# Exploring .NET MAUI: Working with Lists of Data

In this article series, you’ve created several .NET MAUI pages, created a top-level menu system, and programmatically navigated between pages. Using data binding greatly reduces the amount of code you need to write. Using the MVVM and DI design patterns helps you create applications that are reusable, maintainable, and testable. In this article, you’ll display lists of data and navigate from a list item to the detail page for that item. .NET MAUI provides ListView, CollectionView, and CarouselView controls for displaying lists. Each list control is illustrated, and you’re provided with guidance on what each control is best at displaying.

## Display a List of Users Using a ListView

Now that you’ve learned about the standard controls for input, let's turn your attention to working with lists of data. You’ve already seen the Picker control for displaying a small set of data. Other list controls are ListView, CollectionView, and CarouselView. The **ListView** control displays a scrolling vertical list of items. You define what each item looks like by creating a **DataTemplate** within which you use a **Cell**. A DataTemplate is a set of Cells used to display each record within your collection. As the list control iterates over your data collection, each record is displayed according to the control layout within the DataTemplate.

There are three **Cell** types used most often in a DataTemplate to define how the list of data looks. A **TextCell** displays two pieces of information on each row of the ListView: a piece of large text is set using the **Text** property, then, on the next line, a piece of smaller text is set using the **Detail** property. An **ImageCell** adds an **ImageSource** property while still using the **Text** and **Detail** properties to display large and small text next to the image. Finally, **ViewCell** allows complete flexibility to define the look for each element displayed in the list. I prefer to use ViewCell as it provides the most flexibility.

### Modify the User View Model to Get a List of Users

When using .NET MAUI or WPF, always use the **ObservableCollection<T>** class for all your lists, as opposed to a List<T>. The ObservableCollection<T> object raises the appropriate notifications when the list changes. If an item is added, updated, or removed from the list, or if the whole list is recreated, any bound objects to this collection are informed and redisplay themselves using the new information in the collection.

Before you can display a list of users, you first need to modify the UserViewModel class to retrieve a list of users from the UserRepository. Open the **ViewModelClasses\UserViewModel.cs** file and add a new private property to hold a collection of users.

private ObservableCollection<User> \_Users = new();

Add a new public property named **Users** to hold the list of users and to give a public collection property to bind to on the user list page.

public ObservableCollection<User> Users {

 get { return \_Users; }

 set {

 \_Users = value;

 RaisePropertyChanged(nameof(Users));

 }

}

Locate the GetAsync() method and modify it to use the UserRepository class to retrieve a list of users, as shown in **Listing 1.** Check to ensure that the Repository variable is not equal to null. If it’s not null, call the GetAsync() method on the Repository class to return the set of users and assign that collection to the **Users** property. Set the **RowsAffected** property to the **Count** of the users. Also set the **InfoMessage** property to inform the user how many rows of users were found. Later, you’ll add a label to display this informational message.

### Using the TextCell

Let's modify the user list page to display a list of users using a ListView control and a TextCell. Open the **Views\UserListView.xaml** file and add a few XML namespaces to the partial views' namespace, the view model namespace, and the entity layer namespace.

xmlns:partial="clr-namespace:

 AdventureWorks.MAUI.ViewsPartial"

xmlns:vm="clr-namespace:

 AdventureWorks.MAUI.MauiViewModelClasses"

xmlns:model="clr-namespace:

 AdventureWorks.EntityLayer;

 assembly=AdventureWorks.EntityLayer"

Remove the entire <VerticalStackLayout> element and replace it with the code shown in **Listing 2**. In the ListView control, set the **SeparatorColor** and **SeparatorVisibility** properties to provide separation between each item on the Android and iOS platforms. Set the **ItemsSource** property of the ListView to the **Users** collection property of the view model. Create a <ListView.ItemTemplate> element that encloses the <DataTemplate> element. Add the x:DataType attribute on the DataTemplate starting tag to inform the Cell what type of object is expected for each row of the collection. Within the DataTemplate, add a <TextCell> element with the **Text** and **Detail** properties set to bind to the appropriate properties in the User class.

Open the **Views\UserListView.xaml.cs** file and replace the entire contents of the file with the code shown in **Listing 3**. The constructor for this class needs to have the UserViewModel injected. Assign the instance of the view model class passed into the private **\_ViewModel** variable. In the OnAppearing() method set the **BindingContext** property of the ContentPage to the **\_ViewModel** variable. Finally, call the GetAsync() method on the UserViewModel class to populate the **Users** property so the ListView control can bind to the list of users.

Open the **ExtensionClasses\ServiceExtensions.cs** file and add the UserListView page as a new service to the DI container. This page needs to participate in DI to have the UserViewModel injected into its constructor. Add the following line of code to the AddViewClasses() method.

services.AddScoped<UserListView>();

### Try It Out

Run the application on Windows and click on the **Users** menu to see the list of users, as shown in **Figure 1**. Notice that there’s no separator displayed on a Windows computer. For whatever reason, Microsoft chose not to respect the **SeparatorColor** and **SeparatorVisibility** properties on a Windows computer. Thus, the TextCell is not how I would choose to display data on a ListView. Instead, you should use a ViewCell as that allows you to keep the interface consistent between all platforms.



**Figure 1:** A ListView control using a TextCell for displaying user information on Windows

Run the application on the Android emulator and you should see that there’s a separator displayed, as shown in **Figure 2**. On both Android and iOS platforms, both the **SeparatorColor** and **SeparatorVisibility** properties are respected.



**Figure 2:** A ListView control using a TextCell for displaying user information on Android

### Using the ViewCell

Open the **Views\UserListView.xaml** file and replace the <TextCell> element with the XAML shown in **Listing 4**. When you use a ViewCell control, you have complete flexibility to use any layout you want for each row in your collection. In this ViewCell, create a VerticalStackLayout element with a spacing of five device-independent pixels between each child element. Four HorizontalStackLayout elements are created with Label controls in each to define where the various properties of each User object are to be displayed (see **Figure 3**). The last element within the VerticalStackLayout is a BoxView control with a keyed style attached to it.



**Figure 3:** Create your own custom template using ViewCell.

Open the **Resources\Styles\CommonStyles.xaml** file located in the Common.Library.MAUI file and add the new **keyed** style for the BoxView control. This style is used to separate each user in the ListView control, as shown in the following XAML:

<Style TargetType="BoxView"

 x:Key="Grid.Item.Separator">

 <Setter Property="VerticalOptions"

 Value="Fill" />

 <Setter Property="BackgroundColor"

 Value="Black" />

 <Setter Property="HeightRequest"

 Value="2" />

 <Setter Property="Margin"

 Value="0,0,0,10" />

 </Style>

When you use a ViewCell template, add the HasUnevenRows="True" to the ListView control or the list won’t render correctly on mobile devices. Remove the **SeparatorColor** and **SeparatorVisibility** properties and add the **HasUnevenRows="True"** attribute to the ListView control, as shown in the following code:

<ListView Grid.Row="1"

 **HasUnevenRows="True"**

 ItemsSource="{Binding Users}">

### Try It Out

Run the application and click on the **Users** menu to see the new user template as was shown in **Figure 3**. If you run the application on the Android emulator or on iOS, it should look the same as on Windows. This is the main reason I like to use ViewCell: The results on each platform almost always look the same.

## Display User Detail View from List View

Because you removed the button that read **Navigation to Detail** from the user list page, you removed the ability to navigate to the user detail page. Let's now add that capability back in by clicking on the edit button on one of the users in the list.

### Add Command for Editing

Open the **MauiViewModelClasses\UserViewModel.cs** file and add a new method to navigate to the user detail page. As mentioned previously, .NET MAUI uses a URI-based navigation system. As such, you pass parameters between pages by using a key/value pair as you would for a web application. The first key/value pair is separated from the name of the page to navigate to by a question mark, whereas succeeding key/value pairs are separated using an ampersand. If you type in the following code, be sure there are no spaces within the interpolated string.

