

# **SEMINARARBEIT**

im Studiengang Telekommunikation- und Internettechnologien  
Lehrveranstaltung Projektarbeit

## **Xamarin Framework und Microsoft Azure – Photo App**

Ausgeführt von: Christian Pipp

Begutachter: Dipl. Ing. Dr. Thomas Polzer

Wien, 01.02.2017



# Inhaltsverzeichnis

Einführung.....	3
1 Xamarin.Forms .....	3
1.1 Architektur von Xamarin.....	4
1.2 Aufbau eines Xamarin.Forms Projekts.....	5
1.3 User Interface.....	8
1.3.1 Pages .....	8
1.3.2 Views.....	8
1.3.3 Layouts.....	9
1.4 Eventhandling .....	10
1.5 Data Binding.....	11
1.6 Dependency Service .....	12
1.7 Messaging Center .....	13
2 Aufbau der Photos Applikation.....	13
2.1 Modelklassen .....	13
2.2 Views.....	14
2.2.1 CarouselView .....	14
2.2.2 ListView .....	16
2.2.3 Toolbar .....	18
2.3 Aufruf von nativen Funktionen .....	20
3 Azure .....	
3.1 Architektur von Azure Blob Storage .....	22
3.2 Konfiguration von Azure mit Blob Storage .....	23
3.3 Beispiel Backend C# Client.....	26
4 Aufbau des Photos Azure Client Backend .....	27
4.1 Serviceklassen .....	28
4.2 Administration Blob Container.....	31
Literaturverzeichnis.....	32
Abbildungsverzeichnis .....	33

# Einführung

Xamarin bietet die Möglichkeit mit Xamarin.Forms plattformübergreifend zu entwickeln. Dabei werden die bekanntesten Plattformen wie Android, iOS oder WindowsPhone unterstützt. Dieses Framework wird von Microsoft verbreitet und es gibt dazu zahlreiche Dokumentation welche frei von Microsoft zur Verfügung gestellt wird. Desweiteren gibt es auch Erweiterungen mit denen man externe Anwendungen benutzen kann. Eine davon ist die Unterstützung von Azure. Damit wird es ermöglicht Daten der App persistent in Azure zu speichern.

In diesem Projekt war es Ziel einen Photo App zu entwickeln die es ermöglicht eigene Fotos zu verwalten. Zusätzlich sollte man die Fotos kommentieren können. Die App sollte zumindest auf iOS und Android lauffähig sein.

## 1 Xamarin.Forms

Für unsere Applikation wollen wir Xamarin.Forms verwenden um mit einer Codebasis (Shared Code) die App auf allen Plattformen laufen zu lassen. Für das Projekt wurde „*Visual Studio for Mac*“ benutzt damit wird es ermöglicht Apps für iOS und Android zu entwickeln. Da für die Entwicklung der unterschiedlichen OS das darunterliegende betriebssystemspezifische Framework vorhanden sein muss, gibt es keine Windows Version dieser App. So muss zur Entwicklung von iOS zum Beispiel XCode installiert werden um dieses in Xamarin.Forms zu verwenden. Um wirklich für alle drei Plattformen entwickeln zu können braucht man daher zwei Rechner (bzw. eine VM). Die Applikation kann dann im entsprechenden iOS Simulator oder Android VM direkt angezeigt und getestet werden.

Der Vorteil von Xamarin.Forms liegt auf der Hand: Es ermöglicht das Entwerfen einer GUI, die dann wieder von allen OS verwendet werden kann. Die Programmierung dieser GUI kann über C# erfolgen oder aber auch über die eigene Beschreibungssprache XAML die an XML angelehnt ist und dem Beschreibungscode in der Android Entwicklung ähnelt. Leider gibt es keinen XAML Designer in der derzeitigen Visual Studio Version und somit das Design etwas erschwert.

Zusätzlich kann nativer Code implementiert werden um damit spezifische Funktionalität der Zielplattform zu verwenden. Das wird zum Beispiel benötigt um die jeweiligen Fotobibliotheken aufzurufen. Zusätzlich besteht ein Kommunikationskanal zurück zum Shared Code Xamarin.Forms um Werte von den nativen Plattformen weiterverwenden zu können. Im Fall der Fotobibliothek wäre das ein binärer Stream mit dem man auf Xamarin.Forms Seite einen ImageSource laden kann.

## 1.1 Architektur von Xamarin

Die prinzipielle Architektur von Xamarin.Forms baut darauf auf, dass sich Xamarin.Forms on top der Applikation setzt und als Compiler für diese Betriebssysteme fungiert. Das heißt, es wird tatsächlicher ausführbarer Code für die jeweilige Applikation generiert und Xamarin.Forms agiert nicht als Interpreter. Am besten veranschaulicht untenstehende Abbildung den Aufbau. Xamarin.Forms wird immer dann verwendet, wenn man möglichst wenig nativen Code zu programmieren hat, da hier deutlich die GUI-Entwicklung vereinfacht wird (siehe Abbildung 1).

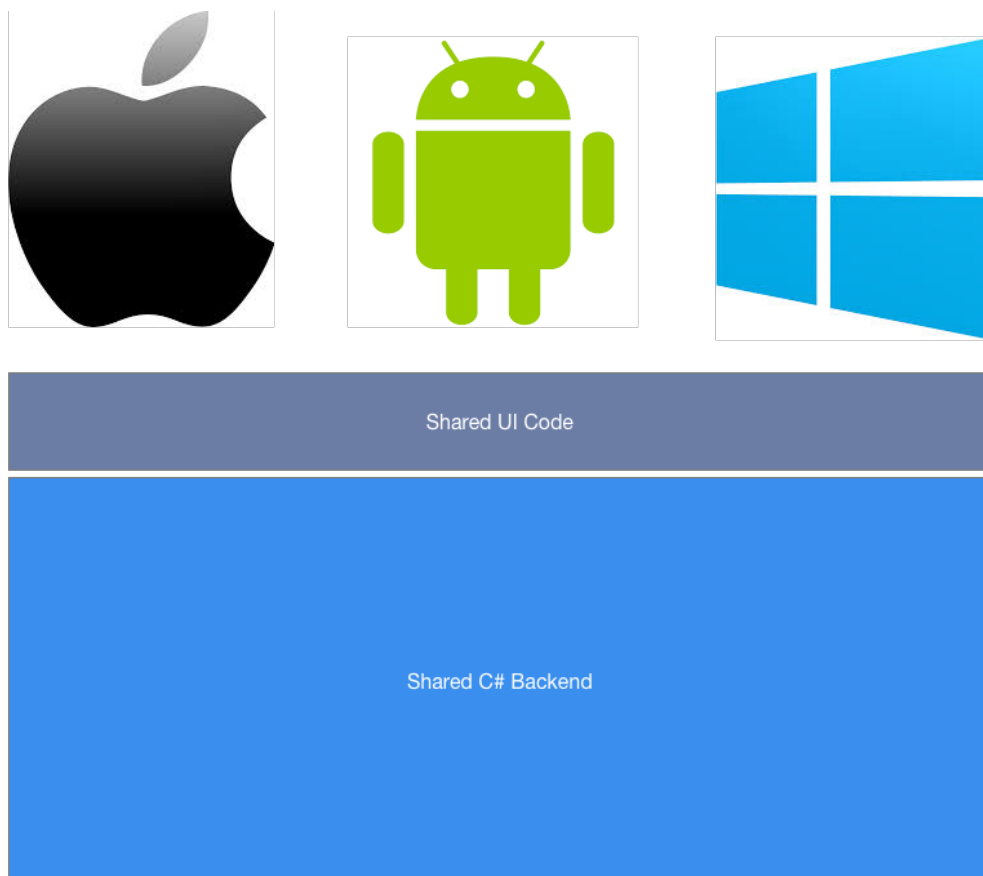


Abbildung 1: Architektur Xamarin.Forms

In früheren Versionen von Xamarin war es noch nicht möglich, einen Shared Code für die UI zu nutzen; mittlerweile ist das über Xamarin.Forms kein Problem. Nach wie vor kann man natürlich für jede Plattform seinen eigenen UI Code in C# entwickeln. Die Xamarin.Forms-Anwendung ist wie jede plattformübergreifende Anwendung aufgebaut, wobei der Code in eine portable Klassenbibliothek (Portable Class Libraries) platziert wird und die jeweiligen nativen Anwendungen diesen dann nutzen. Untenstehende Abbildung sollte dieses Prinzip verdeutlichen [1].

