

# Workshop

# Xamarin y Azure

# Contacto

[alejandro3wade@gmail.com](mailto:alejandro3wade@gmail.com)

@LizardWizardTM

Matsu Software



# UNIA y Club.Net

**UNIA (Universitarios Informáticos de Almería)** es una asociación que promueve la **participación** y el **aprendizaje**.



# UNIA y Club.NET

Dentro de UNIA, tenemos varias verticales de actividades, una de ellas es el **Club.NET Almería**.



# UNIA y Club.NET

- \* Webs
- \* Programas
- \* Aplicaciones multiplataforma
- \* Juegos
- \* Servicios en el cloud
- \* IT

# Materiales

**Los materiales que usaremos en el proyecto podéis encontrarlos en:**

<https://github.com/xamarin/dev-days-labs/tree/master/HandsOnLab>

# Laboratorio

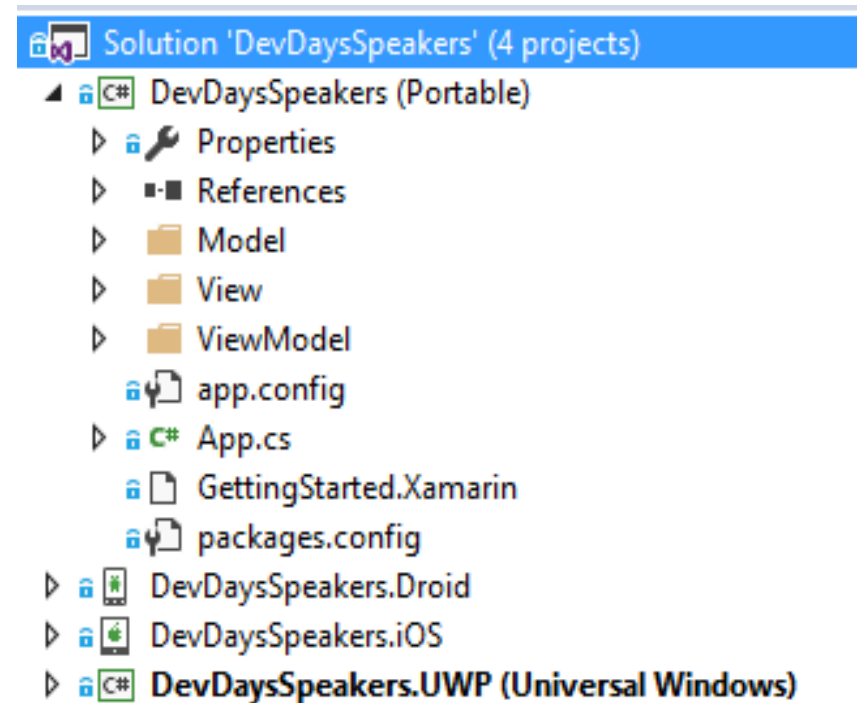
**Vamos a crear una aplicación conectada a la nube que muestre una lista de los conferenciantes de Xamarin Dev Days y sus detalles.**

# Laboratorio

- \* Abrir **Start/DevDaysSpeakers.sln**

- \* **Contiene 4 proyectos:**

- DevDaysSpeakers
- DevDaysSpeakers.Droid
- DevDaysSpeakers.iOS
- DevDaysSpeakers.UWP

