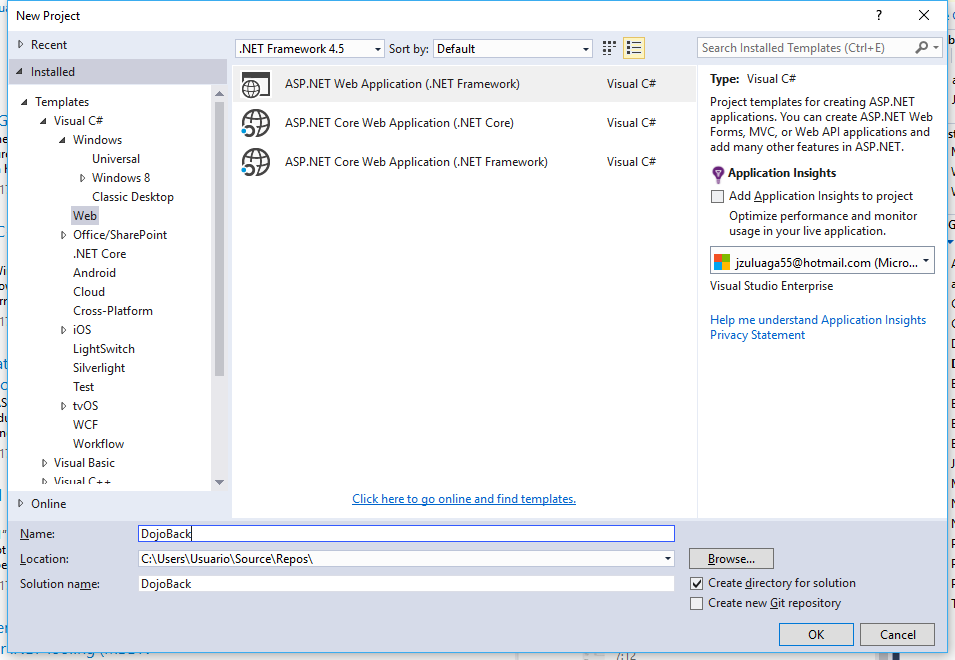
Creating and consuming services from Xamarin Forms App on Azure

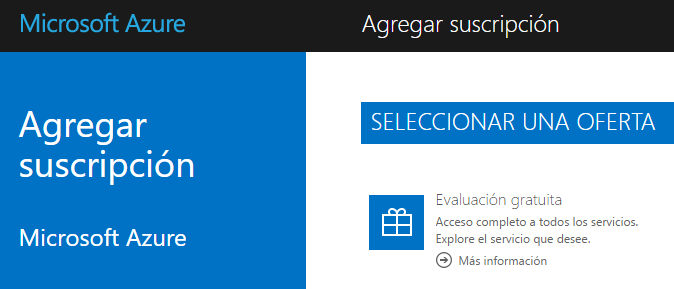
Autor, Juan Carlos Zuluaga

# PARTE I: BACKEND

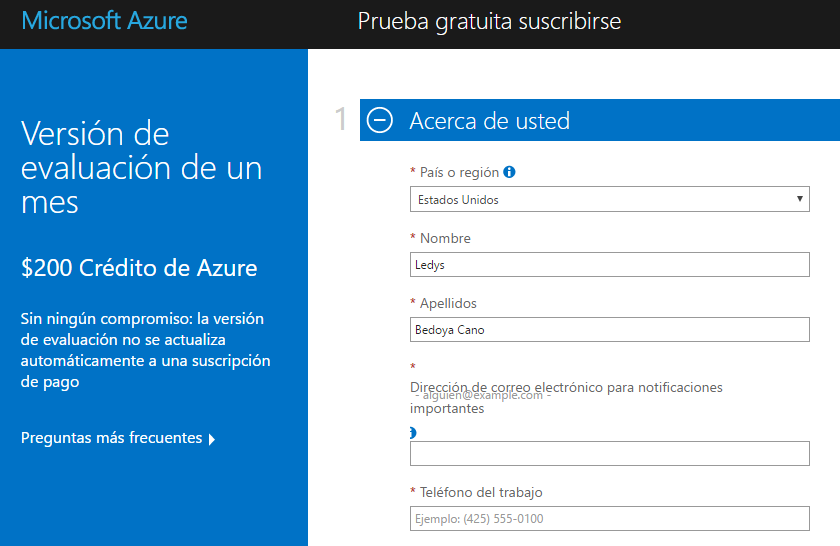
1. Creamos un nuevo proyecto Web, asegúrese que tiene referenciada su cuenta de Azure



Para activar su suscripción de evaluación de Azure, entras a: <https://account.windowsazure.com/signup?showCatalog=True>:

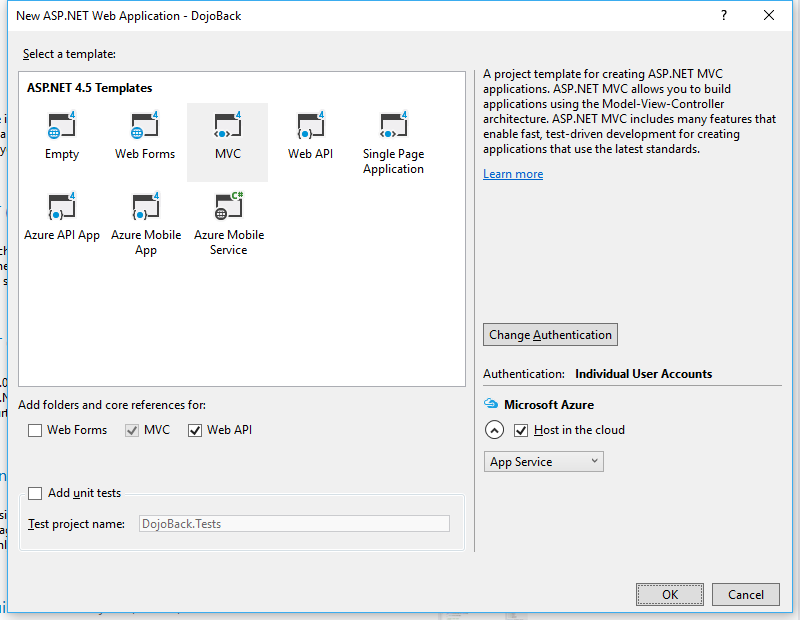


Seleccionas evaluación gratuita y luego llenas este formulario:

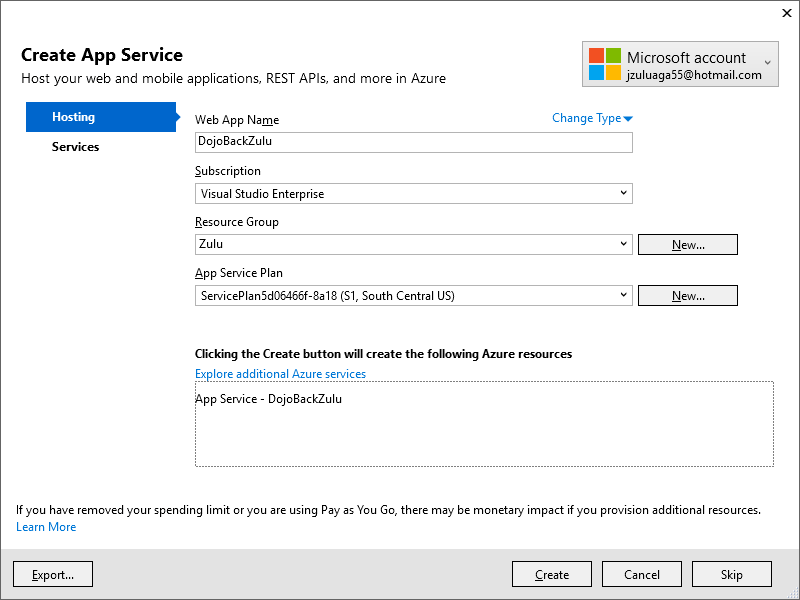


Para completar el proceso de registro vas a necesitar una tarjeta de crédito, aunque ellos no te hacen ningún cargo si no lo autorizas explícitamente. **Nota**: si no dispones de tarjeta de crédito, puedes crear una e-card, esa te sirve, aunque no tenga saldo. Bancolombia tiene el servicio de e-card, no sé si está disponible con otros bancos.

