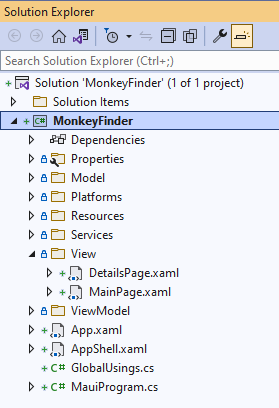
Displaying Data

Open Solution in Visual Studio

1. Open **Part 1 - Displaying Data/MonkeyFinder.sln**

This MonkeyFinder contains 1 project:

* MonkeyFinder - The main .NET MAUI project that targets Android, iOS, macOS, and Windows. It includes all scaffolding for the app including Models, Views, ViewModels, and Services.

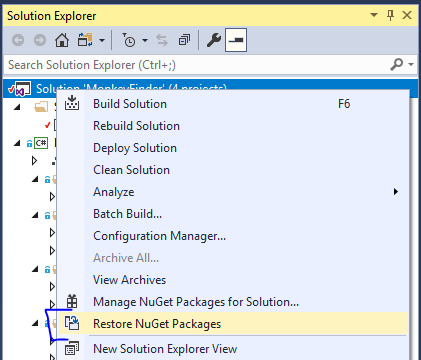


The **MonkeyFinder** project also has blank code files and XAML pages that we will use during the workshop. All of the code that we modify will be in this project for the workshop.

NuGet Restore

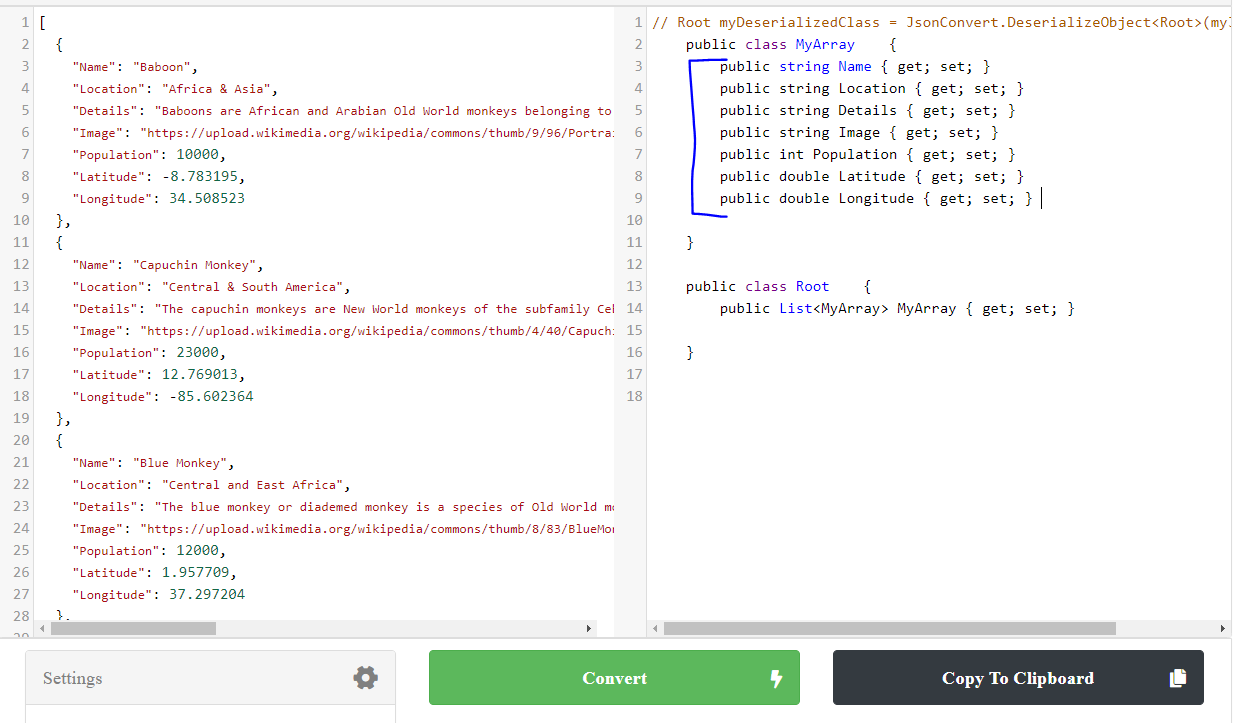
All projects have the required NuGet packages already installed, so there will be no need to install additional packages during the Hands on Lab. The first thing that we must do is restore all of the NuGet packages from the internet.

1. **Right-click** on the **Solution** and select **Restore NuGet packages...**



Model

We will be downloading details about the monkey and will need a class to represent it.



We can easily convert our json file located at [montemagno.com/monkeys.json](https://montemagno.com/monkeys.json) by using [json2csharp.com](https://json2csharp.com/) and pasting the raw json into quicktype to generate our C# classes. Ensure that you set the Name to Monkey and the generated namespace to MonkeyFinder.Model and select C#.

1. Open Model/Monkey.cs
2. In Monkey.cs, copy/paste the properties:

public class Monkey

{

public string Name { get; set; }

public string Location { get; set; }

public string Details { get; set; }

public string Image { get; set; }

public int Population { get; set; }

public double Latitude { get; set; }

public double Longitude { get; set; }

}

Additionally, because we will be using System.Text.Json to deserialize the data, we will want to add a MonkeyContext that will dynamically generate code for better performance. The following code will enable this and we will use it in the future.

[JsonSerializable(typeof(List<Monkey>))]

internal sealed partial class MonkeyContext : JsonSerializerContext

{

}

Displaying Data

We can display hard coded data of any data type in a CollectionView in our MainPage.xaml. This will allow us to build out our user interface by setting the ItemTemplate with some simple images and labels.

We first need to add a new namespace at the top of the MainPage.xaml:

xmlns:model="clr-namespace:MonkeyFinder.Model"

This will allow us to reference the Monkey class above for data binding purposes.

Add the following into the MainPage.xaml's ContentPage:

<CollectionView>

<CollectionView.ItemsSource>

<x:Array Type="{x:Type model:Monkey}">

<model:Monkey

Name="Baboon"

Image="https://raw.githubusercontent.com/jamesmontemagno/app-monkeys/master/baboon.jpg"

Location="Africa and Asia" />

<model:Monkey

Name="Capuchin Monkey"

Image="https://raw.githubusercontent.com/jamesmontemagno/app-monkeys/master/capuchin.jpg"

Location="Central and South America" />

<model:Monkey

Name="Red-shanked douc"

Image="https://raw.githubusercontent.com/jamesmontemagno/app-monkeys/master/douc.jpg"

Location="Vietnam" />

</x:Array>

</CollectionView.ItemsSource>

<CollectionView.ItemTemplate>

<DataTemplate x:DataType="model:Monkey">

<HorizontalStackLayout Padding="10">

<Image

Aspect="AspectFill"

HeightRequest="100"

Source="{Binding Image}"

WidthRequest="100" />

<Label VerticalOptions="Center" TextColor="Gray">

<Label.Text>

<MultiBinding StringFormat="{}{0} | {1}">

<Binding Path="Name" />

<Binding Path="Location" />

</MultiBinding>

</Label.Text>

</Label>

</HorizontalStackLayout>

</DataTemplate>

</CollectionView.ItemTemplate>

</CollectionView>

If we wanted to display the two strings vertically on top of each other, we could wrap two Label controls inside of a VerticalStackLayout and assign font sizes to stand out:

<HorizontalStackLayout Padding="10">

<Image

Aspect="AspectFill"

HeightRequest="100"

Source="{Binding Image}"

