Laboratorio

Accediendo a servicios REST con Xamarin.Forms

Versión: 1.0.0
Diciembre de 2016

Miguel Muñoz Serafín
@msmdotnet

Soporte: mmunoz@mail.ticapacitacion.com

CONTENIDO

Introducción

Integrar un servicio Web dentro de una aplicación es un escenario común. En este laboratorio se muestra la forma de consumir un servicio Web RESTful desde una aplicación Xamarin.Forms. La aplicación Xamarin.Forms será desarrollada implementando el patrón MVVM.

Objetivos

Al finalizar este laboratorio, los participantes serán capaces de:

- Implementar el patrón MVVM en una aplicación Xamarin.Forms.
- Utilizar la clase **HttpClient** para consumir un servicio Web RESTful desde una aplicación Xamarin.Forms.
- Utilizar la clase **JsonConvert** para deserializar datos JSON a objetos .NET.

Requisitos

Para la realización de este laboratorio es necesario contar con lo siguiente:

- Un equipo de desarrollo con sistema operativo Windows 10 y Visual Studio 2015 Community, Professional o Enterprise con la plataforma Xamarin.
- Un equipo Mac con la plataforma Xamarin.
- Una conexión a Internet.

Tiempo estimado para completar este laboratorio: **60 minutos**.

Ejercicio 1: Consumiendo un servicio Web RESTful.

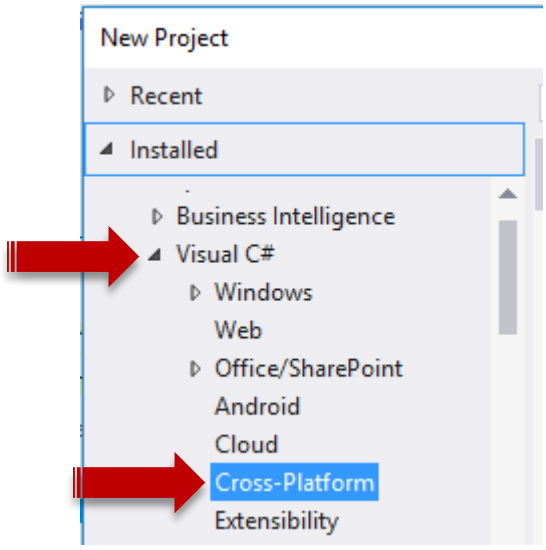
En este ejercicio crearás una aplicación Xamarin.Forms que consumirá un servicio Web RESTful. El servicio RESTful preparado para este laboratorio fue desarrollado con ASP.NET Web API y expone información en formato JSON relacionada con el precio de los gatos más caros del mundo.

La aplicación será desarrollada implementando el patrón MVVM.

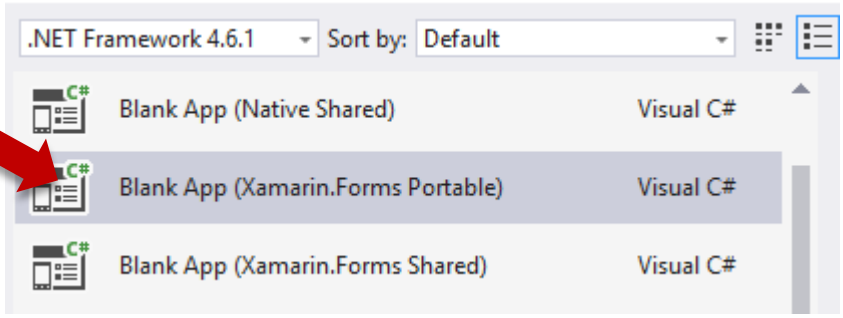
Tarea 1. Crear una aplicación Xamarin.Forms.

En esta tarea crearás una aplicación Xamarin.Forms utilizando Microsoft Visual Studio y la plantilla **Blank App (Xamarin.Forms Portable)**.

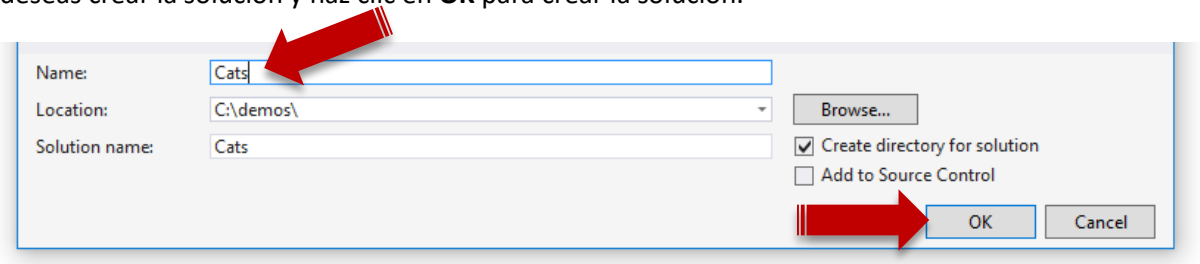
1. Selecciona la opción **File > New > Project** desde Visual Studio.
2. En el panel izquierdo de la ventana **New Project** selecciona **Visual C# > Cross-Platform**.



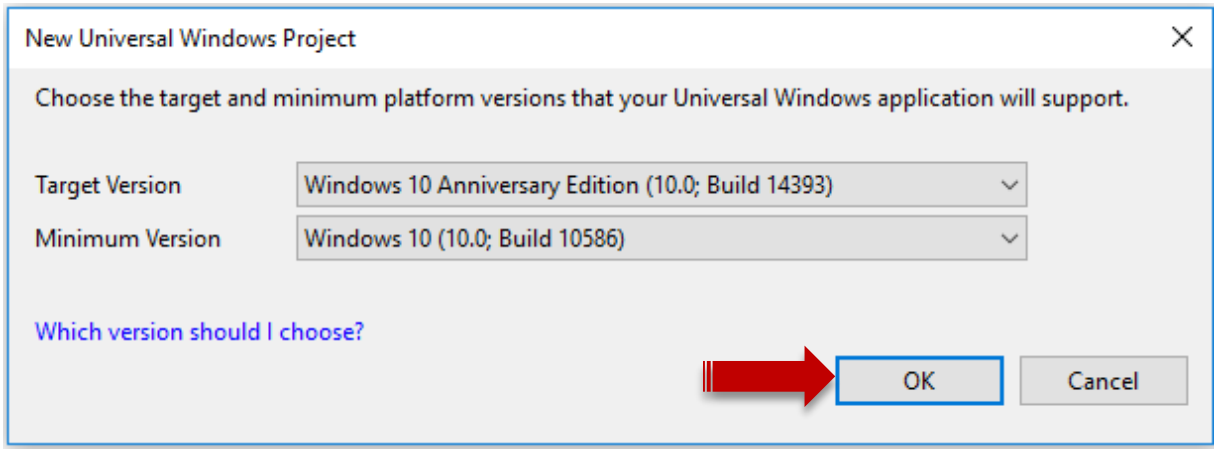
3. Selecciona la plantilla **Blank App (Xamarin.Forms Portable)**.