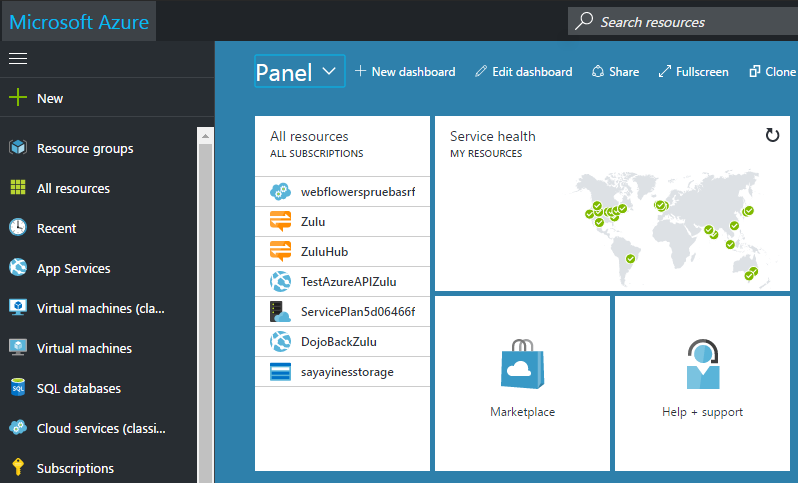
1. Marcamos “Web API” y “Host in the cloud”



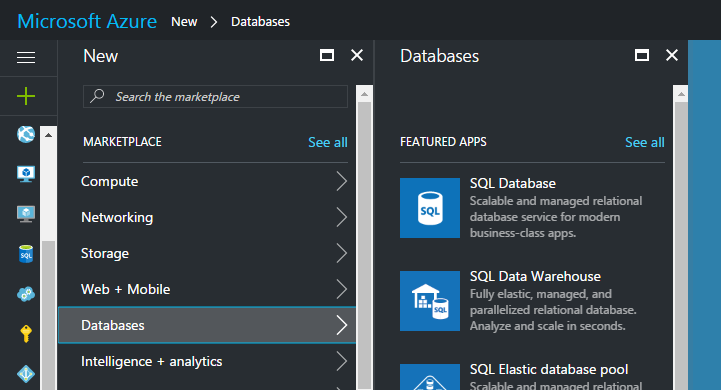
1. Marca los datos del formulario:



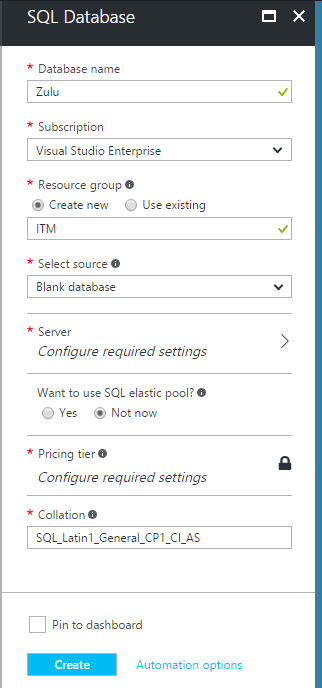
1. Entra al <https://portal.azure.com> y crea una base de datos y el servidor o utiliza una de la que ya tengas creada. Debes tener las credenciales de conexión a la mano.



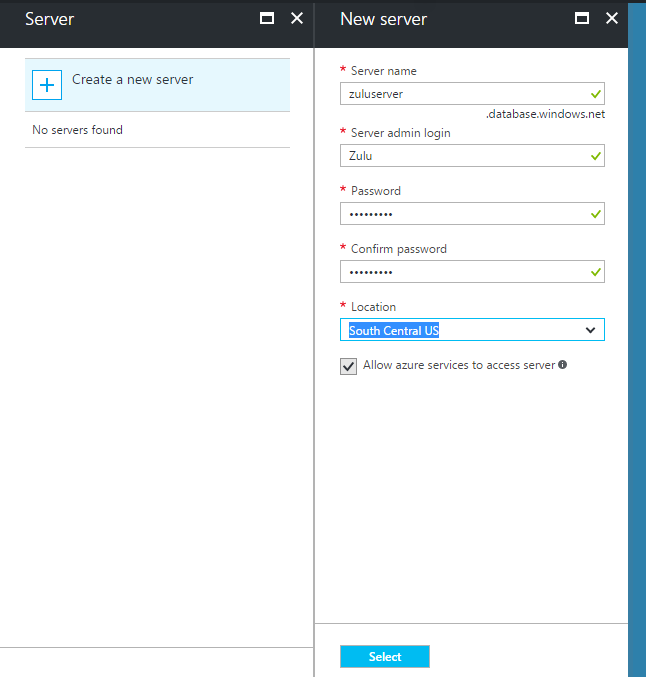
Presionamos New -> Databases -> SQL Database:



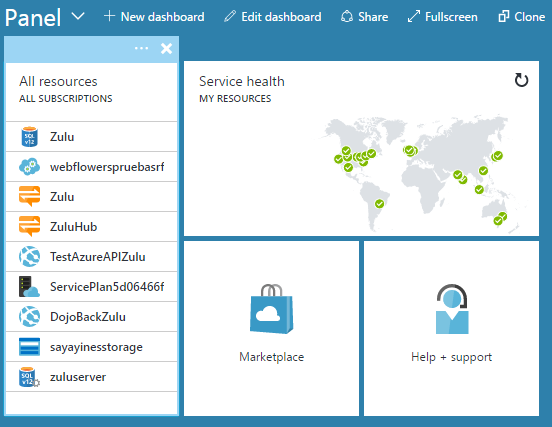
Llenamos los campos:



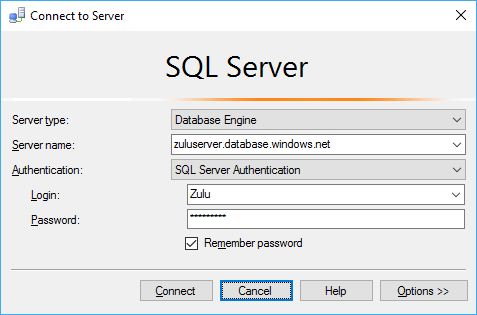
Creamos el servidor:

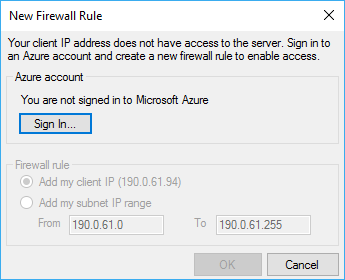


Y confirmamos la creación de la BD y esperamos que el servidor y la BD se creen correctamente.



Es importante mantener la base de datos en S0 o incluso en B, si no la vamos a usar, y subirla a S2 cuando vamos a trabajar. También dependiendo del lugar donde te conectes es posible que toque agregar la dirección IP en el Firewall del servidor. Luego de esto ya te puede conectar con el SQL Management Studio 2016 (disponible en: <https://msdn.microsoft.com/es-co/library/mt238290.aspx>)





1. Cambie los parámetros del **web.config** por los de su base de datos, por ejemplo:

<connectionStrings>

<add name="DefaultConnection"

connectionString="Data Source=zulutest.database.windows.net;Initial Catalog=zulu;Persist Security Info=True;User ID=zulu;Password=Roger1974"

providerName="System.Data.SqlClient" />

</connectionStrings>

1. Agreguemos la clase **DataContext** en la carpeta **Models** con el siguiente código:

public class DataContext : DbContext

{

public DataContext() : base("DefaultConnection")

{

}

}

1. Creemos la clase **Flower** en la carpeta **Models** con el siguiente código:

public class Flower

{

[Key]

public int FlowerId { get; set; }

[Required(ErrorMessage = "You must enter a {0}")]

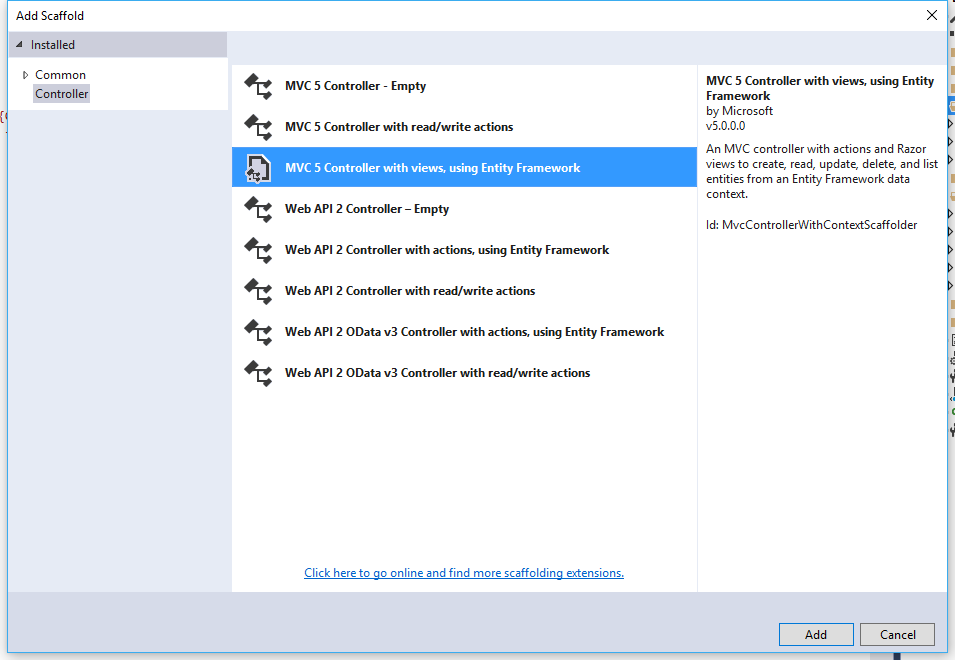
[StringLength(50, ErrorMessage = "The field {0} can contain maximun {1} and minimum {2} characters", MinimumLength = 1)]