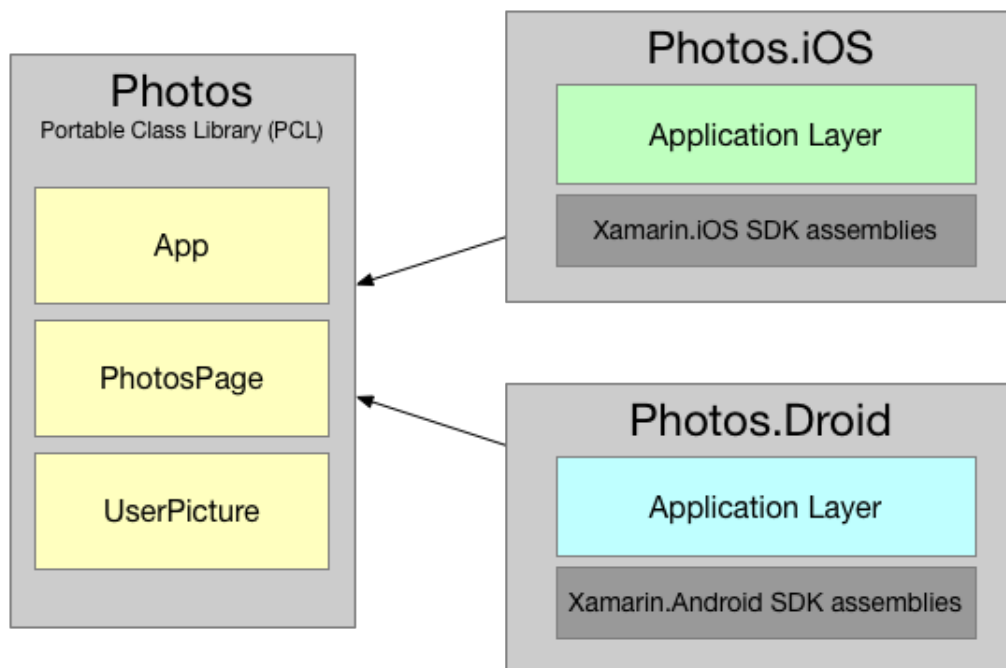


Abbildung 2: Beziehung der Portable Class Libraries

Um das Umzusetzen gibt es in einem Xamarin.Forms drei verschiedene Hauptbereiche die relevant für die Entwicklung sind und je nach Bedarf den unterschiedlichen Code beinhalten. Diese werden im folgenden Kapitel beschrieben.

## 1.2 Aufbau eines Xamarin.Forms Projekts

Wenn man sich die Photos Applikation ansieht gibt es drei Bereiche, wenn man für iOS und Android programmiert. Wenn noch eine Windows Phone Zweig vorhanden wäre, wären es dann vier Bereiche. Diese Bereiche sind immer zwingend in einem Xamarin.Forms Projekt enthalten.

- **Photos:** Dieser Teil enthält den Code der für die SharedCode/PCL benötigt wird
- **Photos.Droid:** Hier befindet sich spezifischer Android Code und bildet auch den Einstiegscode für Android Anwendungen
- **Photos.iOS:** Hier befindet sich spezifischer iOS Code und bildet auch den Einstiegscode für iOS Anwendungen

Unter diesen einzelnen Teilen befindet sich Unterverzeichnisse die unter anderen Referenzen die zum Ausführen der Anwendung erforderlich sind aber auch Pakete, sogenannten NuGet Pakete, die es ermöglichen Bibliotheken von Drittanbietern in das Projekt einzubinden.

Beim Exekutieren einer Applikation wird zum Beispiel bei iOS die *Main.cs* Datei aufgerufen, diese wiederum ruft nur die *AppDelegate.cs* Datei auf, die den Code auf dem iOS generiert. *AppDelegate* Klasse wird jeder nativer iOS Entwickler kennen, bei Android heisst die *Main.cs* dann *MainActivity.cs*, die den nativen Android Code enthält. Untenstehende Abbildung verdeutlicht diese Beziehungen untereinander.

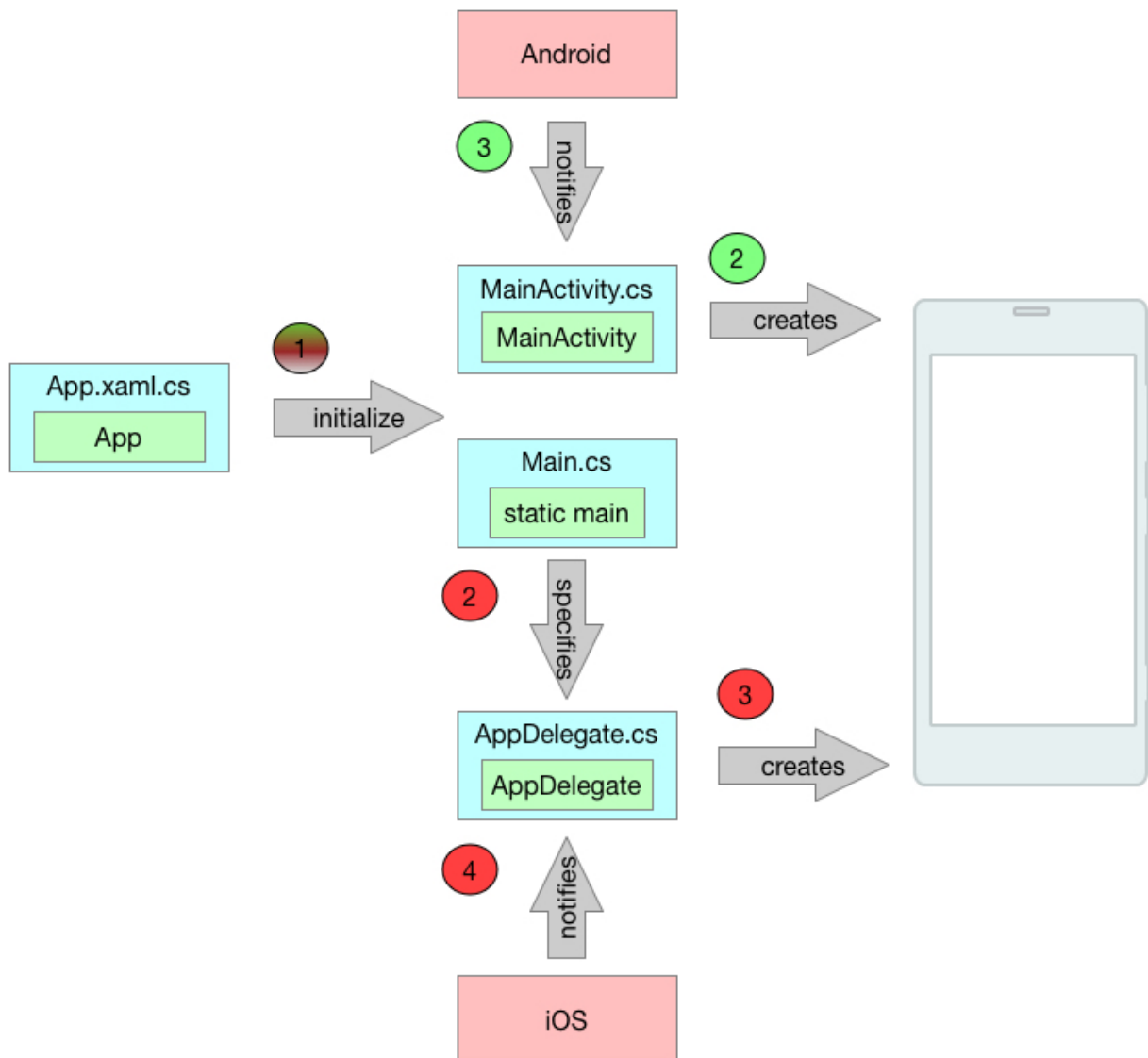


Abbildung 3: Beziehung Xamarin.Forms zu nativen Plattformen (Quelle [modifiziert]: [1])

Bei der oberen Abbildung gibt es auch den Schritt 3 oder 4, der eine Kommunikation von iOS oder Android zum Xamarin.Forms Shared Code ermöglicht. Diese wird im konkreten Fall auch benötigt, da wir die Fotos, die in der iOS Photo Library ausgewählt werden, im Shared Code Teil der Applikation verwenden wollen.

Der Start der Applikation findet in der App.xaml.cs an. Diese initialisiert alle Komponenten und startet den richtigen nativen Code der gerade benötigt wird. Wenn es sich um eine iOS Applikation handelt wird als nächstes die *Main.cs* exekutiert

```
namespace Photos.iOS
{
    public class Application
    {
        // This is the main entry point of the application.
        static void Main(string[] args)
        {
            // if you want to use a different Application Delegate class from "AppDelegate"
            // you can specify it here.
            UIApplication.Main(args, null, "AppDelegate");
        }
    }
}
```

Diese Klasse macht nun wiederum nichts anderes und ruft die AppDelegate Klasse auf:

```
namespace Photos.iOS
{
    [Register("AppDelegate")]
    public partial class AppDelegate : global::Xamarin.Forms.Platform.iOS.FormsApplication
    Delegate
    {
        public override bool FinishedLaunching(UIApplication app, NSDictionary options)
        {
            global::Xamarin.Forms.Forms.Init();
            Microsoft.WindowsAzure.MobileServices.CurrentPlatform.Init();
            LoadApplication(new App());
            return base.FinishedLaunching(app, options);
        }
    }
}
```

Damit werden alle notwendigen Schritte unternommen um die Anwendung auf der iOS Plattform zu starten. Ein spezifischer Aufruf wurde bei diesen Projekt hinzugefügt und ist oben im Code Abschnitt rot markiert. Dieser ist notwendig um die Dienste für Azure zu initialisieren. Allerdings ist das nur im iOS Code notwendig und Android läuft ohne Anpassungen mit Azure.

Viele Klassen in Xamarin.Forms sind (außer die reinen Model oder Service Klassen) eine Kombination aus einem C# und XAML File. Dabei definiert das XAML File das Design der jeweiligen Page, diese Elemente könnten dann über das C# File angesprochen werden, bzw. es werden Events der einzelnen GUI Elemente behandelt. Prinzipiell ist es auch möglich die gesamte GUI über das C# festzulegen, dies bleibt dem Entwickler überlassen. Dieser Aspekt führt uns auch zu den verschiedenen Designelementen von Xamarin.Forms.

