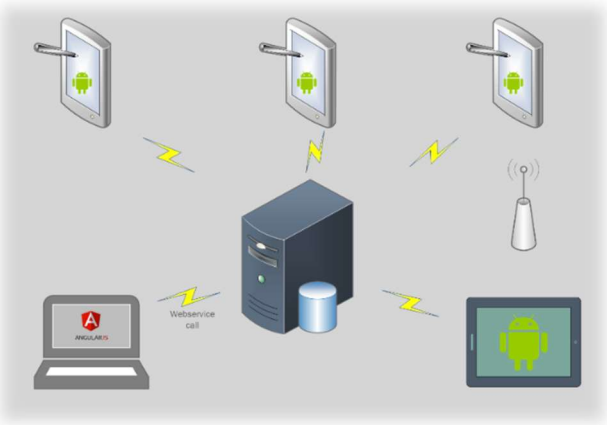
TECHNISCHES HANDBUCH ORDERM8 (APPS)

INHALT

EINLEITUNG

Dieses Dokument dient dazu, nochmals alle technischen, relevanten Aspekte der Implementierung näher zu erläutern.

Basis der beiden Applikationen bildet das plattformübergreifende Entwicklungs-Framework Xamarin. Dies erlaubte uns die Android Applikationen plattformübergreifend mittels C#-Codebasis zu entwickeln.



Die Apps kommunizierten via REST-Services mit dem Webserver. Dies erfolgt in einem lokalen WLAN-Netz, welches während dem Fest aktiviert ist.

Außerdem wurden bei der Entwicklung der beiden Applikationen zwei verschiedene Software-Patterns verwendet, auf die in den folgenden Kapiteln noch näher eingegangen wird.

SOFTWARE PATTERNS

Im Allgemeinen versteht man unter Software Patterns eine Art Muster bzw. Herangehensweise an die Entwicklung der Software-Architektur und der anschließenden Implementierung.

ORDERMAN (MVVM)

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;
        }
    }
```

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.

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.

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)

```
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
    }
}
```

BARMAN (MVC)

Die Architektur der Barman-App baut auf normalen MVC auf.

Dem Model entsprechen die normalen Datenklassen.

```
public class Product
{
    public int fkType { get; set; }
    public int idProduct { get; set; }
    public string name { get; set; }
    public double price { get; set; }
    public int quantity { get; set; }

    public Product(int fkType, int idProduct, string name, double price, int qty)
    {
        this.fkType = fkType;
        this.idProduct = idProduct;
        this.name = name;
        this.price = price;
        quantity = qty;
    }

    public override string ToString()
    {
        return $"[Product fkType={fkType}, idProduct={idProduct}, name={name}, price={price}]";
    }
}
```

Die CodeBehind Files (.cs) entsprechen den Controllern. Ein CodeBehind File gehört immer zu einer View (XAML-File).

```
public partial class MainPage : ContentPage
{
    #region Constructor

    public MainPage()
    {
        InitializeComponent();
        Polling();
    }

    #endregion

    Methods

    #region Methods

    private void SetTimestamp()
    {
        lblLastUpdate.Text = String.Format("Last update: {0:HH:mm:ss}", DateTime.Now);
    }

    private async void Polling()
    {
```

Der View entsprechen die XAML-Files. (XAML ist eine von Microsoft entwickelte Beschreibungssprachen für das Gestalten und Erstellen von Benutzeroberflächen → Findet auch Einsatz in WPF-Anwendungen)

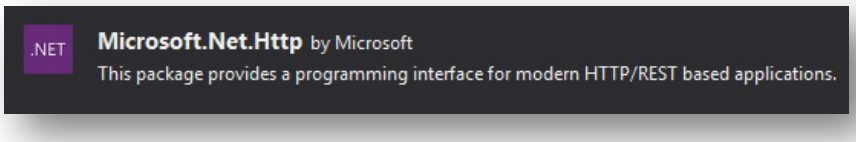
WEBSERVICE – LAYER

Innerhalb der Applikation wurden REST-Calls für folgende Sachen gebraucht:

- Produkttypen abrufen
- Tables / table status abfragen
- Bestellpositionen abfragen / adden
- Bestellpositionen bezahlen / Rechnungen erstellen

```
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();
    }
}
```

TESTING

UI-TESTING

Xamarin bietet die Möglichkeit UITest Projekte zu erstellen, in diesen können User-Inputs und Benutzerinteraktionen simuliert werden. Diese Simulation bzw. die Tests selbst können LIVE am Smartphone / Emulator mitverfolgt werden.

Beispiel Testfall für das Login:

```
[Test]
[Category("Login")]
public void LoginCorrect()
{
    app.EnterText("txtUsername", "wat");
    app.EnterText("txtPassword", "wat");

    app.DismissKeyboard();
    app.Tap("btnLogin");
}
```

Jedes UI-Element, dass angesprochen wird bekommt für das Testen extra eine sogenannte „AutomationId " über diese erfolgt dann der Befehlsaufruf („Tap " für Klick, „EnterText " für Texteingabe)

```
<Button AutomationId="btnLogin" Grid.Row="7" x:Nam
```

```
app.Tap("Bill");
app.WaitForElement("Pay", "Pay View konnte nicht geöffnet werden", TimeSpan.FromSeconds(5));

result = app.Query("Pay");
Assert.IsTrue(result.Any(), "Cannot open pay view");
```

Wie im Beispiel zu sehen wird hier auf den Button Bill geklickt und 5 Sekunden gewartet, bevor dann überprüft wird, ob sich die nächste View tatsächlich geöffnet hat. Auf diese Art kann man verschiedenste Szenarien durchspielen.