WidthRequest="100" />

<VerticalStackLayout VerticalOptions="Center">

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

<Label Text="{Binding Location}" FontSize="18" TextColor="Gray"/>

</VerticalStackLayout>

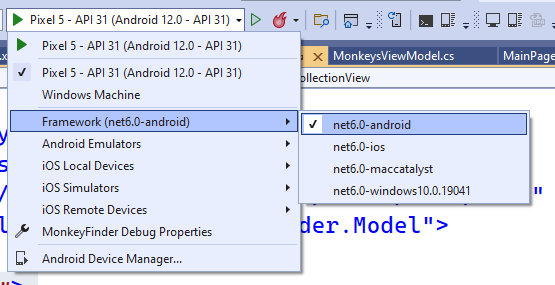
</HorizontalStackLayout>

Run the App

Ensure that you have your machine setup to deploy and debug to the different platforms:

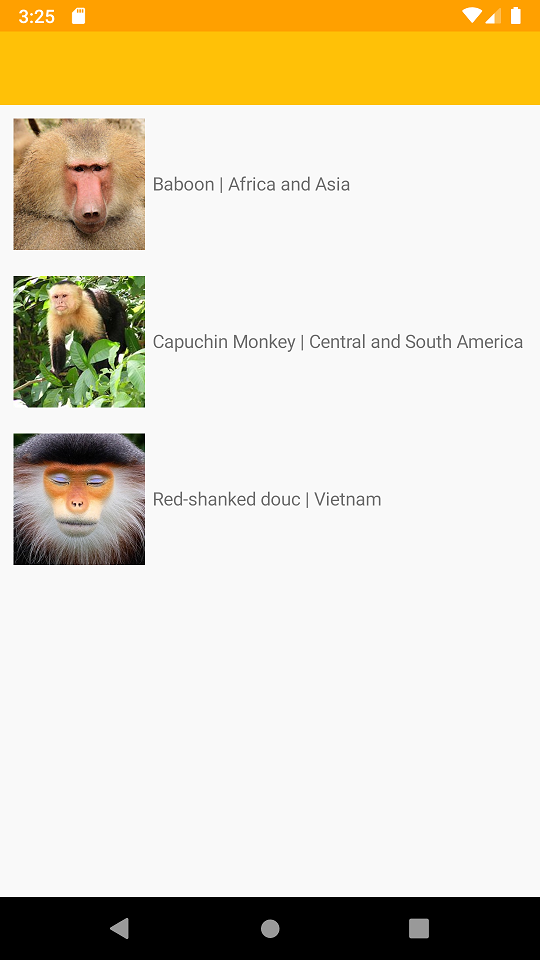
* [Android Emulator Setup](https://docs.microsoft.com/dotnet/maui/android/emulator/device-manager)
* [Windows setup for development](https://docs.microsoft.com/dotnet/maui/windows/setup)

1. In Visual Studio, set the Android or Windows app as the startup project by selecting the drop down in the debug menu and changing the Framework



1. In Visual Studio, click the "Debug" button or Tools -> Start Debugging
   * If you are having any trouble, see the Setup guides for your runtime platform

Running the app will result in a list of three monkeys:



## MVVM & Data Binding

### Implementing INotifyPropertyChanged

INotifyPropertyChanged is important for data binding in MVVM Frameworks. This is an interface that when implemented, lets our view know about changes to the model. We will implement it once in our BaseViewModel so all other view models that we create can inherit from it.

1. In Visual Studio, open ViewModel/BaseViewModel.cs
2. In BaseViewModel.cs, implement INotifyPropertyChanged by changing this

public class BaseViewModel

{

}

to this

public class BaseViewModel : INotifyPropertyChanged

{

}

1. In BaseViewModel.cs, right click on INotifyPropertyChanged
2. Implement the INotifyPropertyChanged Interface
   * (Visual Studio Mac) In the right-click menu, select Quick Fix -> Implement Interface
   * (Visual Studio PC) In the right-click menu, select Quick Actions and Refactorings -> Implement Interface
3. In BaseViewModel.cs, ensure this line of code now appears:

public event PropertyChangedEventHandler PropertyChanged;

1. In BaseViewModel.cs, create a new method called OnPropertyChanged
   * Note: We will call OnPropertyChanged whenever a property updates

public void OnPropertyChanged([CallerMemberName] string name = null) =>

PropertyChanged?.Invoke(this, new PropertyChangedEventArgs(name));

### Implementing Title, IsBusy, and IsNotBusy

We will create a backing field and accessors for a few properties. These properties will allow us to set the title on our pages and also let our view know that our view model is busy so we don't perform duplicate operations (like allowing the user to refresh the data multiple times). They are in the BaseViewModel because they are common for every page.

1. In BaseViewModel.cs, create the backing field:

public class BaseViewModel : INotifyPropertyChanged

{

bool isBusy;

string title;

//...

}

1. Create the properties:

public class BaseViewModel : INotifyPropertyChanged

{

//...

public bool IsBusy

{

get => isBusy;

set

{

if (isBusy == value)

return;

isBusy = value;

OnPropertyChanged();

}

}

public string Title

{

get => title;

set

{

if (title == value)

return;

title = value;

OnPropertyChanged();

}

}

//...

}

Notice that we call OnPropertyChanged when the value changes. The .NET MAUI binding infrastructure will subscribe to our **PropertyChanged** event so the UI will be notified of the change.

We can also create the inverse of IsBusy by creating another property called IsNotBusy that returns the opposite of IsBusy and then raising the event of OnPropertyChanged when we set IsBusy

public class BaseViewModel : INotifyPropertyChanged

{

//...

public bool IsBusy

{

get => isBusy;

set

{

if (isBusy == value)

return;

isBusy = value;

OnPropertyChanged();

// Also raise the IsNotBusy property changed

OnPropertyChanged(nameof(IsNotBusy));

}

}

public bool IsNotBusy => !IsBusy;

//...

}

### Simplifying MVVM with .NET Community Toolkit

Now that you have an understanding of how MVVM works, let's look at a way to simplify development. As applications get more complex, more properties and events will be added. This leads to more boilerplate code being added. The .NET Community Toolkit seeks to simplify MVVM with source generators to automatically handle the code that we used to manually had to write. The CommunityToolkit.Mvvm library has been added to the project and we can start using it right away.

Delete all contents in BaseViewModel.cs and replace it with the following:

namespace MonkeyFinder.ViewModel;

public partial class BaseViewModel : ObservableObject

{

[ObservableProperty]

[NotifyPropertyChangedFor(nameof(IsNotBusy))]

bool isBusy;

[ObservableProperty]

string title;

public bool IsNotBusy => !IsBusy;

}

Here, we can see that our code has been greatly simplified with an ObservableObject base class that implements INotifyPropertyChanged and also attributes to expose our properties.

Note that both isBusy and title have the [ObservableProperty] attribute attached to it. The code that is generated looks nearly identical to what we manually wrote. Additionally, the isBusy property has [NotifyPropertyChangedFor(nameof(IsNotBusy))], which will also notify IsNotBusy when the value changes. To see the generated code head to the project and then expand **Dependencies -> net8.0-android -> Analyzers -> CommunityToolkit.Mvvm.SourceGenerators -> CommunityToolkit.Mvvm.SourceGenerators.ObservablePropertyGenerator** and open MonkeyFinder.ViewModel.BaseViewModel.cs:

Here is what our IsBusy looks like:

[global::System.CodeDom.Compiler.GeneratedCode("CommunityToolkit.Mvvm.SourceGenerators.ObservablePropertyGenerator", "8.0.0.0")]

[global::System.Diagnostics.DebuggerNonUserCode]

[global::System.Diagnostics.CodeAnalysis.ExcludeFromCodeCoverage]

public bool IsBusy

{

get => isBusy;

set

{

if (!global::System.Collections.Generic.EqualityComparer<bool>.Default.Equals(isBusy, value))

{

OnPropertyChanging(global::CommunityToolkit.Mvvm.ComponentModel.\_\_Internals.\_\_KnownINotifyPropertyChangingArgs.IsBusy);

isBusy = value;

OnPropertyChanged(global::CommunityToolkit.Mvvm.ComponentModel.\_\_Internals.\_\_KnownINotifyPropertyChangedArgs.IsBusy);

OnPropertyChanged(global::CommunityToolkit.Mvvm.ComponentModel.\_\_Internals.\_\_KnownINotifyPropertyChangedArgs.IsNotBusy);

}

}

}

This code may look a bit scary, but since it is auto-generated it adds additional attributes to avoid conflicts. It is also highly optimized with caching as well.