## 1.3 User Interface

In Xamarin.Forms gibt es prinzipiell drei Hauptgruppen wie der Content in der App dargestellt wird. Diese werden nun in den folgenden Kapitel beschrieben mit einer kurzen Übersicht welche Möglichkeiten es dazu gibt.

### 1.3.1 Pages

Damit wird der Screen der Applikation abgebildet. Man kann sie mit den View Controller in iOS oder mit der Activity in Android vergleichen (es ist aber keine Activity!).

Dazu gibt es verschieden Pagetypen:

- **ContentPage**: Die einfachste Form seinen Inhalt anzuzeigen
- **MasterDetailPage**: Mit dieser Page werden zwei Seiten visualisiert
- **NavigationPage**: Damit kann man sich durch eine Reihe von Pages einfach navigieren
- **TabbedPage**: Damit kann sich mit Tabs durch einzelnen Pages navigieren
- **TemplatedPage**: Die Basis Klasse für die ContentPage
- **CarouselPage**: Damit kann man sich mit Fingerwischen durch einzelne Pages navigieren wie zum Beispiel einer Foto Galerie.

Pages sind damit das Grundelement zur Darstellung in Applikationen.

### 1.3.2 Views

Views sind die Elemente die zur Interaktion mit dem Benutzer benötigt werden (Button, ListView, Editor, ....). Diese Element sind in der Regel eine Subklasse der Viewklasse.

Xamarin.Forms bietet hier eine Vielzahl von Views an:

- **ActivityIndicator**: Damit kann die Aktivität eines Prozesses visualisiert werden. Damit der User weiß das etwas passiert.
- **BoxView**: Ein simples Rechteck



- **Button**: Ein Schalter der mit diversen Events verknüpft werden kann
- **DatePicker**: Eine View um visuell ein Datum auszusuchen
- **Editor**: Damit können mehrere Zeilen eingegeben werden
- **Entry**: Damit kann man einzeiligen Text eingeben
- **Image**: Mit diesen View kann man Images visualisieren
- **Label**: Hier wird ein nicht editierbarer Text angezeigt werden
- **ListView**: Eine Sammlung von Daten kann hier als Liste dargestellt werden
- **OpenGLView**: Damit kann man OpenGL Content abbilden
- **Picker**: Hiermit kann ein Element aus einer Liste ausgewählt werden
- **ProgressBar**: Mit diesen View kann der Fortschritt eines Prozesses dargestellt werden
- **SearchBar**: Darstellung einer Sucheingabe
- **Slider**: Hiermit kann eine linearer Wert eingegeben werden
- **Stepper**: Damit kann man einen einzelnen Werte aus einer bestimmten Menge auswählen
- **Switch**: Visualisierung einen Schalters mit zwei möglichen Zuständen
- **TableView**: Damit können Reihen und Zellen abgebildet werden
- **TimePicker**: Eine View um visuell eine Zeit auszusuchen
- **WebView**: Damit wird HTML Content visualisiert

Diese Views werden im Allgemeinen Teil des Projekts definiert und werden dann auf das jeweilige native System gerendert.

Zu den jeweiligen Views gibt es auch noch eine Vielzahl von Attributen wie zum Beispiel Farbe, Padding oder Position die definiert werden können. Man kann dabei alle Elemente entweder direkt in der C# Klasse definieren oder im jeweiligen XAML File der Klasse. Bis dato gab es leider keinen Designer für Visual Studio for Mac, der das Design wesentlich einfacher gestalten würde.

### 1.3.3 Layouts

Layouts werden benutzt um eine logische Struktur in das Userinterface zu bekommen. Die Layoutklasse ist eine Subklasse der Viewklasse und kann auch als Container agieren um andere Layouts oder Views aufzunehmen. Layouts kontrollieren damit die Position und die Größe der einzelnen Views.

Untenstehend eine Übersicht über die wichtigsten Layouts:

- **ContentView**: Ist ein Element mit einem einzigen Element. Im Prinzip ist das die BaseView für alle anderen Layouts.

- **StackLayout:** Ein einfaches aber auch sehr effizientes Layout mit dem man Views positionieren kann. Damit werden die Elemente in einer Linie untereinander angeordnet werden. Das kann sowohl horizontal als auch vertikal sein.
- **ScrollView:** Damit kann man durch den jeweiligen Content scrollen
- **AbsoluteLayout:** Damit werden die View an absoluten Positionen verankert
- **Grid:** Eine Layout Art mit Reihen und Spalten
- **RelativeLayout:** Damit können UIs entwickelt die sich auf jede Bildschirmgröße anpassen können. Dabei wird mit sogenannten Constraints gearbeitet

Alle Layouts werden dann entsprechend auf den nativen Geräten abgebildet und es gibt nur eine Stelle in der Applikation wo diese bearbeitet werden müssen um somit auf allen Geräten korrekt angezeigt zu werden.

## 1.4 Eventhandling

Für jede Applikation ist es notwendig ein jeweiliges Eventhandling zu implementieren um mit dem User zu interagieren. Dieses Eventhandling wird auch im gemeinsamen Code der Applikation hinterlegt. Das kann man am einfachsten im entsprechenden XAML Code machen. Dabei hat jedes Element eigene Eventtrigger. Beim Button ist das zum Beispiel ein Trigger mit den Namen *Clicked*. Wie schaut das nun im XAML Code aus:

```
<Button x:Name="b_editor" Image="paper_plane.png" Text="Post Comment" Clicked="comment_button"></Button>
```

Neben anderen Attribute sieht man hier das Clicked Event, das hier die Methode *comment\_button* in der jeweilige C# Klasse aufrufen wird sobald der Button gedrückt. Nun muss man nur dafür Sorgen tragen die entsprechende Methode in der Klasse zu hinterlegen. Das schaut folgendermaßen aus:

```
void comment_button(object sender, EventArgs e)
{
    // Event Code
}
```

Hiermit hat man dann die Methode definiert in der man auf den Button Click reagieren kann. Jedes View Element hat einer dieser Click Events um darauf reagieren zu können.

Anstatt im XAML File einen Methode anzugeben kann man das Ganze auch direkt in der C# Klasse abdecken.

```
b_editor.Clicked += (sender, ea) => {
    // Event Code
};
```

Dieser Aufruf hat die selbe Funktion wie die Definition einer Methode im XAML File. Es bleibt den Entwickler überlassen welchen Ansatz er verwenden will.

## 1.5 Data Binding

In den meisten Fällen will man die eingebenden Daten in der App auch irgendwo wieder anzeigen. Meist wird dafür eine *ListView* verwendet mit der man die Daten schön tabellarisch darstellen kann. Diese *ListView* wiederum braucht eine Datenquelle damit sie weiß was dargestellt werden soll. Am besten definiert man die Daten daher in einer Modelklasse mit den verschiedenen Getter und Setter Methoden und bildet dort einen Konstruktor, der die Attribute belädt. Damit kann man dann folgendermaßen eine sogenannte *ObservableCollection* erstellen:

```
public ObservableCollection<YourClass> yours { get; set; }
```

und mit

```
yours.Add(new YourClass(some_attributes1, some_attributes2));
```

kann man diese Collection dann mit Daten beladen. So einmal vorbereitet ist es relativ einfach diese Collection mit der jeweiligen *ListView* zu verknüpfen.

```
listView.ItemsSource = yours;
```

Zusätzlich muss man natürlich den Konnex in dem XAML File herstellen das erfolgt mit dem Schlüsselwort *Binding* und sieht so aus:

```
<Label Text="{Binding Comment}" />
```

Hier wird mittel dem Binding Keyword das gewünschte Attribut der Modelklasse eingebunden.

Das war alles und bei jeder Veränderung dieser Datenquelle wird auch die jeweilige *ListView* aktualisiert und die Werte aktualisiert.

## 1.6 Dependency Service

Ein wichtiger Punkt für das Projekt waren die Dependency Services da hiermit auf native Funktionen zugegriffen werden kann, wie in unseren Fall auf die Foto Bibliothek. Wir werden uns vorerst die allgemeine Struktur ansehen wie so ein Dependency Service implementiert wird. Dazu werden folgende Elemente benötigt

**Interface:** Dabei wird im gemeinsamen Code die Funktionalität definiert

**Implementation pro Plattform:** Für dieses Interface muss der jeweilige native Code der Plattform hinterlegt werden

**Registrierung:** Mit den Metadaten Attributen muss die jeweilige Klasse registriert werden. Das ist notwendig damit während der Laufzeit die native Klasse gefunden werden kann

**Aufruf des Dependency Service:** Die Klasse muss explizit im gemeinsame Code aufgerufen werden. Im Detail wird das im praktischen Teil des Papers beschrieben. Die untenstehende Abbildung gibt eine gute Übersicht wo etwas definiert oder implementiert werden muss.