#region EditAsync Method

protected async Task EditAsync(int id) {

 await Shell.Current.GoToAsync(

 $"{nameof(Views.UserDetailView)}?id={id}");

}

#endregion

Define an **EditCommand** property of the type ICommand to which you can map the **Command** property on the Edit button, as shown in the following code snippet.

public ICommand? EditCommand { get; private set; }

Create an instance of this new command property in the Init() method and set it to call the EditAsync() method you just created.

EditCommand = new Command<int>(

 async (int id) => await EditAsync(id));

The edit button is contained within the DataTemplate in the ListView control. As such, the **x:DataType** is mapped to the **User** object and not the user view model class. Because the EditCommand property you just created is in the view model, you need a reference back to that view model. Open the **Views\UserListView.xaml** file and add an **x:Name** attribute to the ContentPage.

x:Name="UserListPage"

Now that you’ve defined a name that refers to the ContentPage, and that page has its **x:DataType** equal to the UserViewModel object, this allows you to reference that view model by adding a **Source={x:Reference UserListPage}** within the **Binding** markup extension. Locate the ImageButton control for editing and make it look like the following XAML.

<ImageButton Source="edit.png"

 ToolTipProperties.Text="Edit User"

 **CommandParameter="{Binding UserId}"**

 **Command="{Binding**

 **Source={x:Reference UserListPage},**

 **Path=BindingContext.EditCommand}"** />

### Receive User ID in Detail Page

The UserId is passed to the user detail page as the key ID in your navigation URI. You need some way to retrieve that ID key and assign the value to a property in the UserDetailView class. Open the **Views\UserDetailView.xaml.cs** file and first add a public property named **UserId**.

public int UserId { get; set; }

Next, add a **[QueryProperty]** attribute above the **public partial class UserDetailView** definition, as shown in the code snippet below. The **QueryProperty** attribute reads the value passed as part of the key pair "id=2", for example. It maps that value to the property name specified as the first parameter to the QueryProperty attribute. In this case, it maps it to the **UserId** property you just created.

[QueryProperty(nameof(UserId), "id")]

public partial class UserDetailView

 : ContentPage {

 // REST OF THE CODE HERE

}

Locate the OnAppearing() event procedure and where you call the method GetAsync(1) with the hard-coded value one, use the **UserId** property, as shown in the code below.

// Retrieve a User

await \_ViewModel.GetAsync(UserId);

### Try It Out

Run the application and click on the **User** menu. **Click** on the edit button for the first users and the detail for the first user is displayed on the detail page (**Figure 4**). Notice that a back arrow is displayed in the upper left corner of the window shell. If you click on this, you’re returned to the user list page. If you click on each user in the list, you’re directed to the detail page and the **UserId** value for each user is passed to the detail page. The code behind on the detail page runs and the **UserId** value is passed to the GetAsync(id) method, which in turn fills the **CurrentEntity** property on the view model. Each control on the detail page is bound to a property on the **CurrentEntity** property, thus the data for the current user is displayed on the detail page.



**Figure 4:** After navigating to the detail page, a back arrow is displayed to allow you to go back to the list page.

### Save User and Return to User List View

When the user clicks on the Save button on the detail page, a few different things may happen. The data is saved, and they are returned to the list page to see their changes appear in the list. If there are one or more validation errors that occur due to bad input, a list of values to update is displayed. Another scenario is that an exception happens and an error message is displayed. You have not written code to save the data to a data store yet, but let's at least take the happy path and assume the data is saved, and you want to return the user to the list page. Open the **MauiViewModelClasses\UserViewModel.cs** file and add a **SaveAsync()** method, as shown in the following code:

#region SaveAsync Method

public override async Task<User?> SaveAsync() {

 User? ret = await base.SaveAsync();

 if (ret != null) {

 await Shell.Current.GoToAsync("..");

 }

 return ret;

}

#endregion

This method overrides the SaveAsync() in the UserViewModel base class. It calls the SaveAsync() method in the base class, which, if the user is saved successfully, a valid User object is returned. If the User object is valid, then use the GoToAsync() method to navigate back to the list page. The two dots in the double quotes tell the .NET MAUI navigation system to go back one level, or to the page that called this one. Open the **ViewModelClasses\UserViewModel.cs** file and **remove** the **System.Diagnostics.Debugger.Break();** line of code from the SaveAsync() method so you can see this code navigate back to the list page after clicking on the Save button.

### Try It Out

Run the application and click on the **User** menu. Click on the edit button for a user to see the user detail page appear. Click on the **Save** button and you’re redirected back to the user list page. There’s no code to save any changes yet, but this will come in a later article.

### Cancel Changes and Return to User List View

Besides the Save button, there is also a Cancel button on the user detail page. For the Cancel button, you want to just have it redirect back to the user list page without saving any changes made on the page. Open the **MauiViewModelClasses\UserViewModel.cs** file and add a **CancelAsync()** method. This method simply navigates back to the user list view using the GoToAsync() method.

#region CancelAsync Method

public async Task CancelAsync() {

 await Shell.Current.GoToAsync("..");

}

#endregion

As you’ve done previously, add a **CancelCommand** property to which you can map the **Command** property to the Cancel button.

public ICommand? CancelCommand

 { get; private set; }

Create an instance of this new command property in the Init() method and set it to call the CancelAsync() method you just created.

CancelCommand = new Command(

 async () => await CancelAsync());

Open the **Views\UserDetailView.xaml** file, locate the Cancel button and add a Command attribute that’s bound to the **CancelCommand** property.

<Button Text="Cancel"

 ImageSource="cancel.png"

 ContentLayout="Left"

 ToolTipProperties.Text="Cancel Changes"

 Command="{Binding CancelCommand}" />

### Try It Out

Run the application and click on the **User** menu. Click on the **Edit** button for a user to see the detail page appear. Click on the **Cancel** button and you’re redirected back to the user list page.

## Display a List of Products Using a CollectionView

You’ve seen the ListView control. Let's now build the product list page and product detail page using a **CollectionView** control. The **CollectionView** control is like the ListView in that it’s used to present a vertical list of data to the user. The CollectionView control doesn’t have any Cell controls. Instead, you create the DataTemplate in any format you wish. This provides the same flexibility as using the ViewCell you used on the ListView control. With the CollectionView control, you may select single or multiple items. The CollectionView control also uses virtualization, thus making it potentially more performant than the ListView.

To illustrate the CollectionView control, let's start building the product classes to display a list of products and display a single product object. Create a product entity class to represent each field of a product. Create a Repository class to return a list of product data, as well as a single product. Create a product view model class to support displaying a list of products and selecting a single product, just like you did with the user view model class.

### Create a Product Entity Class

Right mouse-click on the **EntityClasses** in the **AdventureWorks.EntityLayer** project and add a new class named **Product**. Replace the entire contents of this new file with the code shown in **Listing 5**. This class is designed exactly like the User class where it has private variables mapped to public properties, and which raise the PropertyChanged event when modified.

### Create a Product View Model Class

Right mouse-click on the **ViewModelClasses** in the **AdventureWorks.ViewModelLayer** project and add a new class named **ProductViewModel**. Replace the entire contents of this new file with the code shown in **Listing 6**. This view model class uses the same design pattern as the UserViewModel class you created earlier. It has properties for a repository object, a list of products, and the currently selected product. It also contains the same three methods as in the UserViewModel class: GetAsync(), GetAsync(id), and SaveAsync().