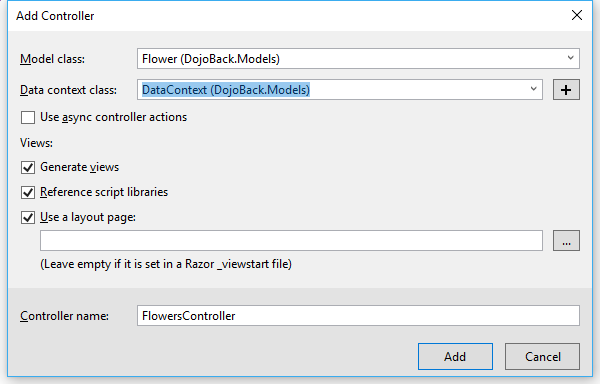
public string Description { get; set; }

public decimal Price { get; set; }

}

1. Compilemos el proyecto.
2. Creamos el controlador para el modelo Flower.





1. Creamos una entrada en el menú de la aplicación para el nuevo controlador, modificamos esta línea en el \_Layout:

<div class="navbar-collapse collapse">

<ul class="nav navbar-nav">

<li>@Html.ActionLink("Home", "Index", "Home")</li>

<li>@Html.ActionLink("About", "About", "Home")</li>

<li>@Html.ActionLink("Contact", "Contact", "Home")</li>

<li>@Html.ActionLink("Flowers", "Index", "Flowers")</li>

</ul>

@Html.Partial("\_LoginPartial")

</div>

1. Antes de correr el proyecto por primera vez, no está de más, habilitar las migraciones automáticas y deshabilitar el borrado en cascada. Primero habilitemos las migraciones automáticas, ejecute el siguiente comando:

Enable-Migrations -ContextTypeName DataContext -EnableAutomaticMigrations –Force

Adiciona la siguiente línea a la clase **Configuration**:

public Configuration()

{

AutomaticMigrationsEnabled = true;

AutomaticMigrationDataLossAllowed = true;

}

Adiciona la siguiente línea al **Global.asax**:

protected void Application\_Start()

{

Database.SetInitializer(new MigrateDatabaseToLatestVersion<Models.DataContext, Migrations.Configuration>());

AreaRegistration.RegisterAllAreas();

GlobalConfiguration.Configure(WebApiConfig.Register);

FilterConfig.RegisterGlobalFilters(GlobalFilters.Filters);

RouteConfig.RegisterRoutes(RouteTable.Routes);

BundleConfig.RegisterBundles(BundleTable.Bundles);

}

Y para deshabilitar el borrado en cascada, agreguemos este método al **DataContext**:

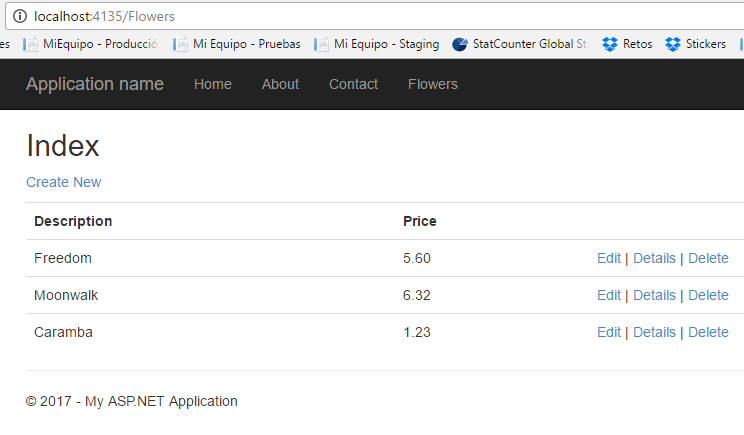
protected override void OnModelCreating(DbModelBuilder modelBuilder)

{

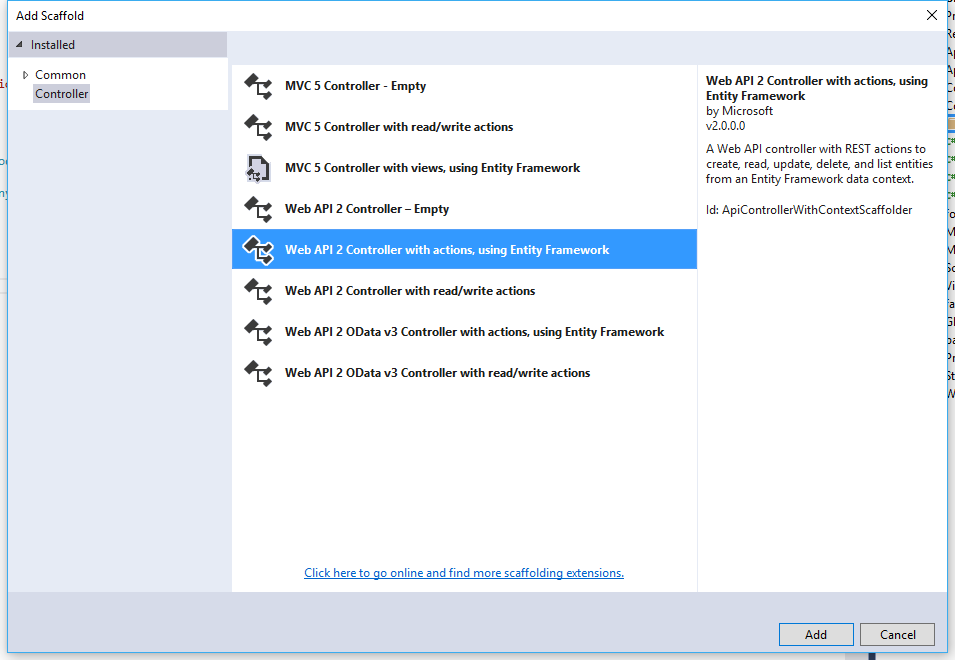
modelBuilder.Conventions.Remove<OneToManyCascadeDeleteConvention>();

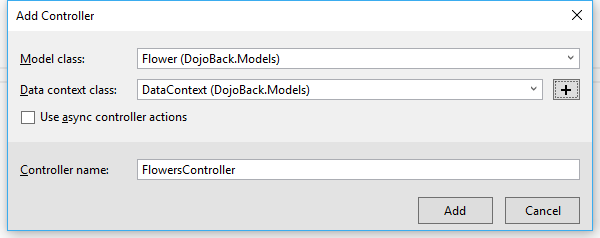
}

1. Corremos el proyecto y creamos unos cuantos registros:

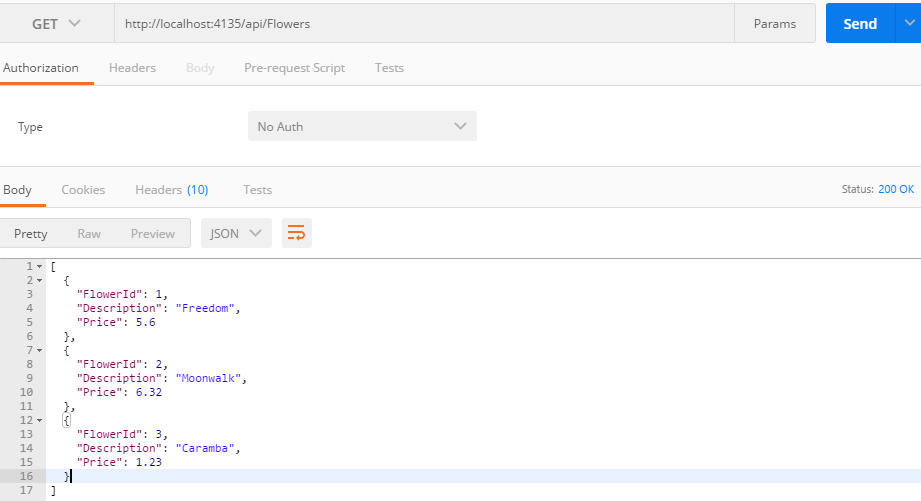


1. Ahora creamos la carpeta **API** dentro de **Controllers** y dentro de esta creamos el controlador API para nuestro modelo:

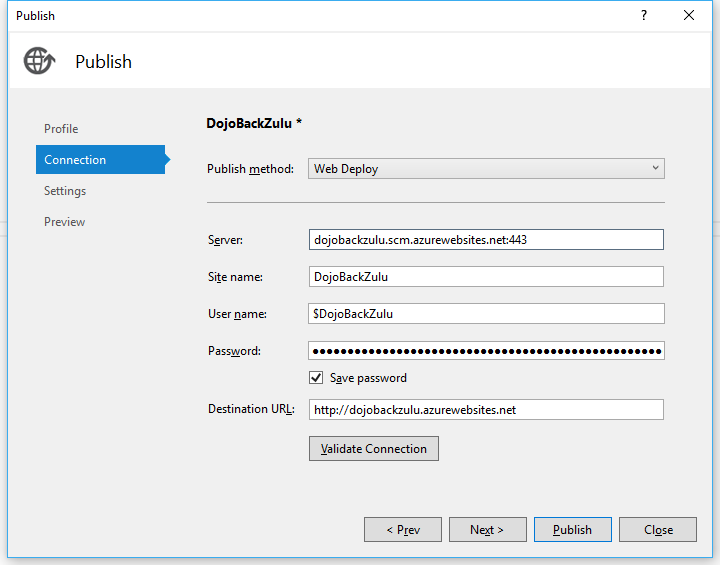


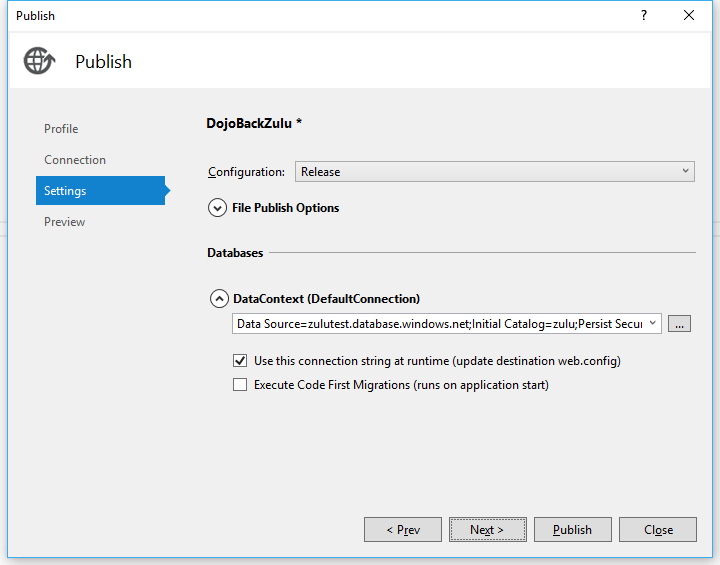


1. Corremos de nuevo la aplicación y probamos nuestro API por Postman:

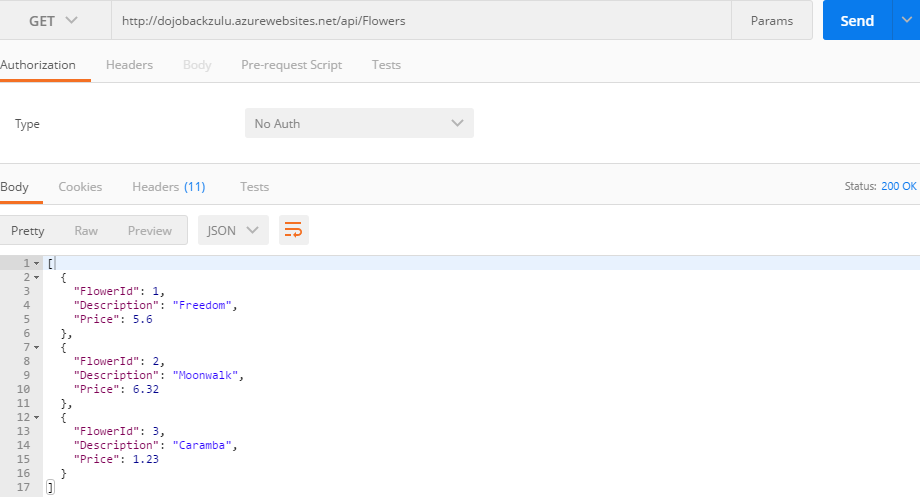


1. Publicamos nuestra aplicación:





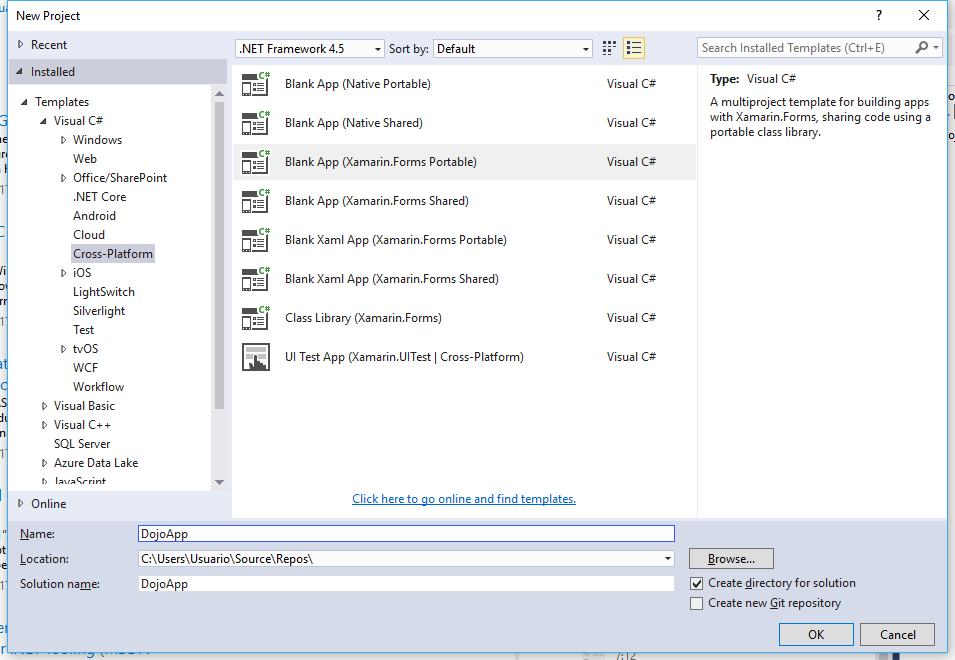
1. Probamos nuevamente en Postman pero ya con la dirección de publicación:



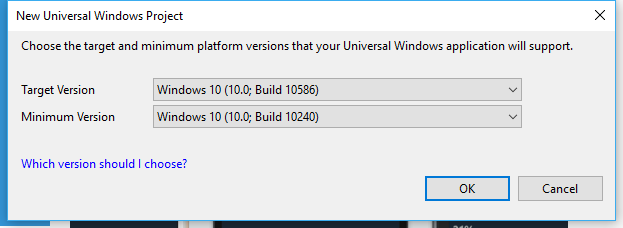
Ya con esto estamos listos con el Backend, ahora vamos a construir la aplicación móvil que consuma estos servicios.

# PARTE II: APP

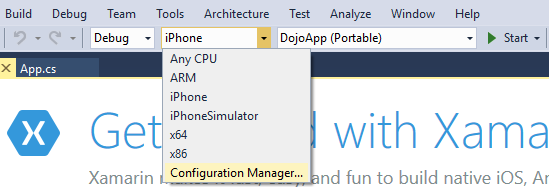
1. Creamos nuestra aplicación de Xamarin Forms

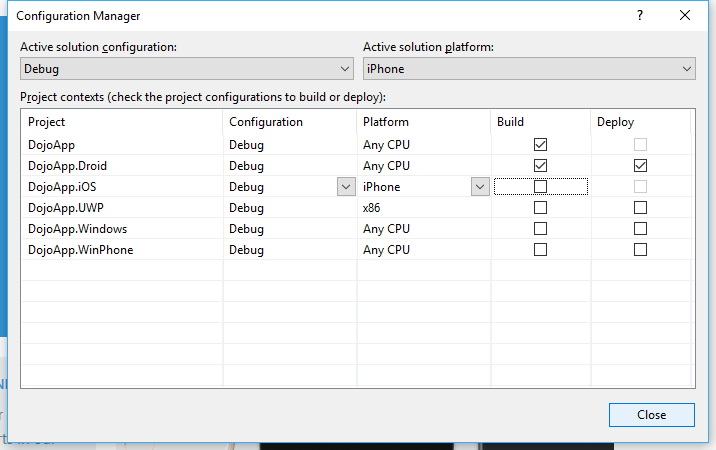


Seleccionamos esto al preguntarnos por las aplicaciones Windows:



Como no le vamos a prestar atención a los proyectos Windows, vamos a desactivarlos para ganar tiempo en la compilación:

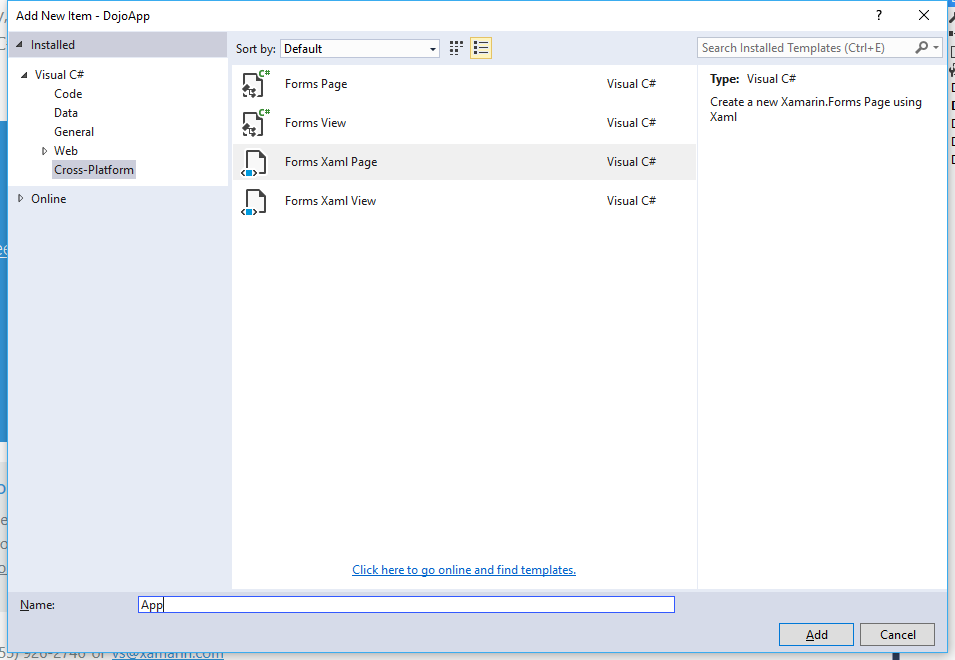




1. Para correrlo por primera vez en Android, debemos borrar el Nuget de Xamarin.Forms, volverlo a instalar, hacer un Clean y Rebuild a la aplicación. Luego de esto corremos y nos debe aparecer el “Hola Mundo” de Xamarin.

|  |  |
| --- | --- |
| Android  C:\Users\Usuario\Desktop\Nexus 4 (Lollipop) Screenshot 5.png | iOS  C:\Users\Usuario\Downloads\Simulator Screen Shot 19.01.2017, 4.41.06 p.m..png |

1. Ahora vemos a crear un diccionario de recursos en nuestra App, para tal fin vamos a copar el código de la clase App en el porta papeles, luego borramos la clase App y agregamos un **Forms Xaml Page** y lo llamamos **App**:



1. Luego vamos al Code Behind de este archivo y lo reemplazamos por lo que tenemos en el porta papeles, y reemplazamos lo que está resaltado en amarillo:

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using Xamarin.Forms;

namespace DojoApp

{

public partial class App : Application

{

public App()

{

InitializeComponent();

// The root page of your application

var content = new ContentPage

{

Title = "DojoApp",

Content = new StackLayout

{

VerticalOptions = LayoutOptions.Center,

Children = {

new Label {

HorizontalTextAlignment = TextAlignment.Center,

Text = "Welcome to Xamarin Forms!"

}

}

}

};

MainPage = new NavigationPage(content);

}

protected override void OnStart()

{

// Handle when your app starts

}

protected override void OnSleep()

{

// Handle when your app sleeps

}

protected override void OnResume()

{

// Handle when your app resumes

}

}

}

1. Cambiamos el código XAML por el siguiente:

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

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

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

x:Class="DojoApp.App">

<Application.Resources>

<ResourceDictionary>

</ResourceDictionary>

</Application.Resources>

</Application>

1. Creamos la carpeta **ViewModels** y dentro de esta creamos la clase **MainViewModel**:

public class MainViewModel

{

}

1. Crearemos esta clase para poder implementar el patrón **Locator**. Para tal fin vamos a crear la carpeta **Infrastructure** y dentro de esta la clase **InstanceLocator** con el siguiente código:

public class InstanceLocator

{

public MainViewModel Main { get; set; }

public InstanceLocator()

{

Main = new MainViewModel();

}

}

1. Vamos hacer que nuestro **InstanceLocator** sea un recurso general para toda la aplicación. Vamos al **App.xaml** y le hacemos las siguientes modificaciones:

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

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

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

xmlns:infra="clr-namespace:DojoApp.Infrastructure;assembly=DojoApp"

x:Class="DojoApp.App">

<Application.Resources>

<ResourceDictionary>

<!-- Locator -->

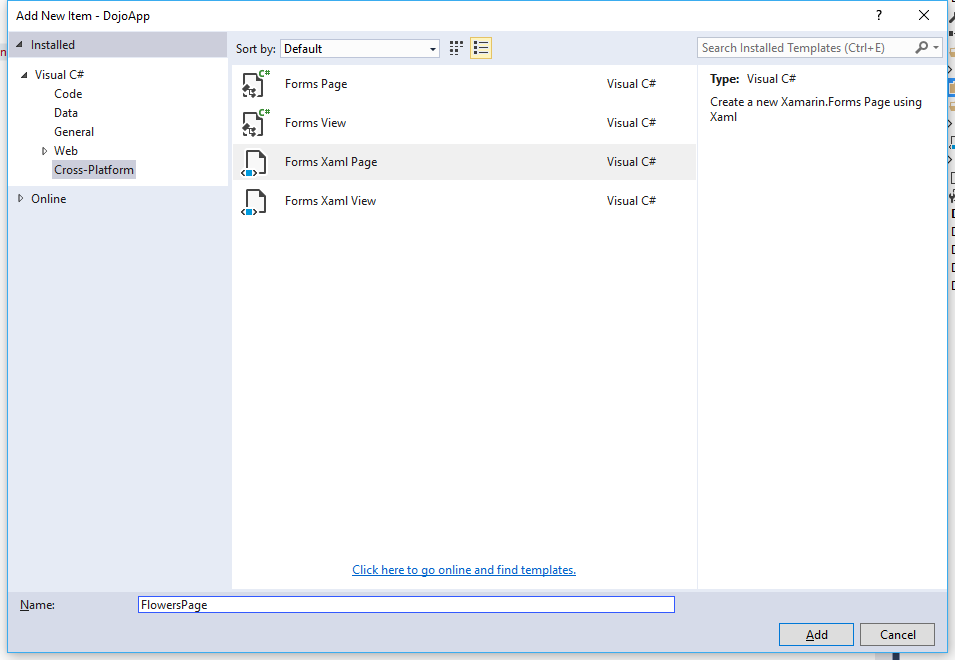
<infra:InstanceLocator x:Key="Locator"></infra:InstanceLocator>

</ResourceDictionary>

</Application.Resources>

</Application>

1. Creamos la carpeta **Pages** y dentro de esta vamos a crear nuestra **FlowersPage**:



<?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="DojoApp.Pages.FlowersPage"

Title="Flowers"

BindingContext="{Binding Main, Source={StaticResource Locator}}">

<Label Text="FlowersPage" VerticalOptions="Center" HorizontalOptions="Center" />

</ContentPage>

1. Cambiamos nuestra página de inicio en el **App.cs**:

public App()

{

InitializeComponent();

MainPage = new NavigationPage(new FlowersPage());

}

1. Probemos que los cambios realizados no nos hayan dañado el proyecto:

|  |  |
| --- | --- |
| Android  C:\Users\Usuario\Desktop\Nexus 4 (Lollipop) Screenshot 6.png | iOS  C:\Users\Usuario\Dropbox\Personal\Documentos\Simulator Screen Shot 20.01.2017, 8.40.04 a.m..png |

1. Ahora vamos a consumir el API para obtener la lista de Flores, primero agregamos a la **SOLUCIÓN** los siguientes Nugets: **Microsoft.BCL.Build**, **Microsoft.BCL**, **Microsoft.Net.Http** y **Newtonsoft.Json**.
2. Creamos la carpeta **Services** y dentro de esta la clase **ApiService**, e implementamos un método genérico que nos sirva para obtener cualquier lista de objetos de un servicio Rest full:

public class ApiService

{

public async Task<List<T>> Get<T>(string urlBase, string servicePrefix, string controller)

{

try

{

var client = new HttpClient();

client.BaseAddress = new Uri(urlBase);

var url = string.Format("{0}{1}", servicePrefix, controller);

var response = await client.GetAsync(url);

if (!response.IsSuccessStatusCode)

{

return null;

}

var result = await response.Content.ReadAsStringAsync();

var list = JsonConvert.DeserializeObject<List<T>>(result);

return list;

}

catch

{

return null;

}

}

}

1. Creamos la carpeta **Models** y dentro de esta la clase **Flower**, la misma que usamos en el backend pero sin los decorados:

public class Flower

{

public int FlowerId { get; set; }

public string Description { get; set; }

public decimal Price { get; set; }

public override int GetHashCode()

{

return FlowerId;

}

}

1. En la carpeta **ViewModels** creamos la clase **FlowerItemViewModel**, esto lo hacemos para mantener el modelo “puro” y seguir los lineamientos del patrón MVVM:

public class FlowerItemViewModel : Flower

{

}

1. Modificamos nuestro **MainViewModel** para que consuma la lista de flores del servicio:

public class MainViewModel

{

#region Attributes

private ApiService apiService;

#endregion

#region Properties

public ObservableCollection<FlowerItemViewModel> Flowers { get; set; }

#endregion

#region Constructors

public MainViewModel()

{

apiService = new ApiService();

Flowers = new ObservableCollection<FlowerItemViewModel>();

LoadFlowers();