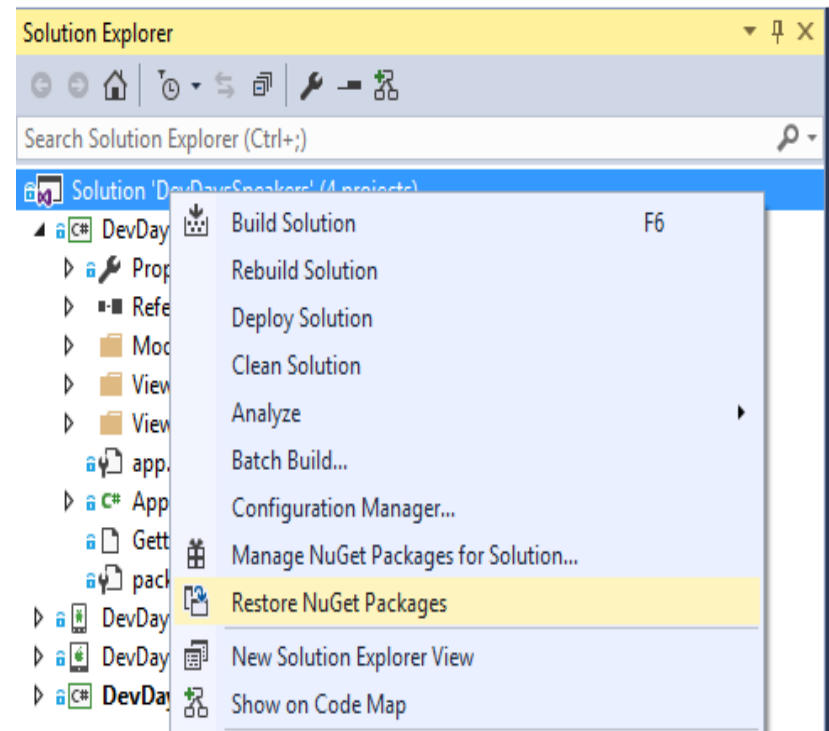




# Laboratorio

Los proyectos ya tienen los **paquetes de NuGet** instalados, por lo que no necesitamos instalar paquetes adicionales. Lo que haremos será restaurar los paquetes de Internet.

Para ello hacemos **click derecho** en la **Solución** y seleccionamos **Restaurar paquetes de NuGet**.



# Laboratorio

Abrimos el archivo **DevDaysSpeakers/Model/Speaker.cs** y añadimos las siguientes propiedades a la clase **Speaker**:

```
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; }
```

# Laboratorio

A continuación editaremos **SpeakersViewModel.cs**:

-Implementamos la interfaz **INotifyPropertyChanged**.

\* **Actualizamos:**

```
public class SpeakersViewModel  
{  
}
```

\* **A:**

```
public class SpeakersViewModel : INotifyPropertyChanged  
{  
}
```

# Laboratorio

Y declaramos el evento **PropertyChanged** con la siguiente línea de código:

```
public event PropertyChangedEventHandler PropertyChanged;
```

Crearemos un método auxiliar llamado **OnPropertyChanged** e invocaremos este método cada vez que cambie una propiedad.

# Laboratorio

\* Si usamos **C# 6 (Visual Studio 2015 o Xamarin Studio en Mac)**:

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

# Laboratorio

\* Si usamos **C# 5 (Visual Studio 2012 o 2013)**:

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

    if (changed == null)
        return;

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

# Laboratorio

- \* Ahora crearemos nuestra primera propiedad:

- \* Primero creamos una booleana:

```
private bool busy;
```

- \* Y a continuación creamos la propiedad:

```
public bool IsBusy
```

```
{
```

```
    get { return busy; }
```

```
    set
```

```
    {
```

```
        busy = value;
```

```
        OnPropertyChanged();
```

```
    }
```

```
}
```

# Laboratorio

Usaremos una **ObservableCollection** que se borrará y después se llenará con los conferenciantes. Gracias a esto no necesitaremos llamar a **OnPropertyChanged** cada vez.

- \* Sobre el constructor de la clase **SpeakersViewModel** declararemos una auto-propiedad:

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



# Laboratorio

- \* Dentro del constructor, creamos una instancia del **ObservableCollection**:

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

# Laboratorio

- \* Ahora estamos preparados para crear un método llamado **GetSpeakers** que recuperará los datos de los conferenciantes de Internet.
- \* En primer lugar, implementaremos esto con una simple solicitud **HTTP**, pero más tarde lo actualizaremos para coger y sincronizar los datos de **Azure**.