4. Proporciona **Cats** como nombre de la solución. Proporciona también la ubicación donde deseas crear la solución y haz clic en **OK** para crear la solución.

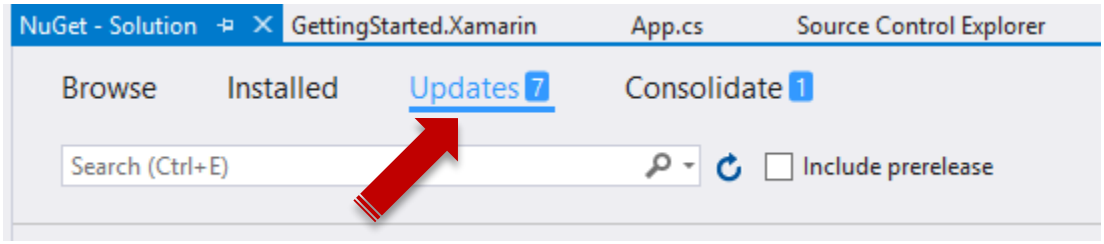


5. Haz clic en el botón **OK** del cuadro de diálogo **New Universal Windows Project** para aceptar las versiones sugeridas para la aplicación UWP que será creada.



6. Después de que la solución con sus proyectos haya sido creada, selecciona la opción **Manage NuGet Packages for Solution…** desde el menú contextual del nombre de la solución.

7. Haz clic en la opción **Updates** para ver las actualizaciones disponibles.



Visual Studio puede indicarte que existen actualizaciones para el paquete NuGet Xamarin.Forms y todas sus dependencias, sin embargo, Xamarin.Forms está configurado para dependencias de versiones específicas. Por lo tanto, aunque Visual Studio te indique que hay nuevas versiones disponibles de paquetes Xamarin.Android.Support, Xamarin.Forms no es necesariamente compatible con esas nuevas versiones.

8. Selecciona el paquete **Xamarin.Forms** y haz clic en **Update** para iniciar la actualización.

Es probable que te sea solicitado aceptar los cambios y reiniciar Visual Studio para concluir la instalación.

Tarea 2. Instalar los paquetes NuGet.

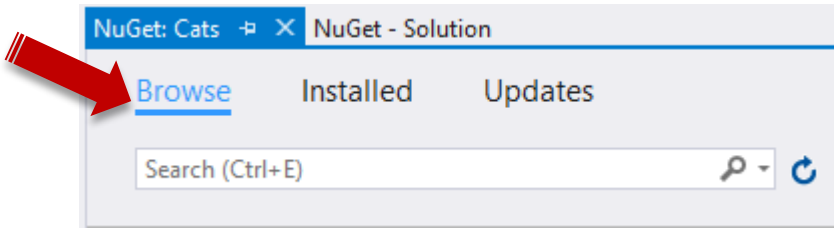
Dos paquetes Nuget son necesarios en nuestra aplicación:

- **Microsoft.Net.Http**. Este paquete proporciona una interface de programación para aplicaciones basadas en HTTP/REST. El paquete incluye la clase **HttpClient** que es utilizada

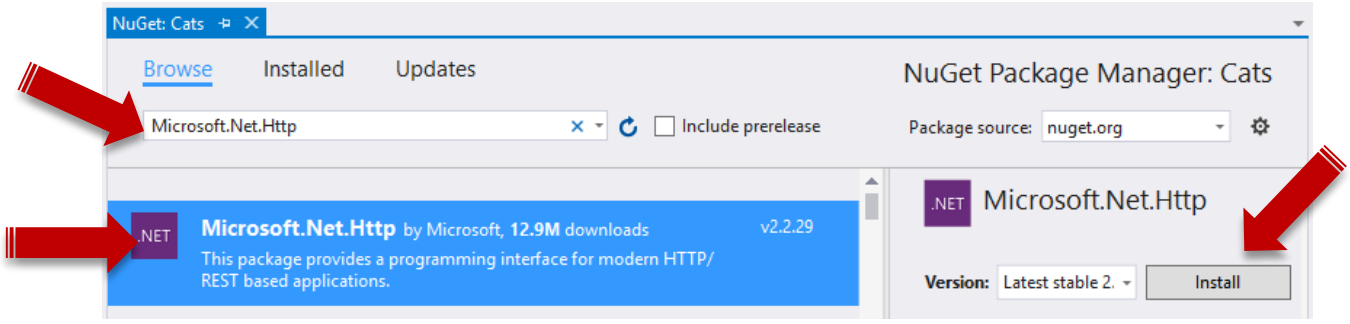
para enviar peticiones sobre HTTP. Incluye además las clases **HttpRequestMessage** y **HttpResponseMessage** para procesar los mensajes HTTP.

- **Newtonsoft.Json**. Este paquete es un Framework que permite serializar y deserializar datos en formato JSON. Es útil para procesar los datos que son enviados o recibidos desde el servicio REST.