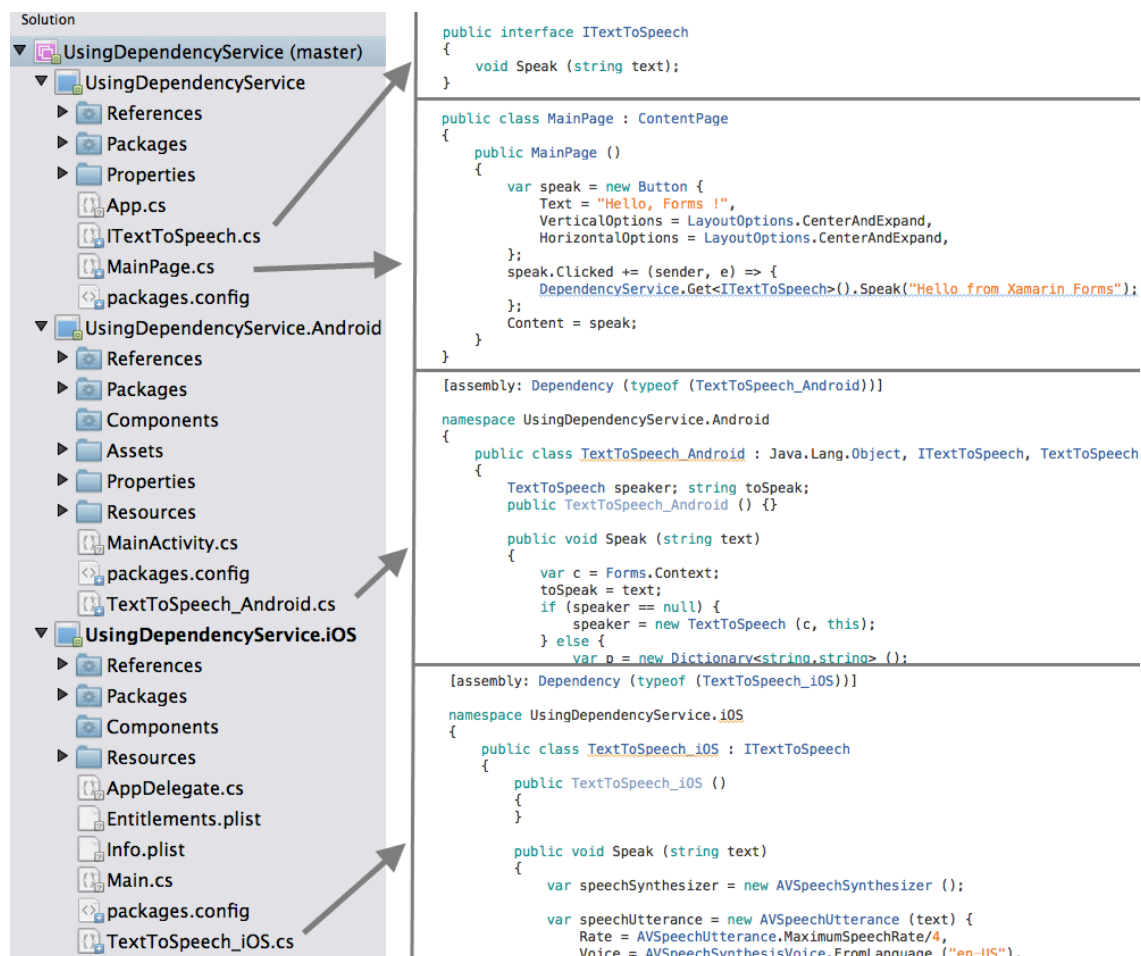


Abbildung 4: Dependency Service Übersicht

Wie man in der Abbildung oben schön sieht sind hier in jedem Teil des Projektes Anpassungen notwendig.

## 1.7 Messaging Center

Um nun eventuelle Rückgabewerte von den nativen Clients zu bekommen braucht man das Messaging Center. Damit ist es möglich mit Komponenten zu kommunizieren, die voneinander nichts wissen. Dazu gibt es zwei Teile:

- **Subscribe:** Hier wird eine Nachricht erwartet und nachdem diese angekommen ist kann man den entsprechenden Code dazu ausführen
- **Send:** Hier wird eine Nachricht versendet die vom jeweiligen Consumer (der sich vorher mit Subscribe angemeldet hat) empfangen werden kann.

Natürlich kann sich auf wieder mit *unsubscribe* vom Empfang der Nachrichten abmelden. Wie dieses relative simple Konstrukt im Details aussieht wird im praktischen Teil vorgestellt.

## 2 Aufbau der Photos Applikation

Nach der kurzen Einführung in Xamarin.Forms und deren wichtigsten Komponenten werden hier die Elemente des Projekts beschrieben. Dabei wurde bewusst der Azure Teil rausgenommen um nur die Xamarin.Forms relevanten Teile zu beschreiben, da Azure in einem extra Kapitel nochmals beschrieben wird.

### 2.1 Modelklassen

Um die Daten in der Applikation abzubilden wurde verschieden Klassen angelegt und diese mit entsprechenden Getter- und Setter Methoden ausgestattet. Vorrauschauend auf die Verwendung von Azure wurde auch immer eine eigene ID mit aufgenommen, damit die Datenbankeinträge einzigartig (unique) sind.

- **UserPicture** : enthält alle Bilder des Users
- **PictureComment**: enthält alle Kommentare zum jeweiligen Bild

Diese beiden Klassen werden dazu verwendet um die entsprechenden *ListView* in der Applikation durch sogenanntes Data Binding mit entsprechenden Daten zu versorgen.

Wie so eine Modelklasse im Detail aussieht zeigt untenstehender Code:

```

namespace Photos
{
    public class UserPicture
    {

        public string Id { get; set; }
        public ImageSource Picture { get; set; }

        public UserPicture(ImageSource picture, string id)
        {
            this.Picture = picture;
            this.Id = id;
        }
        public UserPicture()
        {
        }
    }
}

```

Hier sieht man das für jedes Attribut eine Getter und Setter Methode angelegt wurde und auch ein Konstruktor definiert wurde den wir für das adden der Objekte in die ObservableCollection benötigen. In dieser Klasse wird dabei die *ImageSource* und ein eindeutiger Identifier hinzugefügt. Der Identifier wird benötigt um dann die richtigen Kommentare zum Bild anzuzeigen (aber auch dann für Azure). Zusätzlich wurde im entsprechenden XAML File *PhotosPage.xaml* die ListView definiert in der die Daten dargestellt werden.

## 2.2 Views

Es gibt zwei Views in der Applikation, einer der die Fotos der anzeigen sollte und ein andere mit des es möglich ist Kommentare zu den jeweiligen Fotos abzugeben. Zum Anzeigen der Fotos wurde ein CarouselView gewählt den man mit NuGet in seine Applikation einbinden muss um in auch verwenden zu können. Die dazu notwendigen Schritte werden im folgenden Kapitel beschrieben.

### 2.2.1 CarouselView

Der CarouselView wurde schon in Xamarin.Forms Version 2.2.0 fix aufgenommen, aber auf Grund von diversen Problemen in der Version 2.3.0 wieder herausgenommen, deswegen muss man sich diesen View über NuGet in die projekteigenen Packages installiert werden [2]. Wichtig dabei ist das man in den XAML Dateien den entsprechenden Namespace

inkludiert, aber auch in den jeweiligen Start Methoden der nativen Plattformen den View initialisiert. Im konkreten Fall musste zum Beispielt im iOS Projekt folgende Zeile in die AppDelegate hinzugefügt werden:

```
global::CarouselView.FormsPlugin.iOS.CarouselViewRenderer.Init();
```

Damit wurde sichergestellt, dass auch alle CarouselView Methoden innerhalb der C# Klassen verwendet werden konnten. Um den CarouselView in einem XAML File zu verwenden war es auch notwendig den korrekten Namespace in ContenPage Tag einzupflegen

```
xmlns:cv="clr-
namespace:CarouselView.FormsPlugin.Abstractions;assembly=CarouselView.FormsPlugin.A
bstractions"
```

Dadurch konnte man mit den CarouselView Bilder anzeigen, die in ein sogenannten DataTemplate eingebettet werden mussten, damit auch tatsächlich Daten angezeigt wurden. Des Weiteren wurde mittels dem Attribute *x:name* eine eindeutiger Name vergeben um den View auch im C# Code anzusprechen um so zum Beispiel eine Data Source hinzufügen. *ShowIndicator* legt fest ob man Pfeile in der View zum hin und her schieben der Bilder zeigen will und *Orientation* legt fest in welche Richtung der CarouselView verschoben werden soll.

```
<ContentView Grid.Row="1" Grid.Column="0">
  <cv:CarouselViewControl AnimateTransition="true" ShowArrows="true" ShowIndicators=
"true" Orientation="Horizontal" x:Name="CarouselPics" >
    <cv:CarouselViewControl.ItemTemplate>
      <DataTemplate>
        <Image Source="{Binding Picture}"/>
      </DataTemplate>
    </cv:CarouselViewControl.ItemTemplate>
  </cv:CarouselViewControl>
</ContentView>
```

Zusätzlich musste im C# Code noch mit *ItemSource* die korrekte Modelklasse hinzugefügt werden, damit der CarouselView nach jedem Update der Daten automatisch gerendert wird. In unseren Fall ist das die Klasse *UserPicture*.