# Laboratorio

- \* En primer lugar, creamos un método llamado **GetSpeakers** que es de tipo **async Task**:

```
private async Task GetSpeakers()  
{  
  
}
```

- \* **DENTRO** del método escribiremos lo siguiente:

```
if (IsBusy)  
    return;
```

# Laboratorio

- \* A continuación añadimos un bloque **try/catch/finally** (SEGUIMOS DENTRO DEL MÉTODO):

```
Exception error = null;  
try {  
    IsBusy = true;  
}  
catch (Exception ex) {  
    error = ex;  
}  
finally {  
    IsBusy = false;  
}
```

# Laboratorio

\* **DENTRO** del bloque **try** usaremos **HttpClient**:

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

\* **DENTRO** del **using**, **deserializamos** el **json** y lo convertimos en una lista de conferenciantes con **Json.NET**:

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

# Laboratorio

- \* Seguimos **DENTRO** del **using**. Limpiaremos los conferenciantes y luego los cargaremos en el **ObservableCollection**:

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

# Laboratorio

- \* **DESPUÉS** del bloque **finally** podemos incluir una alerta por si algo fuera mal:

```
if (error != null)
```

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

¡Ya está listo nuestro método principal para obtener datos!

# Laboratorio

- \* En lugar de invocar este método directamente, vamos a usar un **Command**. Primero creamos un nuevo Command llamado **GetSpeakersCommand**:

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



# Laboratorio

- \* En el interior del constructor **SpeakersViewModel**, creamos **GetSpeakersCommand** y pasamos dos métodos: uno para invocar cuando se ejecuta el comando y otro que determina si el comando está habilitado:

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

# Laboratorio

- \* En el **set** de **IsBusy** ahora invocaremos el método **ChangeCanExecute** en el **GetSpeakersCommand** como se muestra a continuación:

```
set {  
    busy = value;  
    OnPropertyChanged();  
    GetSpeakersCommand.ChangeCanExecute();  
}
```

# Laboratorio

- \* Ahora vamos a construir nuestra primera interfaz de usuario **Xamarin.Forms**. Abrimos **View/SpeakersPage.xaml**.
- \* Para la primera página agregaremos unos cuantos controles apilados verticalmente a la página usando un **StackLayout**. Para ello, entre los tags de **ContentPage** añadimos lo siguiente:

```
<StackLayout Spacing="0">
```

```
</StackLayout>
```

# Laboratorio

- \* A continuación, vamos a agregar un botón con un enlace al **GetSpeakersCommand** que hemos creado:

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

- \* Bajo el botón podemos mostrar una barra de carga cuando estamos recolectando datos desde el servidor usando un **ActivityIndicator** y lo vinculamos a la propiedad **IsBusy** que hemos creado:

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

# Laboratorio

- \* Utilizaremos un **ListView** que se vincule a la colección de **Speakers** para mostrar todos los elementos. Podemos usar una propiedad especial llamada **x: Name = ""** para nombrar cualquier control:

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

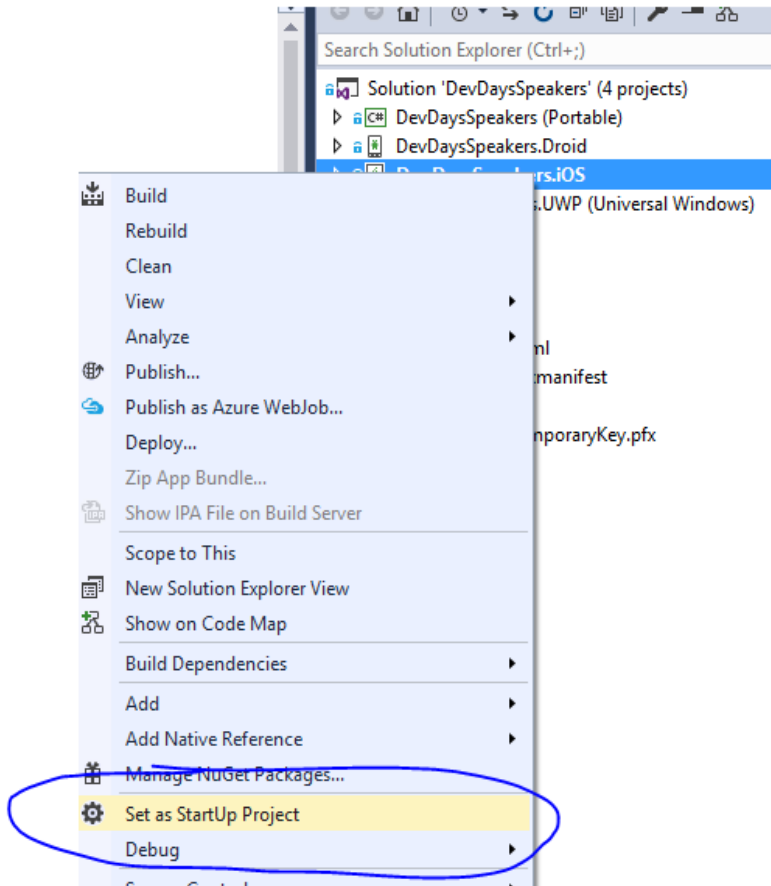
# Laboratorio

\* Reemplazamos <!--Add ItemTemplate Here--> con:

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

# Laboratorio

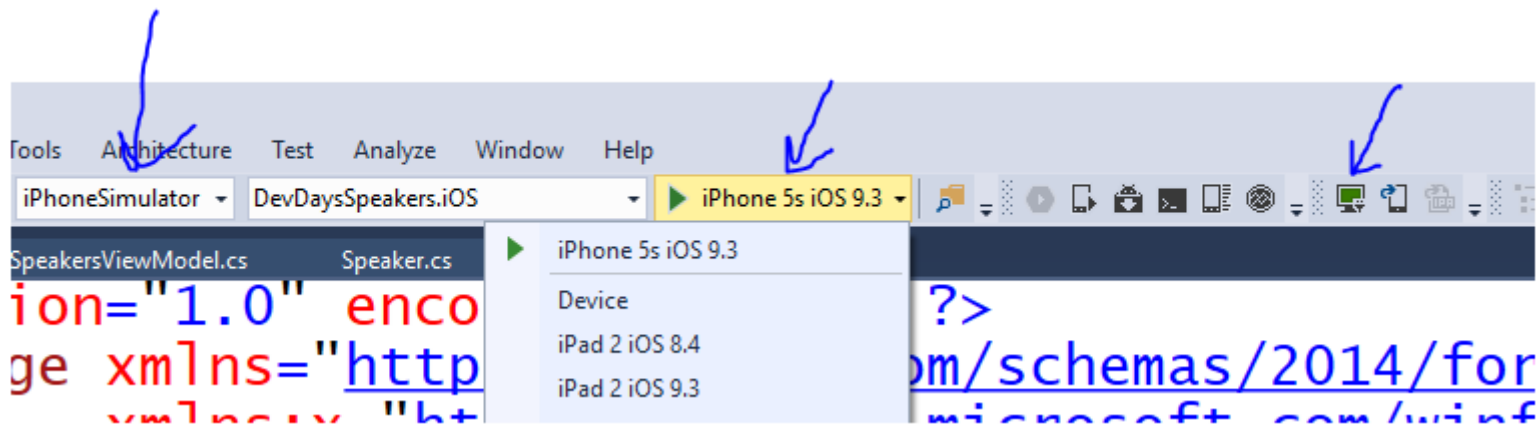
Seleccionamos **iOS, Android, o UWP (Windows/VS2015 solo)** como el proyecto inicial y comenzamos la depuración.



# Laboratorio

## iOS

- \* Si estás en un PC necesitarás estar conectado a un dispositivo macOS con Xamarin instalado para correr y depurar la aplicación.
- \* Si está conectado, verás el icono del estado de conexión verde. Selecciona **iPhoneSimulator** como objetivo y selecciona el simulador donde depurarlo.





# Laboratorio

## Android

- \* Simplemente establece **DevDaysSpeakers.Droid** como el proyecto de inicio y selecciona un simulador donde correrlo.

## Windows 10

- \* Simplemente establece **DevDaysSpeakers.UWP** como el proyecto de inicio y selecciona depurar a **Local Machine**.

# Laboratorio

Ahora abrimos **SpeakersPage.xaml.cs**.

