandler

        public void OnOrderAddToBill(object sender, EventArgs e)
        {
            OrderItem selectedOrderItem = ((Button)sender).BindingContext as OrderItem;

            observableColOrderItem.RemoveAt(observableColOrderItem.IndexOf(selectedOrderItem));
            selectedTable.RemoveOrderItem(selectedOrderItem);

            currentBill.ListOrderItems.Add(selectedOrderItem);

            lblOrderCount.Text = currentBill.ListOrderItems.Count + "";
        }

        public async void OnBillBtnClicked(object sender, EventArgs e)
        {
            await Navigation.PushAsync(new DetailBill(selectedTable, currentBill));
        }
```

Code Behind der View des alten MVC - Patterns

```
namespace OrderM8_mvvm
{
    public partial class TableDetailPay : ContentPage
    {
        TableDetailPayViewModel vm;
        public TableDetailPay(Table t)
        {
            InitializeComponent();
            BindingContext = App.Locator.TableDetailPay;
            vm = App.Locator.TableDetailPay;
            vm.Table = t;
        }

        protected override void OnAppearing()
        {
            base.OnAppearing();

            vm.InitializeData();
        }
    }
}
```

Im Vergleich dazu: Code-Behind im MVVM – Pattern, Logik wurde sozusagen in die ViewModels entkoppelt und macht es so leicht die View einfach auszutauschen.

2.2.2.1.3 ViewModel

Das ViewModel stellt das Model für die View dar. (Nicht mit Code-Behind zu verwechseln!) Durch die im ViewModel implementierten Change Notifications werden Änderungen direkt an die View weitergegeben. Funktionalitäten stehen per Commands zur Verfügung.

Beispiel: ViewModel der **TableDetailPay.**

```
namespace OrderM8_mvvm.ViewModel
{
    public class TableDetailPayViewModel : ViewModelBase
    {
        #region Properties
        public IDataAccessService DataAccessProxy { get; set; }
        public INavigationService NavigationProxy { get; set; }

        private Table _Table;
        public Table Table...

        private ObservableCollection<OrderEntryProductWrapper> _OrderEntries;
        public ObservableCollection<OrderEntryProductWrapper> OrderEntries...

        private ObservableCollection<OrderEntryProductWrapper> _BillEntries;
        public ObservableCollection<OrderEntryProductWrapper> BillEntries...

        public RelayCommand BillClickedCommand { get; set; }
        public RelayCommand<OrderEntryProductWrapper> OrderEntryAddedToBillCommand { get; set; }

        #endregion

        public TableDetailPayViewModel(IDataAccessService _ServiceProxy, INavigationService _NavigationProxy)
        {
            DataAccessProxy = _ServiceProxy;
            NavigationProxy = _NavigationProxy;

            BillClickedCommand = new RelayCommand(OnBillClicked);
            OrderEntryAddedToBillCommand = new RelayCommand<OrderEntryProductWrapper>(OnOrderEntryAddedToBill);
        }

        #region Methods

        public async void InitializeData()
        {
            List<OrderEntryProductWrapper> tmp = await DataAccessProxy.GetOrderEntryProductWrapperAsync(Table.IdTable);
            OrderEntries = new ObservableCollection<OrderEntryProductWrapper>(tmp);
            BillEntries = new ObservableCollection<OrderEntryProductWrapper>();
        }

        #endregion

        #region Command Handler
        private async void OnBillClicked()
        {
            Bill b = await DataAccessProxy.GetNewBillAsync();
            BillOrderEntriesWrapper bw = new BillOrderEntriesWrapper();
            bw.bill = b;
            bw.orderEntryProductWrappers = BillEntries;

            Tuple<Table, BillOrderEntriesWrapper> t = new Tuple<Table, BillOrderEntriesWrapper>(Table, bw);

            NavigationProxy.NavigateTo(ViewModelLocator.DetailBillPage, t);
        }

        private void OnOrderEntryAddedToBill(OrderEntryProductWrapper obj)
        {
            BillEntries.Add(obj);
            OrderEntries.Remove(obj);
        }
        #endregion
    }
}
```

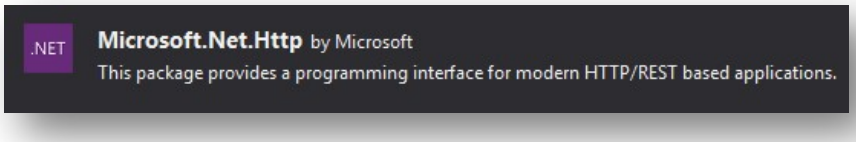
Wie man sieht gibt es Commands, welche Funktionalitäten anbieten und ObservableCollections welche Änderungen an den Auflistungen erkennen und dadurch an das Binding System weitergegeben werden. (ohne dafür zusätzlichen Code schreiben zu müssen)

2.2.3 Webservice – Layer

Ein weiteres Ziel des zweiten Sprints war es, die gesamte Applikation mit REST - Webservice Einbindung zu implementieren.