The same library will also help us handle click events aka Commands in the future.

Note that we changed this class to a partial class so the generated code can be shared in the class.

### Create a Monkey Service

We are ready to create a method that will retrieve the monkey data from the internet. We will first implement this with a simple HTTP request using HttpClient. We will do this inside of our MonkeyService.cs file that is located in the Services folder.

1. Inside of the MonkeyService.cs, let's add a new method to get all Monkeys:

List<Monkey> monkeyList = new ();

public async Task<List<Monkey>> GetMonkeys()

{

return monkeyList;

}

Right now, the method simply creates a new list of Monkeys and returns it. We can now fill in the method use HttpClient to pull down a json file, parse it, cache it, and return it.

1. Let's get access to an HttpClient by added into the contructor for the MonkeyService.

HttpClient httpClient;

public MonkeyService()

{

this.httpClient = new HttpClient();

}

.NET MAUI includes dependency injection similar to ASP.NET Core. We will register this service and dependencies soon.

1. Let's check to see if we have any monkeys in the list and return it if so by filling in the GetMonkeys method:

if (monkeyList?.Count > 0)

return monkeyList;

1. We can use the HttpClient to make a web request and parse it using the built in System.Text.Json deserialization.

var response = await httpClient.GetAsync("https://www.montemagno.com/monkeys.json");

if (response.IsSuccessStatusCode)

{

monkeyList = await response.Content.ReadFromJsonAsync(MonkeyContext.Default.ListMonkey);

}

return monkeyList;

1. Add the following using directive at the top of the file to access the ReadFromJsonAsync extension method:

using System.Net.Http.Json;

#### **No Internet? No Problem!**

If you have internet issues in your current setup don't worry as we have embedded a list of monkeys into the project. Instead of using HttpClient, you can read the file and return it:

using var stream = await FileSystem.OpenAppPackageFileAsync("monkeydata.json");

using var reader = new StreamReader(stream);

var contents = await reader.ReadToEndAsync();

monkeyList = JsonSerializer.Deserialize(contents, MonkeyContext.Default.ListMonkey);

### Call MonkeyService from ViewModel

We now can update our MonkeysViewModel to call our new monkey service and expose the list of monkeys to our user interface.

We will use an ObservableCollection<Monkey> that will be cleared and then loaded with **Monkey** objects. We use an ObservableCollection because it has built-in support to raise CollectionChanged events when we Add or Remove items from the collection. This means we don't call OnPropertyChanged when updating the collection.

1. In MonkeysViewModel.cs declare a property which we will initialize to an empty collection. Also, we can set our Title to Monkey Finder.

public partial class MonkeysViewModel : BaseViewModel

{

public ObservableCollection<Monkey> Monkeys { get; } = new();

public MonkeysViewModel()

{

Title = "Monkey Finder";

}

}

1. We will want to access our new MonkeyService. So let's add the following using directive to the top of the file:

using MonkeyFinder.Services;

1. We also need access to our MonkeyService, which we will inject through the constructor:

public ObservableCollection<Monkey> Monkeys { get; } = new();

MonkeyService monkeyService;

public MonkeysViewModel(MonkeyService monkeyService)

{

Title = "Monkey Finder";

this.monkeyService = monkeyService;

}

1. In MonkeysViewModel.cs, create a method named GetMonkeysAsync that returns async Task:

public class MonkeysViewModel : BaseViewModel

{

//...

async Task GetMonkeysAsync()

{

}

//...

}

1. In GetMonkeysAsync, first ensure IsBusy is false. If it is true, return

async Task GetMonkeysAsync()

{

if (IsBusy)

return;

}

1. In GetMonkeysAsync, add some scaffolding for try/catch/finally blocks
   * Notice, that we toggle IsBusy to true and then false when we start to call to the server and when we finish.

async Task GetMonkeysAsync()

{

if (IsBusy)

return;

try

{

IsBusy = true;

}

catch (Exception ex)

{

}

finally

{

IsBusy = false;

}

}

1. In the try block of GetMonkeysAsync, we can get the monkeys from our MonkeyService.

async Task GetMonkeysAsync()

{

//...

try

{

IsBusy = true;

var monkeys = await monkeyService.GetMonkeys();

}

//...

}

1. Still inside of the try block, clear the Monkeys property and then add the new monkey data:

async Task GetMonkeysAsync()

{

//...

try

{

IsBusy = true;

var monkeys = await monkeyService.GetMonkeys();

if(Monkeys.Count != 0)

Monkeys.Clear();

foreach (var monkey in monkeys)

Monkeys.Add(monkey);

}

//...

}

1. In GetMonkeysAsync, add this code to the catch block to display a popup if the data retrieval fails:

async Task GetMonkeysAsync()

{

//...

catch(Exception ex)

{

Debug.WriteLine($"Unable to get monkeys: {ex.Message}");

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

}

//...

}

1. Ensure the completed code looks like this:

async Task GetMonkeysAsync()

{

if (IsBusy)

return;

try

{

IsBusy = true;

var monkeys = await monkeyService.GetMonkeys();

if(Monkeys.Count != 0)

Monkeys.Clear();

foreach(var monkey in monkeys)

Monkeys.Add(monkey);

}

catch (Exception ex)

{

Debug.WriteLine($"Unable to get monkeys: {ex.Message}");

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

}

finally

{

IsBusy = false;

}

}

1. Finally, let's expose this method via an ICommand that we can data bind to. Normally, we would have to create a backing field such as:

public Command GetMonkeysCommand { get; }

public MonkeysViewModel()

{

//...

GetMonkeysCommand = new Command(async () => await GetMonkeysAsync());

}

However, with the .NET Community Toolkit we simply can add the [RelayCommand] attribute to our method:

[RelayCommand]

async Task GetMonkeysAsync()

{

//..

}

This will automatically create all of the code we need:

// <auto-generated/>

namespace MonkeyFinder.ViewModel

{

partial class MonkeysViewModel

{

/// <summary>The backing field for <see cref="GetMonkeysASyncCommand"/>.</summary>

[global::System.CodeDom.Compiler.GeneratedCode("CommunityToolkit.Mvvm.SourceGenerators.RelayCommandGenerator", "8.0.0.0")]

private global::CommunityToolkit.Mvvm.Input.AsyncRelayCommand? getMonkeysASyncCommand;

/// <summary>Gets an <see cref="global::CommunityToolkit.Mvvm.Input.IAsyncRelayCommand"/> instance wrapping <see cref="GetMonkeysASync"/>.</summary>

[global::System.CodeDom.Compiler.GeneratedCode("CommunityToolkit.Mvvm.SourceGenerators.RelayCommandGenerator", "8.0.0.0")]

[global::System.Diagnostics.CodeAnalysis.ExcludeFromCodeCoverage]

public global::CommunityToolkit.Mvvm.Input.IAsyncRelayCommand GetMonkeysASyncCommand => getMonkeysASyncCommand ??= new global::CommunityToolkit.Mvvm.Input.AsyncRelayCommand(new global::System.Func<global::System.Threading.Tasks.Task>(GetMonkeysASync));

}

}

MAGIC!

Our main method for getting data is now complete!