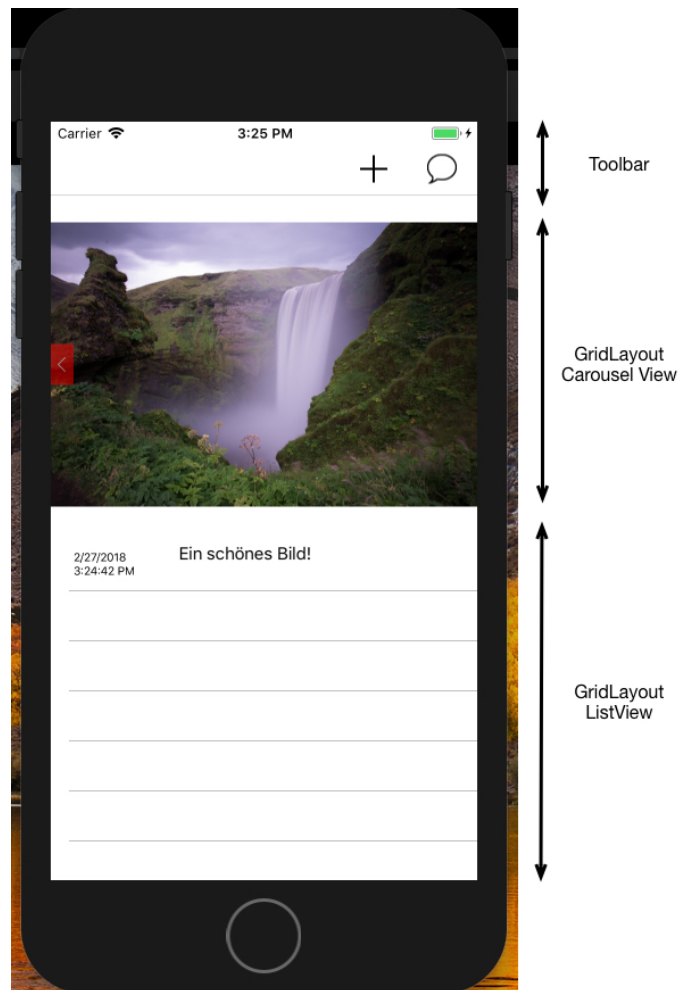


Abbildung 5: CarouselView

In dieser Abbildung sieht man auch welches Layout und welche Art von View in den jeweiligen Bereichen verwendet wurde.

Im unteren Teil der Applikation werden die Kommentare zu den jeweiligen Bilder angezeigt. Dazu wurde eine *ListView* verwendet die im folgenden Kapitel beschreiben wird.

## 2.2.2 ListView

Um die Daten in der *ListView* korrekt zu verwenden greifen wir auf die Modelklasse *PictureComment* zu.

```
<ScrollView Grid.Row="1" Grid.Column="0" Orientation="Both">
  <ListView x:Name="lstView" ItemTapped="tapped_listview">
    <ListView.ItemTemplate >
      <DataTemplate>
        <ViewCell>
```



```

        <Grid>
        <Grid.RowDefinitions>
        <RowDefinition Height="*" />
        </Grid.RowDefinitions>
        <Grid.ColumnDefinitions>
        <ColumnDefinition Width="100" />
        <ColumnDefinition Width="Auto" />
        </Grid.ColumnDefinitions>
        <Label VerticalOptions="Start" Scale="0.6" Text="{Binding CurrentDate, StringFormat='{0:MMMM dd, yyyy hh\\:mm}}'" Grid.Row="0" Grid.Column="0" />
        <Label Scale="0.9" Text="{Binding Comment}" Grid.Row="0" Grid.Column="1" />
    </Grid>
</TableViewCell>
</DataTemplate>
</ListView.ItemTemplate>
</ListView>
</ScrollView>

```

Für die Anzeige Elemente wurde ein `GridLayout` verwendet um die View richtig zu positionieren, aber es wurde auch nochmals ein `GridLayout` innerhalb der Liste definiert um Datum und Kommentar sauber zu trennen. Mit den Binding für die Attribute *CurrentDate* und *Comment* wird dafür gesorgt dass die richtigen Daten aus der Modelklasse kommen.

Im Code selber speichern wir alle Kommentare in einer *ObservableCollection*:

```

comment = new ObservableCollection<PictureComment>();
comment.Where((comment) => comment.PictureId.Contains("0"));

```

Zugleich legen wir auch einen Filter an damit wir auch wirklich nur die Kommentare zu sehen sind die zum jeweiligen Bild passen. Zu Beginn ist der einfach 0. Sobald wir den Picture Stream von unserem nativen Client zurückbekommen können wir diesen hinzufügen.

```

//Set the source of the image view with the byte array
p.Picture = ImageSource.FromStream(() => new MemoryStream((byte[])args));
pic.Add(new UserPicture(p.Picture,counter.ToString()));
counter++;
CarouselPics.Position = counter;

```

Gleichzeit erhöhen wir unseren *counter* damit wir einen eindeutigen Identifier für unser Bild bekommen.

## 2.2.3 Toolbar

Als letztes Element in der Applikation kommt die Toolbar zum Einsatz, die in XAML sehr einfach zu konfigurieren ist. Wichtig dabei das die *ContentPage* selber als *NavigationPage* definiert ist. Diese kann man in der Startdatei im Shared Code festlegen – in der *App.xaml.cs*. Der Aufruf sieht dort wie folge aus:

```
public App()
{
    InitializeComponent();
    MainPage = new NavigationPage(new PhotosPage())
    {
        BarTextColor = Color.Black
    };
    //MainPage = new PhotosPage();
}
```

Ganz unten im auskommentierten Teil sieht man wie der Standard Aufruf aussehen würde. Wir legen nun unsere *PhotosPage* als *NavigationPage* fest. In der Methode sieht man auch dass man noch zusätzliche Attribute – in dem Fall *BarTextColor*, die Farbe der Toolbar – übergeben kann. Damit kann man die Toolbar im *PhotosPage.xaml* folgendermaßen definieren:

```
<ContentPage.ToolbarItems>
  <ToolbarItem Text="Add"
    Priority="1" Order="Primary"
    Clicked="GetPhoto"
    Icon="plus.png"
  />
  <ToolbarItem x:Name="t_comment"
    Priority="1" Order="Primary"
    Clicked="AddComment"
    Icon="message.png"
  />
</ContentPage.ToolbarItems>
```

Wichtig dabei ist dass dieses innerhalb der *ContentPage* Tags passiert. Wie man oben sehen kann man Namen für das *ToolbarItem* vergeben (mit *x:name*), aber auch eigene Icons für die Toolbar verwenden, dabei ist es wichtig das die Images dieser Icons sich an den

nativen Standard der Plattform halten und man muss sie in das richtige Verzeichnis des nativen Teils der Applikation hinterlegen. Bei iOS ist das im Verzeichnis *Resources* und beim Android wäre das zum Beispiel im Verzeichnis *Resources/drawable-hdpi*. Ein wichtiger Punkt ist noch eine Methode zu definieren die aufgerufen wird sobald auf das Item geklickt wird. Dazu verwendet man das Attribut *Clicked*. Im Falle vom Toolbar Item *t\_comment* ist das die Methode *AddComment* die folgende Aktionen auslöst.

```
void AddComment(object sender, EventArgs e)
{
    layout_editor.IsVisible = true;
    CarouselPics.IsVisible = false;
    editor.Focus();
}
```

Hier wird nun einfach der CarouselView versteckt und ein Editor View erscheint um ein Kommentar abzugeben. Das sieht nun so aus:

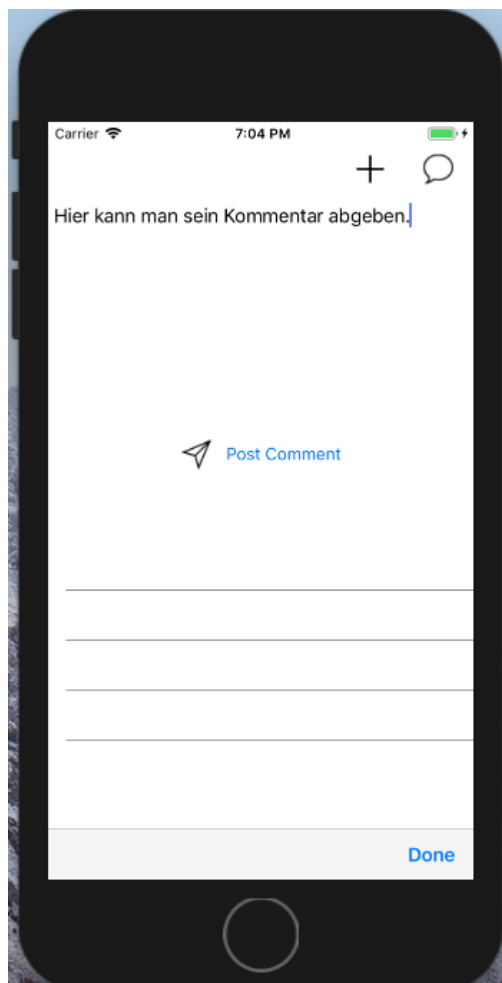


Abbildung 6: Kommentarfunktion

Nach Eingabe des Kommentar und klicken des Button wird folgende Methode ausgeführt:

```
void comment_button(object sender, EventArgs e)
{
    var text = editor.Text;
    comment.Add(new PictureComment(text, current_pic_id.ToString(), DateTime.Now.ToString()));
    listView.ItemsSource = comment.Where((comment) => comment.PictureId.Contains(current_pic_id.ToString()));
    layout_editor.IsVisible = false;
    CarouselPics.IsVisible = true;
    editor.Text = "";
}
```

Diese versteckt wieder alle Elemente und zeigt den CarouselView und fügt aber auch gleich mit *Add* den Kommentartext mit dem entsprechenden Identifier für das Bild in die ObservableCollection. Zusätzlich wird auch *ItemsSource* mit der *Where* Methode auf die richtigen Identifier des Bildes gesetzt.

