Xamarin Dev Days Hands On Lab

Xamarin を使った Cloud Connected Application

2016 年 10 月

Microsoft

内容

概要

ここでは、Xamarin Dev Day で登壇するスピーカーのリストやその詳細情報を表示するクラウド接続型のアプリケーションを Xamarin.Forms を使って開発します。初めに RESTful エンドポイントから JSON で取得するバックエンドのビジネスロジックをいくつか作ってから、それを Azure Mobile App に数行のコードで接続します。

目的

このラボでは、以下の方法を示します。

- Xamarin を使ってクライアントアプリを生成する
- Azure Mobile Apps を使ってクラウド環境を用意する
- Xamarin から Azure のサービスを呼び出す

システム要件

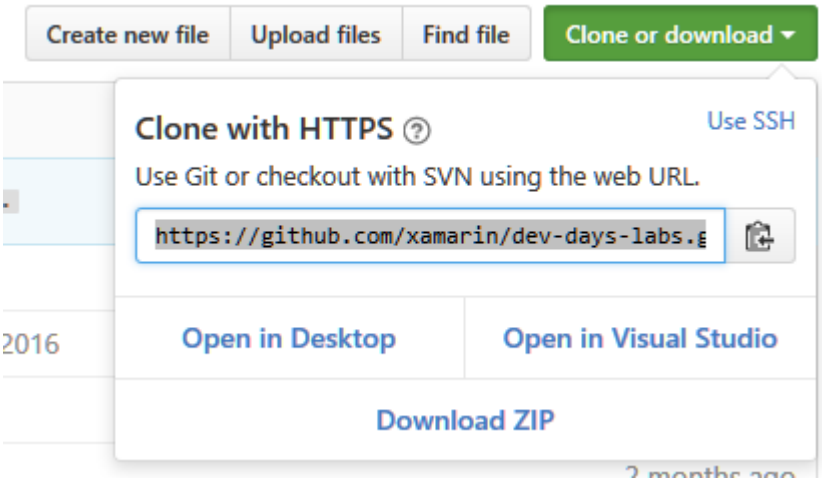
このラボを完了するには、以下の環境やツールが必要です。

- Microsoft Windows 10
- Microsoft Visual Studio 2015 Update 3 および クロスプラットフォーム開発環境
- Microsoft Azure (要アカウント設定)
- Android デバイス（オプショナル）

セットアップ

このラボ用にコンピューターの準備を整えるには、以下の手順を実行する必要があります。

1. Microsoft Windows 10 をインストールします。
2. Microsoft Visual Studio 2015 Update 3 をインストールします。[カスタム] インストールを選択し、[機能の選択] の一覧で [クロスプラットフォーム] を選択します。
3. あらかじめ GitHub への登録を済ませておきます。GitHub の Desktop アプリをインストールしておきます。
4. https://github.com/xamarin/dev-days-labs/tree/2016/HandsOnLab を開き Clone or download から「Open in Desktop」を選択します。



演習

このハンズオン ラボは、以下の演習から構成されています。

1. ―アプリケーションの開発
2. ―追加の宿題

ラボの推定所要時間: **30 ～ 45 分**

演習 1: クライアントアプリの開発

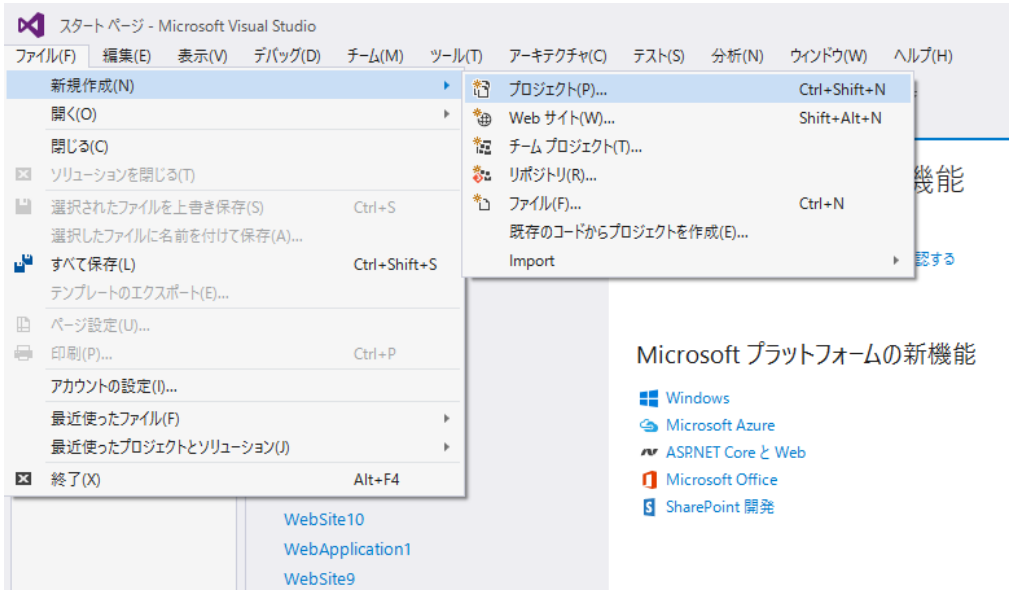
基本的なソリューションはすでにこの Hands on Lab のための GitHub に公開されていますので、準備の手順に従って公開されているソリューションを自分の環境に Clone するか Zip でダウンロードして展開しておきます。

タスク 1 – プロジェクトを開く

初めに Start フォルダにある DevDaysSpeakers.sln を開きます。

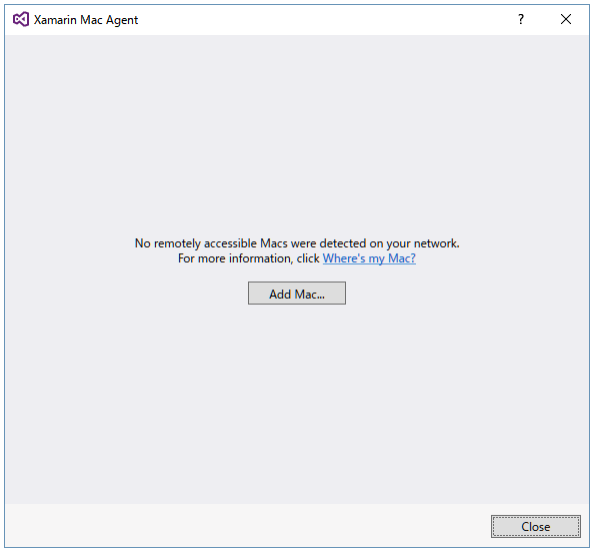
このソリューションは 4 つのプロジェクトで構成されています。

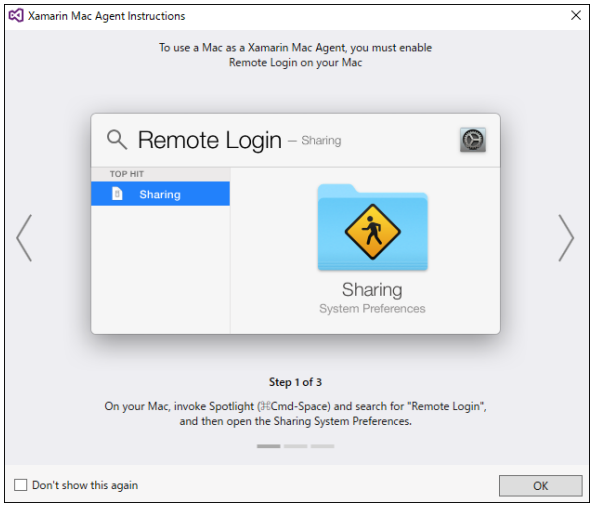
- ✓ DevDaysSpeakers (Portable) – （model, view, view model などの ）共有コードを含む Portable Class Library (PCL)
- ✓ DevDaysSpeakers.Droid - Xamarin.Android アプリケーション
- ✓ DevDaysSpeakers.iOS - Xamarin.iOS アプリケーション
- ✓ DevDaysSpeakers.UWP - Windows 10 UWP アプリケーション
  (Windows 10 と Visual Studio 2015 が必要です)



