**Xamarin Engage Hands-on Lab**

Today, we will be building a cloud connected [Xamarin.Forms](http://xamarin.com/forms) application that will display a list of Speakers at Xamarin Engage and show their details. We will connect it to an Azure Mobile App backend in just a few lines of code.

**Get Started**

Open **Start/Speakers.sln**

This solution contains 4 projects

* Speakers(Portable) - PCL Project that will have all shared code.
* Speakers.Droid - Xamarin.Android application
* Speakers.iOS - Xamarin.iOS application
* Speakers.UWP- Universal Windows Application

The **Speakers** project also has blank code files and XAML pages that we will use during the Hands on Lab.

**NuGet Restore**

All projects have the required NuGet packages already installed, so there will be no need to install additional packages during the Hands on Lab. The first thing that we must do is restore all of the NuGet packages from the internet.

This can be done by **Right-clicking** on the **Solution** and clicking on **Restore NuGet packages...**

**Model**

We will be pulling down information about speakers. Open the **Speakers/Speaker.cs** file and add the following properties inside of the **Speaker** class:

public string ID { get; set; }

public string Name { get; set; }

public string Title { get; set; }

public string Description { get; set; }

**The User Interface**

It is now finally time to build out our first Xamarin.Forms user interface in the **SpeakersPage.xaml**

**SpeakersPage.xaml**

For the first page we will add a few vertically-stacked controls to the page. We can use a StackLayout to do this. Between the ContentPage tags add the following:

<StackLayout Spacing="0">

</StackLayout>

This will be the container where all of the child controls will go. Notice that we specified the children should have no space in between them.

Next, let's add a Button that has a clicked handler and will be executed whenever the user taps the button.

<Button Text="Sync Speakers" Clicked="SyncbuttonCLicked" />

Under the button we can display a loading bar when we are gathering data from the server. We can use an ActivityIndicator to do this and enable and disable it while we are making a call to the server and when we are done:

<ActivityIndicator x:Name="Loader" IsVisible="False" />

We will use a ListView and set the source to the Speakers collection we get from our Azure Call to display all of the items. We can use a special property called *x:Name=""* to name any control:

<ListView x:Name="SpeakersList">

</ListView>

We still need to describe what each item looks like, and to do so, we can use an ItemTemplate that has a DataTemplate with a specific View inside of it. Xamarin.Forms contains a few default Cells that we can use, and we will use the **TextCell** that has two rows of text.

Replace with:

<ListView x:Name="SpeakersList" ItemSelected="SpeakersList\_OnItemSelected">

<ListView.ItemTemplate>

<DataTemplate>

<TextCell Detail="{Binding Title}" Text="{Binding Name}" />

</DataTemplate>

</ListView.ItemTemplate>

</ListView>

The Final Page code would look like this:

<?xml version="1.0" encoding="utf-8"?>

<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"

xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"

x:Class="Speakers.SpeakersPage"

Title="Speakers"

>

<ContentPage.ToolbarItems>

<ToolbarItem Text="+"

Clicked="MenuItem\_OnClicked" />

</ContentPage.ToolbarItems>

<StackLayout Spacing="0">

<Button Text="Sync Speakers" Clicked="SyncbuttonCLicked" />

<ActivityIndicator x:Name="Loader" IsVisible="False" />

<ListView x:Name="SpeakersList" ItemSelected="SpeakersList\_OnItemSelected">

<ListView.ItemTemplate>

<DataTemplate>

<TextCell Detail="{Binding Title}" Text="{Binding Name}" />

</DataTemplate>

</ListView.ItemTemplate>

</ListView>

</StackLayout>

</ContentPage>

**Handle Events in SpeakersPage.xaml.cs**

Now, let's handle the events of the button and set it to call the Speakers list upon click. Let's open up the code-behind for **SpeakersPage.xaml** called **SpeakersPage.xaml.cs**. add the following methods:

using System;

using System.Collections.Generic;

using System.Collections.ObjectModel;

using System.Diagnostics;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

using Xamarin.Forms;

namespace Speakers

{

public partial class SpeakersPage : ContentPage

{

public SpeakersPage()

{

InitializeComponent();

}

private async void SyncbuttonCLicked(object sender, EventArgs e)

{

ObservableCollection<Speaker> speakers = await GetSpeakers();

SpeakersList.ItemsSource = speakers;

}

async Task<ObservableCollection<Speaker>> GetSpeakers()

{

ObservableCollection<Speaker> speakers = new ObservableCollection<Speaker>();

try

{

Loader.IsVisible = true;

Loader.IsRunning = true;

var service = new AzureService();

var items = await service.GetSpeakers();

speakers.Clear();

foreach (var item in items)

speakers.Add(item);

}

catch (Exception ex)

{

Debug.WriteLine("Error: " + ex);

await Application.Current.MainPage.DisplayAlert("Error!", ex.Message, "OK");

}

finally

{

Loader.IsVisible = false;

Loader.IsRunning = false;

}

return speakers;