## 2.3 Aufruf von nativen Funktionen

Im Xamarin.Forms gibt es nicht die Möglichkeit nativen Code direkt aufzurufen, sondern es muss dieser speziellen Code in den jeweiligen Verzeichnissen der Plattformen implementiert werden. Natürlich gibt es eine Schnittstellenfunktion um eventuelle Daten zwischen Xamarin.Forms und dem nativen Code auszutauschen. Dieser erfolgt über das sogenannte MessagingCenter. In diesem kann man sich zu gewissen Themen anmelden (subscribe) und diese auf der Gegenseite konsumieren.

Um die Verbindung herzustellen muss im Xamarin.Forms Strang eine Interface Klasse deklariert werden, die Methoden enthält die auf der nativen Seite aufgerufen werden. Diese können dann über die *Dependency Service* Klasse aufgerufen werden.

```
Device.BeginInvokeOnMainThread(() =>
{
    DependencyService.Get<CameraInterface>().BringUpPhotoGallery();
});
```

Dadurch wird in diesem zum Beispiel die Methode *BringUpPhotoGallery()* aufgerufen, diese wiederum muss im iOS und Android Strang implementiert werden. Sprich man implementiert

eine Klasse und vererbt dort das zuvor angelegte Interface. Dort implementiert man dann die Methode *BringUpPhotoGallery()*. Wichtig dabei ist es dieser Klasse im *DependencyService* zu registrieren:

```
[assembly: Dependency(typeof(Photos.iOS.CameriaIOS))]  
namespace Photos.iOS  
{  
    public class CameriaIOS : CameraInterface  
    {  
        .....  
    }  
}
```

Das geschieht mit den Metadaten Keyword *assembly*. Am Ende der implementierten Methode sollte folgender Aufruf folgen (sofern es Daten zu übertragen gibt).

```
MessagingCenter.Send<byte[]>(myByteArray, "ImageSelected");
```

Damit wird eine Nachricht an die aufrufende Klasse generiert und zurück gesendet. Im obigen Aufruf wird einfach ein Array aus Bytes mit dem „Namen“ *ImageSelected* zurück geschickt. An Hand dieses Namen kann man die Nachricht in der Xamarin.Forms Klasse zuordnen und empfangen über:

```
MessagingCenter.Subscribe<byte[]>(this, "ImageSelected", (args) =>  
{  
    Device.BeginInvokeOnMainThread(() =>  
    {  
        //Set the source of the image view with the byte array  
        p.Picture = ImageSource.FromStream(() => new MemoryStream((byte[])args));  
    });  
});
```

Dadurch erhalten wird im Projekt das Image aus der Foto Gallery des nativen Clients zurückgeschickt und in die entsprechende Klasse zurückgeliefert.

### 3 Azure

Zum persistenten Speichern der Daten wurde Azure gewählt und dazu ein Studenten Account über Imagine Microsoft erstellt um dort entsprechende SQL Datenbanken anzulegen und ein sogenanntes App Service zu hosten. Damit konnte man die Datenbankzugriffe direkt in das Xamarin.Forms Projekt einbinden. Diese erfolgte ein paar Schritten direkt im Portal von Azure um alle notwendigen serverseitigen Einstellungen vorzunehmen wichtig dabei war das sich alle Applikationen in der selben Ressourcengruppe

befinden müssen. Folgender Abschnitt gibt einen Überblick über die Architektur von Azure und zeigt die notwendigen Schritte um Azure in Xamarin.Forms einzubinden.

### 3.1 Architektur von Azure Blob Storage

Im Azure gibt es die Möglichkeit der *BlobStorage* in dem man viele Möglichkeiten hat seine Daten abzuspeichern. Vor allem bietet sich diese Art der Speicherung an Bilder abzuspeichern. Leider ist der BlobStorage nicht im Studenten Account enthalten und man muss kurzfristig auf einen Bezahl Account wechseln. Es gibt auch die Möglichkeit in Visual Studio einen BlobStorage Emulator einzurichten und diesen für das Development und Testen zu verwenden [3].

Wie ein *BlobStorage* organisiert ist zeigt die untere Abbildung.

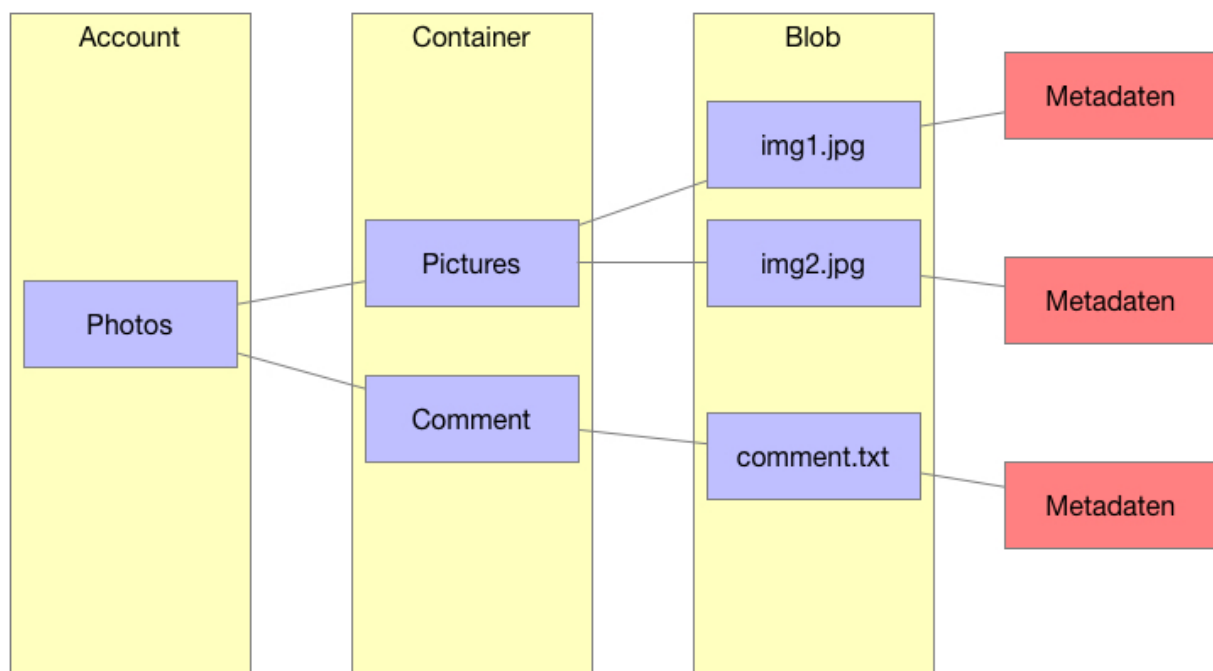


Abbildung 7: Blob Storage

Um den BlobStorage verwenden zu können muss man diesen vorher im Azure Portal dementsprechend einrichten und konfigurieren. Nach der Konfiguration erhält man einen fertigen Connection String mit einem Key um sich mit diesem von der Applikation zu den *BlobContainer* zu verbinden.

Dabei ist eines wichtig, dass man das NuGet Package für Microsoft.Storage einbindet, sowohl im Shared Code als auch in den jeweiligen nativen Projektteilen, sonst wird die Verbindung zum Azure Blob Storage nicht funktionieren.

## 3.2 Konfiguration von Azure mit Blob Storage

Bevor man den Blob Storage verwenden kann sind einige Konfigurationsschritte im Portal von Azure notwendig. Diese werden hier nun in allgemeinen durchgegangen.

Zuerst wählt man am rechten Rand den Menüpunkt Storage Account aus.

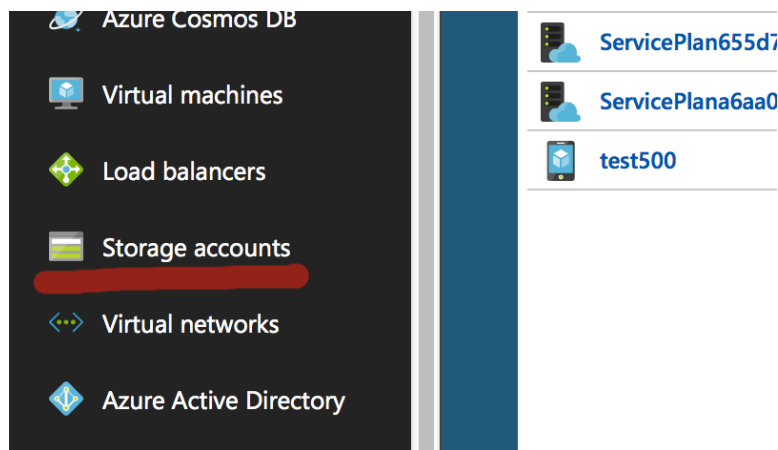


Abbildung 8: Auswahl Storage Account

Dadurch kann man mit *Add* einen neuen Storage anlegen.