DevDaysSpeakers (Portable) には、このハンズオンの中で利用するための、空のコードファイルや、XAML のページが含まれています。

このため、もし Mac の環境が無い場合は、Mac Agent のウィンドウが表示されるので、Close または画面右上の×を押して画面を閉じ、次に進みます。

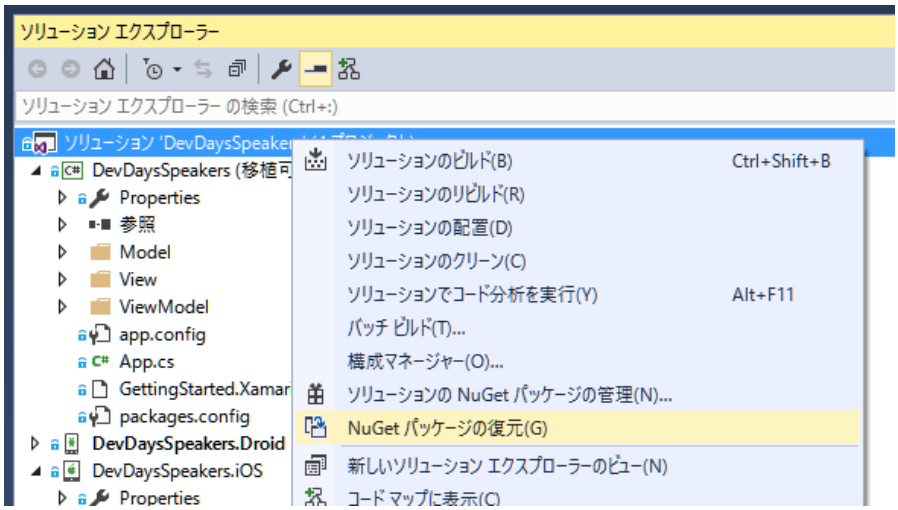




タスク 2 – NuGet Restore

全てのプロジェクトで必要な NuGet パッケージは既にインストール済みですので、このハンズオンの中ではパッケージを追加する必要はありません。そのため、まず初めにやらないければならないことは、インターネットからすべての NuGet パッケージをリストアすることです。

パッケージのリストアは、ソリューションを右クリックし「NuGet パッケージの復元」を選択します。



タスク 3 – Model の実装

スピーカーの情報を取得できるようにするために、Speaker のモデルを実装します。

DevDaysSpeakers プロジェクトの中の Model フォルダにある Speaker.cs ファイルを開きます。開いたら、次のプロパティを **Speaker** クラスの中に追加します。

Speakers.cs

```
public string Id { get; set; }
public string Name { get; set; }
public string Description { get; set; }
public string Website { get; set; }
public string Title { get; set; }
public string Avatar { get; set; }
```

タスク 4 – ViewModel の実装

SpeakersViewModel.cs は Xamarin.Forms のメインとなる View で、データをどのように表示するかのためのすべての機能を提供しています。ここにはスピーカーのリストとサーバーからスピーカーのデータを取得するためのメソッドで構成されています。さらにバックグラウンドタスクでデータを取得するかどうかを示す boolean フラグも含まれています。

InotifyPropertyChanged を実装する

INotifyPropertyChanged は MVVM フレームワークに置いてデータバインディングのための重要インターフェースです。これはモデルつまりオブジェクトの状態が変化すると通知を受け表示させることが出来る機能です。

以下の様に記述し SpeakerViewModel クラスを InotifyPropertyChanged クラスの派生として定義します。

更新前

SpeakersViewMode.cs

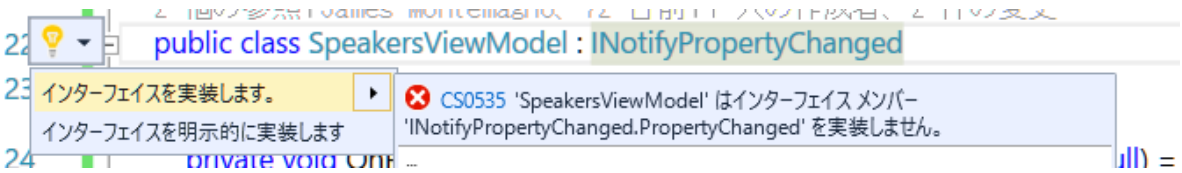
```
public class SpeakersViewModel
{

}
```

更新後

SpeakersViewMode.cs

```
public class SpeakersViewModel : INotifyPropertyChanged
{

}
```

その後、ごクイック操作が表示されたら、「インターフェースを実装します。」を選び、次のコードが追加されることを確認します。



SpeakersViewMode.cs

```
public event PropertyChangedEventHandler PropertyChanged;
```

そして、PropertyChanged イベントが発生すると呼ばれる OnPropertyChanged という名前のヘルパーメソッドの定義を追加します。

SpeakersViewMode.cs / C# 6 (Visual Studio 2015 or Xamarin Studio on Mac)

```
private void OnPropertyChanged([CallerMemberName] string name = null) =>
    PropertyChanged?.Invoke(this, new PropertyChangedEventArgs(name));
```

なお、C# Ver.5 の場合は以下の様に記述します。

SpeakersViewMode.cs / C# 5 (Visual Studio 2012 or 2013)

```
private void OnPropertyChanged([CallerMemberName] string name = null)
{
    var changed = PropertyChanged;

          if (changed == null)
               return;

            changed.Invoke(this, new PropertyChangedEventArgs(name));
}
```

これで、プロパティが変化すると OnPropertyChanged を呼ぶことが出来ます。では続けてプロパティを定義しましょう。

IsBusy プロパティ

では、続けてバッキングフィールドと Boolean プロパティのためのアクセサを作ります。この Boolean 値は、（データを何回も再表示するように）view model が処理中にさらに追加の描画をリクエストしないようにするためのフラグです。

初めにバッキングフィールドを作ります。

SpeakersViewMode.cs

```
private bool busy;
```

続けてプロパティを実装します。

SpeakersViewMode.cs

```
public bool IsBusy
{
    get { return busy; }
    set
    {
        busy = value;
        OnPropertyChanged();
    }
}
```

このように値が変化した場合は常に OnPropertyChanged()を呼び出しています。こうすることで IsBusy が変化すると Xamarin.Forms のバインディング機構によって UI はプロパティの変化を知ることが出来るようになります。

Speaker クラスの ObservableCollection

今回は、読み込まれたスピーカーのリストをクリアし読み込むために ObservableCollection を使います。というのも ObservableCollection にはデータの追加や削除があった時に CollectionChanged イベントが発生する仕組みをはじめから持っているためです。このため OnPropertyChanged を呼ぶ必要はありません。