### Register Services

Before we can run the app, we must register all of our dependencies. Open the MauiProgram.cs file.

1. Add the following using directive to access our MonkeyService:
2. using MonkeyFinder.Services;
3. Find where we are registering our MainPage with builder.Services and add the following above it:
4. builder.Services.AddSingleton<MonkeyService>();
5. builder.Services.AddSingleton<MonkeysViewModel>();

We are registering the MonkeyService and MonkeysViewModel as singletons. This means they will only be created once, if we wanted a unique instance to be created each request we would register them as Transient.

1. In the code behind for the project we will inject our MonkeysViewModel into our MainPage:

public MainPage(MonkeysViewModel viewModel)

{

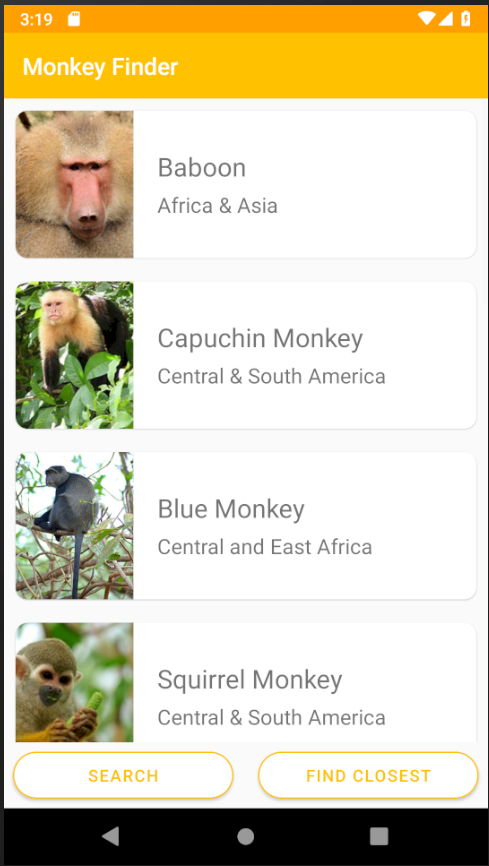
InitializeComponent();

BindingContext = viewModel;

}

## Build The Monkeys User Interface

It is now time to build the .NET MAUI user interface in View/MainPage.xaml. Our end result is to build a page that looks like this:



1. In MainPage.xaml, add a xmlns:viewmodel namespace and a x:DataType at the top of the ContentPage tag, which will enable us to get binding intellisense:

<ContentPage

xmlns="http://schemas.microsoft.com/dotnet/2021/maui"

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

x:Class="MonkeyFinder.View.MainPage"

xmlns:model="clr-namespace:MonkeyFinder.Model"

xmlns:viewmodel="clr-namespace:MonkeyFinder.ViewModel"

x:DataType="viewmodel:MonkeysViewModel">

</ContentPage>

This is called a compiled binding. We are specifying that we will be binding directly to the MonkeysViewModel. This will do error checking and has performance enhancements.

1. We can create our first binding on the ContentPage by adding the Title Property:

<ContentPage

xmlns="http://schemas.microsoft.com/dotnet/2021/maui"

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

x:Class="MonkeyFinder.View.MainPage"

xmlns:model="clr-namespace:MonkeyFinder.Model"

xmlns:viewmodel="clr-namespace:MonkeyFinder.ViewModel"

x:DataType="viewmodel:MonkeysViewModel"

Title="{Binding Title}">

</ContentPage>

1. In the MainPage.xaml, we can add a Grid between the ContentPage tags with 2 rows and 2 columns. We will also set the RowSpacing and ColumnSpacing to

<ContentPage

xmlns="http://schemas.microsoft.com/dotnet/2021/maui"

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

x:Class="MonkeyFinder.View.MainPage"

xmlns:model="clr-namespace:MonkeyFinder.Model"

xmlns:viewmodel="clr-namespace:MonkeyFinder.ViewModel"

x:DataType="viewmodel:MonkeysViewModel"

Title="{Binding Title}">

<!-- Add this -->

<Grid

ColumnDefinitions="\*,\*"

ColumnSpacing="5"

RowDefinitions="\*,Auto"

RowSpacing="0">

</Grid>

</ContentPage>

1. In the MainPage.xaml, we can add a CollectionView between the Grid tags that spans 2 Columns. We will also set the ItemsSource which will bind to our Monkeys ObservableCollection and additionally set a few properties for optimizing the list.

<ContentPage

xmlns="http://schemas.microsoft.com/dotnet/2021/maui"

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

x:Class="MonkeyFinder.View.MainPage"

xmlns:model="clr-namespace:MonkeyFinder.Model"

xmlns:viewmodel="clr-namespace:MonkeyFinder.ViewModel"

x:DataType="viewmodel:MonkeysViewModel"

Title="{Binding Title}">

<!-- Add this -->

<Grid

ColumnDefinitions="\*,\*"

ColumnSpacing="5"

RowDefinitions="\*,Auto"

RowSpacing="0">

<CollectionView ItemsSource="{Binding Monkeys}"

SelectionMode="None"

Grid.ColumnSpan="2">

</CollectionView>

</Grid>

</ContentPage>

1. In the MainPage.xaml, we can add a ItemTemplate to our CollectionView that will represent what each item in the list displays:

<ContentPage

xmlns="http://schemas.microsoft.com/dotnet/2021/maui"

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

x:Class="MonkeyFinder.View.MainPage"

xmlns:model="clr-namespace:MonkeyFinder.Model"

xmlns:viewmodel="clr-namespace:MonkeyFinder.ViewModel"

x:DataType="viewmodel:MonkeysViewModel"

Title="{Binding Title}">

<Grid

ColumnDefinitions="\*,\*"

ColumnSpacing="5"

RowDefinitions="\*,Auto"

RowSpacing="0">

<CollectionView ItemsSource="{Binding Monkeys}"

SelectionMode="None"

Grid.ColumnSpan="2">

<CollectionView.ItemTemplate>

<DataTemplate x:DataType="model:Monkey">

<Grid Padding="10">

<Frame HeightRequest="125" Style="{StaticResource CardView}">

<Grid Padding="0" ColumnDefinitions="125,\*">

<Image Aspect="AspectFill" Source="{Binding Image}"

WidthRequest="125"

HeightRequest="125"/>

<VerticalStackLayout

Grid.Column="1"

VerticalOptions="Center"

Padding="10">

<Label Style="{StaticResource LargeLabel}" Text="{Binding Name}" />

<Label Style="{StaticResource MediumLabel}" Text="{Binding Location}" />

</VerticalStackLayout>

</Grid>

</Frame>

</Grid>

</DataTemplate>

</CollectionView.ItemTemplate>

</CollectionView>

</Grid>

</ContentPage>

1. In the MainPage.xaml, we can add a Button under our CollectionView that will enable us to click it and get the monkeys from the server:

<ContentPage

xmlns="http://schemas.microsoft.com/dotnet/2021/maui"

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

x:Class="MonkeyFinder.View.MainPage"

xmlns:model="clr-namespace:MonkeyFinder.Model"

xmlns:viewmodel="clr-namespace:MonkeyFinder.ViewModel"

x:DataType="viewmodel:MonkeysViewModel"

Title="{Binding Title}">

<Grid

ColumnDefinitions="\*,\*"

ColumnSpacing="5"

RowDefinitions="\*,Auto"

RowSpacing="0">

<CollectionView ItemsSource="{Binding Monkeys}"

SelectionMode="None"

Grid.ColumnSpan="2">

<CollectionView.ItemTemplate>

<DataTemplate x:DataType="model:Monkey">

<Grid Padding="10">

<Frame HeightRequest="125" Style="{StaticResource CardView}">

<Grid Padding="0" ColumnDefinitions="125,\*">

<Image Aspect="AspectFill" Source="{Binding Image}"

WidthRequest="125"

HeightRequest="125"/>

<VerticalStackLayout

Grid.Column="1"

VerticalOptions="Center"

Padding="10">

<Label Style="{StaticResource LargeLabel}" Text="{Binding Name}" />

<Label Style="{StaticResource MediumLabel}" Text="{Binding Location}" />

</VerticalStackLayout>

</Grid>

</Frame>

</Grid>

</DataTemplate>

</CollectionView.ItemTemplate>

</CollectionView>

<!-- Add this -->

<Button Text="Get Monkeys"

Command="{Binding GetMonkeysCommand}"

IsEnabled="{Binding IsNotBusy}"

Grid.Row="1"

Grid.Column="0"

Style="{StaticResource ButtonOutline}"

Margin="8"/>

</Grid>

</ContentPage>

1. Finally, In the MainPage.xaml, we can add a ActivityIndicator above all of our controls at the very bottom or Grid that will show an indication that something is happening when we press the Get Monkeys button.