}

#endregion

#region Methods

private async void LoadFlowers()

{

var flowers = await apiService.Get<Flower>("http://dojobackzulu.azurewebsites.net", "/api", "/Flowers");

Flowers.Clear();

foreach (var flower in flowers)

{

Flowers.Add(new FlowerItemViewModel

{

Description = flower.Description,

FlowerId = flower.FlowerId,

Price = flower.Price,

});

}

}

#endregion

}

1. Modificamos nuestra **FlowersPage** para que pinte la lista de flores:

<?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="DojoApp.Pages.FlowersPage"

Title="Flowers"

BindingContext="{Binding Main, Source={StaticResource Locator}}">

<ListView

ItemsSource="{Binding Flowers}"

HasUnevenRows="True">

<ListView.ItemTemplate>

<DataTemplate>

<ViewCell>

<Grid Padding="8">

<Grid.ColumnDefinitions>

<ColumnDefinition Width="Auto"></ColumnDefinition>

<ColumnDefinition Width="\*"></ColumnDefinition>

</Grid.ColumnDefinitions>

<Image

Source="icon.png"

VerticalOptions="Start"

WidthRequest="50"

HeightRequest="50"/>

<StackLayout Grid.Column="1">

<Label

FontAttributes="Bold"

VerticalOptions="Center"

Text="{Binding Description}"/>

<Label

VerticalOptions="Center"

Text="{Binding Price, StringFormat='{0:C2}'}"/>

</StackLayout>

</Grid>

</ViewCell>

</DataTemplate>

</ListView.ItemTemplate>

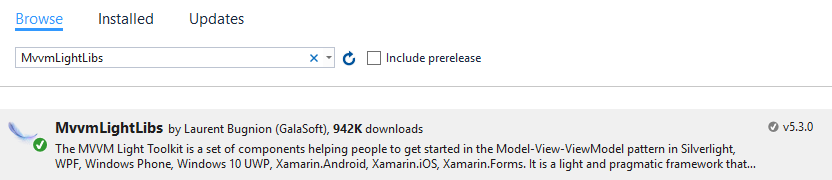
</ListView>

</ContentPage>

1. Para que el ícono también nos salga en iOS, lo agregamos del proyecto Droid al proyecto iOS.
2. Probemos lo que llevamos hasta el momento:

|  |  |
| --- | --- |
| Android  C:\Users\Usuario\Desktop\Nexus 4 (Lollipop) Screenshot 7.png | iOS  C:\Users\Usuario\Downloads\Simulator Screen Shot 20.01.2017, 9.57.04 a.m..png |

1. Ahora agreguemos el Nuget **MvvmLightLibs** para poder incluir comandos que sigan el patrón MVVM:



1. Agreguemos un ícono para adicionar flores (+), recomiendo la página: <https://romannurik.github.io/AndroidAssetStudio/>
2. Creamos la opción de agregar en nuestra barra de herramientas, del FlowersPage:

</ListView>

<ContentPage.ToolbarItems>

<ToolbarItem Icon="add.png" Command="{Binding AddFlowerCommand}"/>

</ContentPage.ToolbarItems>

</ContentPage>

1. Creamos nuestra página **NewFlowerPage**, como inicialmente solo lo vamos a poner a navegar ahí, dejemola temporalmente con este código:

<?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="DojoApp.Pages.NewFlowerPage"

Title="Add Flower"

BindingContext="{Binding Main, Source={StaticResource Locator}}">

<Label Text="NewF lower Page" VerticalOptions="Center" HorizontalOptions="Center" />

</ContentPage>

1. Creamos el **NavigationService** en la carpeta de **Services**:

public class NavigationService

{

public async Task Navigate(string pageName)

{

switch (pageName)

{

case "NewFlowerPage":

await App.Current.MainPage.Navigation.PushAsync(new NewFlowerPage());

break;

default:

break;

}

}

public async Task Back()

{

await App.Current.MainPage.Navigation.PopAsync();

}

}

1. Ahora implementemos el comando en la **MainViewModel**, primero creamos el objeto del servicio de navegación

private NavigationService navigationService;

Y lo instanciamos en el constructor:

navigationService = new NavigationService();

Y creamos el comando:

#region Commands

public ICommand AddFlowerCommand { get { return new RelayCommand(AddFlower); } }

private async void AddFlower()

{

await navigationService.Navigate("NewFlowerPage");

}

#endregion

1. Probemos como quedo la navegación:

|  |  |
| --- | --- |
| Android  C:\Users\Usuario\Desktop\Nexus 4 (Lollipop) Screenshot 10.png  C:\Users\Usuario\Desktop\Nexus 4 (Lollipop) Screenshot 11.png | iOS  C:\Users\Usuario\Downloads\Simulator Screen Shot 20.01.2017, 10.40.16 a.m..png  C:\Users\Usuario\Downloads\Simulator Screen Shot 20.01.2017, 10.40.31 a.m..png |

1. Para evitar que nuestro **MainViewModel** crezca mucho, vamos a crear otro view model llamado **NewFlowerViewModel**:

public class NewFlowerViewModel

{

}

1. En nuestro **MainViewModel** vamos a crear un objeto del nuevo View Model:

public NewFlowerViewModel NewFlower { get; set; }

1. Implementemos un singleton en nuestro **MainViewModel** para poder acceder a este en cualquier momento:

#region Singleton

private static MainViewModel instance;

public static MainViewModel GetInstance()

{

if (instance == null)

{

instance = new MainViewModel();

}

return instance;

}

#endregion

Y lo instanciamos en el constructor:

instance = this;

1. Ahora creamos el nuevo objeto antes de navegar a la página:

public async Task Navigate(string pageName)

{

switch (pageName)

{

case "NewFlowerPage":

var mainViewModel = MainViewModel.GetInstance();

mainViewModel.NewFlower = new NewFlowerViewModel();

await App.Current.MainPage.Navigation.PushAsync(new NewFlowerPage());

break;

default:

break;

}

}

1. Modifiquemos nuestra NewFlowerPage por:

<?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="DojoApp.Pages.NewFlowerPage"

Title="Add Flower"

BindingContext="{Binding Main, Source={StaticResource Locator}}">

<StackLayout

BindingContext="{Binding NewFlower}"

Padding="8">

<Label

Text="Description">

</Label>

<Entry

Text="{Binding Descrpition, Mode=TwoWay}"

Placeholder="Enter a description...">

</Entry>

<Label

Text="Price">

</Label>

<Entry

Text="{Binding Price, Mode=TwoWay}"

Keyboard="Numeric"

Placeholder="Enter a price...">

</Entry>

<ActivityIndicator

IsRunning="{Binding IsRunning, Mode=TwoWay}">

</ActivityIndicator>

<Button

Command="{Binding NewFlowerCommand}"

IsEnabled="{Binding IsEnabled, Mode=TwoWay}"

BorderRadius="20"

HeightRequest="40"

BackgroundColor="Blue"

TextColor="White"

Text="New">

</Button>

</StackLayout>

</ContentPage>

1. Creemos un **DialogService** que lo vamos a necesitar más adelante:

public class DialogService

{

public async Task ShowMessage(string title, string message)

{

await App.Current.MainPage.DisplayAlert(title, message, "Accept");

}

public async Task<bool> ShowConfirm(string title, string message)

{

return await App.Current.MainPage.DisplayAlert(title, message, "Yes", "No");

}

}

1. Ahora creemos los atributos y propiedades en el **NewFlowerViewModel**, para que las propiedades se puedan refrescar debemos de implementar el **INotifyPropertyChanged**

public class NewFlowerViewModel : INotifyPropertyChanged

{

#region Attributes

private ApiService apiService;

private DialogService dialogService;

private NavigationService navigationService;

private string descrpition;

private decimal price;

private bool isRunning;

private bool isEnabled;

#endregion

#region Properties

public string Descrpition

{

set

{

if (descrpition != value)

{

descrpition = value;

PropertyChanged?.Invoke(this, new PropertyChangedEventArgs("Descrpition"));

}

}

get

{

return descrpition;

}

}

public decimal Price

{

set

{

if (price != value)

{

price = value;

PropertyChanged?.Invoke(this, new PropertyChangedEventArgs("Price"));

}

}

get

{

return price;

}

}

public bool IsRunning

{

set

{

if (isRunning != value)

{

isRunning = value;

PropertyChanged?.Invoke(this, new PropertyChangedEventArgs("IsRunning"));

}

}

get

{

return isRunning;

}

}

public bool IsEnabled

{

set

{

if (isEnabled != value)

{

isEnabled = value;

PropertyChanged?.Invoke(this, new PropertyChangedEventArgs("IsEnabled"));

}

}

get

{

return isEnabled;

}

}

#endregion

#region Constructors

public NewFlowerViewModel()

{

apiService = new ApiService();

dialogService = new DialogService();

navigationService = new NavigationService();

IsEnabled = true;

}

#endregion

#region Events

public event PropertyChangedEventHandler PropertyChanged;