SpeakerViewModel クラスの、コンストラクタ定義の前に、プロパティを宣言します。

SpeakersViewMode.cs

```
public ObservableCollection<Speaker> Speakers { get; set; }
```

そしてコンストラクタの中に、新しい ObservableCollection のインスタンスを生成するコードを記述します。

SpeakersViewMode.cs

```
public SpeakersViewModel()
{
    Speakers = new ObservableCollection<Speaker>();
}
```

GetSpeakers 関数

ここではインターネットからスピーカーのデータ取得するための、**GetSpeakers** メソッドを実装します。初めにシンプルな HTTP リクエストを実装しますが、あとでこの部分は Azure からデータを取得・同期するように変更します。

まず GetSpeakers という名前の async Task 型のメソッドを作ります。（非同期型のメソッドを作るために Task 型を使います）

SpeakersViewMode.cs

```
private async Task GetSpeakers()
{

}
```

次のコードは、上で定義した関数の中に記述します。はじめに既にデータを取得したかどうかを判定します。

SpeakersViewMode.cs

```
private async Task GetSpeakers()
{
    if(IsBusy)
        return;
}
```

続けて、try/catch/finally ブロックのためのベースを作ります。

SpeakersViewMode.cs

```
private async Task GetSpeakers()
{
    if (IsBusy)
        return;

    Exception error = null;
    try
    {
        IsBusy = true;

    }
    catch (Exception ex)
    {
        error = ex;
    }
    finally
    {
       IsBusy = false;
    }

}
```

ここで、IsBusy を True にセットし、サーバーに接続するときと終了した時には false にセットします。

そして、上記の try ブロックの中で、HttpClient を使ってデータを json 形式で取得します。

SpeakersViewMode.cs

```
using(var client = new HttpClient())
{
    //grab json from server
    var json = await client.GetStringAsync("http://demo4404797.mockable.io/speakers");
}
```

以下は上記の using 区の中に記述します。ここでは Json.NET を使ってスピーカーリストを json で取得して、それをデシリアライズしています。

SpeakersViewMode.cs

```
using(var client = new HttpClient())
{
    //grab json from server
    var json = await client.GetStringAsync("http://demo4404797.mockable.io/speakers");
    var items = JsonConvert.DeserializeObject<List<Speaker>>(json);
}
```

以下も Using 句の中に実装です。スピーカーリストをクリアし、取得したデータを ObservableCollection に追加していきます。

SpeakersViewMode.cs

```
using(var client = new HttpClient())
{
    //grab json from server
    var json = await client.GetStringAsync("http://demo4404797.mockable.io/speakers");
    var items = JsonConvert.DeserializeObject<List<Speaker>>(json);

    Speakers.Clear();
    foreach (var item in items)
        Speakers.Add(item);
}
```

もし何か問題があっても Catch 句によって例外の発生を防ぐことが出来た上、その後 Pop Up アラートを表示することができます。

SpeakersViewMode.cs

```
if (error != null)
    await Application.Current.MainPage.DisplayAlert("Error!", error.Message, "OK");
```

実装したコードは以下の通りです。

SpeakersViewMode.cs

```
    Exception error = null;
    try
    {
        IsBusy = true;

        using(var client = new HttpClient())
        {
            //grab json from server
            var json = await
client.GetStringAsync("http://demo4404797.mockable.io/speakers");

            //Deserialize json
            var items = JsonConvert.DeserializeObject<List<Speaker>>(json);

            //Load speakers into list
            Speakers.Clear();
            foreach (var item in items)
                Speakers.Add(item);
        }
    }
    catch (Exception ex)
    {
        Debug.WriteLine("Error: " + ex);
        error = ex;
    }
    finally
    {
        IsBusy = false;
    }

    if (error != null)
        await Application.Current.MainPage.DisplayAlert("Error!", error.Message, "OK");
}
```

以上で、データを取得するためのメイン関数の実装は完了です。

GetSpeakers コマンド

メソッドを直接呼び出す代わりに、コマンドを使って起動します。コマンドはメソッドを実行するためのインターフェースで、コマンドが有効かどうか記述するオプションを持っています。

初めに GetSpeakersCommand という生のコマンドを定義します。

SpeakersViewMode.cs

```
public Command GetSpeakersCommand { get; set; }
```

SpeakerViewModel のコンストラクタの中で GetSpeakersCommand を作成し次の 2 つのメソッドを渡します。一つはコマンドが実行された時に関数を呼び出し、もう一つはコマンドが有効かどうかを確認しています。どちらのメソッドも下記の様にラムダ式で記述します。

SpeakersViewMode.cs

```
GetSpeakersCommand = new Command(
                async () => await GetSpeakers(),
                () => !IsBusy);
```

唯一実装が必要なことは、IsBusy プロパティがセットされた時、作成した関数が利用可能か再度評価することです。IsBusy をセットすると下記の様に単純に GetSpeakersCommand の ChangeExecute 関数が呼び出されるようにします。

SpeakersViewMode.cs

```
set
{
    busy = value;
    OnPropertyChanged();
    //実行可能かどうか更新
    GetSpeakersCommand.ChangeCanExecute();
}
```

タスク 5 – ユーザー インターフェース の実装

さて、ようやく View フォルダ以下の SpeakersPage.xaml で Xamarin.Forms のユーザーインターフェースを作成します。

SpeakersPage.xaml

アプリの最初のページとして、縦にスタックされるコントロールをページに追加します。これを実現するために StackLayout を使います。ContentPage のタグの間に以下の内容を記述します。

SpeakersPage.xaml

```
<StackLayout Spacing="0">

  </StackLayout>
```

これはほかのコントロールを子要素として配置することが出来るコンテナです。子要素を配置する場合その間隔を取らないように指定(Spacing=0)指定しています。

次に、GetSpeakersCommand と紐づけたボタンを配置します（下記参照）。コマンドはボタンをタップすると Clicked ハンドラの代わりに実行されます。

SpeakersPage.xaml

```
<Button Text="Sync Speakers" Command="{Binding GetSpeakersCommand}"/>
```

ボタンの下にサーバーからデータを取得する際に表示するローディングバーを配置します。ここでは ActivityIndicator（コントロール）を使用し、前に作った IsBusy プロパティと紐づけています。

SpeakersPage.xaml

```
<ActivityIndicator IsRunning="{Binding IsBusy}" IsVisible="{Binding IsBusy}"/>
```

スピーカー コレクション（リスト）を画面に表示するために ListView を使います。x:Name="コントロール名" のようにプロパティ設定することでコントロールに名前を付けることが出来ます。

SpeakersPage.xaml

```
<ListView x:Name="ListViewSpeakers"
              ItemsSource="{Binding Speakers}">
        <!--Add ItemTemplate Here-->
</ListView>
```

続けて、それぞれのアイテム（スピーカーの情報）がどのように表示されるべきかを記述する必要があります。そのために、表示方法を定義する DataTemplate を含む ItemTemplate を使用します。Xamarin.Forms にはいくつかの既定の Cell が用意されていますが、ここでは画像と 2 つのテキストを行表示できる ImageCell を使います。

上記のコードの <!--Add ItemTemplate Here--> を次のコードで置き換えて下さい。

SpeakersPage.xaml

```
<ListView.ItemTemplate>
    <DataTemplate>
        <ImageCell Text="{Binding Name}"
                   Detail="{Binding Title}"
                   ImageSource="{Binding Avatar}"/>
    </DataTemplate>
</ListView.ItemTemplate>
```