<ContentPage

xmlns="http://schemas.microsoft.com/dotnet/2021/maui"

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

x:Class="MonkeyFinder.View.MainPage"

xmlns:model="clr-namespace:MonkeyFinder.Model"

xmlns:viewmodel="clr-namespace:MonkeyFinder.ViewModel"

x:DataType="viewmodel:MonkeysViewModel"

Title="{Binding Title}">

<Grid

ColumnDefinitions="\*,\*"

ColumnSpacing="5"

RowDefinitions="\*,Auto"

RowSpacing="0">

<CollectionView ItemsSource="{Binding Monkeys}"

SelectionMode="None"

Grid.ColumnSpan="2">

<CollectionView.ItemTemplate>

<DataTemplate x:DataType="model:Monkey">

<Grid Padding="10">

<Frame HeightRequest="125" Style="{StaticResource CardView}">

<Grid Padding="0" ColumnDefinitions="125,\*">

<Image Aspect="AspectFill" Source="{Binding Image}"

WidthRequest="125"

HeightRequest="125"/>

<VerticalStackLayout

Grid.Column="1"

VerticalOptions="Center"

Padding="10">

<Label Style="{StaticResource LargeLabel}" Text="{Binding Name}" />

<Label Style="{StaticResource MediumLabel}" Text="{Binding Location}" />

</VerticalStackLayout>

</Grid>

</Frame>

</Grid>

</DataTemplate>

</CollectionView.ItemTemplate>

</CollectionView>

<Button Text="Get Monkeys"

Command="{Binding GetMonkeysCommand}"

IsEnabled="{Binding IsNotBusy}"

Grid.Row="1"

Grid.Column="0"

Style="{StaticResource ButtonOutline}"

Margin="8"/>

<!-- Add this -->

<ActivityIndicator IsVisible="{Binding IsBusy}"

IsRunning="{Binding IsBusy}"

HorizontalOptions="Fill"

VerticalOptions="Center"

Color="{StaticResource Primary}"

Grid.RowSpan="2"

Grid.ColumnSpan="2"/>

</Grid>

</ContentPage>

### Run the App

1. In Visual Studio, set the iOS, Android, macOS, or Windows project as the startup project
2. In Visual Studio, click "Start Debugging". When the application starts you will see a **Get Monkeys** button that when pressed will load monkey data from the internet!

Navigation

We will use the built-in Shell navigation of .NET MAUI. This powerful navigation system is based on URIs. You can pass additional information while navigating query parameter such as a string, or a full object.

For example, let's say we wanted to navigate to a details page and pass in an identifier.

await Shell.Current.GoToAsync("DetailsPage?name=james");

Then in our details page or view model we should define this property:

[QueryProperty(nameof(Name), "name")]

public partial class DetailsPage : ContentPage

{

string name;

public string Name

{

get => name;

set => name = value;

}

}

When we navigate, the name "james" would be passed along automatically. We can also pass a full object as well using the same mechanism:

var person = new Person { Name="James" };

await Shell.Current.GoToAsync("DetailsPage", new Dictionary<string, object>

{

{ "person", person }

});

Then on our page or view model we would create the property.

[QueryProperty(nameof(Person), "person")]

public partial class DetailsPage : ContentPage

{

Person person;

public Person Person

{

get => person;

set => person = value;

}

}

Here, the Person is automatically serialized and deserialized for us when we navigate.

Now, let's add a click handler to the collection view and pass the monkey to the details page.

Add Selected Event

Now, let's add navigation to a second page that displays monkey details!

1. In MonkeysViewModel.cs, create a method async Task GoToDetailsAsync(Monkey monkey) exposed as an [RelayCommand]:

[RelayCommand]

async Task GoToDetails(Monkey monkey)

{

if (monkey == null)

return;

await Shell.Current.GoToAsync(nameof(DetailsPage), true, new Dictionary<string, object>

{

{"Monkey", monkey }

});

}

* + This code checks to see if the selected item is non-null and then uses the built in Shell Navigation API to push a new page with the monkey as a parameter and then deselects the item.

1. In MainPage.xaml we can add an TapGestureRecognizer event to the Frame of our monkey inside of the CollectionView.ItemTemplate:

Before:

<CollectionView.ItemTemplate>

<DataTemplate x:DataType="model:Monkey">

<Grid Padding="10">

<Frame HeightRequest="125" Style="{StaticResource CardView}">

<Grid Padding="0" ColumnDefinitions="125,\*">

<Image

Aspect="AspectFill"

HeightRequest="125"

Source="{Binding Image}"

WidthRequest="125" />

<VerticalStackLayout

Grid.Column="1"

VerticalOptions="Center"

Padding="10">

<Label Style="{StaticResource LargeLabel}" Text="{Binding Name}" />

<Label Style="{StaticResource MediumLabel}" Text="{Binding Location}" />

</VerticalStackLayout>

</Grid>

</Frame>

</Grid>

</DataTemplate>

</CollectionView.ItemTemplate>

After:

<CollectionView.ItemTemplate>

<DataTemplate x:DataType="model:Monkey">

<Grid Padding="10">

<Frame HeightRequest="125" Style="{StaticResource CardView}">

<!-- Add the Gesture Recognizer-->

<Frame.GestureRecognizers>

<TapGestureRecognizer

Command="{Binding Source={RelativeSource AncestorType={x:Type viewmodel:MonkeysViewModel}}, Path=GoToDetailsCommand}"

CommandParameter="{Binding .}"/>

</Frame.GestureRecognizers>

<Grid Padding="0" ColumnDefinitions="125,\*">

<Image

Aspect="AspectFill"

HeightRequest="125"

Source="{Binding Image}"

WidthRequest="125" />

<VerticalStackLayout

Grid.Column="1"

VerticalOptions="Center"

Padding="10">

<Label Style="{StaticResource LargeLabel}" Text="{Binding Name}" />

<Label Style="{StaticResource MediumLabel}" Text="{Binding Location}" />

</VerticalStackLayout>

</Grid>

</Frame>

</Grid>

</DataTemplate>

</CollectionView.ItemTemplate>

This uses a RelativeSource binding, which means that it isn't binding to the Monkey anymore in the DataTemplate, but instead it is looking up the hierarchy specifically for an AncestorType of MonkeysViewModel. This allows for more advanced scenarios like this.

ViewModel for Details

1. Inside of our ViewModel/MonkeyDetailsViewModel.cs, we will house our logic for assigning the monkey to the view model. Let's first create a bindable property for the Monkey:

public partial class MonkeyDetailsViewModel : BaseViewModel

{

public MonkeyDetailsViewModel()

{

}

[ObservableProperty]

Monkey monkey;

}

1. Next, we will add a QueryProperty to handle passing the monkey data:

//Add QueryProperty

[QueryProperty(nameof(Monkey), "Monkey")]

public partial class MonkeyDetailsViewModel : BaseViewModel

{

public MonkeyDetailsViewModel()

{

}

[ObservableProperty]

Monkey monkey;

}

Registering Routing

Now that we have our details page in place, we need to register it for routing. This is done in both the Shell routing system and with the .NET MAUI dependency service.