### Create a Product View Model for Commanding

Just like you did with the UserViewModel class, create a view model class in the .NET MAUI application to handle commanding for calling methods in the view model. In the **AdventureWorks.MAUI** project, right mouse-click on the **MauiViewModelClasses** folder and create a new class named **ProductViewModel**. Replace the entire contents of this new file with the code shown in **Listing 7**. You’ll add more to this class a little later.

### Create a Product Repository Class

Right mouse-click on the **RepositoryClasses** in the **AdventureWorks.DataLayer.Mock** project and add a new class named **ProductRepository**. Replace the entire contents of this new file with the code shown in **Listing 8**. The code in the ProductRepository follows the same design pattern you already learned building the UserRepository class. The GetAsync() method returns a set of four product objects and the GetAsync(id) returns a single product object.

### Build a Product List Page

You already have a product detail page, so now you need to create a page to display a list of products. Right mouse-click on the **Views** folder and add a new **Content Page (XAML)** named **ProductListView**. Change the **Title** attribute to "Product List". Add three XML namespaces to reference the partial views, the MAUI view models, and the entity layer project.

xmlns:partial="clr-namespace:

 AdventureWorks.MAUI.ViewsPartial"

xmlns:vm="clr-namespace:

 AdventureWorks.MAUI.MauiViewModelClasses"

xmlns:model="clr-namespace:

 AdventureWorks.EntityLayer;

 assembly=AdventureWorks.EntityLayer"

Add a **x:DataType** attribute to the <ContentPage> element to take advantage of compiled bindings on the page.

x:DataType="vm:ProductViewModel"

Replace the <VerticalStackLayout> with the XAML shown in **Listing 9**. Instead of a ListView control, use a CollectionView control. Specify that you only want to allow a single row to be selected at a time by setting the **SelectionMode** property to **Single**. Bind the **ItemsSource** property to the Products collection in the ProductViewModel class. The <CollectionView.ItemTemplate> element specifies the DataTemplate for how to display each item in your collection. Always add the x:DataType attribute to the DataTemplate so it knows what kind of entity object is within each row of your collection.

Why is there an ItemTemplate element around the DataTemplate? It’s because there is also a HeaderTemplate element and a FooterTemplate element that you may add to the CollectionView. These templates may also have a DataTemplate that describes how the header and footer are laid out for the CollectionView. I don't generally use these templates, but you may find them handy in some circumstances. Open the **Views\ProductListView.xaml.cs** file and replace the entire contents of this file with the code shown in **Listing 10**. Once again, this code should look very familiar to you, as it’s almost identical to the code in the User List View code behind.

### Change Application Shell and DI for Products

Now that you added a product list page and modified both the list and the product detail page to have the ProductViewModel injected into each page, you need to ensure that these pages work with DI. Open the **AppShell.xaml** file and change the <ShellContent> element for the Products tab to point to the **ProductListView** page instead of the **ProductDetailView** page, as shown in the following XAML:

<ShellContent Title="Products"

 ContentTemplate="{DataTemplate

 views:ProductListView}" />

Open the **ExtensionClasses\ServiceExtensions.cs** file and, in the AddRepositoryClasses() method, add code to inject the ProductRepository class.

services.AddScoped<IRepository<Product>,

 ProductRepository>();

In the AddViewModelClasses() method, add code to inject the ProductViewModel.

services.AddScoped<MauiViewModelClasses

 .ProductViewModel>();

In the AddViewClasses() method, add code to inject both the product detail and list pages.

services.AddScoped<ProductDetailView>();

services.AddScoped<ProductListView>();

### Try It Out

Run the application and click on the **Products** menu to see the list of products, as shown in **Figure 5**.



**Figure 5:** A product list is created using a CollectionView control.

## Display Product Detail from Collection View

Let's now modify the pages so the list can navigate to the product detail page when the edit button is clicked. Open the **Views\ProductDetailView.xaml** file and add an XML namespace to the product view model class in the MauiViewModelClasses namespace.

xmlns:vm="clr-namespace:

 AdventureWorks.MAUI.MauiViewModelClasses"

Add the **x:DataType** on the ContentPage element to specify that the ProductViewModel class is the source of the data binding for this page.

x:DataType="vm:ProductViewModel"

The XAML shown in **Listing 11** is all the controls on the ProductDetailView page with their data bindings to the appropriate properties to the Product class. This XAML isn’t too different from what you created in an earlier article; it just has the data bindings attached to the controls.

### Replace the Product Detail View Code Behind

Open the **Views\ProductDetailView.xaml.cs** file and replace the entire contents of this file with the code shown in **Listing 12**. The product detail page must accept a ProductId passed from the product list page, so add a [QueryProperty] attribute to the ProductDetailView class. Add the OnAppearing() method to set the **BindingContext** of the page to the view model injected into the constructor. Call the GetAsync(ProductId) method to load the **CurrentEntity** property with the product data to display on the controls.

### Modify Product View Model Commands

Open the **MauiViewModelClasses\ProductViewModel.cs** file you created earlier and add the code shown in **Listing 13** after the constructors. This code should look very familiar, as you used almost the exact same code in the UserViewModel that handles the commands for the user pages.

### Modify Product List View XAML

Add the code to the edit button on the product list page to bind to the **EditCommand** property in the ProductViewModel class. Open the **Views\ProductViewList.xaml** file and add an **x:Name** attribute to the ContentPage starting tag, as shown in the following XAML:

x:Name="ProductListPage"

Now that you’ve defined a name that refers to the ContentPage, and that page has as its x:DataType equal to the ProductViewModel object, this allows you to reference that view model by adding a **Source={x:Reference ProductListPage}** within the Binding markup extension. Locate the ImageButton control for the edit button and make it look like the following XAML:

<ImageButton Source="edit.png"

 ToolTipProperties.Text="Edit Product"

 **CommandParameter="{Binding ProductID}"**

 **Command="{Binding**

 **Source={x:Reference ProductListPage},**

 **Path=BindingContext.EditCommand}"** />

Because you changed the <ShellContent> element to call the ProductListView instead of the ProductDetailView page, that page is no longer in the list of routes. Open the **AppShell.xaml.cs** file and register the route for the ProductDetailView in the constructor, as shown in the following code:

Routing.RegisterRoute(

 nameof(Views.ProductDetailView),

 typeof(Views.ProductDetailView));

### Try It Out

Run the application and click on the **Product** menu. Click on the edit button for different products to see the product detail page appear as shown in **Figure 6**. Click on the **Save** or **Cancel** buttons and you are redirected back to the product list page.



**Figure 6:** Display the product detail information by navigating from a product on the list page.

## Display Colors Using a Carousel View (Mobile Development Only)

By default, a CarouselView control displays items horizontally so you can swipe back and forth to navigate from one item to another. Swiping backward from the first item in the collection displays the last item in the collection. Similarly, swiping forward from the last item in the collection returns to the first item in the collection. Note that the CarouselView does not currently work very well in Windows applications.

CarouselView shares much of its implementation with CollectionView. However, the two controls have different use cases. CollectionView is used to present lists of data of any length, whereas CarouselView is used to display information from a small list of items.

The CarouselView works closely with an IndicatorView control (see **Figure 7**). The IndicatorView control shows how many items are in the CarouselView by using a circular dot (by default). One dot represents a single item in your collection being displayed in the CarouselView control. You may swipe left or right on the CarouselView to display each item in the collection, or you may click on one of the dots in the IndicatorView to move through the collection.



**Figure 7:** Use the CarouselView and IndicatorView controls together.

### Create a Color Entity Class

When you’re using a color on the product detail page, you only need to set the color name. However, in the Color entity class you’re going to create, add a **ColorId** property as well, in case you wish to store colors for your products in a data store that requires a unique primary key. Right mouse-click on the **EntityClasses** folder in the **AdventureWorks.EntityLayer** project and add a new class named **Color**. Replace the entire contents of this new file with the code shown in **Listing 14**.