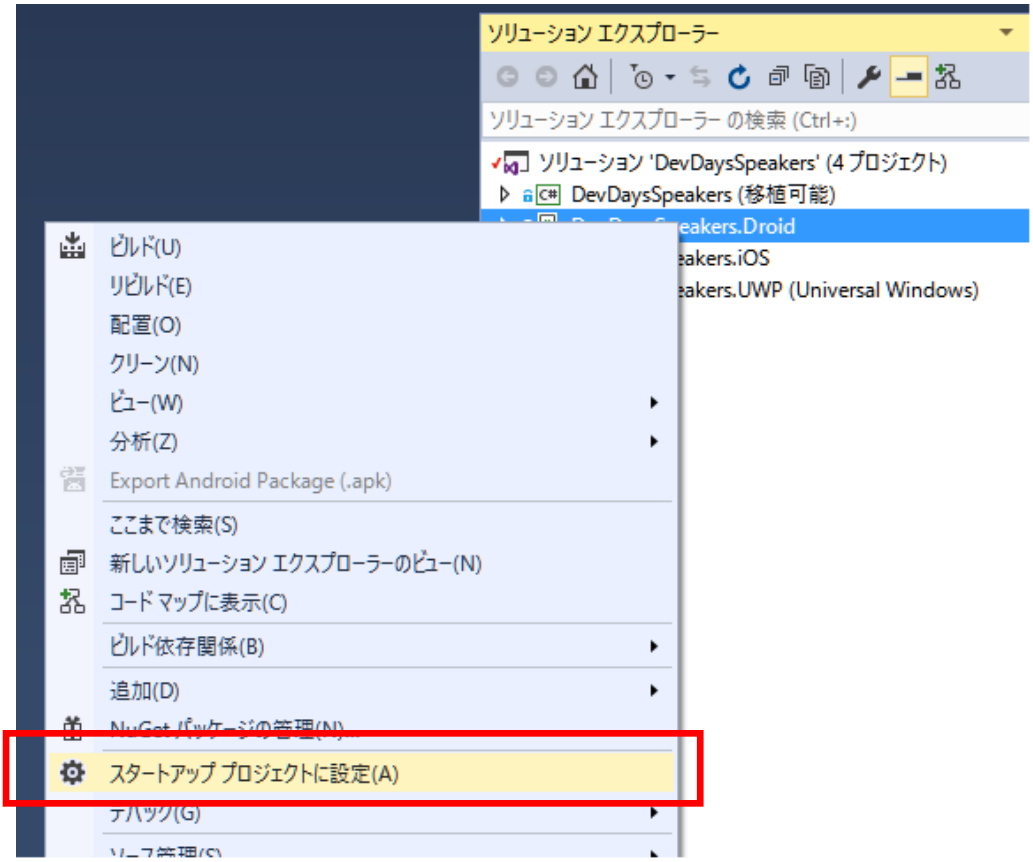
Xamarin.Forms はサーバーの画像を自動的にダウンロード、キャッシュした後に画面表示します。

App.cs の確認

さて、App.cs ファイルを開いてみると、そのコンストラクタである App()があり、ここがアプリケーションの最初のエントリーポイントとなっていることがわかります。この中では SpeakersPage を作成し、それをナビゲーション ページでラップすることで、タイトルバーを生成しています。

アプリを実行してみる

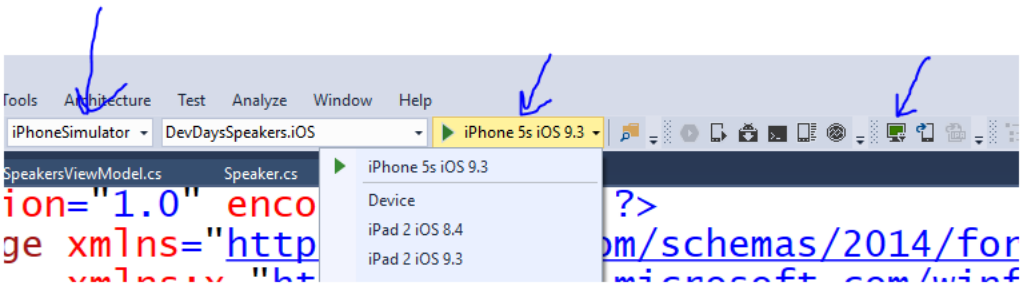
iOS, Android, あるいは UWP（Windows 10/Visual Studio 2015 のみ）のいずれかのプロジェクトをスタートアッププロジェクトとして指定しデバッグを開始します。



iOS

Windows PC を使っている場合、iOS アプリの実行やデバッグを行うためには、Xamarin がインストールされた macOS デバイスに接続する必要があります。

macOS に正しく接続されている場合、画面上部の接続状態は緑になります。デバッグするターゲットデバイスとして iPhoneSimulator を選択し、デバッグを行うシミュレーターの種類を選択します。



Android

DevDaysSpeakers.Droid をスタートアッププロジェクトとして選択し、実行するためのシミュレーターを選択します。最初のコンパイルはサポートパッケージをダウンロードするため少し時間がかかると思いますのでご注意ください。

もし、プロジェクトをビルドした際に、以下のようなエラーがが表示された場合：

> **aapt.exe exited with code** or **Unsupported major.minor version 52**

JDK が正しくセットアップされていないか、新しいビルドツールが既にインストールされている可能性がありますので、詳しくは以下のサポート情報をご覧ください。

> https://releases.xamarin.com/technical-bulletin-android-sdk-build-tools-24/

加えて、Jame's blog も参考画面としてご覧ください。

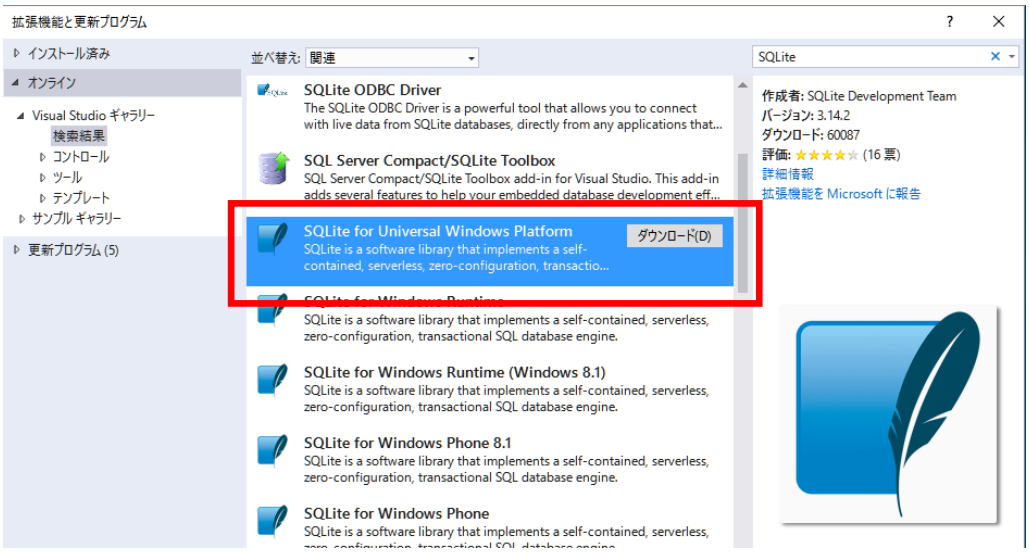
> http://motzcod.es/post/149717060272/fix-for-unsupported-majorminor-version-520