Debajo de **BindingContext = vm** ; vamos a agregar un evento a **ListViewSpeakers** para ser notificado cuando se selecciona un elemento:

```
ListViewSpeakers.ItemSelected += ListViewSpeakers_ItemSelected;
```

# Laboratorio

Implementamos este método para que navegue al **DetailsPage**:

```
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;
}
```

# Laboratorio

Ahora abrimos **DetailsPage.xaml**. De forma similar a la página de **SpeakersPage**, usaremos **StackLayout**, pero lo envolveremos en un **ScrollView** en caso de que tengamos texto largo:

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

# Laboratorio

Ahora, vamos a agregar controles y enlaces para las propiedades del objeto **Speaker**:

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

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

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

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

# Laboratorio

Añadimos dos botones y les ponemos un nombre:

```
<Button Text="Speak" x:Name="ButtonSpeak"/>
```

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

# Laboratorio

Si abrimos **DetailsPage.xaml.cs**, ahora podemos agregar unos cuantos controladores. Comencemos con **ButtonSpeak**, donde usaremos el **Text To Speech Plugin** para leer la descripción del hablante.

En el constructor, agregue un controlador de clic debajo de **BindingContext**:

```
ButtonSpeak.Clicked += ButtonSpeak_Clicked;
```

# Laboratorio

A continuación, podemos agregar el controlador de clic y llamar a la API de varias plataformas para **Text To Speech**:

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



# Laboratorio

Añadimos otro controlador de clic pero esta vez para **ButtonWebsite**:

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

# Laboratorio

A continuación, podemos utilizar la clase **Device** para llamar al método **OpenUri**:

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

¡Ahora, debemos estar listos para compilar y correr como antes!

# Laboratorio

## CONECTAR A AZURE MOBILE APPS

Vamos a actualizar nuestra aplicación para usar un back-end de **Azure Mobile Apps**.

Vaya a **<http://portal.azure.com>** y regístrese para obtener una cuenta.

Una vez en el portal, seguiremos los siguientes pasos...

# Laboratorio

Seleccionamos el botón **+New**, buscamos **mobile apps** y seleccionamos **Mobile Apps Quickstart**

The screenshot shows the Microsoft Azure portal interface. The breadcrumb navigation at the top reads "Microsoft Azure > New > Marketplace > Everything".

**Left Sidebar:** Contains navigation links. The **+ New** button is highlighted with a red box.

**New Pane:** Displays a search bar with "mobile apps" entered, which is also highlighted with a red box. Below the search bar, a list of categories is shown under the heading "MARKETPLACE".


**Marketplace Pane:** Shows a list of categories: Everything, Virtual Machines, Web + Mobile, Data + Storage, Data + Analytics, and Internet of Things.

**Everything Pane:** Displays search results for "mobile apps". The results table is as follows:

NAME	PUBLISHER	CATEGORY
Mobile App	Microsoft	Web + Mobile
Mobile Apps Quickstart	Microsoft	Web + Mobile







The "Mobile Apps Quickstart" entry is highlighted with a red box.

# Laboratorio

**Mobile Apps Quickstart**  
Microsoft

Accelerate your mobile app development with this ready-to-use Mobile App todo sample. Mobile Apps provides a turnkey way to structure storage, authenticate users, and send push notifications. With native and cross-platform SDKs for iOS, Android, Windows, and HTML, as well as a powerful and flexible REST API, Mobile Apps empowers you to build connected applications for any platform and deliver a consistent experience across devices.

- Integrate with SQL, Oracle, SAP, MongoDB, and more.
- Make your app work offline and sync.
- Connect to on-premises data.
- Leverage enterprise single sign-on with Active Directory.
- Integrate with social providers like Facebook, Twitter, and Google.
- Broadcast push notifications across platforms, with customer segmentation.
- Gain insights with mobile analytics.
- Auto-scale to millions of devices.



---

**PUBLISHER**Microsoft

**USEFUL LINKS**[Documentation](#)  
[Github](#)  
[Service Overview](#)  
[Pricing Details](#)

---

Create

Se abrirá la siguiente pestaña de Quickstart. Seleccionamos **Create**.