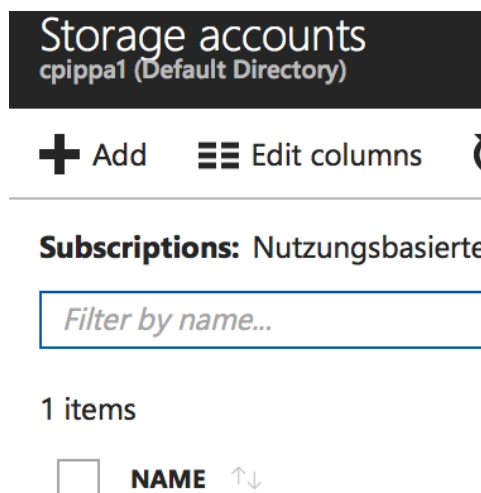


Abbildung 9: Anlegen Storage

Im Wesentlichen wird hier vorerst nur einmal der Name des Storage bekanntgegeben und man muss dieselbe Ressourcengruppe auswählen die man für seinen Windows Server verwendet hat. Wichtig ist auch das der Name des Blobs einzigartig sein muss. Das wird aber gleich nach der Eingabe überprüft. Eines ist auch ganz wichtig: Den Account Type von General Storage auf Blob Storage umzustellen.

\* Name ⓘ

testblob999 ✓

.core.windows.net

Deployment model ⓘ

Resource manager Classic

Account kind ⓘ

Storage (general purpose v1) ▾

Performance ⓘ

Standard Premium

Replication ⓘ

Read-access geo-redundant storage (R... ▾

\* Secure transfer required ⓘ

Disabled Enabled

\* Subscription

Nutzungsbasierte Bezahlung ▾

\* Resource group

☐ Create new ☒ Use existing

xamarin ▾

Abbildung 10: Dateneingabe Storage

Nach betätigen des Create Buttons wird der Storage deployed. Dies kann ein bisschen dauern, aber nach einer Weile hat man nach neuerlichen Auswahl von Storage Account seinen BlobStorage in der Liste angezeigt.



Um das Ganze zu vervollständigen muss man nun einen Container für seinen BlobStorage anlegen, den dorthin kommen dann unsere tatsächlichen Blobs und werden persistent gespeichert. Dazu muss man den richtigen Storage auswählen und kommt zu folgender Übersicht.

The screenshot shows the Azure Storage portal interface. On the left is a sidebar with navigation options: Overview, Activity log, Access control (IAM), Tags, Diagnose and solve problems, and a SETTINGS section containing Containers and Access keys. The 'Containers' option is selected. The main area displays details for the storage account 'xamarinphoto'. At the top, there are buttons for '+ Container', 'Refresh', and 'Delete'. Below these, the storage account details are shown in two columns. The left column lists: Storage account (xamarinphoto), Status (Primary: Available, Secondary: Available), Location (West Europe, North Europe), Subscription (change) (Nutzungsbasierte Bezahlung), and Subscription ID (663e6da9-95de-401e-b787-d5f8601c68d0). The right column lists: Primary blob serv (https://xamarinpl), Secondary blob s (https://xamarinpl), Replication status (Live), and Last synchronizec (3/9/2018, 12:24:4). Below the details is a search bar 'Search containers by prefix' and a table of containers. The table has two columns: NAME and LAST MODIFIED. One container is listed: 'xamarinphotocontainer' with a last modified date of '3/2/2018, 10:00:09 A'.

NAME	LAST MODIFIED
xamarinphotocontainer	3/2/2018, 10:00:09 A

Abbildung 11: Anlage Container

In dieser Übersicht kann man unter Settings -> Containers den gewünschten Container hinzufügen. Unter dem Menüpunkt Containers ist der nächste wichtige Menüpunkt Access Keys.

Da man nicht will, dass jeder in dem Container speichern kann benötigt man einen Schlüssel um die Verbindung aufzubauen.

Storage account name

xamarinphoto

**key1** ↻

Key

VKK/awZJuzJ2TrCQszdFD0ntrphzeVapPxQY8d+UjTQ/ZSWef3KHoAh65klz5b6vtMyHiLaUhtVVIY8IMwh8vQ==

Connection string

DefaultEndpointsProtocol=https;AccountName=xamarinphoto;AccountKey=VKK/awZJuzJ2TrCQszdFD0ntrphzeVapPxQY8d+UjT...

**key2** ↻

Key

gKTZuLdB0KbFO4XksLTydr3nNHk2aCA6awCcUgqu19A2jRZiUhqXfNThKj2iiwU/E5GqHtJRQvSPN6oTRCpoAw==

Connection string

DefaultEndpointsProtocol=https;AccountName=xamarinphoto;AccountKey=gKTZuLdB0KbFO4XksLTydr3nNHk2aCA6awCcUgqu...

Abbildung 12: Schlüssel für Blob Storage

Man sollte den Schlüssel regelmäßig ändern, dazu gibt es auch dann den zweiten Schlüssel damit man diesen verwenden kann wenn man den anderen Schlüssel neu generiert. Zusätzlich sehen wir einen vollständigen Connection String der in der Applikation verwendet werden kann um eine Verbindung zum Storage Account aufzubauen.

Dieser sieht wie folgt aus:

```
DefaultEndpointsProtocol=https;AccountName=xamarinphoto;AccountKey=VKK/awZJuzJ2TrCQszdFD0ntrphzeVapPxQY8d+UjTQ/ZSWef3KHoAh65klz5b6vtMyHiLaUhtVVIY8IMwh8vQ==;EndpointSuffix=core.windows.net
```

Und dieser wird in der Regel wie folgt verwendet:

```
CloudStorageAccount.Parse("DefaultEndpointsProtocol=https;AccountName=xamarinphoto;AccountKey=VKK/awZJuzJ2TrCQszdFD0ntrphzeVapPxQY8d+UjTQ/ZSWef3KHoAh65klz5b6vtMyHiLaUhtVVIY8IMwh8vQ==;EndpointSuffix=core.windows.net");
```

Damit bekommt man die Connection zum Storage Account und kann seinen Blob Storage ansprechen.

### 3.3 Beispiel Backend C# Client

Im folgendem wird allgemein beschrieben wie man das Backend in C# implementieren kann. Es wird sicher gut sein eine eigene Klasse zu implementieren die es ermöglicht die Verbindung zum Storage aufzubauen. Diese Klasse enthält dabei folgende Methodenaufrufe.

```
// Verbindung zum Storage Account mit dem Connection String
CloudStorageAccount storageAccount = CloudStorageAccount.Parse(
    CloudConfigurationManager.GetSetting("StorageConnectionString"));

// Hinzufügen eines Blob Clients
CloudBlobClient blobClient = storageAccount.CreateCloudBlobClient();

// Die Referenz zum angelegten Container
CloudBlobContainer container = blobClient.GetContainerReference("mycontainer");

// Erhalten einer Referenz zum Blob
CloudBlockBlob blockBlob = container.GetBlockBlobReference("myblob");

// Beschreiben des Blobs
using (var fileStream = System.IO.File.OpenRead(@"path\myfile"))
{
    blockBlob.UploadFromStream(fileStream);
}
```

In den Kommentaren des Source Codes kann man schön nachvollziehen wann, wo und zu wen eine Verbindung aufgebaut (bzw. eine Referenz generiert wird). Beim Upload des Blobs ist zu beachten das eventuell vorhandene Blobs mit denselben Namen überschrieben werden. Daher sollte man darauf achten dass dieser Name einzigartig ist um dies zu vermeiden. Es gibt natürlich auch das Gegenstück zu *UploadFromStream* in den man den Blob mit *DownloadFromStream* wieder retour bekommt. Es gibt auch die Möglichkeit falls man eine Liste aller vorhandenen Blobs haben will diese mit

```
foreach (IListBlobItem item in container.ListBlobs(null, true)) { ... }
```

zu iterieren und so zu einer vollständige Liste aller Blobs im Container zu kommen.

Eines ist dabei ganz wichtig bei diesen Operationen. Diese sollten auf keinen Fall eine vorhanden GUI blockieren und sind daher immer in Kombination mit *async* und *await* zu verwenden. Wie dies im Detail funktioniert wird im praktischen Teil besprochen.

## 4 Aufbau des Photos Azure Client Backend

Um die Fotos nun in unseren *BlobStorage* zu speichern wurden zwei neue Klassen implementiert. Eine um die Verbindung zum Storage Account aufzubauen und die zweite um den Datentransfer zwischen Client und Server zu handeln. Da leider keine Daten mit den

Typen *ImageSource* in *Blobs* zu transferieren wurde die Klasse *UserPicture* um eine binäre Repräsentation des Bildes erweitert. Da diese ohne weiteres in *Blobs* gespeichert werden können. Alle weiteren relevanten Daten für das Bild für über Metadaten hinzugefügt.

## 4.1 Serviceklassen

Um den Azure Storage anzusprechen wurden zwei Klassen generiert. Einerseits die Klasse *AzureConnection* die einfach die Verbindung zu Azure aufbauen sollte andererseits die Klasse *AzureRepo*, die wiederum den ganzen Datenverkehr abhandeln sollte. Im *AzureRepo* gab es daher vier Methoden.