Windows 10

UWP アプリ様に SQLite がインストールされていることを確認してください。

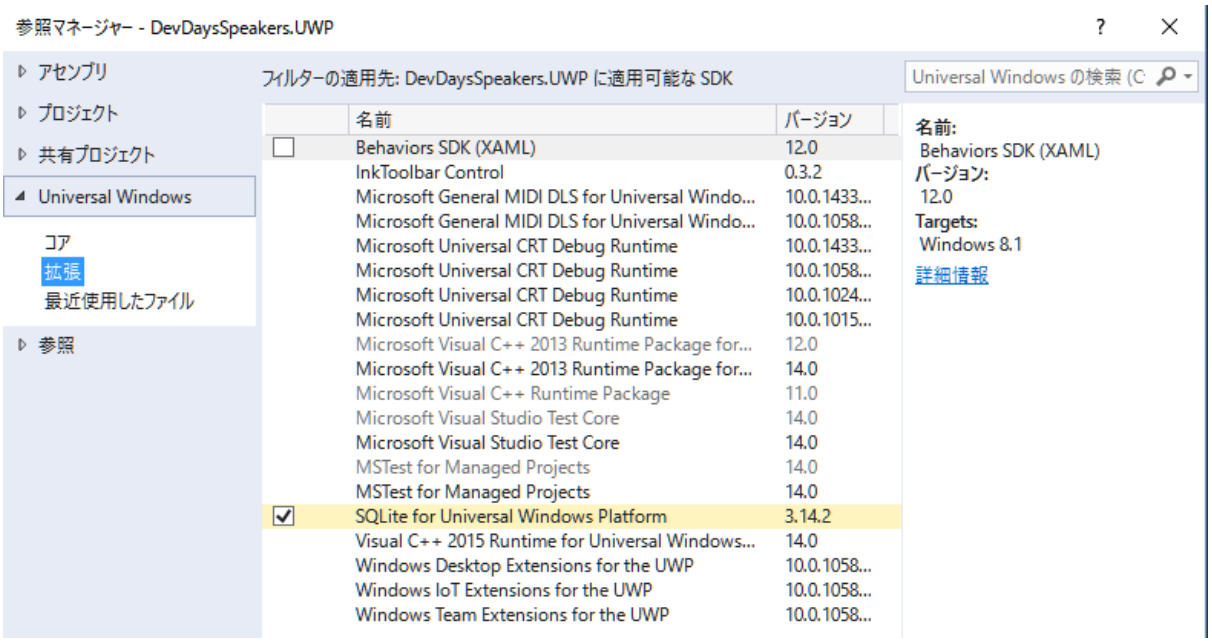
> ツール > 拡張機能と更新プログラム・・・

左のタブはオンラインを選択し、右上の検索項目に SQLite を入力してオンライン検索すると、「SQLite for Universal Windows Platform 」が既にインストールされていることがわかります。（最新バージョンは 3.14.1）



尚、新しいバージョンがあった場合はインストールすると、UWP アプリの参照から SQlite for Universal Windows Platform が削除されます。

その場合は、プロジェクト→参照の追加を指定し、表示されたダイアログから、Universl Windows → 拡張を選択して、SQlite for Universal Windows Platform を確認、追加すれば正しく動くようになるはずです。



デバッグを行う場合は DevDaysSpeakers.UWP をスタートアッププロジェクトとして指定し、ローカル コンピューターをデバッグターゲットとして選択します。

タスク 6 – 詳細機能の実装

ここでは、ナビゲーションと詳細情報表示の機能を実装します。SpeakersPage.xaml のコードビハインドファイルである SpeakersPage.xaml.cs を表示してください。

ItemSelected イベント

コードビハインドファイルを見てみると SpeakerViewModel が既に設定されていることがわかります。**BindingContext = vm;** に続けて、リストビューのアイテムをいずれかを選択した時に呼ばれるイベントを LiveViewSpeakers に追加します。

SpeakersPage.xaml.cs

```
vm = new SpeakersViewModel();
BindingContext = vm;
ListViewSpeakers.ItemSelected += ListViewSpeakers_ItemSelected;
```

DetailsPage に画面遷移するための関数を実装します。

SpeakersPage.xaml.cs

```
 private async void ListViewSpeakers_ItemSelected(object sender,
SelectedItemChangedEventArgs e)
 {
     var speaker = e.SelectedItem as Speaker;
     if (speaker == null)
         return;

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

     ListViewSpeakers.SelectedItem = null;
 }
```

上記のコードでは選択されたアイテムが Null でないかどうかを確認し、それから Navigation API を使って新しいページのインスタンスを作成して Push してから、リスト選択を解除します。

DetailsPage.xaml

DetailsPage を作ります。SpeakersPage と同様に、StackLayout を使いますが、長文に対応するために ScrollView でラップしています。

DetailsPage.xaml

```
<ScrollView Padding="10">
    <StackLayout Spacing="10">
     <!-- Detail controls here -->
    </StackLayout>
  </ScrollView>
```

ここでいくつかのコントロールを追加し、Speaker オブジェクトの各プロパティと紐づけします。

DetailsPage.xaml

```
<Image Source="{Binding Avatar}" HeightRequest="200" WidthRequest="200"/>

<Label Text="{Binding Name}" FontSize="24"/>
<Label Text="{Binding Title}" TextColor="Purple"/>
<Label Text="{Binding Description}"/>
```

更に 2 つのボタンを追加します。それぞれには名前を付け、コードビハインド側の関数を Clicked イベントハンドラーに指定します。

DetailsPage.xaml

```
<Button Text="Speak" x:Name="ButtonSpeak"/>
<Button Text="Go to Website" x:Name="ButtonWebsite"/>
```

Text to Speech（音声読み上げ）

DetailsPage.xaml.cs を開いたら、いくつかの Click ハンドラを追加します。ButtonSpeak では Text To Speech プラグイン（https://github.com/jamesmontemagno/TextToSpeechPlugin）を使ってスピーカーの詳細情報を読み上げます。

コンストラクタの中で、BindingContext の下に Clicked ハンドラーを追加します。

DetailsPage.xaml.cs

```
BindingContext = this.speaker;
ButtonSpeak.Clicked += ButtonSpeak_Clicked;
```

そして、Clicked ハンドラーを追加し、Text to Speech のためのクロスプラットフォーム対応 API を呼んでいます。

DetailsPage.xaml.cs

```
private void ButtonSpeak_Clicked(object sender, EventArgs e)
{
    CrossTextToSpeech.Current.Speak(this.speaker.Description);
}
```

Web サイトの表示

Xamarin.Forms にはクロスプラットフォームに対応したいくつかの便利な API が用意されています。その一つが URL を指定して標準のブラウザを開くための API です。