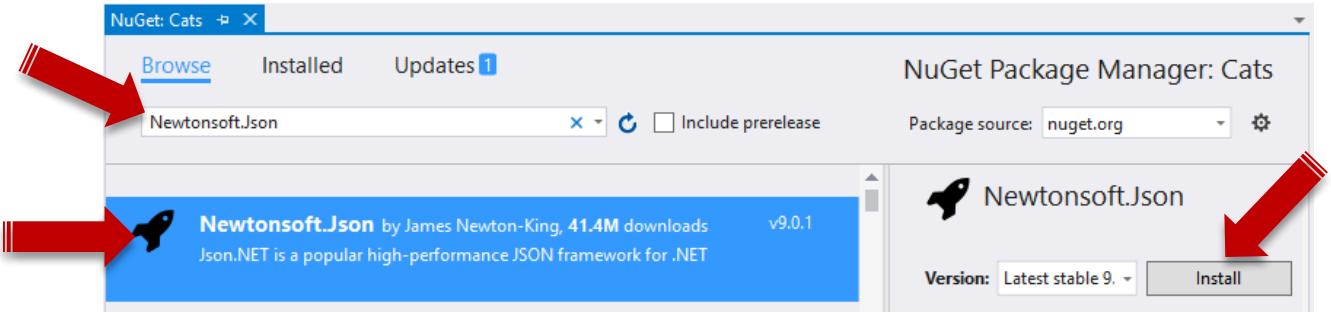
1. Selecciona la opción **Manage NuGet Packages…** del menú contextual del proyecto PCL.
2. En la ventana **NuGet** selecciona la pestaña **Browse**.



3. En el cuadro de búsqueda escribe **Microsoft.Net.Http,** selecciona el paquete **Microsoft.Net.Http** y haz clic en **Install** para instalar el paquete NuGet.



4. Confirma los cambios y el acuerdo de licencias cuando te sea solicitado.
5. Después de que el paquete haya sido instalado, en el cuadro de búsqueda escribe **Newtonsoft.Json**, selecciona el paquete **Newtonsoft.Json** y haz clic en **Install** para instalar el paquete NuGet.

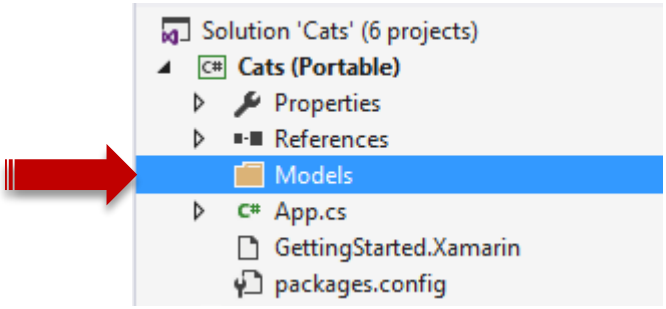


6. Confirma los cambios cuando te sea solicitado.

Tarea 3. Crear el Modelo.

La aplicación obtendrá del servicio REST los datos de las razas de gatos más caras del mundo. En esta tarea crearás la clase modelo que te permitirá almacenar los datos de una raza de gato.

1. En el proyecto PCL agrega un nuevo directorio llamado **Models**.



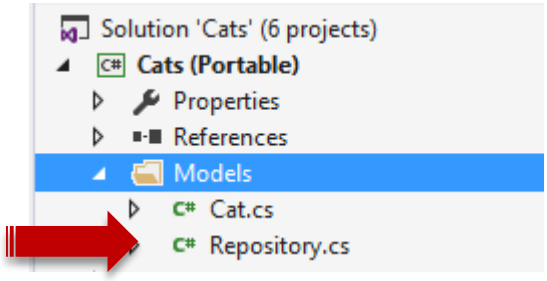
2. Dentro del directorio **Models** agrega la clase **Cat** con las siguientes propiedades públicas:

Nombre	Tipo	Descripción
Id	string	Identificador único de una raza de gato.
Name	string	Nombre de la raza de gato.
Description	string	Descripción de la raza de gato.
Price	int	Precio de un gato de esa raza.
WebSite	string	URL del Sitio Web donde puede encontrarse más información sobre la raza de gato.
Image	string	URL de la imagen de un gato de esa raza.

El código de la clase será similar al siguiente:

```
public class Cat
{
    public string Id { get; set; }
    public string Name { get; set; }
    public string Description { get; set; }
    public int Price { get; set; }
    public string WebSite { get; set; }
    public string Image { get; set; }
}
```

3. Dentro del directorio **Models** agrega una nueva clase pública llamada **Repository**. Esta clase contendrá la lógica de acceso a datos de la aplicación.



4. Agrega el siguiente código a la clase **Repository** para definir e implementar el método **GetCats**. Este método devolverá la lista de datos obtenida desde el servicio REST.

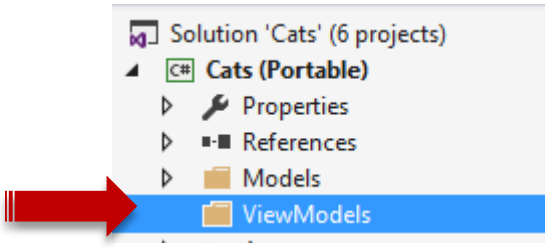
```
public async Task<List<Cat>> GetCats()
{
    List<Cat> Cats;
    var URLWebAPI = "http://demos.ticapacitacion.com/cats";
    using (var Client = new System.Net.Http.HttpClient())
    {
        var JSON = await Client.GetStringAsync(URLWebAPI);
        Cats = Newtonsoft.Json.JsonConvert.DeserializeObject<List<Cat>>(JSON);
    }
    return Cats;
}
```

Puedes notar que el método está definido para ejecutarse de forma asíncrona y que los datos se obtienen desde el servicio **http://demos.ticapacitacion.com/cats** utilizando la clase **HttpClient**. Los datos JSON obtenidos son deserializados utilizando el método **DeserializeObject** del objeto **JsonConvert**.

Tarea 4. Crear el ViewModel.

En esta tarea crearás el ViewModel mediante la clase **CatsViewModel**. Esta clase proporcionará toda la funcionalidad que necesita la Vista Xamarin.Forms principal de la aplicación para mostrar los datos de las razas de gatos. El ViewModel contendrá la lista de objetos **Cat** y un método que podrá ser invocado para obtener la lista de objetos **Cat** del repositorio. También contendrá una propiedad booleana que indicará si estamos obteniendo los datos en una tarea de fondo (Background Task).

1. En el proyecto PCL agrega un nuevo directorio llamado **ViewModels**.



2. En el directorio ViewModels agrega la clase pública llamada **CatsViewModel**.
3. Una clase ViewModel debe ser capaz de notificar los cambios que sucedan en sus propiedades mediante la implementación de la interface **INotifyPropertyChanged**.