1. Open AppShell.xaml.cs code behind and add the following code into the constructor under the InitializeComponent(); invoke:

Routing.RegisterRoute(nameof(DetailsPage), typeof(DetailsPage));

This will register the details page with the route of "DetailsPage", which we used earlier.

1. Open MauiProgram.cs and add both the view model and the page as Transient so a new page and view model is created each time it is navigated to:

builder.Services.AddTransient<MonkeyDetailsViewModel>();

builder.Services.AddTransient<DetailsPage>();

1. Finally, we must inject the view model into our DetailsPage. Open the code behind for the page in DetailsPage.xaml.cs and change the constructor to the following:

public DetailsPage(MonkeyDetailsViewModel viewModel)

{

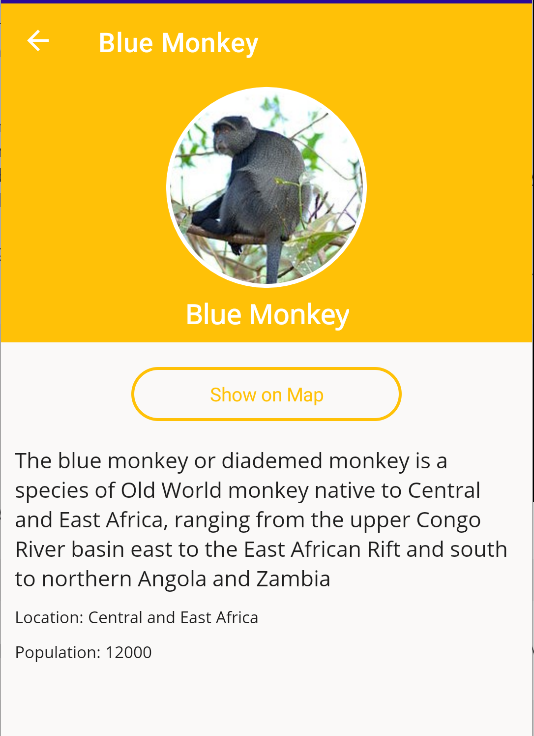
InitializeComponent();

BindingContext = viewModel;

}

Create DetailsPage.xaml UI

Let's add UI to the DetailsPage. Our end goal is to get a fancy profile screen like this:



1. Let's first start by defining our DataType by defining the view model namespace and also setting the title:

<ContentPage

xmlns="http://schemas.microsoft.com/dotnet/2021/maui"

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

x:Class="MonkeyFinder.DetailsPage"

xmlns:viewmodel="clr-namespace:MonkeyFinder.ViewModel"

x:DataType="viewmodel:MonkeyDetailsViewModel"

Title="{Binding Monkey.Name}">

<!-- Add Content Here -->

</ContentPage>

1. At the core is a ScrollView and Grid to layout all of the controls nicely on the screen:

<ScrollView>

<Grid RowDefinitions="Auto,Auto,\*">

<!-- Background, Image of Monkey, Name -->

<!-- Details of Monkey -->

</Grid>

</ScrollView>

1. We can now fill in our Grid with the following code to place a box as the background color of yellow, and then our monkey image cut out in the shape of a circle:

<BoxView

BackgroundColor="{StaticResource Primary}"

Grid.RowSpan="2"

HorizontalOptions="Fill"

VerticalOptions="Fill"/>

<Border StrokeShape="RoundRectangle 80"

Stroke="White"

StrokeThickness="6"

HeightRequest="172"

WidthRequest="172"

Margin="0,8,0,0"

HorizontalOptions="Center"

VerticalOptions="Center">

<Image Aspect="AspectFill"

HeightRequest="160"

HorizontalOptions="Center"

VerticalOptions="Center"

Source="{Binding Monkey.Image}"

WidthRequest="160"/>

</Border>

<Label Style="{StaticResource LargeLabel}"

Grid.Row="1"

TextColor="White"

FontAttributes="Bold"

Text="{Binding Monkey.Name}"

HorizontalOptions="Center"

Margin="0,0,0,8"/>

1. Finally, under the Label, but still inside of the Grid, we'll add a VerticalStackLayout to provide details about the monkey.

<VerticalStackLayout Grid.Row="2" Padding="10" Spacing="10">

<Label Style="{StaticResource MediumLabel}" Text="{Binding Monkey.Details}" />

<Label Style="{StaticResource SmallLabel}" Text="{Binding Monkey.Location, StringFormat='Location: {0}'}" />

<Label Style="{StaticResource SmallLabel}" Text="{Binding Monkey.Population, StringFormat='Population: {0}'}" />

</VerticalStackLayout>

1. Run the application on the desired platform and tap on a monkey to navigate!

Accessing Platform Features

Check for internet

We can easily check to see if our user is connected to the internet with the built in IConnectivity of .NET MAUI

1. First, let's get access to the IConnectivity found inside of .NET MAUI. Let's inject IConnectivity into our MonkeysViewModel constructor:

IConnectivity connectivity;

public MonkeysViewModel(MonkeyService monkeyService, IConnectivity connectivity)

{

Title = "Monkey Finder";

this.monkeyService = monkeyService;

this.connectivity = connectivity;

}

1. Register the Connectivity.Current in our MauiProgram.cs.
2. While we are here let's add both IGeolocation and IMap, add the code:

builder.Services.AddSingleton<IConnectivity>(Connectivity.Current);

builder.Services.AddSingleton<IGeolocation>(Geolocation.Default);

builder.Services.AddSingleton<IMap>(Map.Default);

1. Now, let's check for internet inside of the GetMonkeysAsync method and display an alert if offline.

if (connectivity.NetworkAccess != NetworkAccess.Internet)

{

await Shell.Current.DisplayAlert("No connectivity!",

$"Please check internet and try again.", "OK");

return;

}

Run the app on your emulator and toggle on and off airplane mode to check your implementation.

Find Closest Monkey!

We can add more functionality to this page using the GPS of the device since each monkey has a latitude and longitude associated with it.

1. First, let's get access to the IGeolocator found inside of .NET MAUI. Let's inject IGeolocator into our MonkeysViewModel constructor:

IConnectivity connectivity;

IGeolocation geolocation;

public MonkeysViewModel(MonkeyService monkeyService, IConnectivity connectivity, IGeolocation geolocation)

{

Title = "Monkey Finder";

this.monkeyService = monkeyService;

this.connectivity = connectivity;

this.geolocation = geolocation;

}

1. In our MonkeysViewModel.cs, let's create another method called GetClosestMonkey:

[RelayCommand]

async Task GetClosestMonkey()

{

}

1. We can then fill it in by using .NET MAUI to query for our location and helpers that find the closest monkey to us:

[RelayCommand]

async Task GetClosestMonkey()

{

if (IsBusy || Monkeys.Count == 0)

return;

try

{

// Get cached location, else get real location.

var location = await geolocation.GetLastKnownLocationAsync();

if (location == null)

{

location = await geolocation.GetLocationAsync(new GeolocationRequest

{

DesiredAccuracy = GeolocationAccuracy.Medium,

Timeout = TimeSpan.FromSeconds(30)

});

}

// Find closest monkey to us

var first = Monkeys.OrderBy(m => location.CalculateDistance(

new Location(m.Latitude, m.Longitude), DistanceUnits.Miles))

.FirstOrDefault();

await Shell.Current.DisplayAlert("", first.Name + " " +

first.Location, "OK");

}

catch (Exception ex)

{

Debug.WriteLine($"Unable to query location: {ex.Message}");

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

}

}

1. Back in our MainPage.xaml we can add another Button that will call this new method:

Add the following XAML under the Search button.