### Create a Color Repository Class

Create a mock repository class to return a list of hard-coded Color objects. Right mouse-click on the **RepositoryClasses** folder in the **AdventureWorks.DataLayer.Mock** project and add a new class named **ColorRepository**. Replace the entire contents of this new file with the code shown in **Listing 15**. This code follows the same design pattern as that you used for the user and product repository classes.

### Create a Color View Model

Now that you have the entity and repository classes for colors created, create a color view model class. Right mouse-click on the **ViewModelClasses** folder in the **AdventureWorks.ViewModelLayer** project and add a new class named **ColorViewModel**. Replace the entire contents of this new file with the code shown in **Listing 16**.

You now need a color view model class to handle commanding in .NET MAUI. Right mouse-click on the **MauiViewModelClasses** folder and add a new class named **ColorViewModel**. Replace the entire contents of this new file with the code shown in **Listing 17**.

### Create a Color Detail View Page

After displaying a list of colors, you should also allow the user to add or edit colors by creating a color detail page. Right mouse-click on the Views folder and select **Add > New Item…** and then **.NET MAUI > .NET MAUI ContentPage (XAML)**. Set the Name to **ColorDetailView** and click the Add button. Set the **Title** attribute to **Color Information**. On the **Views\ColorDetailView.xaml** file, add a couple XML namespaces to the partial views' namespace and the view model namespace.

xmlns:partial="clr-namespace:

 AdventureWorks.MAUI.ViewsPartial"

xmlns:vm="clr-namespace:

 AdventureWorks.MAUI.MauiViewModelClasses"

Add an x:DataType attribute to the page to take advantage of compiled bindings.

x:DataType="vm:ColorViewModel"

Replace the <HorizontalStackLayout> element with the code shown in **Listing 18**. You don't need to add or edit the **ColorId** property from the Color class, so just add a single Label and an Entry control for the **ColorName** property.

Open the **Views\ColorDetailView.xaml.cs** file and replace the entire contents of this file with the code shown in **Listing 19**. This code should look very familiar by now, as it follows the same design pattern as the one established in the user and product detail pages.

### Modify AppShell

The color list page is called from a ShellContent element in the AppShell, so it’s already registered in route navigation. However, you need to register the color detail page, as it’s not referenced anywhere in XAML. Open the **AppShell.xaml.cs** file and register the route to the ColorDetailView in the constructor.

Routing.RegisterRoute(

 nameof(Views.ColorDetailView),

 typeof(Views.ColorDetailView));

### Modify the Color List Page

Let's fix up the color list page you added before and use the CarouselView control. Open the **Views\ColorListView** file. Add XML namespaces for the commanding view models, entity layer, and partial views, as shown in the following code:

xmlns:partial="clr-namespace:

 AdventureWorks.MAUI.ViewsPartial"

xmlns:vm="clr-namespace:

 AdventureWorks.MAUI.MauiViewModelClasses"

xmlns:model="clr-namespace:

 AdventureWorks.EntityLayer;

 assembly=AdventureWorks.EntityLayer"

Add an **x:DataType** attribute to the <ContentPage.Resources> element to take advantage of compiled data bindings.

x:DataType="vm:ColorViewModel"

Add an x:Name attribute so the edit button on the list can reference back to the page to call the EditCommand property in the color view model.

x:Name="ColorListPage"

When you created the color list page, you added a single <Label> element to display text that described this page. Replace the <Label> element with the following XAML.

<Border Style="{StaticResource Screen.Border}">

 <Grid Padding="20">

 <Grid.RowDefinitions>

 <RowDefinition Height="Auto" />

 <RowDefinition Height="\*" />

 <RowDefinition Height="Auto" />

 </Grid.RowDefinitions>

 <partial:HeaderView

 ViewTitle="Color List"

 ViewDescription="The list of colors

 in the system." />

 </Grid>

</Border>

Just after the <partial:HeaderView…> element, add the XAML shown in **Listing 20**. The CarouselView control has an **IndicatorView** property to which you set the name of the IndicatorView control. As you can see, the x:Name attribute defined on the IndicatorView is set to **colorIndicators**, thus the **IndicatorView** property on the CarouselView control is set to this same name.

As you swipe through the CarouselView, or if you click a dot on the IndicatorView, the same element in the collection is selected, thus updating the other control. In the DataTemplate for the CarouselView a <Rectangle> element is used to display the color name as an actual color on the screen (as you saw in **Figure 7**). This is accomplished by binding the BackgroundColor property to the **ColorName** property on the Color object being displayed.

To load the colors, pass the ColorViewModel class to the color list page. Open the **ColorListView.xaml.cs** file and make it look like the code in **Listing 21**.

Open the **ExtensionClasses\ServiceExtensions.cs** file and in the AddRepositoryClasses() method, add the ColorRepository to the DI container.

services.AddScoped<IRepository<EntityLayer.Color>,

 ColorRepository>();

In the AddViewModelClasses() method, add the ColorViewModel to the DI container.

services.AddScoped<MauiViewModelClasses

 .ColorViewModel>();

In the AddViewClasses() method, add the ColorListView and ColorDetailView to the DI container.

services.AddScoped<ColorDetailView>();

services.AddScoped<ColorListView>();

### Try It Out

Within Visual Studio, switch to the **Android Emulator** and run the application. Click on the **Maintenance > Colors** menu. Swipe through the colors to see the IndicatorView control update and view the different colors, as shown in **Figure 7**. Click on the edit button to view the color detail page.

## Summary

In this article, you used different list controls available in .NET MAUI to display a collection of data. Use the ObservableCollection class for collections as any changes made to the collection are immediately reflected in any bound list controls. The ListView control has a few different ways to present data, but the most flexible is to use the ViewCell. The CollectionView control is the preferred control to use as it’s the most flexible and performant of any of the list controls. If you have a small set of data, you might use the CarouselView combined with the IndicatorView. However, be aware that currently, this control is only optimized to work on the Android and iOS platforms. Coming up in the next article, you’ll learn to display information and error messages, display pop-up dialogs, and to validate data in .NET MAUI.

Getting the Sample Code

You can download the sample code for this article by visiting [www.CODEMag.com](http://www.codemag.com/) under the issue and article, or by visiting [www.pdsa.com/downloads](http://www.pdsa.com/downloads). Select "Articles" from the Category drop-down. Then select "Exploring .NET MAUI: Working with Lists of Data" from the Item drop-down.

Sponsored Sidebar

Ready to Modernize a Legacy App?

Need advice on migrating yesterday’s legacy applications to today’s modern platforms?

Take advantage of CODE Consulting’s years of experience and contact us today to schedule a FREE consulting call to discuss your options.

No strings. No commitment.

For more information: [www.codemag.com/consulting](http://www.codemag.com/consulting) or email us at info@codemag.com.

Paul D. Sheriff

http://www.pdsa.com

Paul has been working in the IT industry since 1985. In that time, he has successfully assisted hundreds of companies' architect software applications to solve their toughest business problems. Paul has been a teacher and mentor through various mediums such as video courses, blogs, articles and speaking engagements at user groups and conferences around the world. Paul has multiple courses in the www.pluralsight.com library (<https://bit.ly/3gvXgvj>) and on YouTube.com ([https://www.youtube.com/@pauldsheriff](https://www.youtube.com/%40pauldsheriff)) on topics ranging from C#, LINQ, JavaScript, Angular, MVC, WPF, XML, jQuery, and Bootstrap. Contact Paul at psheriff@pdsa.com.

Listing 1: Retrieve a list of users from the UserRepository class and set the Users property in the view model.