ではもう一方 ButtonWebsite の Clicked ハンドラを、前と同じようにコンストラクタの中に記述します。

DetailsPage.xaml.cs

```
ButtonWebsite.Clicked += ButtonWebsite_Clicked;
```

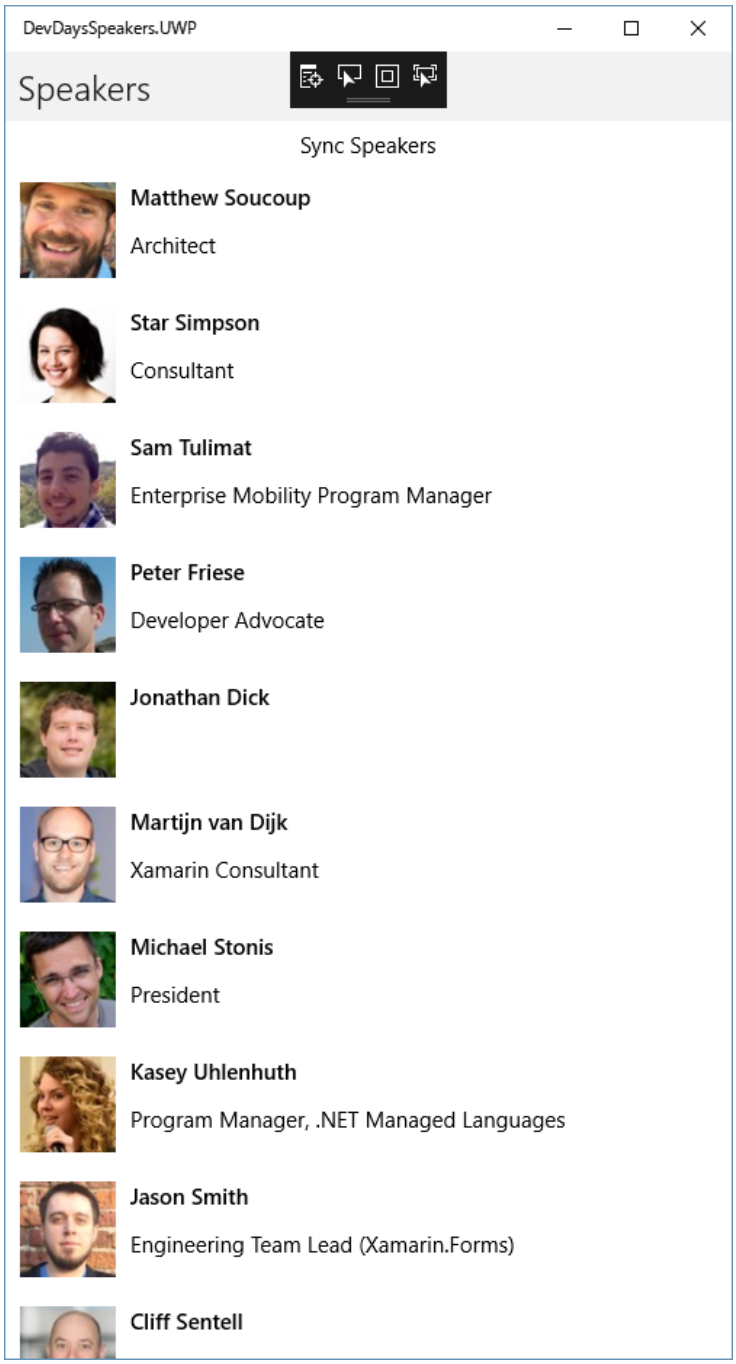
OpenUri メソッドを呼ぶために Device クラスを利用します。

DetailsPage.xaml.cs

```
private void ButtonWebsite_Clicked(object sender, EventArgs e)
{
    if (speaker.Website.StartsWith("http"))
        Device.OpenUri(new Uri(speaker.Website));
}
```

コンパイルして実行

では、ここでいったんコンパイルして実行してみましょう。



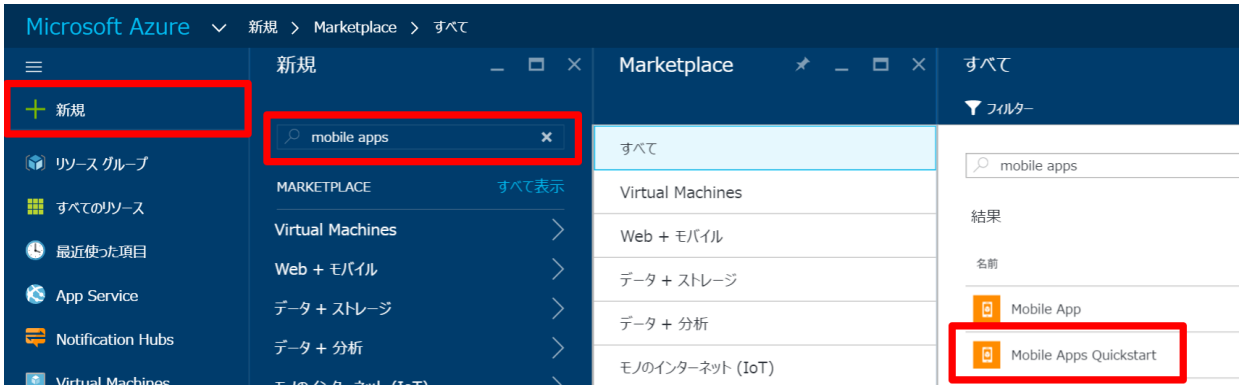
演習 2: Azure Mobile Apps への接続

ここまで実装したような、RESTful エンドポイントからデータを取得できることは重要です。しかし、フル機能のバックエンドについてはどうでしょう？そこで Azure Mobile Apps の登場です。ここまで作ってきたアプリケーションを、こんどは Azure Mobile Apps バックエンドを使うようにアップグレードしていきましょう。

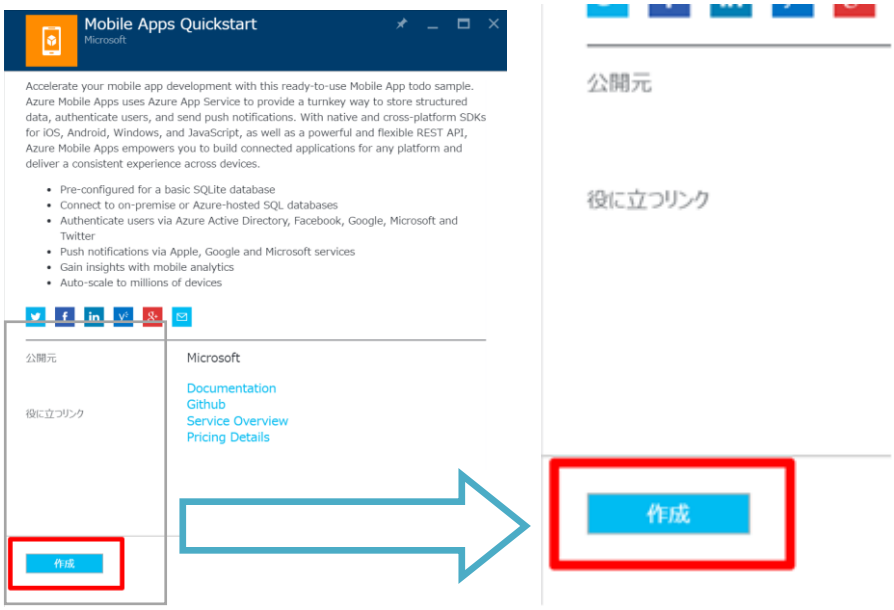
なお、事前に http://portal.azure.com にアクセスして、アカウントを取得しておいてください。