}

private async void SpeakersList\_OnItemSelected(object sender, SelectedItemChangedEventArgs e)

{

var Speaker = e.SelectedItem as Speaker;

if (Speaker == null)

return;

await Navigation.PushAsync(new DetailsPage(Speaker));

SpeakersList.SelectedItem = null;

}

private async void MenuItem\_OnClicked(object sender, EventArgs e)

{

await Navigation.PushAsync(new AddSpeakerPage(new Speaker()));

}

}

}

**Details**

Now, let's do some navigation and display some Details. In the code-behind for **SpeakersPage.xaml** called **SpeakersPage.xaml.cs**.the following code snippet handles the navigation on the list and goes to the details page:

private async void SpeakersList\_OnItemSelected(object sender, SelectedItemChangedEventArgs e)

{

var Speaker = e.SelectedItem as Speaker;

if (Speaker == null)

return;

await Navigation.PushAsync(new DetailsPage(Speaker));

SpeakersList.SelectedItem = null;

}

In the above code we check to see if the selected item is not null and then use the built in **Navigation** API to push a new page and then deselect the item.

**DetailsPage.xaml**

Let's now fill in the DetailsPage. Similar to the SpeakersPage, we will use a StackLayout, but we will wrap it in a ScrollView in case we have long text.

<ScrollView Padding="10">

<StackLayout Spacing="10">

<!-- Controls would go in here--!>

</StackLayout>

</ScrollView>

Now, let's add controls and bindings for the properties in the Speaker object:

<Label Text="{Binding Name}" FontSize="24"/>

<Label Text="{Binding Title}" TextColor="Purple"/>

<Label Text="{Binding Description}"/>

The Final Code would look like this:

<?xml version="1.0" encoding="utf-8" ?>

<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"

xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"

x:Class="Speakers.DetailsPage"

Title="{Binding Name}"

>

<ScrollView Padding="10">

<StackLayout Spacing="10">

<Label Text="{Binding Name}" FontSize="24"/>

<Label Text="{Binding Title}" TextColor="Purple"/>

<Label Text="{Binding Description}"/>

</StackLayout>

</ScrollView>

</ContentPage>

**Add Page**

Now, let's do some navigation to a page and Add a new Speaker to the Database. In the code-behind for **SpeakersPage.xaml** called **SpeakersPage.xaml.cs**.the following code snippet handles the navigation to the next page from the tooolbar item:

private async void MenuItem\_OnClicked(object sender, EventArgs e)

{

await Navigation.PushAsync(new AddSpeakerPage(new Speaker()));

}

In the above code we receivet the click from the toolbaritem and navigates to the add item page.

**DetailsPage.xaml**

Let's now fill in the AddSpeakerPage. Similar to the SpeakersPage and DetailsPage, we will use a StackLayout. The Final Code would look like this:

<?xml version="1.0" encoding="utf-8"?>

<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"

xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"

x:Class="Speakers.AddSpeakerPage"

Title="Add Speaker">

<StackLayout VerticalOptions="FillAndExpand">

<Entry Placeholder="Speaker Name..." Text="{Binding Name}" />

<Entry Placeholder="Speaker Title..." Text="{Binding Title}" />

<Entry Placeholder="Speaker Description..." Text="{Binding Description}" />

<Button Text="Add Speaker" Clicked="Button\_OnClicked" VerticalOptions="End" />

<ActivityIndicator x:Name="Loader" IsVisible="False" />

</StackLayout>

</ContentPage>

Now to handle the click of the button to add the Speaker, we go to the code behind, **AddSpeakerPage.xaml.cs** and add the code:

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

using Xamarin.Forms;

namespace Speakers

{

public partial class AddSpeakerPage : ContentPage

{

Speaker Speaker;

public AddSpeakerPage(Speaker Speaker)

{

InitializeComponent();

Speaker = Speaker;

BindingContext = Speaker;

}

private async void Button\_OnClicked(object sender, EventArgs e)

{

await AddUser();

await Navigation.PopAsync();

}

//Add a New User

async Task AddUser()

{

try

{

Loader.IsVisible = true;

Loader.IsRunning = true;

var service = new AzureService();

await service.AddSpeaker(Speaker);

}

catch (Exception ex)

{

await Application.Current.MainPage.DisplayAlert("Error", ex.Message, "OK");

}

finally

{

Loader.IsVisible = false;

Loader.IsRunning = false;

}

}

}

}

**Run the App!**

Set the iOS, Android, or UWP project as the startup project and start debugging.

**iOS**

If you are on a PC then you will need to be connected to a macOS device with Xamarin installed to run and debug the app.

If connected, you will see a Green connection status. Select iPhoneSimulator as your target, and then select the Simulator to debug on.

**Android**

Simply set the Speakers.Droid as the startup project and select a simulator to run on. The first compile may take some additional time as Support Packages are downloaded, so please be patient.

**UWP**

Simply set the Speakers.UWP as the startup project and select debug to **UWP Emulator, Device or Local Machine**.