#region GetAsync Method

public async

 Task<ObservableCollection<User>> GetAsync() {

 RowsAffected = 0;

 try {

 if (\_Repository == null) {

 LastErrorMessage = REPO\_NOT\_SET;

 }

 else {

 Users = await \_Repository.GetAsync();

 RowsAffected = Users.Count;

 InfoMessage =

 $"Found {RowsAffected} Users";

 }

 }

 catch (Exception ex) {

 PublishException(ex);

 }

 return Users;

}

#endregion

Listing 2: Create a ListView with a TextCell to display all users in a list.

<Border Style="{StaticResource Border.Page}">

 <Grid Style="{StaticResource Grid.Page}">

 <Grid.RowDefinitions>

 <RowDefinition Height="Auto" />

 <RowDefinition Height="\*" />

 </Grid.RowDefinitions>

 <partial:HeaderView

 ViewTitle="User List"

 ViewDescription="The list of users

 in the system." />

 <ListView Grid.Row="1"

 SeparatorColor="Black"

 SeparatorVisibility="Default"

 ItemsSource="{Binding Users}">

 <ListView.ItemTemplate>

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

 <TextCell Text="{Binding FullName}"

 Detail="{Binding Email}" />

 </DataTemplate>

 </ListView.ItemTemplate>

 </ListView>

 </Grid>

</Border>

Listing 3: Call the GetAsync() method on the \_ViewModel variable to retrieve the list of users to display on this page.

using AdventureWorks.MAUI.MauiViewModelClasses;

namespace AdventureWorks.MAUI.Views;

public partial class UserListView

 : ContentPage {

 public UserListView(UserViewModel viewModel) {

 InitializeComponent();

 \_ViewModel = viewModel;

 }

 private readonly UserViewModel \_ViewModel;

 protected async override void OnAppearing() {

 base.OnAppearing();

 BindingContext = \_ViewModel;

 await \_ViewModel.GetAsync();

 }

}

Listing 4: Use a ViewCell to provide the most consistent UI between all platforms.

<ViewCell>

 <VerticalStackLayout Spacing="5">

 <HorizontalStackLayout>

 <Label FontAttributes="Bold"

 FontSize="Title"

 Text="{Binding LastNameFirstName}" />

 </HorizontalStackLayout>

 <HorizontalStackLayout>

 <Label FontAttributes="Bold"

 Text="Email" />

 <Label Text="{Binding Email}" />

 </HorizontalStackLayout>

 <HorizontalStackLayout>

 <Label FontAttributes="Bold"

 Text="Phone" />

 <Label Text="{Binding Phone}" />

 </HorizontalStackLayout>

 <HorizontalStackLayout>

 <ImageButton Source="edit.png"

 ToolTipProperties.Text="Edit User" />

 <ImageButton Source="trash.png"

 ToolTipProperties.Text="Delete User" />

 </HorizontalStackLayout>

 <BoxView Style="{StaticResource

 Grid.Item.Separator}" />

 </VerticalStackLayout>

</ViewCell>

Listing 5: Create a Product entity class to represent a single product.

using Common.Library;

namespace AdventureWorks.EntityLayer;

public partial class Product : EntityBase {

 #region Private Variables

 private int \_ProductID;

 private string \_Name = string.Empty;

 private string \_ProductNumber

 = string.Empty;

 private string \_Color = string.Empty;

 private decimal \_StandardCost = 1;

 private decimal \_ListPrice = 2;

 private string? \_Size = string.Empty;

 private decimal? \_Weight;

 private int? \_ProductCategoryID;

 private int? \_ProductModelID;

 private DateTime \_SellStartDate;

 private DateTime? \_SellEndDate;

 private DateTime? \_DiscontinuedDate;

 private DateTime \_ModifiedDate;

 #endregion

 #region Public Properties

 public int ProductID {

 get { return \_ProductID; }

 set {

 \_ProductID = value;

 RaisePropertyChanged(nameof(ProductID));

 }

 }

 public string Name {

 get { return \_Name; }

 set {

 \_Name = value;

 RaisePropertyChanged(nameof(Name));

 }

 }

 public string ProductNumber {

 get { return \_ProductNumber; }

 set {

 \_ProductNumber = value;

 RaisePropertyChanged(nameof(ProductNumber));

 }

 }

 public string Color {

 get { return \_Color; }

 set {

 \_Color = value;

 RaisePropertyChanged(nameof(Color));

 }

 }

 public decimal StandardCost {

 get { return \_StandardCost; }

 set {

 \_StandardCost = value;

 RaisePropertyChanged(nameof(StandardCost));

 }

 }

 public decimal ListPrice {

 get { return \_ListPrice; }

 set {

 \_ListPrice = value;

 RaisePropertyChanged(nameof(ListPrice));

 }

 }

 public string? Size {

 get { return \_Size; }

 set {

 \_Size = value;

 RaisePropertyChanged(nameof(Size));

 }

 }

 public decimal? Weight {

 get { return \_Weight; }

 set {

 \_Weight = value;

 RaisePropertyChanged(nameof(Weight));

 }

 }

 public int? ProductCategoryID {

 get { return \_ProductCategoryID; }

 set {

 \_ProductCategoryID = value;

 RaisePropertyChanged(nameof(ProductCategoryID));

 }

 }

 public int? ProductModelID {

 get { return \_ProductModelID; }

 set {

 \_ProductModelID = value;

 RaisePropertyChanged(nameof(ProductModelID));

 }

 }

 public DateTime SellStartDate {

 get { return \_SellStartDate; }

 set {

 \_SellStartDate = value;

 RaisePropertyChanged(nameof(SellStartDate));

 }

 }

 public DateTime? SellEndDate {

 get { return \_SellEndDate; }

 set {

 \_SellEndDate = value;

 RaisePropertyChanged(nameof(SellEndDate));

 }

 }

 public DateTime? DiscontinuedDate {

 get { return \_DiscontinuedDate; }

 set {

 \_DiscontinuedDate = value;

 RaisePropertyChanged(nameof(DiscontinuedDate));

 }

 }

 public DateTime ModifiedDate {

 get { return \_ModifiedDate; }

 set {

 \_ModifiedDate = value;

 RaisePropertyChanged(nameof(ModifiedDate));

 }

 }

 #endregion

}

Listing 6: Create a Product view model for data binding on the product pages.

using AdventureWorks.EntityLayer;

using Common.Library;

using System.Collections.ObjectModel;

namespace AdventureWorks.ViewModelLayer;

public class ProductViewModel : ViewModelBase {

 #region Constructors

 public ProductViewModel() : base() {

 }

 public ProductViewModel(

 IRepository<Product> repo) : base() {

 \_Repository = repo;

 }

 #endregion

 #region Private Variables

 private readonly IRepository<Product>? \_Repository;

 private ObservableCollection<Product> \_Products = new();

 private Product? \_CurrentEntity = new();

 #endregion

 #region Public Properties

 public ObservableCollection<Product> Products {

 get { return \_Products; }

 set {

 \_Products = value;

 RaisePropertyChanged(nameof(Products));

 }

 }

 public Product? CurrentEntity {

 get { return \_CurrentEntity; }

 set {

 \_CurrentEntity = value;

 RaisePropertyChanged(nameof(CurrentEntity));

 }

 }

 #endregion

 #region GetAsync Method

 public async

 Task<ObservableCollection<Product>>

 GetAsync() {

 RowsAffected = 0;

 try {

 if (\_Repository == null) {

 LastErrorMessage = REPO\_NOT\_SET;

 }

 else {

 Products = await \_Repository.GetAsync();

 RowsAffected = Products.Count;

 InfoMessage = $"Found {RowsAffected} Products";

 }

 }

 catch (Exception ex) {

 PublishException(ex);

 }

 return Products;