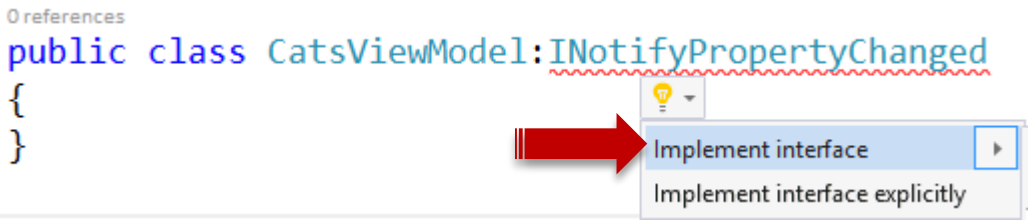
   Agrega al inicio del archivo **CatsViewModel.cs** el siguiente código para importar el espacio de nombres de la interface **INotifyPropertyChanged**.

   ```
   using System.ComponentModel;
   ```

4. Modifica la definición de la clase **CatsViewModel** para indicar que implementa la interface **INotifyPropertyChanged**.

   ```
   public class CatsViewModel:INotifyPropertyChanged
   {
   }
   ```

   INotifyPropertyChanged es importante para el enlace a datos en Frameworks MVVM. Es una interface que cuando es implementada permite a nuestra Vista conocer los cambios del ViewModel.
5. Con la ayuda del intellisense de Visual Studio implementa la interface **INotifyPropertyChanged**.



   Esto agregará la siguiente línea de código:

   ```
   public event PropertyChangedEventHandler PropertyChanged;
   ```

6. Agrega el siguiente código para definir un método de ayuda llamado **OnPropertyChanged** que lanzará el evento **PropertyChanged**. Invocaremos este método de ayuda cuando una propiedad del ViewModel cambie.

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

   Observa que estamos aplicando el atributo **CallerMemberName** al parámetro **propertyName.** Este atributo permite obtener el nombre del miembro de la clase que invoca a este método. Esto evitará que debamos especificar mediante código duro el nombre de la propiedad que ha cambiado al momento de invocar a este método.

7. El siguiente paso es crear una propiedad que le permita a la vista determinar si el ViewModel se encuentra ocupado. De esta forma podremos evitar realizar operaciones duplicadas, como por ejemplo cuando el usuario actualiza los datos múltiples veces.

   Agrega el siguiente código para crear el campo de respaldo de la propiedad.

   ```
   private bool Busy;
   ```

8. Agrega el siguiente código para crear la propiedad **IsBusy**.

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

   Nota que estamos invocando al método **OnPropertyChanged** cuando el valor de la propiedad cambia. La infraestructura de enlace de Xamarin.Forms se suscribirá a nuestro evento **PropertyChanged** para que la interfaz de usuario sea notificada del cambio.

9. Agrega el siguiente código para definir una propiedad **Cats** que almacenará la lista de objetos **Cat**.

   ```
   public ObservableCollection<Cat> Cats { get; set; }
   ```

   Nota que estamos utilizando **ObservableCollection** debido a que esta clase tiene soporte para eventos **CollectionChanged** que ocurren cuando agregamos o eliminamos elementos de la colección. Esto es muy útil ya que no tenemos que invocar al método **OnPropertyChanged** por cada cambio en los elementos de la colección.

10. Agrega el siguiente código al inicio del archivo para importar el espacio de nombres de las clases modelo de la aplicación y de la clase **ObservableCollection**.

    ```
    using System.Collections.ObjectModel;
    using Cats.Models;
    ```

11. Agrega el siguiente código para definir el constructor del ViewModel. El código inicializará la propiedad **Cats**.

    ```
    public CatsViewModel()
    {
        Cats = new ObservableCollection<Models.Cat>();
    }
    ```

12. Agrega el siguiente código para definir un método asíncrono llamado **GetCats** que obtendrá los datos de las razas de gatos desde el repositorio.

```
async Task GetCats()
{
}
```

13. Dentro del método **GetCats** agrega el siguiente código que permitirá detectar si actualmente el ViewModel se encuentra ocupado obteniendo los datos.

```
if(!IsBusy)
{
}
return;
```

14. Dentro del bloque **if** agrega el siguiente código **try/catch/finally**.

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

Nota que estamos estableciendo **IsBusy** a **true** y posteriormente a **false** cuando iniciamos la recuperación de la información desde el repositorio y cuando terminamos de obtener los datos.

15. Agrega el siguiente código dentro del bloque **try** para obtener los datos del repositorio.

```
try
{
    IsBusy = true;
    var Repository = new Repository();
    var Items = await Repository.GetCats();
}
```

16. Dentro del bloque **try,** debajo del código anterior agrega el siguiente código para limpiar la lista actual de objetos **Cat** y cargarlos desde la colección **Items**.

```
Cats.Clear();
foreach(var Cat in Items)
{
    Cats.Add(Cat);
}
```

Si algo funciona mal, el bloque **catch** guardará la excepción y después del bloque **finally** podremos mostrar un mensaje emergente.

17. Agrega el siguiente código justo después del bloque **finally** para mostrar un mensaje en caso de que se haya generado una excepción.

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

El método principal del ViewModel para obtener los datos ha sido completado. En lugar de invocar el método directamente, lo expondremos con un **Command**. Un objeto **Command** tiene una interface que sabe que método invocar y tiene una forma opcional de describir si el **Command** está habilitado.

18. Agrega el siguiente código a la clase **CatsViewModel** para crear un nuevo comando llamado **GetCatsCommand**.

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

19. Agrega el siguiente código al inicio del archivo para importar el espacio de nombres de la clase **Command**.

```
using Xamarin.Forms;
```

20. Dentro del constructor de **CatsViewModel** agrega el siguiente código para inicializar el comando **GetCatsCommand** pasándole dos métodos: uno para invocarse cuando el comando sea ejecutado y otro para determinar cuando el comando esté habilitado. Ambos métodos están implementados como expresiones lambda.

```
GetCatsCommand = new Command(
    async () => await GetCats(),
    () => !IsBusy
    );
```

La única modificación que tenemos que hacer es para el caso en que el valor de la propiedad **IsBusy** cambie. En ese caso, tenemos que reevaluar la función que determina si el comando está habilitado.