<Button Text="Find Closest"

Command="{Binding GetClosestMonkeyCommand}"

IsEnabled="{Binding IsNotBusy}"

Grid.Row="1"

Grid.Column="1"

Style="{StaticResource ButtonOutline}"

Margin="8"/>

Re-run the app to see geolocation in action after you load monkeys!

This project is pre-configured with all required permissions and features needed for Geolocation. You can read the documentation to find out more about setup, but here is a quick overview.

1. .NET MAUI is pre-configured in all .NET MAUI applications including handling permissions.
2. Android manifest information was pre-configured in **MonkeyFinder -> Platforms -> Android -> AssemblyInfo.cs**
3. iOS/macOS manifest information was configured in the **info.plist** file for each platform
4. Windows manifest information was configured in the **Package.appxmanifest**

Opening Maps

.NET MAUI provides over 60 platform features from a single API and opening the default map application is built in!

1. Inject IMap into our MonkeyDetailsViewModel:

IMap map;

public MonkeyDetailsViewModel(IMap map)

{

this.map = map;

}

1. Open the MonkeyDetailsViewModel.cs file and add a method called OpenMap that calls into the Map API passing it the monkey's location:

[RelayCommand]

async Task OpenMap()

{

try

{

await map.OpenAsync(Monkey.Latitude, Monkey.Longitude, new MapLaunchOptions

{

Name = Monkey.Name,

NavigationMode = NavigationMode.None

});

}

catch (Exception ex)

{

Debug.WriteLine($"Unable to launch maps: {ex.Message}");

await Shell.Current.DisplayAlert("Error, no Maps app!", ex.Message, "OK");

}

}

Update DetailsPage.xaml UI

Above the monkey's name, let's add a button that calls the OpenMapCommand.

<Button Text="Show on Map"

Command="{Binding OpenMapCommand}"

HorizontalOptions="Center"

WidthRequest="200"

Margin="8"

Style="{StaticResource ButtonOutline}"/>

<Label Style="{StaticResource MediumLabel}" Text="{Binding Monkey.Details}" />

Run the application, navigate to a monkey, and then press Show on Map to launch the map app on the specific platform.

iOS Safe Area Layouts

In addition to accessing cross-platform device APIs, .NET MAUI also includes platform specific integrations. If you have been running the Monkey Finder app on an iOS device with a notch, you may have noticed that the buttons on the bottom overlap the bar on the bottom of the device. iOS has the concept of Safe Areas and you must progmatically set this. However, thanks to platform specifics, you can set them directly in the XAML.

1. Open MainPage.xaml and add a new namespace for iOS specifics:

xmlns:ios="clr-namespace:Microsoft.Maui.Controls.PlatformConfiguration.iOSSpecific;assembly=Microsoft.Maui.Controls"

1. On the ContentPage node, you can now set the following property:

ios:Page.UseSafeArea="True"

Re-run the application on an iOS simulator or device and notice the buttons have automatically been shifted up.

## Adding Pull-to-Refresh

The .NET MAUI ListView has built in support for pull-to-refresh, however a RefreshView enables developers to add pull-to-refresh to other controls such as ScrollView & CollectionView.

Let's add the new RefreshView to add pull-to-refresh to our CollectionView.

Update the CollectionView logic by wrapping it with a RefreshView from:

<CollectionView

Grid.ColumnSpan="2"

ItemsSource="{Binding Monkeys}"

SelectionMode="None">

<!-- Template -->

</CollectionView>

to:

<RefreshView

Grid.ColumnSpan="2"

Command="{Binding GetMonkeysCommand}"

IsRefreshing="{Binding IsRefreshing}">

<ContentView>

<CollectionView

ItemsSource="{Binding Monkeys}"

SelectionMode="None">

<!-- Template -->

</CollectionView>

</ContentView>

</RefreshView>

Notice that we moved the Grid.ColumnSpan="2" to the RefreshView since it is the new parent view in the Grid.

Since the user can initiate a refresh, we will want to create a new variable in our code behind to bind to stop refreshing when we are done.

1. Open MonkeysViewModel.cs and add a new property:

[ObservableProperty]

bool isRefreshing;

1. In the finally of the GetMonkeysAsync set IsRefreshing to false:

finally

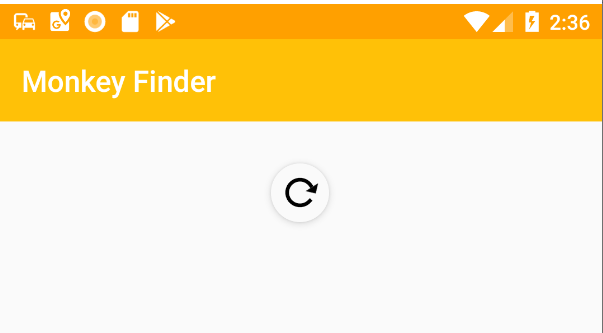
{

IsBusy = false;

IsRefreshing = false;

}

This will enable pull-to-refresh on iOS, Android, macOS, and Windows (on touch screen):



Important Note: If you are on iOS there currently is a bug which makes the UI look incorrect. It is recommended to remove the RefreshView when testing on iOS for the rest of the workshop.

## Layout

CollectionView will automatically layout items in a vertical stack layout. There are several built in ItemsLayout that can be used. Let's explore.

### LinearItemsLayout

This is the default layout that can display items in either vertical or horizontal orientations. You can set the ItemsLayout property to VerticalList or HorizontalList.

To access additional properties on the LinearItemsLayout we will need to set a sub-property:

<CollectionView

ItemsSource="{Binding Monkeys}"

SelectionMode="None">

<!-- Add ItemsLayout -->

<CollectionView.ItemsLayout>

<LinearItemsLayout Orientation="Vertical" />

</CollectionView.ItemsLayout>

<!-- ItemTemplate -->

</CollectionView>

### GridItemsLayout

More interesting is the ability to use GridItemsLayout that automatically spaces out items with different spans.

Let's use the GridItemsLayout and change the span to 3

<CollectionView

ItemsSource="{Binding Monkeys}"