タスク 1 - Azure Mobile Apps の設定

Microsoft Azure のポータルサイトにアクセスしてあら、＋**新規** を選択し、表示された **New** 列で **mobile Apps** を検索してみてください。結果が表示されたら **Mobile Apps Quickstart** を選択します。



Quickstart の項目が表示されるので、一番下にある **作成** をクリックします。



続けて、4 つの設定項目を含むページが表示されます。

アプリ名

アプリのバックエンドを設定するために必要な一意のアプリ名です。グローバルで一意の名前が必要です。(例 yourlastnamespeakers)

サブスクリプション

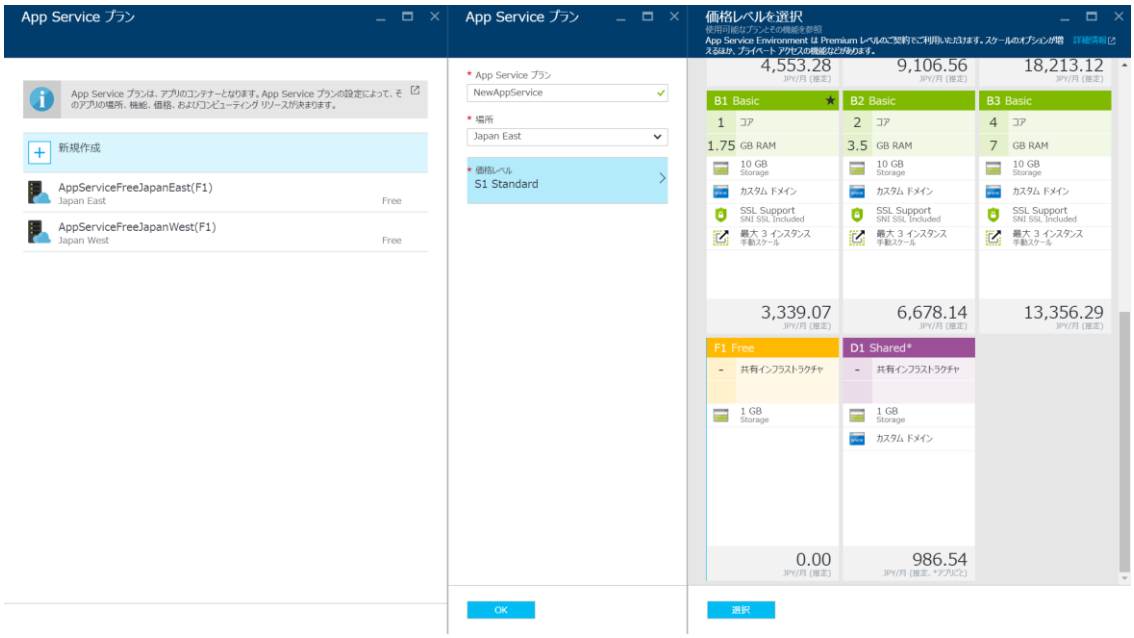
サブスクリプションか従量課金のアカウントを選択します。(このサービスでは費用は発生しません)

リソースグループ

新規作成 を選択して DevDaysSpeakers と入力します。
リソースグループは後で関連するサービスを簡単にまとめて削除するためのグループです。

App Service プラン/場所

ここにある **新規作成** をクリックし、一意の名前を付けます。続けてサーバーの場所を選択（通常は自分から近い場所を選択します）します。最後に価格レベルから F1 Free を選択します。



最後に、**ダッシュボードにピン留めする** をチェックして、**作成** ボタンをクリックして完了です。

ここからセットアップが完了するまで、3 ～5 分ほどかかりますので、いったんコードの方に戻りましょう。

タスク 2 - アプリケーションの対応

App.cs の更新

アプリに Azure App Service Helpers library のコードを 4 行だけ加えて Azure バックエンドを追加します。

DevDaysSpeakers /App.cs ファイルの Azure Client のコンストラクターの上に Static プロパティを追加します。

App.cs

```
public static IEasyMobileServiceClient AzureClient { get; set; }
```

コンストラクタの中で、クライアントを作成してテーブルを登録するための以下のコードを追加します。

App.cs

```
AzureClient = EasyMobileServiceClient.Create();
AzureClient.Initialize("https://YOUR-APP-NAME-HERE.azurewebsites.net");
AzureClient.RegisterTable<Model.Speaker>();
AzureClient.FinalizeSchema();
```

YOUR-APP-NAME-HERE のところはあなたのアプリ名に置き換えてください。

SpeakersViewModel.cs の更新

ViewModel で、テーブルを参照するための別のプライベートプロパティを追加します。コンストラクタの上に以下のコードを追加します。

SpeakersViewModel.cs

```
ITableDataStore<Speaker> table;
```

コンストラクタの中で、スタティックな AzureClient プロパティを使ってテーブルを取得します。

SpeakersViewModel.cs

```
table = App.AzureClient.Table<Speaker>();
```

async Task GetSpeakers() の更新

ここで、文字列を取得するために HttpClient の代わりに、テーブルのクエリを使います。

try ブロックのコードを以下のように変更します。

SpeakersViewModel.cs

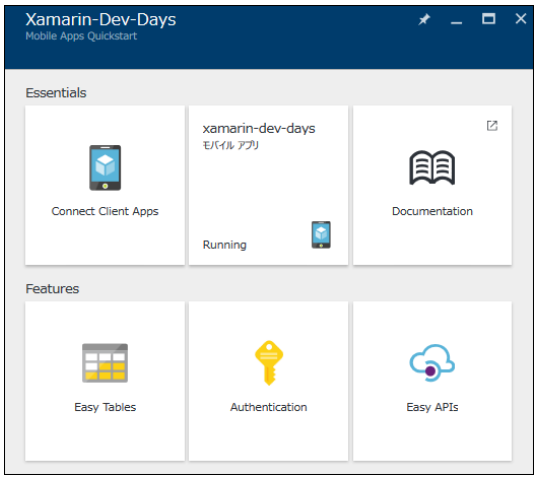
```
 try
{
    IsBusy = true;

    var items = await table.GetItemsAsync();

    Speakers.Clear();
    foreach (var item in items)
        Speakers.Add(item);
}
```

以上で、アプリで必要なコードの実装がすべて終了しました。たった 7 行のコードを追加するだけで、App Service Helper が自動的に Azure のバックエンドとの通信やオンライン・オフラインでの同期を管理するので、日接続状態でもアプリは動作しますしデータの整合性の問題も解決してくれるのです。