- *AddComment*
- *AddPicture*
- *getPictureBlob*
- *getCommentBlob*

Die ersten beiden waren für den Upload der jeweiligen Daten zuständig. Dabei war es wichtig alle Methoden mit dem Keyword *async* zu definieren. Nur so wurde es ermöglicht, dass die aufrufende Methode nicht blockiert wurde. Die Methode *AddPicture* wurde zum Beispiel im *DependencyService* Teil aufgerufen die Lambda Expression musste daher als *async* markiert werden.

```
// async lambda expression because of blob storage
Device.BeginInvokeOnMainThread(async () =>
{
```

Damit konnte man mittels

```
await ar.AddPicture(up);
```

Somit wurde das Picture in den Blob Storage geladen. Die Methode *AddPicture* selber war als *async* definiert und sieht folgendermaßen aus.

```
public async Task AddPicture(UserPicture pic)
{
    CloudBlobClient blobClient = _storageAccount.CreateCloudBlobClient();
    CloudBlobContainer container = blobClient.GetContainerReference("xamarinphotocontainer"
);
    await container.CreateIfNotExistsAsync();
    CloudBlockBlob blockBlob = container.GetBlockBlobReference("photos_blob" + pic.Id);
    blockBlob.Metadata.Add("Id", pic.Id);
```

```

        await blockBlob.UploadFromByteArrayAsync(pic.BytePicture, 0, pic.BytePicture.Length);
    }

```

Hier wurde wie schon im theoretischen Teil besprochen der Blob angelegt und die Daten transferiert. Wichtig dabei zu erwähnen ist, dass der Blob eindeutig sein muss, damit er nicht immer wieder überschrieben wird. Das wurde gewährleistet durch Verwendung der PictureId. Zusätzlich wurde der Blob auch mit Metadaten angereichert um später wieder die richtige PictureId zurückzubekommen. Der Upload selber geschieht wieder mit der await Direktive, damit der Main Thread nicht blockiert werden kann.

Ähnlich gestaltet sich die AddComment Methode mit dem Unterschied dass hier ein eigener Container genommen wurde um die Daten nachher leichter auszulesen.

Wie geschah nun das Laden der Daten. Anhand der *getCommentBlob()* Methode soll dies nun dargestellt werden.

```

public async Task getCommentBlob(PhotosPage page)
{
    CloudBlobClient blobClient = _storageAccount.CreateCloudBlobClient();
    CloudBlobContainer container = blobClient.GetContainerReference("xamarincommentcontainer");
    await container.CreateIfNotExistsAsync();
    BlobContinuationToken token = null;
    var s = await container.ListBlobsSegmentedAsync(token);
    PictureComment pc;
    foreach (IListBlobItem item in s.Results)
    {
        if (item.GetType() == typeof(CloudBlockBlob))
        {
            var PictureId = "";
            var Id = "";
            var PictureDate = "";
            CloudBlockBlob blob = (CloudBlockBlob)item;
            //byte[] blobBytes = new byte[blob.Properties.Length];
            Debug.WriteLine("Block blob of length {0}: {1}", blob.Properties.Length, blob.Uri);
            var comment = await blob.DownloadTextAsync();
            foreach (var metadataItem in blob.Metadata)
            {
                if (metadataItem.Key == "PictureId")

```

```

        {
            PictureId = metadataItem.Value.ToString();
        }
        else if (metadataItem.Key == "Id")
        {
            Id = metadataItem.Value.ToString();
        }
        else if (metadataItem.Key == "PictureDate")
        {
            PictureDate = metadataItem.Value.ToString();
        }
    }
    Debug.WriteLine("Block blob of length {0}: {1}", PictureId, comment)
    pc = new PictureComment(comment,PictureId,Id,PictureDate);
    page.comment.Add(pc);

```

...

Durch definieren des Containers können mit *IListBlobItem* alle Blob Einträge durchiteriert werden und die dazugehörigen Metadaten ausgelesen werden. Da von der MainPage die Referenz übergeben wurde konnte man dort direkt die jeweiligen ObservableCollection mit den korrekten Daten beladen. Damit wurde auch alle View richtig gerendert und dargestellt. Man musste sich nur überlegen wo man diese Methode aufrufen sollte. Da man diese auf keinen Fall im Konstruktor der MainPage aufrufen soll. Vor allem aber kann man den Konstruktor auch nicht als *async* definieren. Somit bot sich die *OnCreate()* Methode in der *App.xaml.cs* an.

```

protected async override void OnStart()
{
    // Handle when your app starts
    AzureRepo ar = new AzureRepo();
    await ar.getPictureBlob(pp);
    await ar.getCommentBlob(pp);
}

```

Diese konnte man als *async* deklarieren und man blockierte damit nicht den Main Thread, da die Daten im Hintergrund geladen wurden und so langsam in die jeweiligen Views geladen wurden.

## 4.2 Administration Blob Container

Im Azure Portal gibt es auch die Möglichkeit sich die jeweiligen Container mit den Daten anzusehen.




NAME	MODIFIED	ACCESS TIER	BLOB TYPE	SIZE	LEASE STATE	
 comments_blob0	3/9/2018, 3:43:06 PM	Hot (Inferred)	Block blob	5 B	Available	...
 comments_blob1	3/9/2018, 3:41:05 PM	Hot (Inferred)	Block blob	5 B	Available	...
 comments_blob2	3/9/2018, 6:08:03 PM	Hot (Inferred)	Block blob	9 B	Available	...

Abbildung 13: Übersicht Blobs

Man sieht dort sehr gut wie groß der Blob ist, wann er angelegt wurde, aber man kann sich die Daten bzw. die Metadaten auch selber anzeigen lassen oder bearbeiten. Durch anklicken der drei Punkte kommt man zu Menüauswahl und kann dort den Blob editieren oder löschen.

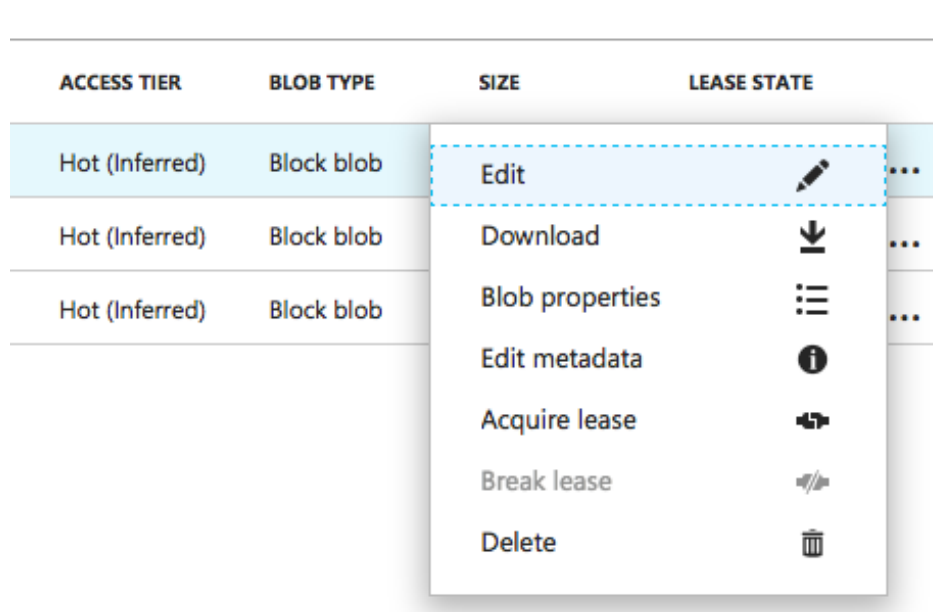


Abbildung 14: Bearbeiten Blobs

## Literaturverzeichnis

- [1] „Ausführliche Erläuterungen zu Xamarin.Forms - Xamarin“. [Online]. Verfügbar unter: <https://developer.xamarin.com/de-de/guides/xamarin-forms/getting-started/hello-xamarin-forms/deepdive/>. [Zugegriffen: 17-Feb-2018].
- [2] M. M. | X. M. | X. C. D. | E. M. F. | X. F. D. | Melbourne und Australia, „Carousel View in Xamarin Forms“, *Xamarin Help*, 18-Apr-2016. [Online]. Verfügbar unter: <https://xamarinhelp.com/carousel-view-xamarin-forms/>. [Zugegriffen: 19-Feb-2018].
- [3] tamram, „Erste Schritte mit Azure Blob Storage (Objektspeicher) mit .NET“. [Online]. Verfügbar unter: <https://docs.microsoft.com/de-de/azure/storage/blobs/storage-dotnet-how-to-use-blobs>. [Zugegriffen: 08-März-2018].
- [4] Martin Sauer, *Grundkurs Mobile Kommunikationssysteme*, 5. Aufl. Wiesbaden: Springer Vieweg, 2013.



# Abbildungsverzeichnis

Abbildung 1: Architektur Xamarin.Forms .....	4
Abbildung 2: Beziehung der Portable Class Libraries .....	5
Abbildung 3: Beziehung Xamarin.Forms zu nativen Plattformen (Quelle [modifiziert]: [1]) .....	6
Abbildung 4: Dependency Service Übersicht .....	12
Abbildung 5: CarouselView .....	16
Abbildung 6: Kommentarfunktion .....	19
Abbildung 7: Blob Storage .....	22
Abbildung 8: Auswahl Storage Account .....	23
Abbildung 9: Anlegen Storage .....	23
Abbildung 10: Dateneingabe Storage .....	24
Abbildung 11: Anlage Container .....	25
Abbildung 12: Schlüssel für Blob Storage .....	26
Abbildung 13: Übersicht Blobs .....	31
Abbildung 14: Bearbeiten Blobs .....	31