SelectionMode="None">

<!-- Change ItemsLayout to GridItemsLayout-->

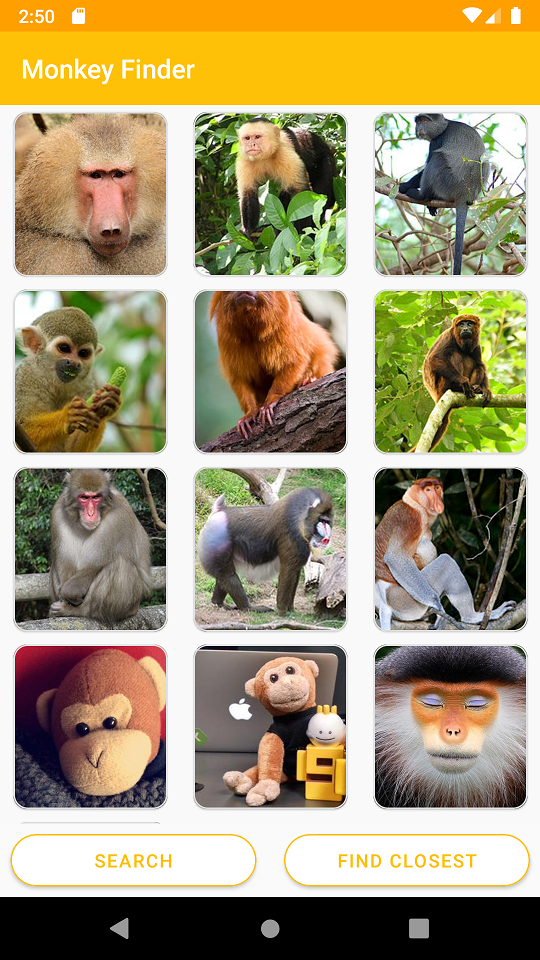
<CollectionView.ItemsLayout>

<GridItemsLayout Orientation="Vertical" Span="3" />

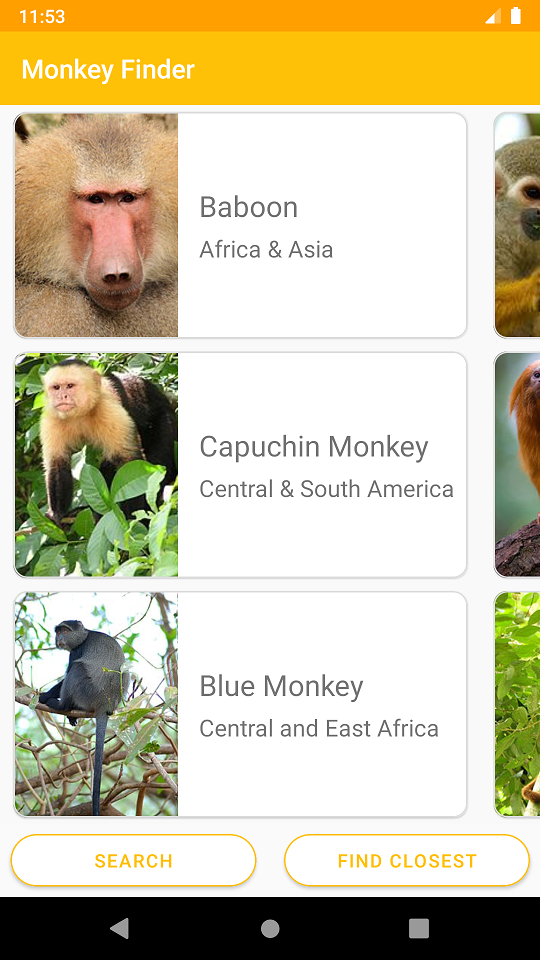
</CollectionView.ItemsLayout>

<!-- ItemTemplate -->

</CollectionView>



We can change the Orientation to Horizontal and now our CollectionView will scroll left to right!



<CollectionView.ItemsLayout>

<GridItemsLayout Orientation="Horizontal" Span="5" />

</CollectionView.ItemsLayout>

Let's go back to our original single column CollectionView:

<CollectionView.ItemsLayout>

<LinearItemsLayout Orientation="Vertical" />

</CollectionView.ItemsLayout>

## EmptyView

Important Note: There is currently an issue on Android in which the EmptyView will not go away. It is recommended to remove it when testing on Android at this time.

There are many neat features to CollectionView including grouping, header, footers, and the ability to set a view that is displayed when there are no items.



Let's add an image centered in the EmptyView:

<CollectionView

ItemsSource="{Binding Monkeys}"

SelectionMode="None">

<!-- Add EmptyView -->

<CollectionView.EmptyView>

<StackLayout Padding="100">

<Image

HorizontalOptions="Center"

Source="nodata.png"

HeightRequest="160"

WidthRequest="160"

VerticalOptions="Center" />

</StackLayout>

</CollectionView.EmptyView>

<!-- ItemTemplate & ItemsLayout-->

</CollectionView>

## App Themes

Up to this point, we have used a standard light theme on the application. .NET MAUI has the concept of reusable Application Resources and resources that can automatically adapt to the theme of the device.

## Reusable Resources

Open the App.xaml file and notice that there are several Color entries and Styles. These were configured ahead of time for some basic colors and styles that we used throughout the application. For example, we have defined a light color for the main background color:

<Color x:Key="LightBackground">#FAF9F8</Color>

It can be referenced later by any UI element or by a shared style that can be reused. For example our ButtonOutline style applies to the Button control and gives it a rounded corner, sets colors for the text, border, and background:

<Style x:Key="ButtonOutline" TargetType="Button">

<Setter Property="Background" Value="{StaticResource LightBackground}" />

<Setter Property="TextColor" Value="{StaticResource Primary}" />

<Setter Property="BorderColor" Value="{StaticResource Primary}" />

<Setter Property="BorderWidth" Value="2" />

<Setter Property="HeightRequest" Value="40" />

<Setter Property="CornerRadius" Value="20" />

</Style>

This is a great way to share code across your entire application.

## Theme Changes - Light/Dark Theme

What happens when you want to respond to the user changing their device to use dark mode? Well, .NET MAUI has the concept of an AppThemeBinding for values. Let's take a Label's TextColor property. We can define two new colors to use:

<Color x:Key="LabelText">Black</Color>

<Color x:Key="LabelTextDark">White</Color>

We would want the text to be Black when the background color is light, and White when the background color is dark. Normally, we would set the color to a single color such as:

<Label Text="Hello, world!" TextColor="{StaticResource LabelText}"/>

However, this will not adjust to app theme changes. We could make it a DynamicResource, listen for app theme changes, and update the LabelText value, or we can use an AppThemeBinding:

<Label Text="Hello, world!"

TextColor="{AppThemeBinding Light={StaticResource LabelText}, Dark={StaticResource LabelTextDark}}"/>

We now have the option of creating a re-usable style that we reference by name or a style that applies to every element of a specific type:

<Style TargetType="Label" x:Key="DefaultLabel">

<Setter Property="TextColor" Value="{AppThemeBinding Light={StaticResource LabelText}, Dark={StaticResource LabelTextDark}}" />

</Style>

<Label Text="Hello, world!"

Style="{StaticResource DefaultLabel}"/>

If we leave out the x:Key, then it will apply automatically to every Label in our app.

<Style TargetType="Label">

<Setter Property="TextColor" Value="{AppThemeBinding Light={StaticResource LabelText}, Dark={StaticResource LabelTextDark}}" />

</Style>

## Update Resources

Now, let's add in light/dark theme support throughout our entire application.

1. Let's add some new colors we will use into our ResourceDictionary:

<Color x:Key="CardBackground">White</Color>

<Color x:Key="CardBackgroundDark">#1C1C1E</Color>

<Color x:Key="LabelText">#1F1F1F</Color>

<Color x:Key="LabelTextDark">White</Color>

1. Let's update background colors on pages from:

<Style ApplyToDerivedTypes="True" TargetType="Page">

<Setter Property="BackgroundColor" Value="{StaticResource LightBackground}" />

</Style>

to:

<Style ApplyToDerivedTypes="True" TargetType="Page">

<Setter Property="BackgroundColor" Value="{AppThemeBinding Light={StaticResource LightBackground}, Dark={StaticResource DarkBackground}}" />

</Style>

1. Update the BaseLabel's TextColor value:

<Setter Property="TextColor" Value="{AppThemeBinding Light={StaticResource LabelText}, Dark={StaticResource LabelTextDark}}" />

1. Add the Background on our RefreshView

<Style ApplyToDerivedTypes="True" TargetType="RefreshView">

<Setter Property="RefreshColor" Value="{StaticResource Primary}" />

<!--Add this-->

<Setter Property="Background" Value="{AppThemeBinding Light={StaticResource LightBackground}, Dark={StaticResource DarkBackground}}" />

</Style>

1. Update the Background on the ButtonOutline

<Setter Property="Background" Value="{AppThemeBinding Light={StaticResource LightBackground}, Dark={StaticResource DarkBackground}}" />

1. Update the Background on the CardView

<Setter Property="Background" Value="{AppThemeBinding Light={StaticResource CardBackground}, Dark={StaticResource CardBackgroundDark}}" />

Now, let's run the app and change the theme:

You did it! Congratulations! You built your first .NET MAUI application, loaded data from the internet, implemented navigation, added platform features, and themed the app.