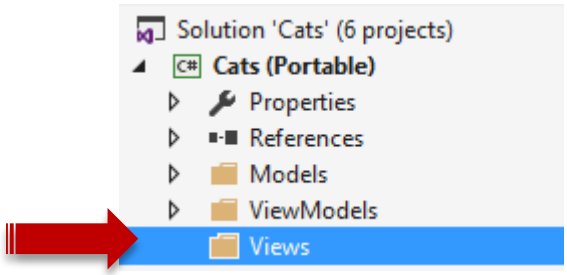
21. Agrega el siguiente código al final del bloque **set** de la propiedad **IsBusy** para invocar el método **ChangeCanExecute** del comando **GetCatsCommand**. Al ejecutar el método **ChangeCanExecute**, la función que determina si el comando está habilitado será reevaluada.

```
GetCatsCommand.ChangeCanExecute();
```

Tarea 5. Crear la Vista.

Finalmente, es tiempo de construir la interfaz de usuario Xamarin.Forms que constituirá el elemento View de nuestra aplicación MVVM.

1. Agrega un directorio **Views** en la raíz del proyecto PCL.



2. Dentro del directorio **Views** agrega un nuevo elemento **Forms Xaml Page** llamado **CatsPage.xaml**.
   Para esta página agregaremos controles alineados verticalmente. Podemos utilizar un control **StackLayout** para hacer eso.
3. Remplaza el elemento **Label** dentro de **ContentPage** por el siguiente código.

```
<StackLayout Spacing="0">

</StackLayout>
```

   Este será el contenedor donde todos los controles hijos serán colocados. Nota que hemos especificado que los hijos no tendrán espacio entre ellos.
4. Agrega el siguiente código para crear un botón con un enlace al comando **GetCatsCommand** del ViewModel. El comando toma el lugar de un manejador del evento **Clicked** y será ejecutado cuando el usuario toque el botón.

```
<Button Text="Sincronizar" Command="{Binding GetCatsCommand}"/>
```

   Debajo del botón podemos mostrar un indicador para informar al usuario cuando la aplicación esté obteniendo los datos del servidor. Para hacer esto, podemos utilizar un control **ActivityIndicator** y enlazarlo a la propiedad **IsBusy** del ViewModel.
5. Agrega el siguiente código para definir el control **ActivityIndicator**.

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

   Utilizaremos un **ListView** que se enlace a la colección **Cats** para mostrar todos los elementos. Podemos utilizar una propiedad especial llamada ***x.Name=””*** para nombrar cualquier control.
6. Agrega el siguiente código para definir el elemento **ListView**.

```
<ListView x:Name="ListViewCats" ItemsSource="{Binding Cats}" >

</ListView>
```

Ahora necesitamos describir la forma en que serán mostrados los elementos de la colección. Para poder hacer esto podemos utilizar un **ItemTemplate** que tenga un **DataTemplate** con una vista especifica. Xamarin.Forms contiene algunas celdas (Cells) que podemos utilizar. Utilizaremos **ImageCell** que tiene una imagen y dos renglones de texto.

7. Agrega el siguiente código XAML dentro del **ListView** para definir la forma en que serán mostrados los elementos de la colección **Cats**.

```
<ListView.ItemTemplate>
  <DataTemplate>
    <ImageCell Text="{Binding Name}"
               Detail="{Binding Price, StringFormat='{0:c} dólares'}"
               ImageSource="{Binding Image}"/>
  </DataTemplate>
</ListView.ItemTemplate>
```

8. Agrega el siguiente código dentro de la etiqueta de elemento **ContentPage** para definir un alias al espacio de nombres del ViewModel y agregar un título a la página.

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
     xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
     x:Class="Cats.Views.CatsPage"
     xmlns:VM="clr-namespace:Cats.ViewModels"
     Title="Cats">
```

9. Finalmente, agrega el siguiente código XAML para definir el contexto del enlace especificado en las propiedades de los controles agregados a la página.

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="Cats.Views.CatsPage"
             xmlns:VM="clr-namespace:Cats.ViewModels"
             Title="Cats">
  <ContentPage.BindingContext>
    <VM:CatsViewModel/>
  </ContentPage.BindingContext>
```

Xamarin.Forms automáticamente cargará y mostrará la imagen desde el servidor.

Tarea 6. Probar la aplicación.

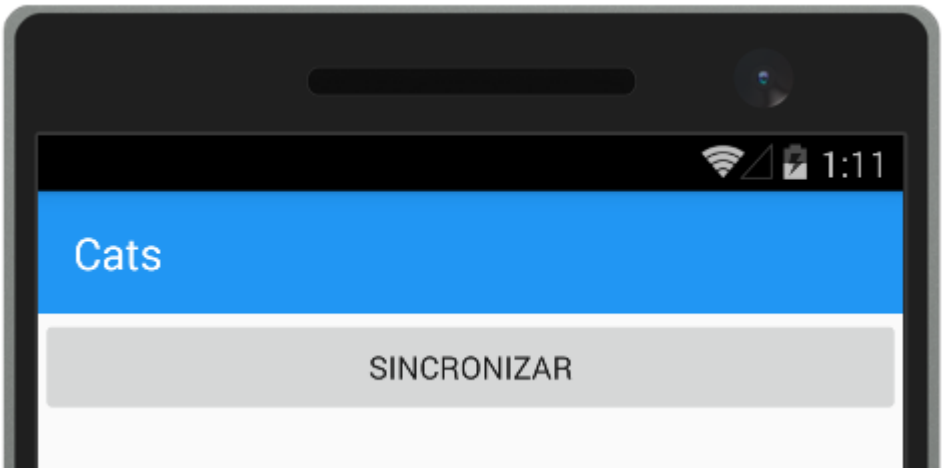
Antes de probar la aplicación tenemos que hacer algunas modificaciones.