データベースの作成

では、Azure ポータルに戻ってデータベースを確認してみましょう。

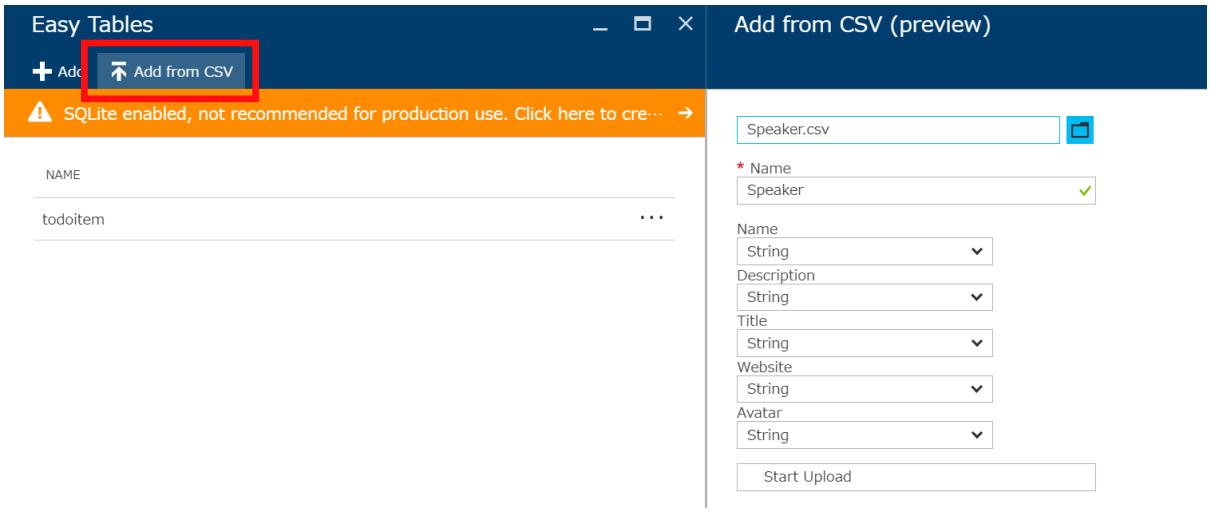
Quickstart が終了していれば、以下の画面が表示されているはずです。なければダッシュボードのピンをタップして表示することができます。



Features の中の **Easy Tables** を選択します。

作成された TodoItem が見えると思いますが、ここでは新しいテーブルを作成して、デフォルトのデータセットを メニューにある **Add from CSV** を選んでデータをアップロードします。

このレポジトリをダウンロードして、Speakers.csv ファイルがフォルダにあることを確認して下さい。ファイルを選択すると、新しいテーブル名が追加されフィールドを検索して追加してくれます。その後 **Start Upload** をクリックします。



では、アプリを再度実行してみます。Azure からデータを取得できていれば成功です。

課題

更に 2 つの課題にチャレンジしてみましょう。

課題 1 – Cognitive Services

Cognitive Serivce Emotion API を追加して、詳細ページにスピーカーの写真から幸福度を測るボタンを追加してみます。

http://microsoft.com/cognitive から、新しいアカウントを作成し Emotion Service の API キーを取得します。

実装手順

1) **Microsoft.ProjectOxford.Emotion** nuget パッケージを全てのプロジェクトに追加します

2) EmotionService クラスを以下の様に追加します。(GetHappinessAsync 関数内の API キーは取得したものに変更してください)

DetailsPage.xaml.cs

```
public class EmotionService
{
    private static async Task<Emotion[]> GetHappinessAsync(string url)
    {
        var client = new HttpClient();
        var stream = await client.GetStreamAsync(url);
        var emotionClient = new EmotionServiceClient("INSERT_EMOTION_SERVICE_KEY_HERE");

        var emotionResults = await emotionClient.RecognizeAsync(stream);

        if (emotionResults == null || emotionResults.Count() == 0)
        {
            throw new Exception("Can't detect face");
        }

        return emotionResults;
    }


    //Average happiness calculation in case of multiple people
    public static async Task<float> GetAverageHappinessScoreAsync(string url)
    {
        Emotion[] emotionResults = await GetHappinessAsync(url);
```

```
        float score = 0;
        foreach (var emotionResult in emotionResults)
        {
            score = score + emotionResult.Scores.Happiness;
        }

        return score / emotionResults.Count();
    }

    public static string GetHappinessMessage(float score)
    {
        score = score * 100;
        double result = Math.Round(score, 2);

        if (score >= 50)
            return result + " % :-)";
        else
            return result + "% :-(";
    }
}
```

3) 詳細ページに新しいボタンを追加し、**x:Name="ButtonAnalyze"** と指定します。

4) Clicked のハンドラを追加し、async キーワードを追加します。

5) そして、ハンドラの中で以下のコードを実装します。

DetailsPage.xaml.cs

```
var level = await EmotionService.GetAverageHappinessScoreAsync(this.speaker.Avatar);
```

6) 続けてポップアップアラートを表示するように以下の様にコードを追加します。

DetailsPage.xaml.cs

```
await DisplayAlert("Happiness Level", EmotionService.GetHappinessMessage(level), "OK");
```

課題 2 – Speaker Details の修正機能

この課題では、スピーカーのタイトルを編集可能にします。

DetailsPage.xaml を開き、Title を表示しているラベルを変更します。

DetailsPage.xaml

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

Entry コントロールを以下の様に OneWay のデータバインディングになるように変更し、Name も指定します。

DetailsPage.xaml

```
<Entry Text="{Binding Title, Mode=OneWay}"
             TextColor="Purple"
             x:Name="EntryTitle"/>
```

Go To Website ボタンの下に Save ボタンを追加します。

DetailsPage.xaml

```
<Button Text="Save" x:Name="ButtonSave"/>
```

SpeakersViewModel の更新

SpeakersViewModel を開いて UpdateSpeaker(Speaker speaker) の関数を追加します。ここではスピーカー情報を更新し、同期し、最後に表示をリフレッシュしています。

DetailsPage.xaml.cs

```
 public async Task UpdateSpeaker(Speaker speaker)
{
    await table.UpdateAsync(speaker);
    await table.Sync();
    await GetSpeakers();
}
```

DetailsPage.xaml.cs の更新

DetailPage 内の SpeakerViewModel クラスの情報を DetailPage のコンストラクタ内で渡します。

変更前

DetailsPage.xaml.cs

```
Speaker;
public DetailsPage(Speaker item)
{
    InitializeComponent();
    this.speaker = item;
    ...
}
```

変更後

DetailsPage.xaml.cs

```
Speaker;
SpeakersViewModel vm;
public DetailsPage(Speaker item, SpeakersViewModel viewModel)
{
    InitializeComponent();
    this.speaker = item;
    this.vm = viewModel;
    ...
}
```

の Clicked ハンドラに続けて、ButtonSave のための Clicked ハンドラを追加します。

DetailsPage.xaml.cs

```
ButtonSave.Clicked += ButtonSave_Clicked;
```

ボタンがクリックされたら、スピカー情報を更新・保存し前のページに戻ります。

DetailsPage.xaml.cs

```
private async void ButtonSave_Clicked(object sender, EventArgs e)
{
    speaker.Title = EntryTitle.Text;
    await vm.UpdateSpeaker(speaker);
    await Navigation.PopAsync();
}
```

最期に、SpeakersPage.xaml.cs の中の ListViewSpeakers_ItemSelected が呼ばれた時に、画面遷移する際に ViewModel を渡すように実装します。

SpeakersPage.xaml.cs

```
//Pass in view model now.
await Navigation.PushAsync(new DetailsPage(speaker, vm));
```

以上で処理は完了です。

関連リンク

- Visual Studio - Xamarin クロスプラットフォーム開発
  https://www.microsoft.com/ja-jp/dev/campaign/vs-xamarin.aspx
- Microsoft Azure Mobile Apps のドキュメント
  https://azure.microsoft.com/ja-jp/documentation/services/app-service/mobile/
- 公式 Xamarin ドキュメント(英語)
  https://developer.xamarin.com/guides/
- 無料版「Creating Mobile Apps with Xamarin.Forms」電子書籍 (英語)
  https://developer.xamarin.com/guides/xamarin-forms/creating-mobile-apps-xamarin-forms/


Microsoft