 }

 #endregion

 #region GetAsync(id) Method

 public async Task<Product?>

 GetAsync(int id) {

 try {

 // Get a Product from a data store

 if (\_Repository != null) {

 CurrentEntity = await \_Repository.GetAsync(id);

 if (CurrentEntity == null) {

 InfoMessage = $"Product id={id} was not found.";

 }

 else {

 InfoMessage = "Found the Product";

 }

 }

 else {

 LastErrorMessage = REPO\_NOT\_SET;

 InfoMessage = "Found a MOCK Product";

 // MOCK Data

 CurrentEntity =

 await Task.FromResult(new Product {

 ProductID = id,

 Name = "A New Product",

 Color = "Black",

 StandardCost = 10,

 ListPrice = 20,

 SellStartDate = Convert.ToDateTime("7/1/2023"),

 Size = "LG"

 });

 }

 RowsAffected = 1;

 }

 catch (Exception ex) {

 RowsAffected = 0;

 PublishException(ex);

 }

 return CurrentEntity;

 }

 #endregion

 #region SaveAsync Method

 public async virtual Task<Product?>

 SaveAsync() {

 // TODO: Write code to save data

 return await Task.FromResult(new Product());

 }

 #endregion

}

Listing 7: Create a view model in the .NET MAUI application for working with product data.

using AdventureWorks.EntityLayer;

using Common.Library;

namespace AdventureWorks.MAUI.MauiViewModelClasses;

public class ProductViewModel :

 AdventureWorks.ViewModelLayer.ProductViewModel {

 #region Constructors

 public ProductViewModel() : base() {

 }

 public ProductViewModel(

 IRepository<Product> repo) : base(repo) {

 }

 #endregion

}

Listing 8: Create a product repository class to return a set of product objects.

using AdventureWorks.EntityLayer;

using Common.Library;

using System.Collections.ObjectModel;

namespace AdventureWorks.DataLayer;

/// <summary>

/// Creates a set of Product mock data

/// </summary>

public partial class ProductRepository

 : IRepository<Product> {

 #region GetAsync Method

 public async

 Task<ObservableCollection<Product>>

 GetAsync() {

 return await Task.FromResult(

 new ObservableCollection<Product>

 {

 new() {

 ProductID = 680,

 Name = @"HL Road Frame - Black, 58",

 ProductNumber = @"FR-R92B-58",

 Color = @"Black",

 StandardCost = 100.0000m,

 ListPrice = 1431.5000m,

 Size = @"58",

 Weight = 1016.04m,

 ProductCategoryID = 18,

 ProductModelID = 6,

 SellStartDate

 = new DateTime(2002, 6, 1),

 SellEndDate = null,

 DiscontinuedDate = null,

 ModifiedDate

 = new DateTime(2008, 3, 11),

 },

 new() {

 ProductID = 707,

 Name = @"Sport-100 Helmet, Red",

 ProductNumber = @"HL-U509-R",

 Color = @"Red",

 StandardCost = 13.0863m,

 ListPrice = 34.9900m,

 Size = null,

 Weight = 3.4m,

 ProductCategoryID = 35,

 ProductModelID = 33,

 SellStartDate

 = new DateTime(2005, 7, 1),

 SellEndDate = null,

 DiscontinuedDate = null,

 ModifiedDate

 = new DateTime(2008, 3, 11),

 },

 new() {

 ProductID = 712,

 Name = @"AWC Logo Cap",

 ProductNumber = @"CA-1098",

 Color = @"Multi",

 StandardCost = 6.9223m,

 ListPrice = 8.9900m,

 Size = null,

 Weight = 0.80m,

 ProductCategoryID = 23,

 ProductModelID = 2,

 SellStartDate

 = new DateTime(2005, 7, 1),

 SellEndDate = null,

 DiscontinuedDate = null,

 ModifiedDate

 = new DateTime(2008, 3, 11),

 },

 new() {

 ProductID = 713,

 Name = @"Long-Sleeve Logo Jersey, S",

 ProductNumber = @"LJ-0192-S",

 Color = @"Multi",

 StandardCost = 38.4923m,

 ListPrice = 49.9900m,

 Size = "S",

 Weight = null,

 ProductCategoryID = 25,

 ProductModelID = 11,

 SellStartDate

 = new DateTime(2005, 7, 1),

 SellEndDate = null,

 DiscontinuedDate = null,

 ModifiedDate

 = new DateTime(2008, 3, 11),

 }

 });

 }

 #endregion

 #region GetAsync(id) Method

 public async Task<Product?> GetAsync(int id) {

 ObservableCollection<Product> list

 = await GetAsync();

 Product? entity = list.Where(

 row => row.ProductID == id)

 .FirstOrDefault();

 return entity;

 }

 #endregion

}

Listing 9: Create the product list page using a CollectionView control.

<Border Style="{StaticResource Border.Page}">

 <Grid Style="{StaticResource Grid.Page}">

 <Grid.RowDefinitions>

 <RowDefinition Height="Auto" />

 <RowDefinition Height="\*" />

 </Grid.RowDefinitions>

 <partial:HeaderView ViewTitle="Product List"

 ViewDescription="The list of products in

 the system." />

 <CollectionView Grid.Row="1"

 SelectionMode="Single"

 ItemsSource="{Binding Products}">

 <CollectionView.ItemTemplate>

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

 <VerticalStackLayout Spacing="5">

 <HorizontalStackLayout>

 <Label FontAttributes="Bold"

 FontSize="Title"

 Text="{Binding Name}" />

 </HorizontalStackLayout>

 <HorizontalStackLayout>

 <Label FontAttributes="Bold"

 Text="Color" />

 <Label Text="{Binding Color}" />

 </HorizontalStackLayout>

 <HorizontalStackLayout>

 <Label FontAttributes="Bold"

 Text="Price" />

 <Label Text="{Binding ListPrice,

 StringFormat='{0:c}'}" />

 </HorizontalStackLayout>

 <HorizontalStackLayout>

 <ImageButton Source="edit.png"

 ToolTipProperties.Text="Edit Product" />

 <ImageButton Source="trash.png"

 ToolTipProperties.Text="Delete Product" />

 </HorizontalStackLayout>

 <BoxView Style="{StaticResource Grid.Item.Separator}" />

 </VerticalStackLayout>

 </DataTemplate>

 </CollectionView.ItemTemplate>

 </CollectionView>

 </Grid>

</Border>

Listing 10: Add code to get all products for displaying on the product list page.

using AdventureWorks.MAUI.MauiViewModelClasses;

namespace AdventureWorks.MAUI.Views;

public partial class ProductListView

 : ContentPage {

 public ProductListView(ProductViewModel viewModel) {

 InitializeComponent();

 \_ViewModel = viewModel;

 }

 private readonly ProductViewModel \_ViewModel;

 protected async override void OnAppearing() {

 base.OnAppearing();

 BindingContext = \_ViewModel;

 await \_ViewModel.GetAsync();

 }

}

Listing 11: Map the data entry controls on the product detail page to the appropriate properties in the Product class.