1. Abre el archivo **App.cs** del proyecto PCL.
2. Remplaza el código del constructor por el siguiente.

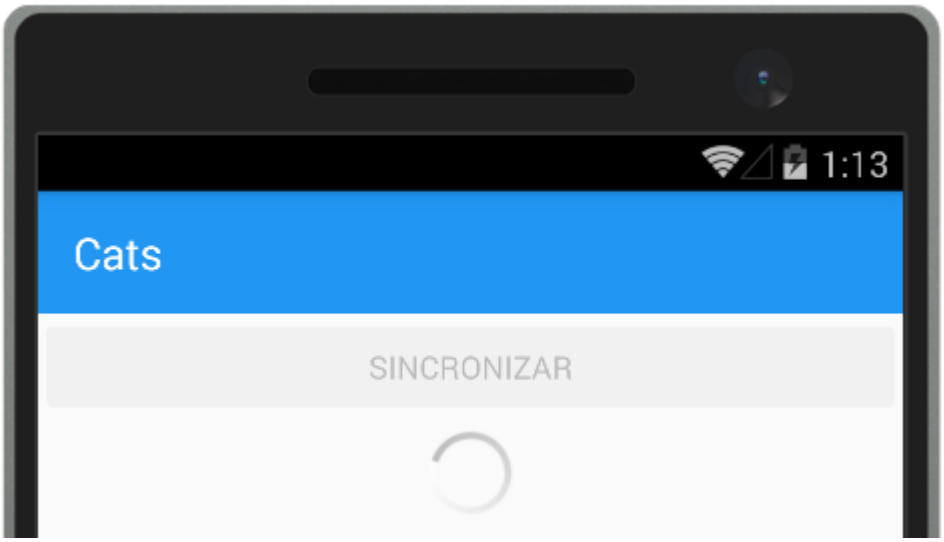
```
public App()
{
    // The root page of your application
    var content = new Views.CatsPage();

    MainPage = new NavigationPage(content);

}
```

Este código es el punto de entrada de la aplicación. El código simplemente crea una instancia de **CatsPage** y la envuelve en una página de navegación para que sea mostrada al usuario.

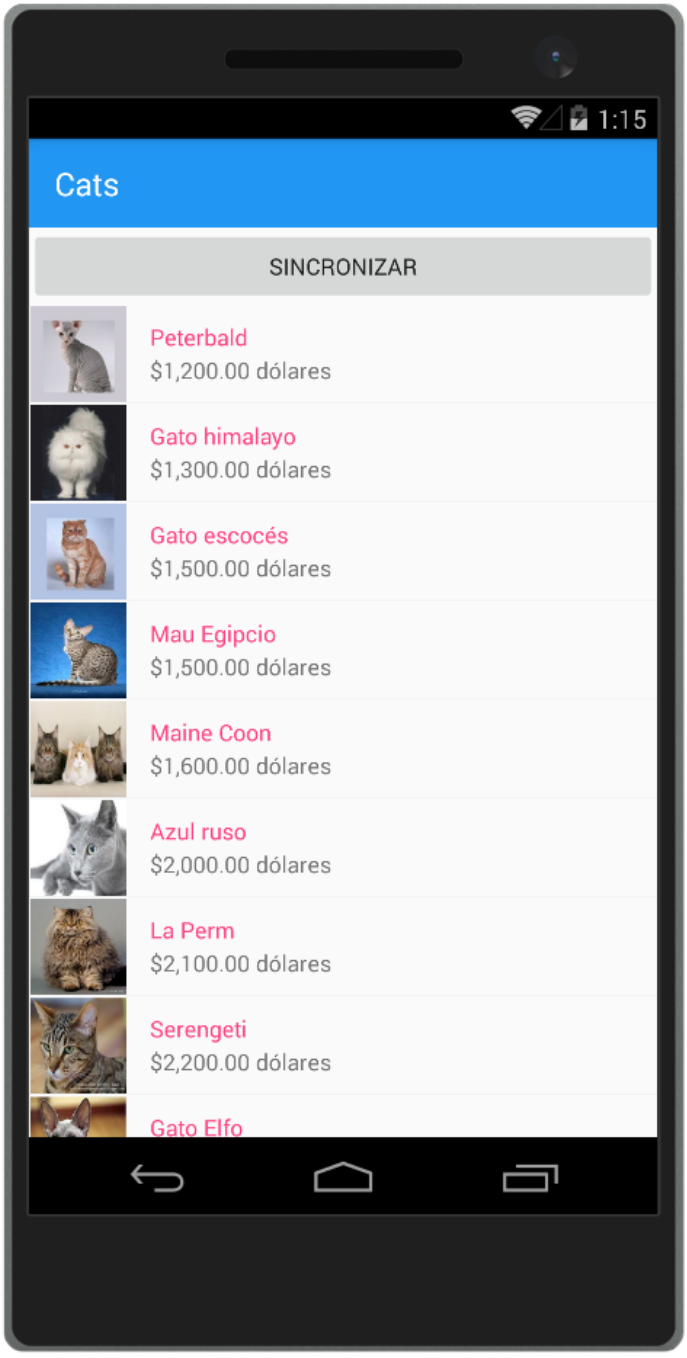
3. Selecciona el proyecto Android como proyecto de inicio.
4. Ejecuta la aplicación en el emulador de tu preferencia. Se mostrará una pantalla similar a la siguiente.