# Laboratorio

Se abrirá una pestaña de configuración con 4 opciones:

- \* **App name:**

Este es un nombre único para la aplicación que necesitará al configurar el back-end. Tendrá que elegir un nombre globalmente único.

- \* **Subscription:**

Selecciona una suscripción.

- \* **Resource Group:**

Selecciona **Create new** y llámala **DevDaysSpeakers**.

# Laboratorio

- \* **App Service plan/Location:**

Haz click en este campo y selecciona **Create New**, dale un nombre único, selecciona una localización y después selecciona **F1 Free tier**.

# Laboratorio

## App Service plan

## App Service plan

## Choose your pricing tier

Browse the available plans and their features

App Service Environments are available in the Premium tier. They offer even greater scale options, private access, and more. [Learn more](#)



An App Service plan is the container for your app. The App Service plan settings will determine the location, features, cost and compute resources associated with your app.



Create New



Default1(B1)  
West US

1 instances, 8 ap...



Default1(B1)  
East US 2

1 instances, 1 ap...



MontemagnoSpeakersPlan(F1)  
West US

0 instances, 1 ap...



MyShoppeDemo2Plan(B1)  
East US 2

1 instances, 4 ap...



MyShopServicePlan(F1)  
West US

0 instances, 2 ap...



SampleAppsServicePlan(S1)  
West US

1 instances, 1 ap...

\* App Service plan

MyNewPlanName

\* Location

West US

\* Pricing tier

S1 Standard

OK



Traffic Manager  
Geo availability

44.64

USD/MONTH (ESTIMATED)



Traffic Manager  
Geo availability

89.28

USD/MONTH (ESTIMATED)



Traffic Manager  
Geo availability

178.56

USD/MONTH (ESTIMATED)

B1 Basic

1

Core

1.75

GB RAM

10 GB

Storage

Custom domains

Up to 3 instances

Manual scale

B2 Basic

2

Core

3.5

GB RAM

10 GB

Storage

Custom domains

Up to 3 instances

Manual scale

B3 Basic

4

Core

7

GB RAM

10 GB

Storage

Custom domains

Up to 3 instances

Manual scale

32.74

USD/MONTH (ESTIMATED)

65.47

USD/MONTH (ESTIMATED)

130.94

USD/MONTH (ESTIMATED)

F1 Free

-

Shared infrastructure

1 GB

Storage

D1 Shared\*

-

Shared infrastructure

1 GB

Storage

Custom domains

0.00

USD/MONTH (ESTIMATED)

9.67

USD/MONTH (ESTIMATED, \*PER AP...

Select



# Laboratorio

Finalmente marque **Pin to dashboard** y haz click en **Create**:

☒ Pin to dashboard

Create

# Laboratorio

Vamos a usar el **Azure Mobile Apps SDK** para agregar un back-end Azure a nuestra aplicación móvil en sólo unas pocas líneas de código.

En el archivo **DevDaysSpeakers / Services / AzureService.cs** vamos a agregar nuestra **url** al método **Initialize**.

```
var appUrl = "https://TU-NOMBRE-DE-LA-APP.azurewebsites.net";
```

# Laboratorio

## GetSpeakers

En este método, necesitaremos inicializar, sincronizar y consultar la tabla de elementos. Podemos utilizar consultas complejas LINQ para ordenar los resultados:

```
await Initialize();  
await SyncSpeakers();  
return await table.OrderBy(s => s.Name).ToEnumerableAsync();
```

# Laboratorio

## SyncSpeakers

Nuestro back-end de Azure tiene la capacidad de empujar cualquier cambio local y luego tirar de todos los datos más recientes desde el servidor mediante el siguiente código que se puede agregar al **try** dentro del método **SyncSpeakers**:

```
await Client.SyncContext.PushAsync();  
await table.PullAsync("allSpeakers", table.CreateQuery());
```

# Laboratorio

Ahora vamos a actualizar el **SpeakersViewModel.cs**:

Ahora, en lugar de usar el **HttpClient** para obtener un string, vamos a consultar la tabla. Para ello cambiamos el bloque try por:

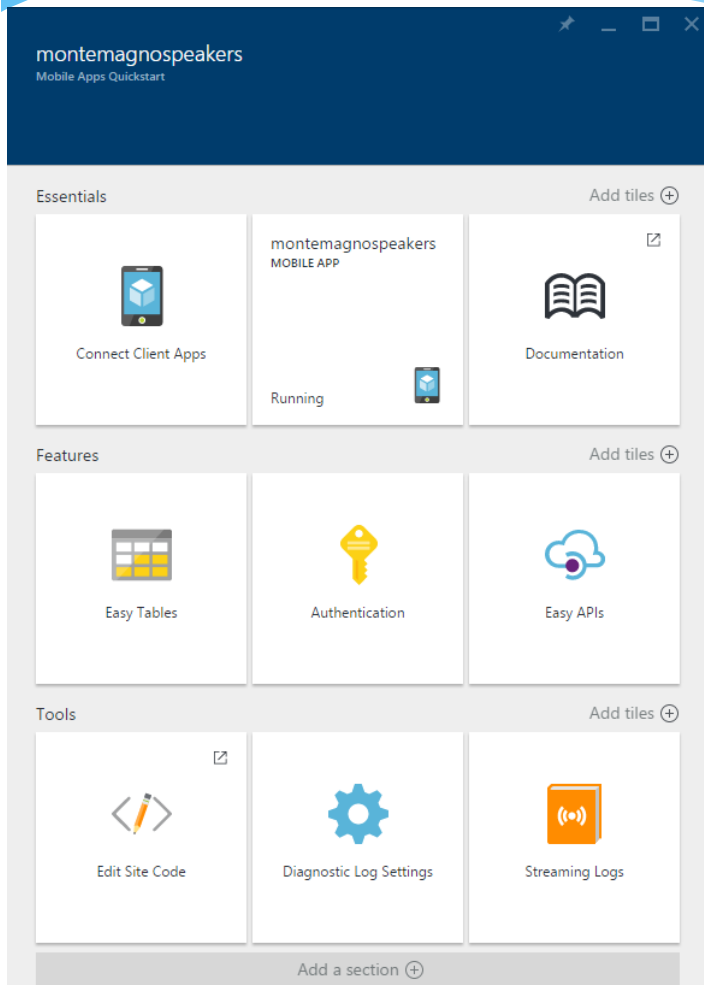
```
try
{
    IsBusy = true;
    var service = DependencyService.Get<AzureService>();
    var items = await service.GetSpeakers();

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

# Laboratorio

**Ya hemos implementado todo el código necesario en nuestra aplicación, ahora volvamos al portal de Azure.**

# Laboratorio



Cuando termine el **Quickstart** debería ver la siguiente pantalla, o puede ir a ella tocando el pin en el **Dashboard**.  
Selecciona bajo **Features** la opción de **Easy Tables**.

# Laboratorio

Se habrá creado un **TodoItem**, que deberías ver, pero podemos crear una nueva tabla y subir un conjunto predeterminado de datos seleccionando **Add from CSV** en el menú.

Asegúrese de que ha descargado este repo y tiene el archivo **Speaker.csv** que se encuentra en esta carpeta.

Seleccione el archivo y agregará un nuevo nombre de tabla y encontrará los campos que hemos enumerado. Luego pulsa **Start Upload**.



# Laboratorio

## Easy Tables



Add



Add from  
CSV

⚠ SQLite enabled, not recommended for production use. [Click here to create a SQL...](#) →

NAME

Speaker

todoitem



## Add from CSV (preview)

Speaker.csv



\* Name

Speaker



Name

String



Description

String



Title

String



Website

String



Avatar

String



Start Upload

# Laboratorio

**¡Ahora puedes volver a correr tu aplicación y coger los datos de Azure!**

# Laboratorio

**GRACIAS POR SU ATENCIÓN**

**URL:** <https://github.com/xamarin/dev-days-labs/tree/master/HandsOnLab>

**alejandro3wade@gmail.com**

**@LizardWizardTM**

**Matsu Software**