<Label Text="Product Name"

 Grid.Row="1" />

<Entry Grid.Column="1"

 Grid.Row="1"

 Text="{Binding CurrentEntity.Name}" />

<Label Text="Product Number"

 Grid.Row="2" />

<Entry Grid.Row="2"

 Grid.Column="1"

 Text="{Binding CurrentEntity.ProductNumber}" />

<Label Text="Color"

 Grid.Row="3" />

<Entry Grid.Row="3"

 Grid.Column="1"

 Text="{Binding CurrentEntity.Color}" />

<Label Text="Cost"

 Grid.Row="4" />

<HorizontalStackLayout Grid.Row="4"

 Grid.Column="1">

 <Entry Text="{Binding CurrentEntity.StandardCost}" />

 <Stepper Value="{Binding CurrentEntity.StandardCost}"

 Minimum="1"

 Maximum="{Binding CurrentEntity.ListPrice}"

 Increment="1" />

</HorizontalStackLayout>

<Label Text="Price"

 Grid.Row="5" />

<HorizontalStackLayout Grid.Row="5"

 Grid.Column="1">

 <Entry Text="{Binding CurrentEntity.ListPrice}" />

 <Stepper Value="{Binding CurrentEntity.ListPrice}"

 Minimum="{Binding CurrentEntity.StandardCost}"

 Maximum="9999"

 Increment="1" />

</HorizontalStackLayout>

<Label Text="Size"

 Grid.Row="6" />

<Entry Grid.Row="6"

 Grid.Column="1"

 Text="{Binding CurrentEntity.Size}" />

<Label Text="Weight"

 Grid.Row="7" />

<VerticalStackLayout Grid.Row="7"

 Grid.Column="1">

 <Slider x:Name="weight"

 Value="{Binding CurrentEntity.Weight}"

 Minimum="1"

 Maximum="99999" />

 <Label Text="{Binding CurrentEntity.Weight}" />

</VerticalStackLayout>

<Label Text="Category"

 Grid.Row="8" />

<Entry Grid.Row="8"

 Grid.Column="1"

 Text="{Binding CurrentEntity.ProductCategoryID}" />

<Label Text="Model"

 Grid.Row="9" />

<Entry Grid.Row="9"

 Grid.Column="1"

 Text="{Binding CurrentEntity.ProductModelID}" />

<Label Text="Selling Start Date"

 Grid.Row="10" />

<DatePicker Grid.Row="10"

 Grid.Column="1"

 Date="{Binding CurrentEntity.SellStartDate}" />

<Label Text="Selling End Date"

 Grid.Row="11" />

<DatePicker Grid.Row="11"

 Grid.Column="1"

 Date="{Binding CurrentEntity.SellEndDate}" />

<Label Text="Discontinued Date"

 Grid.Row="12" />

<DatePicker Grid.Row="12"

 Grid.Column="1"

 Date="{Binding CurrentEntity.DiscontinuedDate}" />

<Label Text="Product Picture"

 Grid.Row="13" />

<Image Grid.Row="13"

 Grid.Column="1"

 HorizontalOptions="Start"

 Aspect="Center"

 Source="bikeframe.jpg" />

<Label Text="Product Notes"

 Grid.Row="14" />

<Editor Grid.Row="14"

 Grid.Column="1"

 HeightRequest="100" />

<HorizontalStackLayout Grid.Row="15"

 Grid.Column="1">

 <Button Text="Save"

 ImageSource="save.png"

 ContentLayout="Left"

 ToolTipProperties.Text="Save Data"

 Command="{Binding SaveCommand}" />

 <Button Text="Cancel"

 ImageSource="cancel.png"

 ContentLayout="Left"

 ToolTipProperties.Text="Cancel Changes"

 Command="{Binding CancelCommand}" />

</HorizontalStackLayout>

Listing 12: Add code in the product detail page to retrieve a single product object.

using AdventureWorks.MAUI.MauiViewModelClasses;

namespace AdventureWorks.MAUI.Views;

[QueryProperty(nameof(ProductId), "id")]

public partial class ProductDetailView

 : ContentPage {

 public ProductDetailView(

 ProductViewModel viewModel) {

 InitializeComponent();

 \_ViewModel = viewModel;

 }

 private readonly ProductViewModel \_ViewModel;

 public int ProductId { get; set; }

 protected async override void OnAppearing() {

 base.OnAppearing();

 // Set the Page BindingContext

 BindingContext = \_ViewModel;

 // Retrieve a Product

 await \_ViewModel.GetAsync(ProductId);

 }

}

Listing 13: Add the commands to handle editing, saving, and cancelling on the product pages.

#region Commands

public ICommand? SaveCommand { get; private set; }

public ICommand? CancelCommand { get; private set; }

public ICommand? EditCommand { get; private set; }

#endregion

#region Init Method

public override void Init() {

 base.Init();

 // Create commands for this view

 SaveCommand = new Command(

 async () => await SaveAsync());

 CancelCommand = new Command(

 async () => await CancelAsync());

 EditCommand = new Command<int>(

 async (int id) => await EditAsync(id));

}

#endregion

#region SaveAsync Method

protected new async Task<Product?> SaveAsync() {

 Product? ret = await base.SaveAsync();

 if (ret != null) {

 await Shell.Current.GoToAsync("..");

 }

 return ret;

}

#endregion

#region CancelAsync Method

protected async Task CancelAsync() {

 await Shell.Current.GoToAsync("..");

}

#endregion

#region EditAsync Method

protected async Task EditAsync(int id) {

 await Shell.Current.GoToAsync(

 $"{nameof(Views.ProductDetailView)}?id={id}");

}

#endregion

Listing 14: Create an entity class to represent a color.

using Common.Library;

namespace AdventureWorks.EntityLayer;

public class Color : EntityBase {

 #region Private Variables

 private int \_ColorId = 0;

 private string \_ColorName = string.Empty;

 #endregion

 #region Public Properties

 public int ColorId {

 get { return \_ColorId; }

 set {

 \_ColorId = value;

 RaisePropertyChanged(nameof(ColorId));

 }

 }

 public string ColorName {

 get { return \_ColorName; }

 set {

 \_ColorName = value;

 RaisePropertyChanged(nameof(ColorName));

 }

 }

 #endregion

}

Listing 15: Create a repository class to return a set of colors.

using AdventureWorks.EntityLayer;

using Common.Library;

using System.Collections.ObjectModel;

namespace AdventureWorks.DataLayer;

/// <summary>

/// Creates fake data for Colors.

/// </summary>

public partial class ColorRepository

 : IRepository<Color> {

 #region GetAsync Method

 public async

 Task<ObservableCollection<Color>>

 GetAsync() {

 return await Task.FromResult(

 new ObservableCollection<Color>

 {

 new() {

 ColorId = 1,

 ColorName = "Black",

 },

 new() {

 ColorId = 2,

 ColorName = "Blue",

 },

 new() {

 ColorId = 3,

 ColorName = "Gray",

 },

 new() {

 ColorId = 4,

 ColorName = "Multi",

 },

 new() {

 ColorId = 5,

 ColorName = "Red",

 },

 new() {

 ColorId = 6,

 ColorName = "Silver",

 },

 new() {

 ColorId = 7,

 ColorName = "Silver/Black",

 },

 new() {

 ColorId = 8,

 ColorName = "White",

 },

 new() {

 ColorId = 9,

 ColorName = "Yellow",

 }

 });

 }

 #endregion

 #region GetAsync(id) Method

 public async Task<Color?> GetAsync(int id) {

 ObservableCollection<Color> list

 = await GetAsync();

 Color? entity = list.Where(

 row => row.ColorId == id)

 .FirstOrDefault();

 return entity;

 }

 #endregion

}

Listing 16: Create a Color View Model class for working with the color data store.

using AdventureWorks.EntityLayer;

using Common.Library;

using System.Collections.ObjectModel;

namespace AdventureWorks.ViewModelLayer;

public class ColorViewModel

 : ViewModelBase {

 #region Constructors

 public ColorViewModel() : base() {

 }

 public ColorViewModel(

 IRepository<Color> repo) : base() {

 \_Repository = repo;

 }

 #endregion

 #region Private Variables

 private readonly IRepository<Color>?