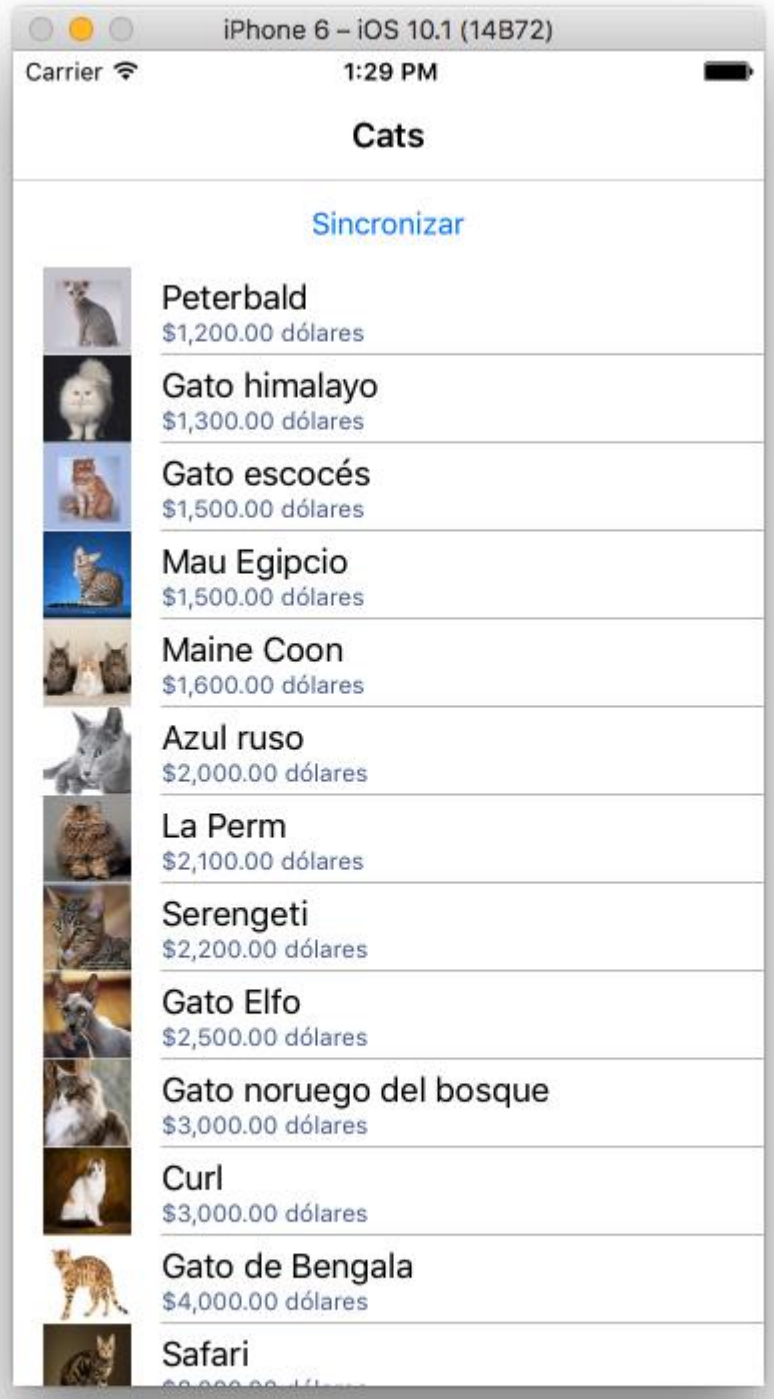
5. Toca el botón sincronizar. El indicador será mostrado.



Después de cargar los datos desde el servicio Web, se mostrará una pantalla similar a la siguiente.



6. Detén la aplicación y regresa a Visual Studio.
7. Prueba la aplicación en las demás plataformas. La siguiente imagen muestra la aplicación ejecutándose en el emulador de iOS.

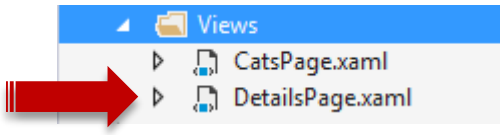


La siguiente imagen muestra la aplicación ejecutándose en un emulador Windows 10 Mobile.

Tarea 7. Agregar la página de detalles.

Ahora es tiempo de realizar algo de navegación y mostrar algunos detalles de los datos.

1. Dentro del directorio **Views** agrega un nuevo elemento **Forms Xaml Page** llamado **DetailsPage.xaml**. Esta página permitirá mostrar el detalle de un elemento de la lista seleccionado por el usuario.



Al igual que con la página **CatsPage**, utilizaremos un **StackLayout** pero lo pondremos dentro den un **ScrollView** por si hay demasiado texto a desplegar.

2. Remplaza el elemento **Label** por el siguiente código.

```
<ScrollView Padding="10">
  <StackLayout Spacing="10">

  </StackLayout>
</ScrollView>
```

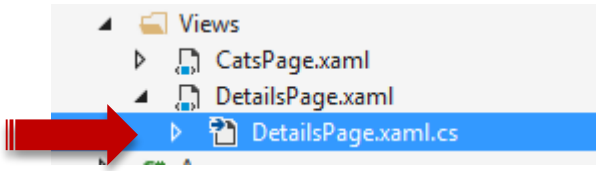
3. Dentro del **StackLayout** agrega ahora el siguiente código para definir controles y enlaces para las propiedades del objeto **Cat**.

```
<Image Source="{Binding Image}" HeightRequest="200" WidthRequest="200"/>
<Label Text="{Binding Name}" FontSize="24"/>
<Label Text="{Binding Price, StringFormat='{0:c} dólares'}" TextColor="Red"/>
<Label Text="{Binding Description}" />
```

4. Agrega el siguiente código para para permitir al usuario navegar al sitio Web del elemento seleccionado.

```
<Button Text="Ir al Sitio Web" x:Name="ButtonWebSite"/>
```

5. Guarda los cambios.
6. Abre el archivo code-behind de **DetailsPage.xaml** llamado **DetailsPage.xaml.cs**.



7. Modifica el constructor de la clase para que acepte como parámetro un objeto **Cat** que representa al objeto seleccionado por el usuario.

```
public DetailsPage(Models.Cat selectedCat)
{
    InitializeComponent();

}
```

8. Agrega el siguiente código dentro de la clase para declarar una variable que almacene los datos del objeto **Cat** seleccionado.

```
Models.Cat SelectedCat;
```

9. En el constructor de la clase, agrega el siguiente código para almacenar los datos del elemento seleccionado y definir el contexto de enlace de la página.

```
Models.Cat SelectedCat;
public DetailsPage(Models.Cat selectedCat)
{
    InitializeComponent();
    this.SelectedCat = selectedCat;
    BindingContext = this.SelectedCat;
 }
```

10. Xamarin.Forms tiene predefinidas algunas APIs interesantes para funcionalidad multiplataforma, tal como abrir un URL en el navegador predeterminado.