#endregion

#region Commands

public ICommand NewFlowerCommand { get { return new RelayCommand(NewFlower); } }

private async void NewFlower()

{

if (string.IsNullOrEmpty(Descrpition))

{

await dialogService.ShowMessage("Error", "You must enter a description");

return;

}

if (Price <= 0)

{

await dialogService.ShowMessage("Error", "The price must be greather than zero");

return;

}

}

#endregion

}

1. Creamos la carpeta **Classes** y dentro de esta carpeta creamos la clase **Response**:

public class Response

{

public bool IsSuccess { get; set; }

public string Message { get; set; }

public object Result { get; set; }

}

1. Creemos ahora nuestro método genérico en el **ApiService** agreguemos los métodos genéricos que nos permitan hacer el POST, PUT y DELETE a los controladores API:

public async Task<Response> Put<T>(string urlBase, string servicePrefix, string controller, T model)

{

try

{

var request = JsonConvert.SerializeObject(model);

var content = new StringContent(request, Encoding.UTF8, "application/json");

var client = new HttpClient();

client.BaseAddress = new Uri(urlBase);

var url = string.Format("{0}{1}/{2}", servicePrefix, controller, model.GetHashCode());

var response = await client.PutAsync(url, content);

if (!response.IsSuccessStatusCode)

{

return new Response

{

IsSuccess = false,

Message = response.StatusCode.ToString(),

};

}

var result = await response.Content.ReadAsStringAsync();

var newRecord = JsonConvert.DeserializeObject<T>(result);

return new Response

{

IsSuccess = true,

Message = "Record updated OK",

Result = newRecord,

};

}

catch (Exception ex)

{

return new Response

{

IsSuccess = false,

Message = ex.Message,

};

}

}

public async Task<Response> Post<T>(string urlBase, string servicePrefix, string controller, T model)

{

try

{

var request = JsonConvert.SerializeObject(model);

var content = new StringContent(request, Encoding.UTF8, "application/json");

var client = new HttpClient();

client.BaseAddress = new Uri(urlBase);

var url = string.Format("{0}{1}", servicePrefix, controller);

var response = await client.PostAsync(url, content);

if (!response.IsSuccessStatusCode)

{

return new Response

{

IsSuccess = false,

Message = response.StatusCode.ToString(),

};

}

var result = await response.Content.ReadAsStringAsync();

var newRecord = JsonConvert.DeserializeObject<T>(result);

return new Response

{

IsSuccess = true,

Message = "Record added OK",

Result = newRecord,

};

}

catch (Exception ex)

{

return new Response

{

IsSuccess = false,

Message = ex.Message,

};

}

}

public async Task<Response> Delete<T>(string urlBase, string servicePrefix, string controller, T model)

{

try

{

var client = new HttpClient();

client.BaseAddress = new Uri(urlBase);

var url = string.Format("{0}{1}/{2}", servicePrefix, controller, model.GetHashCode());

var response = await client.DeleteAsync(url);

if (!response.IsSuccessStatusCode)

{

return new Response

{

IsSuccess = false,

Message = response.StatusCode.ToString(),

};

}

return new Response

{

IsSuccess = true,

Message = "Record deleted OK",

};

}

catch (Exception ex)

{

return new Response

{

IsSuccess = false,

Message = ex.Message,

};

}

}

1. Ahora si podemos terminar el comando para la creación de una nueva flor:

private async void NewFlower()

{

if (string.IsNullOrEmpty(Descrpition))

{

await dialogService.ShowMessage("Error", "You must enter a description");

return;

}

if (Price <= 0)

{

await dialogService.ShowMessage("Error", "The price must be greather than zero");

return;

}

IsRunning = true;

IsEnabled = false;

var newFlower = new Flower

{

Description = Descrpition,

Price = Price,

};

var result = await apiService.Post("http://dojobackzulu.azurewebsites.net", "/api", "/Flowers", newFlower);

IsRunning = false;

IsEnabled = true;

if (!result.IsSuccess)

{

await dialogService.ShowMessage("Error", result.Message);

return;

}

Descrpition = string.Empty;

Price = 0;

await navigationService.Back();

}

1. Probemos como quedo la agregación de registros:

|  |  |
| --- | --- |
| Android  C:\Users\Usuario\Desktop\Nexus 4 (Lollipop) Screenshot 12.png | iOS  C:\Users\Usuario\Downloads\Simulator Screen Shot 20.01.2017, 12.01.46 p.m..png |

1. Ya funciona, sin embargo, hay un problema con el refresco de la lista, vamos a solucionarlo. Primero pasemos la definición de la lista de flores al diccionario de recursos. Así queda el diccionario de registros:

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

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

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

xmlns:infra="clr-namespace:DojoApp.Infrastructure;assembly=DojoApp"

x:Class="DojoApp.App">

<Application.Resources>

<ResourceDictionary>

<!-- Locator -->

<infra:InstanceLocator x:Key="Locator"></infra:InstanceLocator>

<!-- Data Templates -->

<DataTemplate x:Key="FlowersDataTemplate">

<ViewCell>

<Grid Padding="8">

<Grid.ColumnDefinitions>

<ColumnDefinition Width="Auto"></ColumnDefinition>

<ColumnDefinition Width="\*"></ColumnDefinition>

</Grid.ColumnDefinitions>

<Image

Source="icon.png"

VerticalOptions="Start"

WidthRequest="50"

HeightRequest="50"/>

<StackLayout Grid.Column="1">

<Label

FontAttributes="Bold"

VerticalOptions="Center"

Text="{Binding Description}"/>

<Label

VerticalOptions="Center"

Text="{Binding Price, StringFormat='{0:C2}'}"/>

</StackLayout>

</Grid>

</ViewCell>

</DataTemplate>

</ResourceDictionary>

</Application.Resources>

</Application>

Y así queda nuestra página:

<?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="DojoApp.Pages.FlowersPage"

Title="Flowers"

BindingContext="{Binding Main, Source={StaticResource Locator}}">

<ListView

ItemTemplate="{StaticResource FlowersDataTemplate}"

ItemsSource="{Binding Flowers}"

HasUnevenRows="True"

IsPullToRefreshEnabled="True"

RefreshCommand="{Binding RefreshCommand}"

IsRefreshing="{Binding IsRefreshing, Mode=TwoWay}">

</ListView>

<ContentPage.ToolbarItems>

<ToolbarItem Icon="add.png" Command="{Binding AddFlowerCommand}"/>

</ContentPage.ToolbarItems>

</ContentPage>

1. Ahora implementamos la propiedad y el nuevo comando de refresco. Cambie el **MainViewModel** para que implemente la interfaz **INotifyPropertyChanged**

public class MainViewModel : INotifyPropertyChanged

Y cree el evento de la interfaz:

#region Events

public event PropertyChangedEventHandler PropertyChanged;

#endregion

Creamos en el **MainViewModel** el atributo y la propiedad del refresh:

private bool isRefreshing;

Y la propiedad:

public bool IsRefreshing

{

set

{

if (isRefreshing != value)

{

isRefreshing = value;

PropertyChanged?.Invoke(this, new PropertyChangedEventArgs("IsRefreshing"));

}

}

get

{

return isRefreshing;

}

}

Cambiamos el método de carga de flores por:

private async void LoadFlowers()

{

var flowers = await apiService.Get<Flower>("http://dojobackzulu.azurewebsites.net", "/api", "/Flowers");

ReloadFlowers(flowers);

}

private void ReloadFlowers(List<Flower> flowers)

{

Flowers.Clear();

foreach (var flower in flowers)

{

Flowers.Add(new FlowerItemViewModel

{

Description = flower.Description,

FlowerId = flower.FlowerId,

Price = flower.Price,

});

}

}

Y obvio, crear el comando del Refresh:

public ICommand RefreshCommand { get { return new RelayCommand(Refresh); } }

private void Refresh()

{

LoadFlowers();

IsRefreshing = false;

}

Con esto debe refrescar al bajar el **ListView**, probemos. Pero para que refresque automáticamente debemos agregar estas líneas al code behind del **FlowersPage**:

public FlowersPage()

{

InitializeComponent();

var mainViewModel = MainViewModel.GetInstance();

base.Appearing += (object sender, EventArgs e) =>

{

mainViewModel.RefreshCommand.Execute(this);

};

}

1. Probemos.
2. Ahora agreguemos la página para la edición de flores **EditFlowerPage**:

<?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="DojoApp.Pages.EditFlowerPage"

Title="Edit Flower"

BindingContext="{Binding Main, Source={StaticResource Locator}}">

<StackLayout

BindingContext="{Binding EditFlower}"

Padding="8">

<Label

Text="Description">

</Label>

<Entry

Text="{Binding Descrpition, Mode=TwoWay}"

Placeholder="Enter a description...">

</Entry>

<Label

Text="Price">

</Label>

<Entry

Text="{Binding Price, Mode=TwoWay}"

Keyboard="Numeric"

Placeholder="Enter a price...">

</Entry>

<ActivityIndicator

IsRunning="{Binding IsRunning, Mode=TwoWay}">

</ActivityIndicator>

<StackLayout

Orientation="Horizontal

">

<Button

Command="{Binding SaveFlowerCommand}"