 \_Repository;

 private ObservableCollection<Color>

 \_Colors = new();

 private Color? \_CurrentEntity = new();

 #endregion

 #region Public Properties

 public ObservableCollection<Color> Colors {

 get { return \_Colors; }

 set {

 \_Colors = value;

 RaisePropertyChanged(nameof(Colors));

 }

 }

 public Color? CurrentEntity {

 get { return \_CurrentEntity; }

 set {

 \_CurrentEntity = value;

 RaisePropertyChanged(nameof(CurrentEntity));

 }

 }

 #endregion

 #region GetAsync Method

 public async

 Task<ObservableCollection<Color>>

 GetAsync() {

 RowsAffected = 0;

 try {

 if (\_Repository == null) {

 LastErrorMessage = REPO\_NOT\_SET;

 }

 else {

 Colors = await \_Repository.GetAsync();

 RowsAffected = Colors.Count;

 InfoMessage =

 $"Found {RowsAffected} Colors";

 }

 }

 catch (Exception ex) {

 PublishException(ex);

 }

 return Colors;

 }

 #endregion

 #region GetAsync(id) Method

 public async Task<Color?> GetAsync(int id) {

 try {

 // Get a Color from a data store

 if (\_Repository != null) {

 CurrentEntity = await \_Repository.GetAsync(id);

 if (CurrentEntity == null) {

 InfoMessage = $"Color id={id} was not found.";

 }

 else {

 InfoMessage = "Found the Color";

 }

 }

 else {

 LastErrorMessage = REPO\_NOT\_SET;

 InfoMessage = "Found a MOCK Color";

 // MOCK Data

 CurrentEntity = await Task.FromResult(new Color {

 ColorId = 1,

 ColorName = "Black",

 });

 }

 RowsAffected = 1;

 }

 catch (Exception ex) {

 PublishException(ex);

 }

 return CurrentEntity;

 }

 #endregion

 #region SaveAsync Method

 public async virtual Task<Color?> SaveAsync() {

 // TODO: Write code to save data

 return await Task.FromResult(new Color());

 }

 #endregion

}

Listing 17: Create the color view model class for commanding.

using Common.Library;

using System.Windows.Input;

namespace AdventureWorks.MAUI.MauiViewModelClasses;

public class ColorViewModel :

 AdventureWorks.ViewModelLayer.ColorViewModel

{

 #region Constructors

 public ColorViewModel() : base()

 {

 }

 public ColorViewModel(

 IRepository<EntityLayer.Color> repo) : base(repo)

 {

 }

 #endregion

 #region Commands

 public ICommand? SaveCommand { get; private set; }

 public ICommand? CancelCommand { get; private set; }

 public ICommand? EditCommand { get; private set; }

 #endregion

 #region Init Method

 public override void Init()

 {

 base.Init();

 // Create commands for this view

 SaveCommand = new Command(

 async () => await SaveAsync());

 EditCommand = new Command<int>(

 async (int id) => await EditAsync(id));

 CancelCommand = new Command(

 async () => await CancelAsync());

 }

 #endregion

 #region SaveAsync Method

 public override async

 Task<EntityLayer.Color?> SaveAsync()

 {

 EntityLayer.Color? ret

 = await base.SaveAsync();

 if (ret != null) {

 await Shell.Current.GoToAsync("..");

 }

 return ret;

 }

 #endregion

 #region CancelAsync Method

 public async Task CancelAsync()

 {

 await Shell.Current.GoToAsync("..");

 }

 #endregion

 #region EditAsync Method

 protected async Task EditAsync(int id)

 {

 await Shell.Current.GoToAsync(

 $"{nameof(Views.ColorDetailView)}?id={id}");

 }

 #endregion

}

Listing 18: Create a color detail view to edit colors.

<Border Style="{StaticResource Border.Page}">

 <ScrollView>

 <Grid Grid.Row="0"

 Grid.Column="1"

 RowDefinitions="Auto,Auto,Auto,Auto,Auto"

 ColumnDefinitions="Auto,\*"

 Style="{StaticResource Grid.Page}">

 <partial:HeaderView Grid.Row="0"

 Grid.ColumnSpan="2"

 ViewTitle="Color Information"

 ViewDescription="Use this screen

 to modify color information." />

 <Label Grid.Row="1"

 Text="Color Name" />

 <Entry Grid.Row="1"

 Grid.Column="1"

 Text="{Binding

 CurrentEntity.ColorName}" />

 <HorizontalStackLayout Grid.Row="2"

 Grid.Column="1">

 <Button Text="Save"

 ImageSource="save.png"

 ToolTipProperties.Text="Save Data"

 ContentLayout="Left"

 Command="{Binding SaveCommand}" />

 <Button Text="Cancel"

 ImageSource="cancel.png"

 ContentLayout="Left"

 ToolTipProperties.Text="Cancel Changes"

 Command="{Binding CancelCommand}" />

 </HorizontalStackLayout>

 </Grid>

 </ScrollView>

</Border>

Listing 19: Add code to display a color on the detail page.

using AdventureWorks.MAUI.MauiViewModelClasses;

namespace AdventureWorks.MAUI.Views;

[QueryProperty(nameof(ColorId), "id")]

public partial class ColorDetailView

 : ContentPage {

 public ColorDetailView(

 ColorViewModel viewModel) {

 InitializeComponent();

 \_ViewModel = viewModel;

 }

 private readonly ColorViewModel \_ViewModel;

 public int ColorId { get; set; }

 protected async override void OnAppearing() {

 base.OnAppearing();

 // Set the BindingContext to the ViewModel

 BindingContext = \_ViewModel;

 // Retrieve a Color

 await \_ViewModel.GetAsync(ColorId);

 }

}

Listing 20: Use a CarouselView, a Frame, and an IndicatorView for small lists of items.

<CarouselView Grid.Row="1"

 IndicatorView="colorIndicators"

 ItemsSource="{Binding Path=Colors}">

 <CarouselView.ItemTemplate>

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

 <StackLayout>

 <Frame HasShadow="True"

 BorderColor="DarkGray"

 CornerRadius="5"

 HorizontalOptions="Center"

 VerticalOptions="CenterAndExpand">

 <Grid Margin="20"

 RowDefinitions="Auto,Auto"

 ColumnDefinitions="Auto,\*"

 RowSpacing="20">

 <Rectangle Grid.Row="0"

 Grid.Column="0"

 BackgroundColor="{Binding ColorName}"

 WidthRequest="40"

 HeightRequest="40"

 Margin="10" />

 <Label Grid.Row="0"

 Grid.Column="1"

 FontSize="Large"

 Text="{Binding ColorName}" />

 <HorizontalStackLayout Grid.Row="1"

 Grid.ColumnSpan="2">

 <ImageButton Source="edit.png"

 ToolTipProperties.Text="Edit Color"

 CommandParameter="{Binding ColorId}"

 Command="{Binding

 Source={x:Reference ColorListPage},

 Path=BindingContext.EditCommand}" />

 <ImageButton Source="trash.png"

 ToolTipProperties.Text="Delete Color" />

 </HorizontalStackLayout>

 </Grid>

 </Frame>

 </StackLayout>

 </DataTemplate>

 </CarouselView.ItemTemplate>

</CarouselView>

<IndicatorView Grid.Row="2"

 x:Name="colorIndicators"

 Margin="5"

 IndicatorSize="20"

 IndicatorColor="LightGray"

 SelectedIndicatorColor="DarkGray"

 HorizontalOptions="Center" />

Listing 21: Load the colors for displaying in the CarouselView.

using AdventureWorks.MAUI.MauiViewModelClasses;

namespace AdventureWorks.MAUI.Views;

public partial class ColorListView : ContentPage {

 public ColorListView(ColorViewModel viewModel) {

 InitializeComponent();

 \_ViewModel = viewModel;

 }

 private readonly ColorViewModel \_ViewModel;

 protected async override void OnAppearing() {

 base.OnAppearing();

 BindingContext = \_ViewModel;

 await \_ViewModel.GetAsync();

 }

}