    En el constructor de la clase, agrega el siguiente código para definir un manejador del evento **Clicked** del botón **ButtonWebSite**.

```
ButtonWebSite.Clicked += ButtonWebSite_Clicked;
```

11. Agrega el siguiente código para implementar el manejador del evento **Clicked** utilizando la clase **Device** para invocar al método **OpenUri**.

```
private void ButtonWebSite_Clicked(object sender, EventArgs e)
{
    if (SelectedCat.WebSite.StartsWith("http"))
    {
        Device.OpenUri(new Uri(SelectedCat.WebSite));
    }
}
```

12. Abre el archivo code-behind de **CatsPage.xaml** llamado **CatsPage.xaml.cs**.
13. Agrega el siguiente código dentro del constructor de la clase para definir un manejador del evento **ItemSelected** del control **ListViewCats**. Esto nos permitirá ser informados cuando un elemento de la lista sea seleccionado.

```
public CatsPage()
{
    InitializeComponent();
    ListViewCats.ItemSelected += ListViewCats_ItemSelected;
}
```

14. Agrega el siguiente código para implementar el manejador del evento que permita navegar a la página **DetailsPage**.

```
private async void ListViewCats_ItemSelected(object sender,
    SelectedItemChangedEventArgs e)
{
```

```
        var SelectedCat = e.SelectedItem as Models.Cat;
        if (SelectedCat != null)
        {
            await Navigation.PushAsync(new Views.DetailsPage(SelectedCat));
            ListViewCats.SelectedItem = null;
        }
    }
```

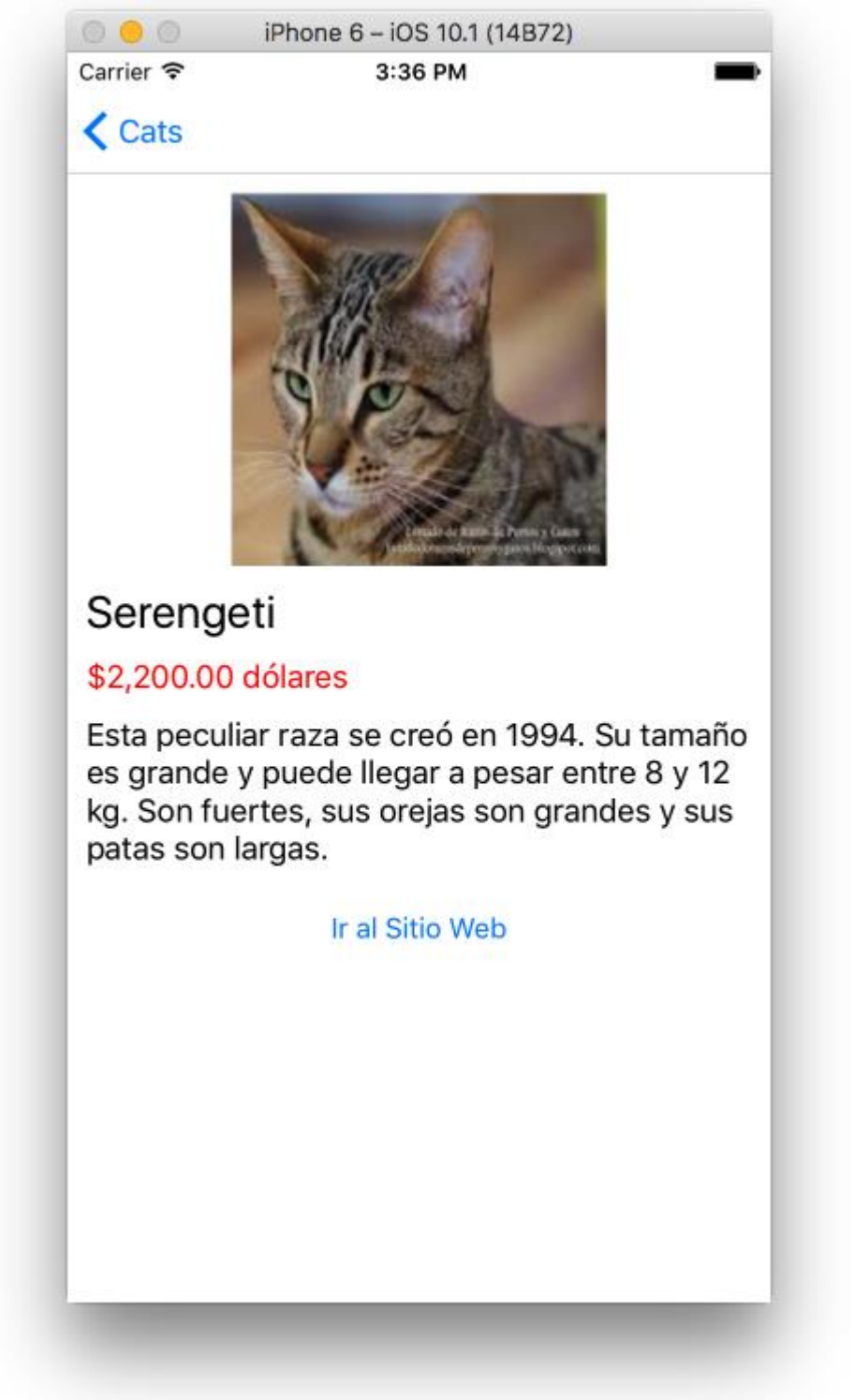
El código verifica si hay un elemento seleccionado y después utiliza la API predefinida **Navigation** para poner (push) una nueva página. Finalmente, el código quita la selección del elemento.

15. Ejecuta nuevamente la aplicación en alguno de los emuladores de la plataforma Android.
16. Toca el botón sincronizar.
17. Toca uno de los elementos. Nota que la página **DetailsPage** aparece mostrando el detalle del elemento seleccionado.

18. Toca el botón **Ir al Sitio Web**. Se mostrará la página Web solicitada.

19. Detén la aplicación y regresa a Visual Studio.

20. Prueba la aplicación en las demás plataformas. La siguiente imagen muestra la aplicación ejecutándose en el emulador de iOS.



La siguiente imagen muestra la aplicación ejecutándose en un emulador Windows 10 Mobile.


Serengeti
$2,200.00 dólares
Esta peculiar raza se creó en 1994. Su tamaño es grande y puede llegar a pesar entre 8 y 12 kg. Son fuertes, sus orejas son grandes y sus patas son largas.
Ir al Sitio Web

Resumen

En este laboratorio desarrollaste una aplicación Xamarin.Forms implementando el patrón MVVM.

La aplicación consume un servicio Web RESTful utilizando la clase **HttpClient**.

En el siguiente laboratorio modificarás el repositorio de datos para consumir los datos desde una aplicación backend hospedada en un **Azure App Service** de Microsoft Azure.

Cuando hayas finalizado este laboratorio publica el siguiente mensaje en Twitter y Facebook:

¡He finalizado el #Lab04 del #XamarinDiplomado y conozco la forma de consumir un servicio Web RESTful desde aplicaciones Xamarin.Forms implementando el patrón MVVM!