IsEnabled="{Binding IsEnabled, Mode=TwoWay}"

BorderRadius="20"

HeightRequest="40"

BackgroundColor="Blue"

HorizontalOptions="FillAndExpand"

TextColor="White"

Text="Save">

</Button>

<Button

Command="{Binding DeleteFlowerCommand}"

IsEnabled="{Binding IsEnabled, Mode=TwoWay}"

BorderRadius="20"

HeightRequest="40"

BackgroundColor="Red"

HorizontalOptions="FillAndExpand"

TextColor="White"

Text="Delete">

</Button>

</StackLayout>

</StackLayout>

</ContentPage>

1. Y agregamos nuestro **EditFlowerViewModel**, inicialmente vacío:

public class EditFlowerViewModel

{

}

Declaramos e instanciamos este elemento en el **MainViewModel**:

public EditFlowerViewModel EditFlower { get; set; }

Modificamos el método de navegación para instanciar el objeto e ir a la nueva página de edición de flores:

public async Task Navigate(string pageName)

{

var mainViewModel = MainViewModel.GetInstance();

switch (pageName)

{

case "NewFlowerPage":

mainViewModel.NewFlower = new NewFlowerViewModel();

await App.Current.MainPage.Navigation.PushAsync(new NewFlowerPage());

break;

case "EditFlowerPage":

mainViewModel.EditFlower = new EditFlowerViewModel();

await App.Current.MainPage.Navigation.PushAsync(new EditFlowerPage());

break;

default:

break;

}

}

1. Ahora vamos a colocar un **Gesture Recognizer** a nuestro grid para que identifique cuando hagan un toque en algún elemento de la cuadricula, para eso editamos el **App.xaml** que es donde tenemos la definición de nuestra cuadricula:

<!-- Data Templates -->

<DataTemplate x:Key="FlowersDataTemplate">

<ViewCell>

<Grid Padding="8">

<Grid.GestureRecognizers>

<TapGestureRecognizer Command="{Binding EditFlowerCommand}"/>

</Grid.GestureRecognizers>

<Grid.ColumnDefinitions>

<ColumnDefinition Width="Auto"></ColumnDefinition>

<ColumnDefinition Width="\*"></ColumnDefinition>

</Grid.ColumnDefinitions>

<Image

Source="icon.png"

VerticalOptions="Start"

WidthRequest="50"

HeightRequest="50"/>

<StackLayout Grid.Column="1">

<Label

FontAttributes="Bold"

VerticalOptions="Center"

Text="{Binding Description}"/>

<Label

VerticalOptions="Center"

Text="{Binding Price, StringFormat='{0:C2}'}"/>

</StackLayout>

</Grid>

</ViewCell>

</DataTemplate>

1. En el contexto de la cuadricula, es decir, el **FlowerItemViewModel**, agregamos el comando para que navegue a la nueva página:

public class FlowerItemViewModel : Flower

{

#region Attributes

private NavigationService navigationService;

#endregion

#region Constructors

public FlowerItemViewModel()

{

navigationService = new NavigationService();

}

#endregion

#region Commands

public ICommand EditFlowerCommand { get { return new RelayCommand(EditFlower); } }

private async void EditFlower()

{

await navigationService.Navigate("EditFlowerPage");

}

#endregion

}

1. Probemos:

|  |  |
| --- | --- |
| Android  C:\Users\Usuario\Desktop\Nexus 4 (Lollipop) Screenshot 13.png | iOS  C:\Users\Usuario\Downloads\Simulator Screen Shot 23.01.2017, 9.49.21 a.m..png |

1. Ahora mostremos los datos de la flor que deseamos editar. Primero cambiamos nuestro EditFlowerViewModel para que reciba la flor en el constructor:

public class EditFlowerViewModel : Flower

{

public EditFlowerViewModel(Flower flower)

{

Description = flower.Description;

FlowerId = flower.FlowerId;

Price = flower.Price;

}

}

Cambiamos nuestro servicio de navegación, donde vamos a crear un método independiente para la edición de la flor:

public class NavigationService

{

public async Task Navigate(string pageName)

{

var mainViewModel = MainViewModel.GetInstance();

switch (pageName)

{

case "NewFlowerPage":

mainViewModel.NewFlower = new NewFlowerViewModel();

await App.Current.MainPage.Navigation.PushAsync(new NewFlowerPage());

break;

default:

break;

}

}

public async Task Back()

{

await App.Current.MainPage.Navigation.PopAsync();

}

public async Task EditFlower(Flower flower)

{

var mainViewModel = MainViewModel.GetInstance();

mainViewModel.EditFlower = new EditFlowerViewModel(flower);

await App.Current.MainPage.Navigation.PushAsync(new EditFlowerPage());

}

}

Y cambiamos el comando que llama la edición de la flor (**FlowerItemViewModel**), para pasar la misma por parámetros:

private async void EditFlower()

{

var flower = new Flower

{

Description = Description,

FlowerId = FlowerId,

Price = Price,

};

await navigationService.EditFlower(flower);

}

1. Probemos:

|  |  |
| --- | --- |
| Android  C:\Users\Usuario\Desktop\Nexus 4 (Lollipop) Screenshot 14.png | iOS  C:\Users\Usuario\Downloads\Simulator Screen Shot 23.01.2017, 10.04.48 a.m..png |

1. Cambiemos la clase EditFlowerViewModel por este código y probemos la edición de flores:

public class EditFlowerViewModel : Flower, INotifyPropertyChanged

{

#region Attributes

private ApiService apiService;

private DialogService dialogService;

private NavigationService navigationService;

private string descrpition;

private decimal price;

private bool isRunning;

private bool isEnabled;

#endregion

#region Properties

public new string Description

{

set

{

if (descrpition != value)

{

descrpition = value;

PropertyChanged?.Invoke(this, new PropertyChangedEventArgs("Descrpition"));

}

}

get

{

return descrpition;

}

}

public new decimal Price

{

set

{

if (price != value)

{

price = value;

PropertyChanged?.Invoke(this, new PropertyChangedEventArgs("Price"));

}

}

get

{

return price;

}

}

public bool IsRunning

{

set

{

if (isRunning != value)

{

isRunning = value;

PropertyChanged?.Invoke(this, new PropertyChangedEventArgs("IsRunning"));

}

}

get

{

return isRunning;

}

}

public bool IsEnabled

{

set

{

if (isEnabled != value)

{

isEnabled = value;

PropertyChanged?.Invoke(this, new PropertyChangedEventArgs("IsEnabled"));

}

}

get

{

return isEnabled;

}

}

#endregion

#region Constructors

public EditFlowerViewModel(Flower flower)

{

Description = flower.Description;

FlowerId = flower.FlowerId;

Price = flower.Price;

apiService = new ApiService();

dialogService = new DialogService();

navigationService = new NavigationService();

IsEnabled = true;

}

#endregion

#region Events

public event PropertyChangedEventHandler PropertyChanged;

#endregion

#region Commands

public ICommand SaveFlowerCommand { get { return new RelayCommand(SaveFlower); } }

private async void SaveFlower()

{

if (string.IsNullOrEmpty(Description))

{

await dialogService.ShowMessage("Error", "You must enter a description");

return;

}

if (Price <= 0)

{

await dialogService.ShowMessage("Error", "The price must be greather than zero");

return;

}

IsRunning = true;

IsEnabled = false;

var flower = new Flower

{

Description = Description,

FlowerId = FlowerId,

Price = Price,

};

var result = await apiService.Put("http://dojobackzulu.azurewebsites.net", "/api", "/Flowers", flower);

IsRunning = false;

IsEnabled = true;

if (!result.IsSuccess)

{

await dialogService.ShowMessage("Error", result.Message);

return;

}

Description = string.Empty;

Price = 0;

await navigationService.Back();

}

#endregion

}

1. Ahora implementemos la opción de borrado con este código:

public ICommand DeleteFlowerCommand { get { return new RelayCommand(DeleteFlower); } }

private async void DeleteFlower()

{

var answer = await dialogService.ShowConfirm("Confirm", "Are you sure to delete this record?");

if (!answer)

{

return;

}

IsRunning = true;

IsEnabled = false;

var flower = new Flower

{

Description = Description,

FlowerId = FlowerId,

Price = Price,

};

var result = await apiService.Delete("http://dojobackzulu.azurewebsites.net", "/api", "/Flowers", flower);

IsRunning = false;

IsEnabled = true;

if (!result.IsSuccess)

{

await dialogService.ShowMessage("Error", result.Message);

return;

}

Description = string.Empty;

Price = 0;

await navigationService.Back();

}

1. Probemos:

|  |  |
| --- | --- |
| Android  C:\Users\Usuario\Desktop\Nexus 4 (Lollipop) Screenshot 15.png | iOS  C:\Users\Usuario\Downloads\Simulator Screen Shot 23.01.2017, 10.33.22 a.m..png |

¡Listo hemos terminado!