```
public class DataAccessService : IDataAccessService
{
    public DataAccessService()
    {
        Url = "http://192.168.193.235:8085/orderm8/api/";
        client = new HttpClient();
        client.MaxResponseContentBufferSize = 256000;
    }

    #region Fields / Properties

    public static HttpClient client;

    public TokenResponse TokenResponse { get; set; }
    public string Url { get; set; }

    public List<ProductType> ProductTypes { get; set; }
    public List<Product> Products { get; set; }
    public List<TableStatusWrapper> TableOrderWrappers { get; set; }

    #endregion

    #region Async Methods

    Auth

    Product / ProductTypes
```

Die REST-Schnittstelle bildete unser *DataAccessService* welcher mit den Standard .NET http-Client auf die einzelnen Services unserer API zugreift.



JSON Objekte wurden mit der Zusatzbibliothek *„Newtonsoft“* serialisiert und deserialisiert.

Beispiel eines unserer Calls:

GetProductTypesAsync() liefert alle verfügbaren Produkttypen.

```
//Get all productTypes
public async Task<List<ProductType>> GetProductTypesAsync()
{
    var request = new HttpRequestMessage()
    {
        RequestUri = new Uri(Url + "producttype"),
        Method = HttpMethod.Get
    };

    request.Headers.Add("x-access-token", TokenResponse.Token);
    var response = await client.SendAsync(request);

    if (response.IsSuccessStatusCode)
    {
        var content = await response.Content.ReadAsStringAsync();
        ProductTypes = JsonConvert.DeserializeObject<List<ProductType>>(content);
        return ProductTypes;
    }
    else
    {
        throw new Exception();
    }
}
```

Da es sehr aufwendig, aber vor allem unperformant wäre mit den ganzen FOREIGN KEYs clientseitig durchzuschleifen (wenn man z.B. den Namen eines Produktes haben will, aber im OrderEntry nur den FOREIGN KEY des Produktes hat) wurde hauptsächlich mit Wrapper-Klassen gearbeitet.

Beispiel einer Wrapper-Klasse:

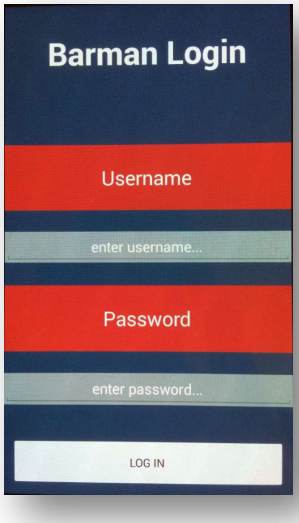
```
namespace OrderM8_mvvm.Model
{
    public class OrderEntryProductWrapper
    {
        public OrderEntry orderEntry { get; set; }
        public Product product { get; set; }

        public OrderEntryProductWrapper(OrderEntry _orderEntry, Product _product)
        {
            orderEntry = _orderEntry;
            product = _product;
        }
    }
}
```

3 BARMAN

Ziel der Barman – App ist es, der Theke schon nachdem der Kellner die Bestellung aufgenommen hat zu zeigen, was bestellt wurde und somit vorbereitet werden muss.

Die App wurde in Xamarin.Forms (C# -> Android) unter Verwendung des MVC-Pattern implementiert. Webservice-Anbindung mit Data-Polling ist auch schon dabei.

3.1 Login



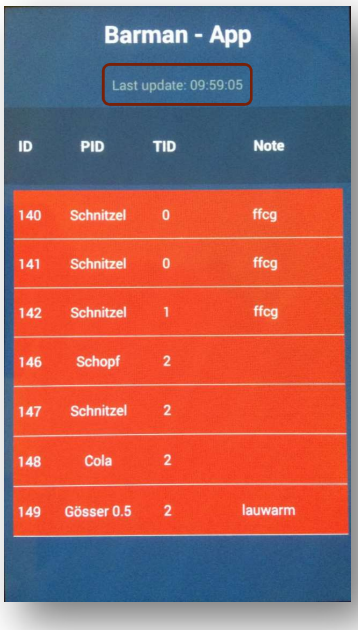
Bei falscher Anmeldung bzw. Errors die in Zusammenhang mit der Authentifikation am Webserver zusammenhängen wird eine Error-Message als Information ausgegeben. (siehe Orderman - App)

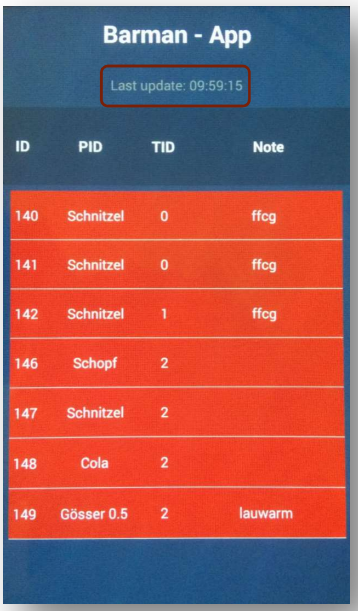
3.2 Hauptansicht

Die App gibt eine übersichtliche Darstellung über alle durch Kellner erfassten Bestellungen, und dient so als eine Art „TODO“ – Liste für das Team hinter der Theke.

Legende der angezeigten Table:

- ID: Identifikationsnummer der jeweiligen Bestellposition
- PID: Identifikationsnummer des jeweiligen Produktes
- TID: Identifikationsnummer des jeweiligen Tisches
- Note: Zusatzinformation zur angelegten Bestellposition (z.B. „ohne Ketchup“, „warm/kalt“, etc.)





Um zu sehen wie aktuell die Daten sind, gibt es zwischen den Table-Headers und dem Titel ein Label, welches den Timestamp der letzten Aktualisierung zeigt. Wie im Screenshot zu sehen, läuft das Datenabfragen (Polling) alle 10 Sekunden ab.